HAL
open science

A New Persistent Labelling Scheme for XML

Alban Gabillon, M. Fansi

To cite this version:

Alban Gabillon, M. Fansi. A New Persistent Labelling Scheme for XML. Journal of Digital Information Management, 2006, 4 (2), 5 p. hal-00994700

HAL Id: hal-00994700

https://hal.science/hal-00994700v1

Submitted on 8 Jul 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A New Persistent Labelling Scheme for XML[1]

Alban Gabillon, Majirus Fansi[2]
Université de Pau et des Pays de l'Adour
IUT des Pays de l'Adour
LIUPPA/CSYSEC
40000 Mont-de-Marsan, France
alban.gabillon@univ-pau.fr
janvier-majirus.fansi@etud.univ-pau.fr

***ABSTRACT**: With the growing importance of XML in data exchange, much research has been done in providing flexible query facilities to extract data from structured XML documents. Thereby, 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. The goal of the research reported in this paper is to design a persistent structural labelling scheme, namely a labelling scheme where labels encode ancestor-descendant relationships and sibling relationship between nodes but need not to be changed when the document is updated.*

Categories and Subject Descriptors
H.2.3 [Data Description Languages]; **D.3.2** [Language Classifications]; Extensible langauges

General Terms
Data management, XML

Keywords: Data exchange, Labeling scheme, XML Database, XML standards

Received 10 Sep. 2005; Reviewed and accepted 27 Oct. 2005

1. Introduction

XML is becoming the new standard for the exchange and publishing of data over the Internet (Bray et al., 2000). Documents obeying the XML standard can be viewed as trees. Query language like XPath (Clark et al., 1999) uses path expressions to traverse XML data. 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 document for results. Normally, a labelling scheme assigns identifiers to elements such that the hierarchical orders of the elements can be re-established based on their identifiers. Since hierarchical orders are used extensively in processing XML queries, the reduction of the computing workload for the hierarchy re-establishment is desirable.

With the growing importance of XML in data exchange and in order to achieve effective indexing, a number of labelling schemes for XML data have been proposed (Marian et al., 2001), (Abiteboul et al., 2001), (Li et al., 2001), (Cohen et al., 2002), (Alstrup et al., 2002), (Wang et al., 2003), (Bremer et al., 2003), (Chen et al., 2004), (O'Neil et al., 2004), (Kha et al., 2004), (Duong et al., 2005), (Weigel et al., 2005). All these techniques help to facilitate query processing. However, the main drawback in most of these works is the following: if deletions and/or insertions occur regularly, then expensive re-computing of affected labels is needed. Frequently re-computing large amount of elements each time XML data is updated takes time and reduces performances.

The goal of the research reported in this paper is to design a persistent structural labelling scheme for XML trees which are frequently updated. i.e. a labelling scheme where labels need not to be changed when the document is updated. Therefore, the contribution of this paper to the existing labelling schemes for XML document can briefly be summarized as follows:

- We compute the label of the newly inserted nodes without any need of re-computing existing labels.
- Our proposal supports the representation of ancestor/descendant relationships and sibling relationship. One can quickly determine the relationship between any two given nodes by looking at their unique codes.
- Given the unique code of a node, one can easily determine its level and its ancestors.

The remainder of this paper is organized as follows: we start with preliminary definitions in section 2. In section 3 we first describe the characteristic properties that distinguish identification schemes known from the literature. We then point out the limitations of these proposals assuming documents are frequently updated. Section 4 presents our proposed labelling scheme. In section 5, we conduct some experiments to evaluate our scheme. Finally, section 6 concludes this paper.

2. Preliminaries

We start by defining some of the basic terms used in the sequel. Some of these notions were recently defined in (Cohen et al., 2002). We distinguish two families of labelling scheme:

- *Static structural labelling scheme*
- *Persistent structural labelling scheme*

A static structural labelling scheme is a triple, <ancestor,sibling,l>, where ancestor and sibling are predicates over unique codes and l is a labelling function that given a tree t assigns a distinct code l(v) to each node $v \in t$. Predicates ancestor and sibling and the labelling function l are such that for every tree t and every two nodes $v,u \in t$, ancestor(l(v),l(u)) evaluates to TRUE iff v is an ancestor of u and sibling(l(v),l(u)) evaluates to TRUE iff v and u are siblings.

A persistent structural labelling *scheme* is also a triple <ancestor,sibling,l> where ancestor and sibling are as before. The labelling function l, however, supports updates into a tree without any need of re-labelling existing nodes. Each insertion is of the form "insert node u as a child of node v, after or before node w". l does not know the insertion sequence in advance. l assigns a label to each inserted node. That label cannot be changed subsequently.

Following techniques can be used to assign labels to nodes:

- *Interval scheme*
- *Range labelling scheme*
- *Prefix labelling scheme*
- *Number-based scheme*

The interval scheme requires numbering the leaves from left to right and labelling each node with a pair consisting of the smallest and largest labels attached to its descendant leafs. An ancestor test then amounts to an interval containment test on the labels. However, if the tree is updated and new leaves are added then labels need to be recomputed. One may try to fix this by leaving some "gaps" between the numbers of the leaves. But if one part of the document is heavily updated then we may run out of available numbers and need re-labelling.

The range labelling scheme comes equipped with some order relation $\leq$ over unique codes. The label of a node v is interpreted as a pair of strings a_v, b_v and the predicate ancestor is such that a node v is an ancestor of u iff $a_v \leq a_u \leq b_u \leq b_v$. The interval scheme described above is an example of such labelling; a_v and b_v are interpreted as integers with $\leq$ being the standard order relation

[1] This work is supported by funding from the French ministry for research under "ACI Sécurité Informatique 2003-2006. Projet CASC".
[2] Majirus Fansi holds a Ph.D Scholarship granted by the Conseil Général des Landes.

over integers.

In the prefix labelling scheme (or *path-based labelling scheme*) the predicate `ancestor` is such that a node `v` is an ancestor of `u` iff `l(v)` is a prefix of `l(u)`. Consequently, labels are of varying length.

The number-based scheme uses atomic numbers to identify nodes. Ancestor/descendant relationships and sibling relationship can be computed with some arithmetic operations, according to the numbering procedure.

As we shall see in section 4, our labelling scheme is a persistent, prefix-based and path-based labelling scheme.

In (Weigel et al., 2005), authors point out that the numbering scheme is complementary to optimization techniques and that it should help solving the reconstruction and decision problems which can be defined as follows:

- *Reconstruction problem*: how parts of the tree structure of the database can be reconstructed without accessing the database, given the label of a node.
- *Decision problem*: how to determine relations between two nodes without accessing the database, given their labels.

In section 4, we also present the reconstruction and decision procedures of our scheme.

3. Related Work

The problem of designing a persistent labelling scheme for identifying nodes has been studied recently (Wang et al., 2003), (Kha et al., 2004), (O'Neil et al., 2004), (Chen et al., 2004), (Duong et al., 2005), (Weigel et al., 2005). In this section we report the characteristics of some major existing identification schemes.

3.1 The Work of O'Neil et al. (O'Neil et al., 2004)

O'Neil et al. suggest a prefix based labelling scheme called ORDPATH. To our knowledge it is the first scheme to allow for arbitrary updates without changing any existing label. ORDPATH is similar conceptually to the Dewey Order described in (Tatarinov et al., 2002). ORDPATH encodes the parent-child relationship by extending the parent's ORDPATH label with a component for the child. E.g.: 1.5.3.9 is the parent, 1.5.3.9.1 the child. The various child components reflect the children relative sibling order, so that byte-by-byte comparison of the ORDPATH labels of two nodes yields the proper document order. The main difference between ORDPATH and Dewey order is that, in ORDPATH even number is reserved for further node insertions. An example of tree labelling using ORDPATH is depicted in figure 1.

Update with ORDPATH

ORDPATH assigns only positive, odd integers during an initial labelling; even and negative component values are reserved for later insertions into an existing tree, as explained below.

After an initial load, authors label a newly inserted node to the right of all existing children of a node by adding +2 to the last ordinal of the last child. In order to insert a node on the left of all existing children of the node, they label a newly inserted node by adding -2 to the last ordinal of the first child, using negative ordinal values when needed.

Authors insert a new node y between any two siblings of a parent node x by creating a component with an even ordinal falling between the final (odd) ordinals of the two siblings, then following this with a new odd component, usually 1. For example, the sequence to fall between sibling nodes 3.5.5 and 3.5.7, by providing the new siblings with the even caret 6 are: 3.5.6.1, 3.5.6.3, 3.5.6.5, …. The value 6 in component 3 (or any even value in any non-terminal component) represents a caret only, that is, it does not count as a component that increases the depth of the node in the tree.

However, this approach is not suitable for deep trees. In order to cope with such trees, ORDPATH uses labels that do not reflect ancestry and thereby looses some of its expressivity, neither supporting decision of the next sibling relation nor reconstruction of sibling or child nodes. Therefore, as in (Marian et al., 2001), their labels cannot be used for structural queries.

3.2 The Work of Duong et al. (Duong et al., 2005)

In (Duong et al., 2005), Duong et al. propose a labelling scheme, called LSDX, which aims to support the demand of updating `XML` data without the need of re-labelling existing labels. LSDX uses a combination of numbers and letters to create unique codes for `XML` data. The approach works as follows: given a node `v` with `n` child nodes: $u_1, u_2, \ldots, u_n$, a unique code of u_1 consists of its level followed by the code of its parent node followed by "`.`" followed by `b`. The unique code of u_2 consists of its level followed by the code of its parent node followed by "`.`" followed by `c`. The labelling continues for the remaining child nodes in alphabetical order.

In order to cope with further updates, authors state a rule for generating labels for new nodes without altering existing labels. However there are cases where this scheme leads to collisions between labels.

Let us consider figure 2. Let us assume we need to add a node between nodes "1a.z" and "1a.zb". According to the LSDX scheme, the inserted node is labelled with "1a.zbb". Now, if we insert a node between nodes "1a.zb" and "1a.zc" then the new node will also be assigned label "1a.zbb". A consequence of this collision is that the LSDX scheme is not injective and then is not applicable if updates occur.

3.3 The Work of Weigel et al. (Weigel et al., 2005)

Weigel et al. suggest a node identification scheme called BIRD numbers (Balanced Index-based numbering scheme for Reconstruction and Decision) where node identifiers are integers. BIRD scheme works as follows: First, a structural summary called `Ind(DB)` is constructed from the tree database (`DB`). `Ind(DB)` is built using the DataGuide technique (Goldman et al., 1997) together with an index mapping $I: N' \longrightarrow M$, where `N` is the set of nodes of `DB` and `M` is the set of nodes (also called index) of `Ind(DB)`. Surjective mapping `I` preserves the root and child relationship in the obvious sense. For $m \in M$, the set $I^{-1}(m)$ is called the set of database nodes with index node `m`. Each node `m` of `Ind(DB)` is associated with a weight, $w(m)=\max\{childCount(n)+1 \mid n \in I^{-1}(m)\}$; `childCount(n)` denotes the number of children of the database node `n`.

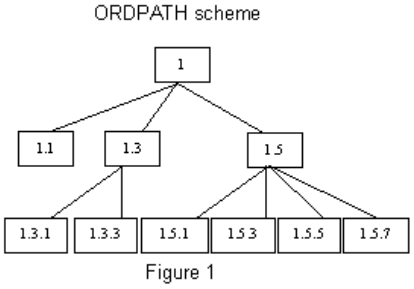

Figure 1

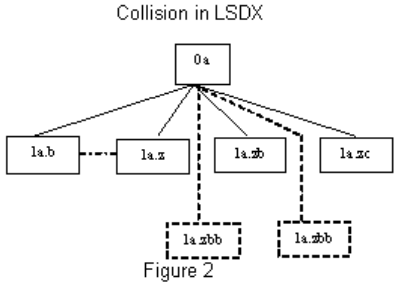

Figure 2

While enumerating the nodes of the database, authors reserve the weight $w(m)$ for all subtrees rooted at any of the nodes n in $I^{-1}(m)$. BIRD number $Id(n)$ of n is then defined in a way that all node identifiers in the subtree of node n are guaranteed to fall into the interval $[Id(n), Id(n)+w(I(n)))$. Since not all the subtrees rooted at $n \in I^{-1}(m)$ are of the same size, some numbers remain unused in the enumeration so that one could further find room to insert other nodes. Thus, BIRD allows for a limited number of node insertions, until an overflow occurs when a node has more children than its ID range allows for. If this happens, then a global reallocation of IDs becomes necessary.

3.4. Other Node Identification Schemes

In (Grust, 2002) the author proposes an index structure, the XPath accelerator scheme which is based on preorder and postorder ranks of each node v. The label of a node v is a triple $(pre(v), post(v), par(v))$ where:

- $pre(v)$ and $post(v)$ are respectively the preorder and the postorder ranks for node v
- $par(v)$ is the parent preorder rank for node v

All XPath axes (descendant, ancestor, following and preceding, parent, child, next sibling, etc.) can be determined relative to an arbitrary context node. The author states that it is necessary to update all labels in the set of following nodes and in the ancestor axe of a newly inserted node.

Path-based node identification schemes such as Dewey Order (Tatarinov et al., 2002) use the entire root path $<c_0, \ldots, c_k>$ of a node at level k as node ID. Each offset c_i denotes the position of the ancestor of level i. This path encoding implies that node IDs have no fixed size and may vary according to the depth of a node and its position among its siblings. Since the offsets are independent of each other, Dewey Order supports (limited) updates without altering all IDs assigned to other nodes. As shown in (Tatarinov et al., 2002), renumbering is restricted to the descendants and following siblings of the node being inserted. (Lee et al., 1996), (Li et al., 2001), (Abiteboul et al., 2001), (Cohen et al., 2002), (Alstrup et al., 2002), (Wang et al., 2003), (Bremer et al., 2003), (Chen et al., 2004), (Kha et al., 2004) propose labelling schemes which need re-labelling of existing labels when updates occur.

4. A New Persistent Labelling Scheme

In this section we present our own labelling scheme. It is a persistent structural labelling scheme which supports an infinite number of insertions without renumbering. Moreover, our scheme does not require a space of reserved IDs. Our solution is based on the fact that there exist an infinite number of real numbers within an interval $[a,b]$; a, b being real numbers.

Static Labelling

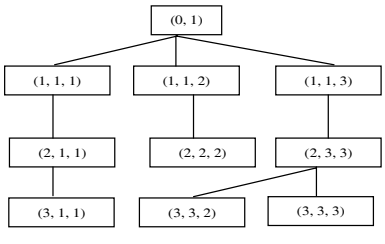

Figure 3

Dynamic Labelling

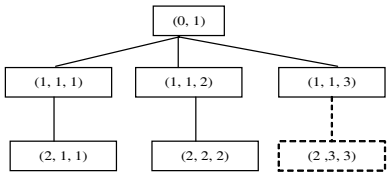

Figure 4b

4.1. Static Labelling Step

Our labelling scheme needs modest storage capacities. We label each node with a triple (p, n_p, n):

- p is the level of the node in the tree (p is an unsigned integer).
- n is the local label of the node (n is a signed real number).
- n_p is the local label of the parent node (n_p is a signed real number).

Given a level, local codes are unique.

We assign label $(0,1)$ to the root element. This label consists of two numbers only since the root element has no parent. 0 is the level. 1 is the local code.

Given a level p, if we assume that nodes are visited from left to right then the local label of a node is i where i is the position of the node at level p. Figure 3 shows an example of tree after the static labelling step.

4.2. Dynamic Labelling Step

In this section, we show how newly inserted nodes are dynamically labelled without changing the label of existing nodes.

Rules for creating labels for new nodes

Let v be a node to be inserted at level p.

- If v is the first node to be inserted at level p then its local code is 1
- If v is inserted immediately before the node of local code i and if there is no other node before i then the local code of v is $i-1$. See an example of such insertion in figure 4a.
- If v is inserted immediately after the node of local code i and if there is no other node after i then the local code of v is $i+1$. See an example of such insertion in figure 4b.
- If v is inserted immediately before the node of local code i and immediately after the node of local code j then the local code of v is k with k=(i+j)/2, see an example of such insertion in figure 4c.

As we can see our scheme allows for an infinite number of insertions without re-labelling. This is of course if we assume that we can have real numbers with infinite decimal expansion. In practice, real numbers are always represented as floating point numbers, with obvious limitations on the precision that can be attained. Considering such limitations, the worst case of insertion is the following: given an existing node n, insert consecutively several nodes *immediately* after node n. Consider for example node of label (2,1,1) in figure 3 and assume real numbers are represented with double precision (64 bits) floating-point numbers. Table 1 shows the labels of the new nodes which are inserted one after the other. The third column indicates the binary value of the local code. The length of this value is 52 bits since the mantissa of double precision floating-point numbers is stored in 52 bits. At the 52nd insertion the limit of precision is reached i.e. the label of the newly inserted node is the same as the label of the previously inserted node.

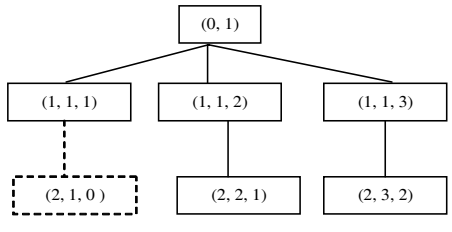

Figure 4a

Dynamic Labelling

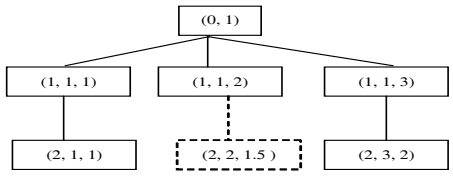

Figure 4c

Repeated insertion (reals)

insertion	label	mantissa
1	(2,1,1.5)	1 .100000000000000…0000000000
2	(2,1,1.25)	1 .010000000000000…0000000000
3	(2,1,1.125)	1 .001000000000000…0000000000
...		

Table1

In a previous version of this paper (Gabillon et al.,2005), we used rational numbers instead of real numbers and we represented each code as a pair of signed integers (e.g. rational 3/2 as the pair (3,2)). With such representation, label (2,1,1.5) becomes (2,(1/1),(3/2)). Recall that with double precision floating-point numbers the length of a code is 64 bits. Therefore, with the rational number representation, we should use integers stored in 32 bits in order to have codes of 64 bits. Consider the aforementioned case of consecutive insertions. Table 2 shows the labels of the new nodes which are inserted one after the other. Since the highest number that a signed integer stored in 32 bits can represent is 2^{n-1}-1, an overflow occurs at the 31st insertion.

Repeated insertion (rationals)

insertion	label	Computation of the local code
1	(2,1/1,3/2)	$1 + (1/2)^1$
2	(2,1/1,5/4)	$1 + (1/2)^2$
3	(2,1/1,9/8)	$1 + (1/2)^3$
...		

Table2

4.3. Reconstruction and Decision Procedures

According to (Weigel et al., 2005), reconstruction and decision problems can be defined as follows:

- *Reconstruction problem*: how parts of the tree structure of the database can be reconstructed without accessing the database, given the label of a node.
- *Decision problem*: how to determine relations between two nodes without accessing the database, given their labels.

Reconstruction procedure

Following the approach used in (Weigel et al., 2005), we build the ancestor structural summary s (e.g. a DataGuide (Goldman et al., 1997)) of the source tree t which consists of a labelled tree constructed as follows:

- nodes having the same parent in the source tree t are represented by a unique node labelled with the local code of that parent in the summary tree s
- the root of s represents the nodes of t having the root of `t` as parent
- each node `v` of tree `s` represents the nodes of `t` whose parent is represented by the parent of node `v`

For example, tree in figure 5 summarises source tree in figure 3. The ancestor structural summary tree `s` is held in memory and is used by the reconstruction procedure. Let us consider the node `v` of code (p,n_p,n). Without access to the source tree `t`, we can reconstruct the code (p',n'_p,n') of its `i`-ancestor[1] as follows ($1 \le i \le l$):

If `i=p` then

- the code of the `p`-ancestor of `v` is `(0,1)`

else

- `p'=p-i`
- let `u` be the unique node of `s` which has label n_p and which is at level `p-1`. Let `w` be the `(i-1)`-ancestor of `u`. Node `w` has label `n'`.
- label of the parent of node `w` is n'_p

[1] Let n be a node. Let i>0 be an integer. We define the *i-ancestor* of n as the node which can be reached from n with exactly i parent steps. The parent of n is the 1-ancestor of n.

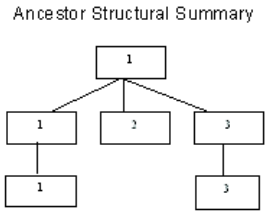

Figure 5

Decision procedure

Let `r` be any of the following `XPath` axes: `parent`, `ancestor`, `followingsibling`. Given two nodes `v` and `v'` of the database, we write `r(v,v')` iff the relation `r` holds between `v` and `v'`. For example `parent(v,v')` means node `v` is the parent of node `v'`. We can decide about `parent` and `followingsibling` relations simply by looking at the labels of nodes `v` and `v'`. In order to decide about the ancestor relation we also need to access to the ancestor structural summary `s` of the source tree `t`. Let (p,n_p,n) be the code of `v` and let (p',n'_p,n') be the code of `v'`:

- `parent(v,v')` iff `p=p'+1` and $n'_p=n$
- `followingsibling(v,v')` iff $p=p'$, $n'_p=n_p$ and `n>n'`.
- `ancestor(v,v')` iff `p=p'` and `v` is the `(p'-p)`-ancestor of `v'`.

5. Experiments

In this section, we evaluate our scheme and compare it to other proposals. We consider the following parameters:

- Robustness: ability to insert new nodes without altering existing labels
- Runtime performance when inserting a large number of nodes
- Storage consumption

5.1. Robustness

Among all identification schemes discussed in this paper, only ORDPATH and our scheme support unlimited updates. Still ORDPATH looses its expressivity in its updatable version. BIRD allows only for a limited number of node insertions until an overflow occurs when a node has more children than its ID range allows for. If this happens, a global reallocation of IDs is necessary. LSDX aims to support unlimited updates. Unfortunately, as we have shown earlier in this work, the scheme is not applicable in its dynamic version due to collisions between labels.

5.2. Runtime performance

We implemented our labelling scheme in java 1.5 and used `DOM` as `XML` parser. We used the data generator of `XMark` (Schmidt et al., 2002) to generate `XML` documents of various sizes. `XMark` is a framework to analyze the capabilities and performance of the `XML` repositories.

Our experiments were performed on the Pentium IV 2.4 GHz with 1GB of RAM under windows XP.

We evaluated our scheme based on the time used to generate labels. In order to compare our scheme with other proposals, we implemented the LSDX scheme in the aforementioned environment. Figure 6 depicts the time (in seconds) needed to generate labels for `XML` documents of various size. The result reported represents the average time needed for 20 executions.

As shown in figure 6, our scheme is close to LSDX scheme in terms of runtime performance. Comparison between LSDX and other schemes can be found in (Duong et al., 2005).

5.3. Storage Consumption

As mentioned in (Cohen et al., 2002), the length of the assigned labels is an important criterion in the quality of a labeling scheme. Indeed, this length determines the size of the index structure that contains the labels and thereby the feasibility of keeping this index in memory. The storage consumption of LSDX and our scheme on `XMark` is depicted in figure 7. The result shows that our scheme

needs less space capacity than does LSDX. In fact the length of labels when using LSDX grows very quickly with the size of the document. This behavior characterizes schemes that extend the label of a node with a component in order to get the ID of child nodes. In our scheme the length of a label is constant.

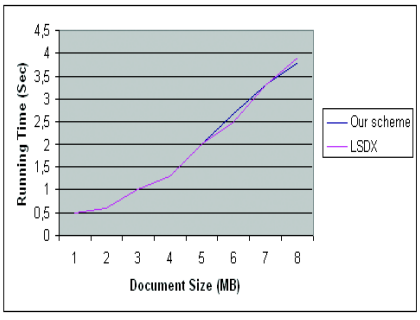

Figure 6. Runtime performance

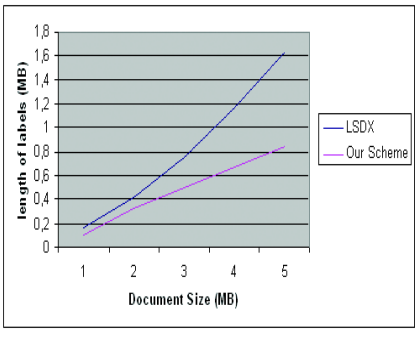

Figure 7. Storage consumption

6. Conclusion

In this paper, we point out the limitations of existing labelling schemes for XML data assuming documents are frequently updated. We present a persistent structural labelling scheme where labels encode ancestor-descendant relationships and sibling relationship between nodes but need not to be modified when the document is updated. Our labelling scheme requires modest storage capabilities. It is based on the fact that there exists an infinite number of reals between a given interval `[a,b]` of reals. It (theoretically) supports an infinite number of updates. We also conduct some experiments showing that performances of our scheme are satisfactory.

As an application of this work, we are currently integrating this scheme into a prototype of a secure XML database (Gabillon, 2005) in which updates are expected to occur frequently. We are also planning to extend our proposal in order to deal with the reconstruction and decision properties of more XPath axes.

Acknowledgment

Authors would like to thank Ioana Manolescu for her fruitful comments on an early version of this paper.

References

1. Abiteboul, S., Kaplan, H., Milo, T (2001). Compact labelling schemes for ancestor queries. Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA).

2. Alstrup, S., Rauhe, T (2002). Improved labelling scheme for ancestor queries. Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA).

3. Bremer, J.M., Gertz, M (2003). An efficient XML Node Identification and Indexing Scheme. Technical Report CSE-2003-04, Department of Computer Science, University of California at Davis.

4. Bray, T. et al. (2000) "eXtensible Markup Language (XML) 1.0" World Wide Web Consortium (W3C). http://www.w3.org/TR/REC-xml

5. Chen, Y., Mihaila, G.A., Bordawekar, R., Padmanabhan, S (2004). L-Tree: a Dynamic Labeling Structure for Ordered XML Data. LNCS 3268 – Current Trends in Database Technology Springer-Verlag.

6. Clark, J., & DeRose S. (1999). "XML Path Language (XPath) version 1.0". World Wide Web Consortium (W3C). http://www.w3.org/TR/xpath

7. Cohen, E., Kaplan, H., Milo, T (2002). Labeling Dynamic XML trees. Proceedings of the Symposium on Principles of Database Systems (PODS).

8. Duong, M., Zhang, Y (2005). LSDX: A new Labelling Scheme for Dynamically Updating XML Data. Proceedings of the 16th Australasian Database Conference.

9. Gabillon, A. (2005). A Formal Access Control Model for XML Databases. Proc. of the second VLDB Workshop on Secure Data Management. Trondheim, Norway.

10. Gabillon, A. Fansi, M. (2005). A Persistent Labelling Scheme for XML and Tree Databases. Proc. of the IEEE conference on Signal and Image Technology and Internet Based Systems. November 2005, Yaoundé Cameroon.

11. Goldman, R., Widom, J. (1997). DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB).

12. Grust, T. (2002). Accelerating XPath location steps. Proceedings of ACM SIGMOD International Conference on Management of Data.

13. Kha, D.D., Yoshikawa, M., Uemura, S. (2004). A Structural Numbering Scheme for Processing Queries by Structure and Keyword on XML Data. IEICE Transactions on Information Systems.

14. Lee, Y.K., Yoo, S.J., Yoon, K., Berra, P.B. (1996). Index structures for structured documents. Proceedings of the 1st ACM International Conference on Digital Libraries.

15. Li, Q., Moon, B. (2001). Indexing and querying XML data for regular path expressions. Proceedings of the 27th International Conference on Very Large Data Bases (VLDB).

16. Marian, A., Abiteboul, S., Cobena, G., Mignet, L.(2001). Change-centric management of versions in an XML warehouse. Proceedings of the 27th International Confefrence on Very Large Data Bases (VLDB).

17. O'Neil, P., O'Neil, E., Pal, S., ICseri, I., Schaller, G., Westbury, N. ORDPATHs: Insert-Friendly XML Node Labels. Proceedings of the 2004 ACM SIGMOD International Conference on Management of data.

18. Schmidt, A., Wass, F., Kersten, M., Carey, J., Manolescu, I. And Busse, R. (2002). Xmark: A Benchmark for XML Data Management. Proceedings of the 28th International Conference on Very Large Data Bases (VLDB).

19. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C. (2002). Storing and Querying Ordered XML Using a Relational Database System. Proceedings of the ACM SIGMOD International Conference on Management of Data.

20. Wang, H., Park, S., Fan, W., Yu, P.S. (2003). ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. Proceedings of the 22nd SIGMOD International Conference.

21. Weigel, F., Schulz, K.U., Meuss, H. (2005). The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations using Efficient Arithmetic Operations. Proceedings of the 3rd International XML Database Symposium, XSym 05.