A Persistent Labelling Scheme for XML and tree Databases[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. Supported update operations are insertion of new sub-trees, deletion of existing sub-trees and modification of existing nodes.

1. Introduction

XML is becoming the new standard for the exchange and publishing of data over the Internet [18]. Documents obeying the XML standard can be viewed as trees (see figure 2). Query language like XPath [9] 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 [1][4][5][2][7][8][11][15][17][3][6][12][16]. 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

[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.

then point out the limitations of these propositions assuming documents are frequently updated. Section 4 presents our proposed labelling scheme. Finally, section 5 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 [1]. We distinguish two families of 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 ∈ t. Predicates ancestor and sibling and the labelling function l are such that for every tree t and every two nodes v,u ∈ 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:

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.

A *range labelling scheme* comes equipped with some order relation ≤ 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 ≤ being the standard order relation over integers.

In a *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.

A *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 [8], 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 [17][15][11][8][5][7]. In this section we report the characteristics of some major existing identification schemes.

3.1 The Work of O'Neil et al. [11]

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 [10]. 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.

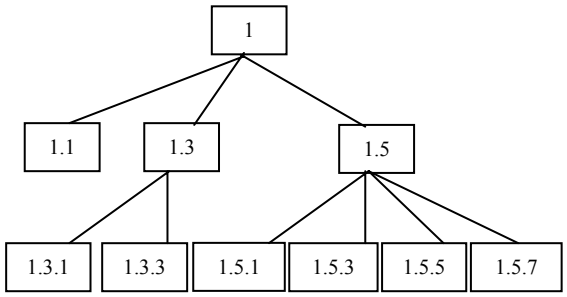

Figure 1: ORDPATH scheme

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 [4], their labels cannot be used for structural queries.

3.2 The Work of Duong et al. [17]

In [17] 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.

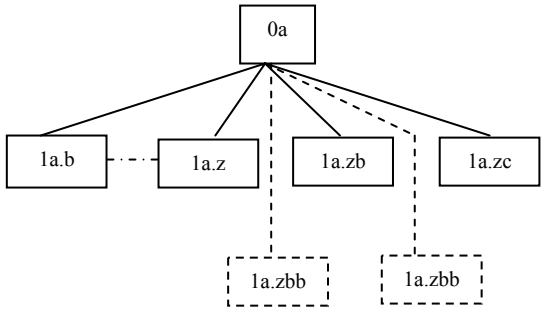

Figure 2: Example of collision

3.3 The Work of Weigel et al. [8]

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 DataGuides technique [19] together with an index mapping I: N→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.

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 [14] 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 [10] 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 [10], renumbering is restricted to the descendants and following siblings of the node being inserted.

[1], [5], [7], [2], [3], [6], [12], [13], [15],[16] propose labelling schemes which need re-labelling of existing labels when updates occur.

4. Our Proposal

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 rationals within an interval [a,b]; a, b being rationals.

4.1. Static Labelling Step

We label nodes by using rationals. We represent a rational by a pair which consists of a signed integer and a strictly positive integer, e.g. we represent rational 5/2 by the pair (5,2) and the rational -5/2 by the pair (-5,2).

Our labelling scheme needs modest storage capacities. We label each node with a quintuple $(l,(n_p, d_p),(n, d))$:

- l is the level of the node in the tree.
- (n,d) is the local label of the node. Pair (n,d) represents the n/d rational.
- (n_p,d_p) is the local label of the parent node.

Given a level, local codes are unique.

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

Given a level l, if we assume that nodes are visited from left to right then the local label of a node is (i,1) where i is the position of the node at level l. Figure 3 shows an example of tree after the static labelling step.

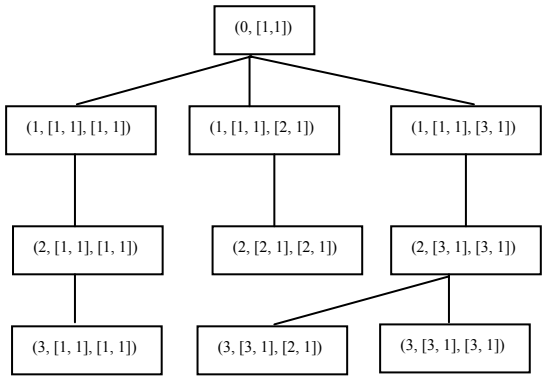

Figure 3: Static Labelling

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 l.

- If v is the first node to be inserted at level l then its local code is (1,1)
- If v is inserted immediately before the node of local code (i,j) and if there is no other node before (i,j) then the local code of v is (i-j,j). See an example of such insertion in figure 4a.
- If v is inserted immediately after the node of local code (i,j) and if there is no other node after (i,j) then the local code of v is (i+j,j). See an example of such insertion in figure 4b.
- If v is inserted immediately before the node of local code (i,j) and immediately after the node of local code (k,h) then the local code of v is (a,b) with a=(i.h+k.j)\d and b=2.h.j\d. \ denotes the integer division. d is the highest common factor of (i.h+k.j) and 2.h.j. See an example of such insertion in figure 4c.

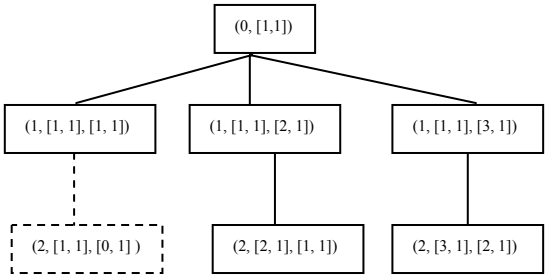

Figure 4a: Dynamic Labelling

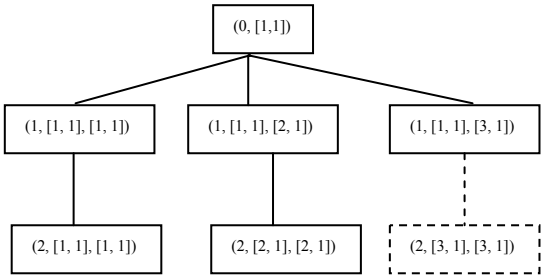

Figure 4b: Dynamic Labelling

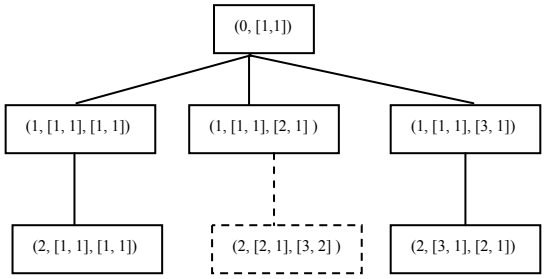

Figure 4c: Dynamic Labelling

4.3. Reconstruction and Decision Procedures

According to [8], 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 [8], we build the ancestor structural summary s (e.g. a DataGuide [19]) 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 $(l,(n_p,d_p),(n,d))$. Without access to the source tree `t`, we can reconstruct the code $(l',(n'_p,d'_p),(n',d'))$ of its `i`-ancestor[3] as follows ($1 \le i \le l$):

If $i = l$ then
- the code of the `l`-ancestor of `v` is $(0,(1,1))$

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

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 $(l,[n_p, d_p],[n, d])$ be the code of `v` and let $(l',[n'_p, d'_p],[n', d'])$ be the code of `v'`:

- `parent(v,v')` iff $l=l'+1$ and $[n'_p,d'_p]=[n,d]$
- `followingsibling(v,v')` iff $l=l'$, $[n'_p,d'_p]=[n_p,d_p]$ and $n/d > n'/d'$.
- `ancestor(v,v')` iff $l<l'$ and `v` is the $(l'-l)$-ancestor of `v'`.

Evaluating the performances of the reconstruction and the decision procedures and comparing them with similar procedures in other labelling schemes remains work to be done.

[3] 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`.

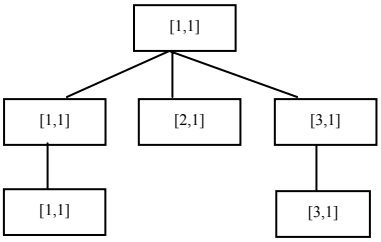

Figure 5: Ancestor Structural Summary

5. 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 rationals between a given interval [a, b] of rationals. It supports an infinite number of updates.

References

[1] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling Dynamic XML trees. In PODS, pages 271-281, 2002.

[2] S. Abiteboul, H. Kaplan, and T. Milo. Compact labelling schemes for ancestor queries. In Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA), January 2001.

[3] S. Alstrup and T. Rauhe. Improved labelling scheme for ancestor queries. In Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA), January 2002.

[4] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In Proc. Int. Conf. on Very Large Data Bases (VLDB), September 2001.

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

[6] Q. Li an B. Moon. Indexing and querying XML data for regular path expressions. In Proc. Of the 27th VLDB Conf., Pages 361-370, 2001.

[7] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In SIGMOD 2003.

[8] F. Weigel, K.U. Schulz and H. Meuss. The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations using Efficient Arithmetic Operations. Third International XML Database Symposium, XSym 05, Trondheim,Norway. September 2005.

[9] J. Clark and S. DeRose. "XML Path Language (XPath) version 1.0". World Wide Web Consortium (W3C). http://www.w3.org/TR/xpath, November 16, 1999.

[10] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In proceedings of the ACM SIGMOD International Conference on Management of Data, pages 204-215, 2002.

[11] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In Proceedings of the 2004 ACM SIGMOD International Conference on Management of data, pages 903 – 908, 2004.

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

[13] Y.K. Lee, S. J. Yoo, K. Yoon, and P.B. Berra. Index structures for structured documents. In Proceedings of the 1st ACM International Conference on Digital Libraries, pages 91-99, 1996.

[14] T. Grust. Accelerating XPath location steps. In proceedings of ACM SIGMOD International Conference on Management of Data, pages 109 – 120, 2002.

[15] D. D. Kha, M. Yoshikawa and S. Uemura: A Structural Numbering Scheme for Processing Queries by Structure and Keyword on XML Data', IEICE Transactions on Information Systems, 2004.

[16] Y. K. Lee, S-J. Yoo, K. Yoon, P. B. Berra. Index Structures for structured documents. ACM first Inter. Conf. on Digital Libraries, Maryland, 91 – 99, 1999.

[17] M. Duong and Y. Zhang. LSDX: A new Labelling Scheme for Dynamically Updating XML Data. In Proc. Of the 16th Australasian Database Conference, Newcastle, Australia, 2005.

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

[19] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Proc. 23rd VLDB Conf, pages 436-445, 1997.