# ORDPATHs: Insert-Friendly XML Node Labels

Patrick O'Neil, Elizabeth O'Neil[1]
University of Massachusetts Boston
{poneil, eoneil}@cs.umb.edu

Shankar Pal, Istvan Cseri, Gideon Schaller, Nigel Westbury
Microsoft Corporation
{shankarp, istvanc,gideons}@microsoft.com

[1] Work of these authors was performed at Microsoft, while on sabbatical from the University of Massachusetts at Boston.

## ABSTRACT
We introduce a hierarchical labeling scheme called ORDPATH that is implemented in the upcoming version of Microsoft® SQL Server™. ORDPATH labels nodes of an XML tree without requiring a schema (the most general case—a schema simplifies the problem). An example of an ORDPATH value display format is "1.5.3.9.1". A compressed binary representation of ORDPATH provides document order by simple byte-by-byte comparison and ancestry relationship equally simply. In addition, the ORDPATH scheme supports insertion of new nodes at arbitrary positions in the XML tree, their ORDPATH values "careted in" between ORDPATHs of sibling nodes, without relabeling any old nodes.

## 1. INTRODUCTION
Relational database systems are now commonly used for XML data storage and update [9], [8], [11]. Typically, an XML document is "shredded" into rows of relational tables; see our *NODE* table below, each row of which stores information for an individual node in the XML tree.

Several node labeling schemes have been proposed in the literature: [9] compares three schemes. Aside from its efficient insertion and compression, ORDPATH is similar conceptually to the Dewey Order described in [9]. While previous schemes are adequate for the structure of static XML data, insertion within a tree remains a challenging issue. ORDPATH provides efficient insertion at any position of an XML tree, and also supports extremely high-performance query plans for native XML queries.

Structural modifications to the XML tree can occur in several ways: new sub-trees may be inserted, sub-trees may be deleted, and sub-trees may be moved around within the tree. Even single node insertions can be very costly for existing labeling schemes, requiring a large number of nodes to be relabeled for each new node inserted. 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** might be the parent ORDPATH, **1.5.3.9.1** the child. The various child components reflect the children's relative sibling order, so that byte-by-byte comparison of the ORDPATH labels of two nodes yields the proper document order.

A new node (possibly a root node of a sub-tree) can be inserted under any designated parent node in an existing tree. Its label is generated using an additional intermediate "careting" component that falls between the components of its left and right siblings. This is referred to as "careting in". Leftmost and rightmost insertion in a group of siblings is even more efficiently supported by range extensibility of component numbering on both ends. In all cases, a need for relabeling is avoided.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2004, June 13–18, 2004, Paris, France.
Copyright 2004 ACM 1-58113-859-8/04/06 …$5.00

## 2. MOTIVATING EXAMPLE
We define a relational table (see Figure 2.3), known as the *NODE* table; a *shredding* transformation [2], [8], [9] will

```
<BOOK ISBN="1-55860-438-3">
    <SECTION>
        <TITLE> Bad Bugs</TITLE>
            Nobody loves bad bugs.
        <FIGURE CAPTION="Sample bug"/>
    </SECTION>
    <SECTION>
        <TITLE> Tree Frogs </TITLE>
            All right-thinking people
            <BOLD> love </BOLD> tree frogs.
    </SECTION>
</BOOK>
```

**Figure 2.1 Sample XML data**

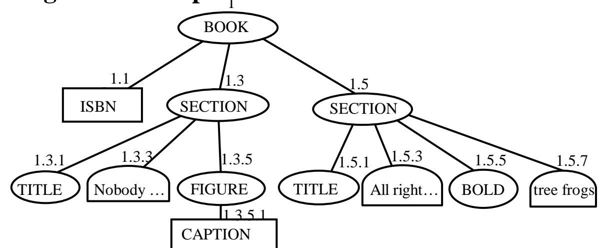


**Figure 2.2 XML tree for XML data of Figure 2.1**

| ORDPATH | TAG | NODE TYPE | VALUE |
|---|---|---|---|
| 1. | 1 (BOOK) | 1 (Element) | null |
| 1.1 | 2 (ISBN ) | 2 (Attribute) | '1-55860-438-3' |
| 1.3 | 3 (SECTION) | 1 (Element) | null |
| 1.3.1 | 4 (TITLE) | 1 (Element) | 'Bad Bugs' |
| 1.3.3 | -- | 4 (Value) | 'Nobody loves bad bugs.' |
| 1.3.5 | 5 (FIGURE) | 1 (Element) | null |
| 1.3.5.1 | 6 (CAPTION) | 2 (Attribute) | 'Sample bug' |
| 1.5 | 3 (SECTION) | 1 (Element) | null |
| 1.5.1 | 4 (TITLE) | 1 (Element) | 'Tree frogs' |
| 1.5.3 | -- | 4 (Value) | 'All right-thinking people' |
| 1.5.5 | 7 (BOLD) | 1 (Element) | 'love ' |
| 1.5.7 | -- | 4 (Value) | 'tree frogs' |

**Figure 2.3**. **XML "Shredded" into relational Node Table**

transform all XML node data into rows of this table. In the most general case, the XML data has no schema. XML data and a tree representing the XML hierarchy are shown in Figures 2.1 and 2.2, respectively, with the corresponding NODE table shredded from Figure 2.1 shown in Figure 2.3.

Successive nodes of the XML tree in XML document order are traversed during the initial load of the NODE table, and the ORDPATH labels are generated at that time. In ORDPATH values of Figure 2.3 (such as "1.3.5.1"), each dot separated component value ("1", "3", "5", "1") reflects a numbered tree edge at successive levels down the path from the root (itself having a 0-length ORDPATH) to the node represented. Note that only *positive*, *odd* integers are assigned during an initial load; even-numbered and negative integer component values are reserved for later insertions into an existing tree, as explained below in Section 3.3. ORDPATH values stored are not the dotted-decimal strings displayed ("1.3.5.1"), but rather a compressed binary representation defined in Section 3.3.

The NODE TYPE column of Figure 2.3 contains coded values for various node types: 1 for an element, 2 for an attribute, and so on. The TAG column contains coded tags. The VALUE column contains variable-type data that is associated with some nodes.

**Primary Index**. An ORDPATH primary key (with a clustered index) on the NODE table provides efficient query access to XML data. For example, a query that retrieves all the descendents of X will find them clustered on disk just after X, in ORDPATH order (i.e., document order), so retrieval is optimal.

## 3. FUNDAMENTAL ORDPATH CONCEPTS

Figure 3.1 illustrates successive variable-length $L_i/O_i$ bitstrings of the compressed ORDPATH format.

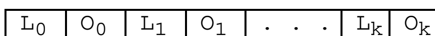



**Figure 3.1 Compressed ORDPATH Format**

In all $L_i/O_i$ component pairs of Figure 3.1, each $L_i$ bitstring specifies the length in bits of the succeeding $O_i$ bitstring. $L_i$ bitstrings are represented using a form of prefix-free encoding, defined in Section 3.1, to provide a number of important properties, as follows. (1) given that we know where an $L_i$ bitstring starts (as we do with $L_0$), we can identify where it stops; (2) each $L_i$ bitstring specifies the length in bits of the succeeding $O_i$ bitstring; (3) from (1) and (2), we see how to parse all ORDPATH bitstrings, left to right, into their $L_i/O_i$ components; (4) the $L_i$ bitstrings are generated to maintain document order; (5) $L_i/O_i$ components can specify *negative* ordinals $O_i$ as well as positive ones; negative ordinals support multiple inserts of nodes to the left of a set of existing siblings.

## 3.1 Detailed $L_i/O_i$ Pair Design

Of the many possible prefix encoding schemes for $L_i$ bitstrings, we examine two described in Figures 3.2a and 3.2b. In Figure 3.2a, the $L_i$ bitstring 01 identifies a component $L_i/O_i$ encoding with assigned length $L_i$ = 3, indicating a 3-bit $O_i$ bitstring. The following $O_i$ bitstrings (000, 001, 010, . . ., 111) represent $O_i$ values of the first eight integers, (0, 1, 2, . . ., 7). Thus 01101 is the bitstring for ORDPATH "5". In the next row in Figure 3.2a, bitstring 100 identifies an encoding with $L_i$ = 4 and the 4-bit $O_i$ bitstrings that follow represent the range [8, 23]; in particular, $O_i$ = 8 is represented by bitstring 0000, 9 by bitstring 0001,…, up to 23 by bitstring 1111. Similarly, $O_i$ in the range [-8, -1] is associated with the $L_i$ bitstring 001, with -8 represented by the lowest bitstring, 000.

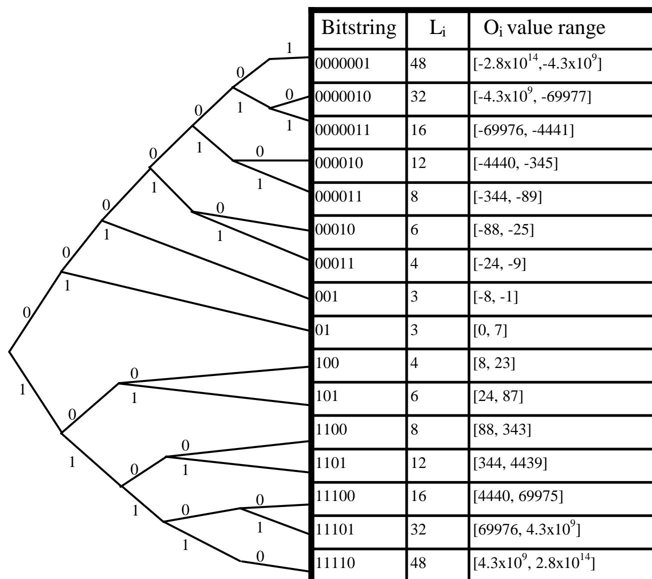
**Example 3.1.** Using $L_i$ values of Figure 3.2a, we would generate ORDPATH = "1.5.3.-9.11" as follows:

| 01 | 001 | 01 | 101 | 01 | 011 | 00011 | 1111 | 100 | 0011 |
|---|---|---|---|---|---|---|---|---|---|
| $L_0$=3 | $O_0$=1 | $L_1$=3 | $O_1$=5 | $L_2$=3 | $O_2$=3 | $L_3$=4 | $O_3$=-9 | $L_4$=4 | $O_4$=11 |

Of course there are no actual spaces in the ORDPATH bit pattern; spaces have been added for ease of reading. Using the $L_i$ values of Figure 3.2b (note that the $L_i$ bitstring 01 requires $O_i$ of length 0 to represent 1), we would have the following for "1.5.3.-9.11":

| 01 | | 110 | 01 | 10 | 1 | 00001 | 1100 | 1110 | 0011 |
|---|---|---|---|---|---|---|---|---|---|
| $L_0$=0 | ($O_0$=1) | $L_1$=2 | $O_1$=5 | $L_2$=1 | $O_2$=3 | $L_3$=4 | $O_3$=-9 | $L_4$=3 | $O_4$=11 |

After an initial load, we can label a newly inserted node to the right of all existing children of a node by adding two to the last ordinal of the last child, coding $L_i$ and $O_i$ as needed. We can insert new children of the node on the left of all existing children by adding -2 to the last ordinal of the first child, using negative ordinal values when needed.



| Bitstring | $L_i$ | $O_i$ value range |
|---|---|---|
| 0000001 | 48 | $[-2.8\times10^{14}, -4.3\times10^{9}]$ |
| 0000010 | 32 | $[-4.3\times10^{9}, -69977]$ |
| 0000011 | 16 | [-69976, -4441] |
| 000010 | 12 | [-4440, -345] |
| 000011 | 8 | [-344, -89] |
| 00010 | 6 | [-88, -25] |
| 00011 | 4 | [-24, -9] |
| 001 | 3 | [-8, -1] |
| 01 | 3 | [0, 7] |
| 100 | 4 | [8, 23] |
| 101 | 6 | [24, 87] |
| 1100 | 8 | [88, 343] |
| 1101 | 12 | [344, 4439] |
| 11100 | 16 | [4440, 69975] |
| 11101 | 32 | $[69976, 4.3\times10^{9}]$ |
| 11110 | 48 | $[4.3\times10^{9}, 2.8\times10^{14}]$ |

**Figure 3.2a**

| Bitstring | $L_i$ | $O_i$ value range |
|---|---|---|
| 000000001 | 20 | [-1118485, -69910] |
| 00000001 | 16 | [-69909, -4374] |
| 0000001 | 12 | [-4373, -278] |
| 000001 | 8 | [-277, -22] |
| 00001 | 4 | [-21, -6] |
| 0001 | 2 | [-5, -2] |
| 001 | 1 | [-1, 0] |
| 01 | 0 | [1, 1] |
| 10 | 1 | [2, 3] |
| 110 | 2 | [4, 7] |
| 1110 | 4 | [8, 23] |
| 11110 | 8 | [24, 279] |
| 111110 | 12 | [280, 4375] |
| 1111110 | 16 | [4376, 69911] |
| 11111110 | 20 | [69912, 1118487] |

**Figure 3.2b**

**Figure 3.2. Two Tables of Lengths $L_i$ with $O_i$ Ranges Represented**

We will describe in Section 3.3 how to "caret in" a node between any two existing children, using even ordinals.

In Figure 3.2a, all $L_i$ values are shown sitting at the leaves of a binary tree. The 0-1 encoding of each $L_i$ is determined by the path through the tree from the root to a leaf: a 0 bit is used for each tree edge going up, and 1 bit for each edge going down. A similar binary tree could be constructed for the more regular set of values of Figure 3.2b.

Deriving each $L_i$ bitstring from 0-1 paths through a binary tree clearly provides a *prefix free encoding*, i.e., no $L_i$ bitstring can be a prefix of another $L_i$ bitstring. We always know when a particular $L_i$ bitstring ends by following the 0-1 path through the tree until a leaf is reached. At that point the length of the subsequent $O_i$ bitstring value is known from the identified $L_i$ bitstring, so the entire sequence of bits that make up the $L_i/O_i$ component pairs of an ORDPATH of Figure 3.1 can be parsed.

## 3.2 Comparing ORDPATH Values

The prefix tree representing any $L_i$ can be concatenated, leaf to root, with simple n-way trees for each set of $O_i$ bitstrings to make a larger prefix tree representing all bitstring values for one-component ORDPATHs. Furthermore, two-component ORDPATHs correspond to a larger prefix tree of leaf-to-root connected one-component prefix trees, and so on. This construction orders all possible ORDPATHs as binary strings, and preserves document order. Thus simple bitstring (or byte by byte) comparison yields document order.

We can also determine ancestor-descendent relationships between any two ORDPATHs X and Y. If X is a strict substring of Y or vice versa, there is an ancestry relationship. The part that is longer can be parsed (using the prefix-free property) to find how far apart they are (parent, grandparent, etc.)

## 3.3 Arbitrary ORDPATH Insertions

We have shown how to insert new nodes to the right of all existing children of any node, or on the left of all children. For the remaining case, we can insert a new node Y between any two siblings of a parent node X (known as ***careting in***) 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.

Indeed we can caret in a sequence of K siblings with parent node having ORDPATH 3.5, the sequence to fall between sibling nodes 3.5.5 and 3.5.7, by providing the new siblings with the even caret 6: 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 doesn't count as a component that increases the depth of the node in the tree. However the caret does have an effect on ORDPATH order, since comparisons of Section 3.2 give 3.5.5 < 3.5.6.1, 3.5.6.3, 3.5.6.5 < 3.5.7. Using this approach we can caret in entire sub-trees falling between the sibling nodes 3.5.5 and 3.5.7, using only the one even ordinal between 5 and 7. For example: 3.5.6.1, 3.5.6.1.1, 3.5.6.3, 3.5.6.3.1, 3.5.6.3.3, 3.5.6.3.3.1, 3.5.6.3.3.3, 3.5.6.3.5, 3.5.6.5, 3.5.6.5.1, etc. The siblings of 3.5.5 and 3.5.7 are

underlined in this example, and the other nodes in the document order sequence are descendents of the siblings.

In interpreting an ORDPATH, the even components (carets) simply don't count for ancestry: 3.5.6.2.1 is a child of 3.5, and a grandchild of 3. Note that the construction ensures that all node labels end in an odd component. That means that the very last bit of a node label is always on, signaling the end of the bitstring in the last byte of a (zero-padded) byte string. This property allows the bit length to be determined from the byte length and the final byte of the ORDPATH.

New insertions can always be careted in between any two existing sibling nodes. For example, we can insert a node between 3.5.6.1 and 3.5.6.2.1 by using 3.5.6.2.-1.

Multiple levels of carets are normally extremely rare in practice. In order for K carets to exist in an ORDPATH, there must have been a decision at some point to perform an insert of a multiple node sub-tree (in text XML, this might be adding an intermediate section of a book), then there must have been a decision to add another multiple node sub-tree within the first (adding a new intermediate paragraph within the new section), and another within that (adding a new intermediate sentence within that paragraph), and so on, for K successive multiple node sub-tree additions, one within another, not at either end. Clearly this is a rarity.

The fact that insertions require no relabelings of old nodes is extremely important to insert performance and possible concurrency of operations. With Dewey Order [9], all right-side siblings and their descendents must be relabeled. These are updates to the primary key values, a particularly costly operation involving the primary index and all secondary indexes. With global numbering [9], all nodes of higher number must be renumbered, and end-descendent ids must be changed for the root node and many other nodes outside the set of renumbered nodes.

## 3.4 ORDPATH Primitives

We describe primitive functions to determine the parent and an upper bound on all descendents of a given node

(1) ORDPATH **PARENT**(ORDPATH X). The ORDPATH of the parent of X has the rightmost component of X removed (always an odd ordinal) and then all rightmost even ordinal components.

(2) ORDPATH **GRDESC**(ORDPATH X). To derive the smallest ORDPATH-like value greater than any descendent of a node with ORDPATH X, we increment the last ordinal component of X (an odd ordinal) to generate the next (even) ordinal and return this value as Y. The value Y is not a valid node-label because its last ordinal component is even; however it has the right format for comparison with descendents of X; the value Y is never used to label nodes without an additional odd component affixed.

## 3.5 ORDPATH Query Plans

### 3.5.1 Secondary Indexes

We now list the most important secondary indexes. The primary index by ORDPATH identifier has already been discussed at the end of Section 2.

- Element and Attribute TAG (its integer id) index, supporting fast look up of elements and attributes by name.
- Element and Attribute VALUE index.

See Section 3.5.2 for examples of use. Naturally there is a concern about the length of ORDPATHs in indexes. Note that the primary key index contains the rows themselves, so the length of the key is not the only contributor to relevant length. In the secondary indexes, the key length is more significant.

### 3.5.2 Query Plans

We now illustrate how efficient query plans for XML queries can be generated using the primary key ORDPATH on the relational NODE table. Consider the following XPath ancestor/descendent query:

[3.1] //Book//Publisher[. = "Random House"]

In general, descendent connections between node sets that are independently described may be treated as joins. In query [3.1] three techniques suggest themselves. If the number of descendent nodes below any Book element is small and likely to fall on the same disk page, one could retrieve the set of all Book elements (using the index on the TAG column of Section 3.5.1) and then explore all descendants; we can do this by treating each Book element as a node with ORDPATH X, by exploring the range of ORDPATHs from X to GRDESC(X) (see primitive (2) of Section 3.4).

Alternatively, if there are a great many descendents of each Book node, we might wish to separately locate the sequence of Book elements and the sequence of Publisher elements that have "Random House" as a value (using an index on VALUE of Section 3.5.1), then merge join the two sequences. Both node sequences will be in increasing order of ORDPATH values, since each of them is located by a single value in an index; furthermore, a merge join by ancestor//descendent can be treated in much the same way as an equal-match join (see [11]).

Finally, if the number of descendent elements is extremely small, we might wish to start at the descendent and look for a Book element ancestor. Successive applications of the elementary PARENT( ) primitive of Section 3.4 allows us

to locate all ancestors efficiently, although testing whether they are Book elements might be costly.

ORDPATH, together with a secondary index on LEVEL (of the node tree) provide efficient means to locate nodes on all XPATH axes of hierarchy and precedence, including axes such as sibling. Efficient ORDPATH query plans use relational primitives, extended by a few functions accessing the implicit information in the ORDPATH labeling values.

## 4. ORDPATH Length

We wish to demonstrate that large XML trees can normally be represented with ORDPATHs that are reasonable in length, an important consideration for a primary key. For any compressed-path representation, the worst cases of (simple) trees are the ones of small fan-out at each level. Let us construct a "random" binary XML tree by inserting n elements at random into an empty binary search tree (with no rebalancing taking place). In [1], Section 4.2, it is stated that the average depth P(n) of such a tree obeys the inequality.

[4.1] $P(n) \leq 1 + 1.4 \cdot \log_2 n$

Transposition gives the new inequality:

[4.2] $n \geq 2^{(P(n) - 1)/1.4}$

The average ORDPATH string length of an XML tree loaded from such a binary tree can be calculated from the average depth P(n) by determining the number of bits at each component level. Such a binary tree will have ordinal values at each level of 1 and 3. Using the encoding of Figure 3.2b, the $L_i$ will be 01 or 10 at each level and $O_i$ will have 0 or 1 bit, respectively, so there will be a total of at most $3 \cdot P(n)$ bits. If we allow a 20-byte ORDPATH, it will consist of 160 bits of components, which will hold P(n) = 53 3-bit components. Using inequality [4.2], this will support an average binary tree having $2^{(53-1)/1.4} = 1.4 \times 10^{11}$ = = 140G nodes.

XML trees will normally have larger average fan-out than binary, and this will normally imply many more nodes for a tree having the same number of components. For example, a fan-out of 4 can be represented by 5-bit components using either of the encoding schemes in Figure 3.2a or 3.2b ($O_i$ will have values 1, 3, 5, and 7 in this case), and a 20-byte ORDPATH will represent about $1.84 \times 10^{19}$ nodes for this 4-fan-out tree. If binary fan-out is very common with the XML trees of interest, one should use the encoding of the $L_i/O_i$ components given in Figure 3.2b (which has 3 bits per component in a binary situation, compared to 5 bits in Figure 3.2a); if a high fan-out (say 50) is very common, the encoding of Figure 3.2b requires 13 bits per component, whereas the encoding of Figure 3.2a only requires 9 bits per component. One can generate a large number of encoding schemes to choose from with our $L_i/O_i$ scheme, and it seems possible to base the scheme on statistics of trees for a given application.

## 5. Length Measurement Studies

A study [5] that gathered statistics on 190,417 XML trees worldwide found a maximum depth of 135. Examining the source of the deepest tree in this study, however (at URL http://edgarscan.pwcglobal.com/servlets/accession/0000950 123-01-505010.xml), showed the document was translated erroneously from html to give a long succession of <page> elements, each a *child* of the one before; there is a closing sequence of </page> tags at the end of the document.

In [5], 99% of the documents have less than 8 levels, i.e. less than depth 8. Almost all of the remaining 1% has depth between 8 and 30. Only a smattering of the documents has more than 30 levels. Further, there are rough power-law dependencies for the number of children per element that show how statistically rare the high fan-out cases are, at least in this collection. In such an environment, we should avoid jumping from 16 to 32 bits of $O_i$ representation in our $L_i$ definitions of Figure 3.2, as in Figure 3.2a, but rather proceed by smaller steps: bitlengths of 12, then 16, then 20, as done in Figure 3.2b. The big jump is good if the typical $O_i$ is expected to be anywhere in the $O_i$ range, but if it is highly probable to be at the lower end of the range, the smaller jump is better. This line of reasoning was prompted by length statistics of Timo Böhme and Erhard Rahm (private communication.)

We ran a prototype experiment to measure ORDPATH length (using the encoding in Figure 3.2a) in two well known cases. For the XMark benchmark [7], which models an auction scenario, we used scale 0.1 and generated an XML instance of length 10MB with 325755 XML nodes. The maximum and average ORDPATH lengths were 12 and 6 bytes, respectively. For the XMach-1 benchmark [6], which models a document management scenario, we created 1000 documents conforming to XMACH-1 DTDs. Document sizes ranged from 0.9KB to 120KB with average size 16KB. Maximum ORDPATH length was 15 bytes with the average of 4 bytes. Thus, the average and maximum ORDPATH lengths are quite small even for large amounts of data.

## 6. Insert-Friendly IDs for Global Numbering

If deep XML trees should appear in practice, we can save most of the label bitlength by using labels that do not reflect ancestry. The technique of “careting-in” used in ORDPATH can still be used for global numbering, that is, identifiers that maintain document order but do not contain path information.

To assign such labels, we pass through the XML tree in document order and generate single-component $L_0/O_0$ pairs with ordinal values 1, 3, 5, . . . for all nodes, regardless of

ancestry. Later insertions within the tree can then use the standard method of ORDPATH careting-in, creating an even-numbered ordinal component (say 4) for two-component identifiers in proper document order; successive nodes inserted in document order receive 4.1, 4.3, 4.5, . . .. As usual, repetitively inserting subtrees within subtrees will result in multiple even-$O_i$ components.

The resulting identifier is quite short under most circumstances (a single $L_0/O_0$ component), and it can be the primary key in the relational Node table of Figure 2.3 to ensure that rows are clustered in document order. We now consider how to represent ancestry information. One method adds an ORDPATH column to the Node table and creates a secondary index on it. For certain query classes this can lead to multiple accesses to rows in the Node table that are not needed when ORDPATH is the primary key, leading to relatively inefficient queries. In another method, as in the global numbering case treated in [9], every node is assigned a corresponding end-descendent (ED) id. The range of descendent ids lies between the reference node id and its ED. Then an insert of a node N requires changes to all the ED values for ancestors X where N lies on the rightmost descendent path.

## 7. Literature Review and Observations

In [9], three labeling schemes for XML trees are considered that encode document order: Global order, Local Order, which labels children of each node with integers starting from 1, and Dewey Order, which is comparable to ORDPATH (with Dewey paths like 1.2.1.1 for example). The Dewey order provides byte string comparisons via UTF-8 encoding [9] for each component of the path. Compression is poor for small ordinals, e.g. 1.2.1.1 uses four one-byte components, compared to four 3-bit components in Figure 3.2b or four 5-bit components in Figure 3.2a. On the other hand, 50.50.50.50 also uses four one-byte components in UTF-8, vs. four 9-bit components from Figure 3.2a. The major drawback of UTF-8, then, is its inflexibility. Further, our concept of "careting" inserts in the middle of a sequence of siblings in ORDPATHs is missing, so the need to relabel large numbers of nodes is a serious problem when updates occur.

In [4], each node x is marked by node type and labeled with two numberings, preorder (pre(x)) and postorder (post(x)), together with a parent identifier. From this information, all XPath axes (descendent, ancestor, following and preceding, parent, child, next sibling, prior sibling, etc.) can be determined relative to an arbitrary context node. The authors state that it is necessary to update all node labels in the following set of nodes and ancestor axes of a newly inserted node. But using ids with caret-in capability appears to provide easier inserts with short average length ids.

In [3], important theoretical results are derived for bounds on lengths of Ancestor Class (i.e. path-based) labelings, where no sibling order is maintained; this is equivalent to always allowing inserts of new nodes on the end of a sibling list. Indeed, our inequality [4.1] is derived in this paper: that for trees with maximum depth d and maximum out-degree t, the maximum bitlength length L of labels that are optimally assigned is bounded by: $d \cdot \log_2 t - 1 \leq L \leq 4d \cdot \log_2 t$. This bound, which is linear in depth, is for dynamic trees, where new nodes can be inserted at random.

## REFERENCES


[1] A. Aho, J. Hopcroft, J. Ullman, Data Structures and Algorithms, Addison-Wesley 1983.

[2] P. Bohannon, J. Freire, P. Roy, J. Simeon. From XML Schema to Relations: A Cost-Based Approach to XML Storage. ICDE 2002.

[3] E. Cohen, H. Kaplan, T. Milo. Labeling Dynamic XML. PODS 2002.

[4] T. Grust. Accelerating XPath Location Steps. SIGMOD 2002

[5] L. Mignet, D. Barbosa, P. Veltri. The XML Web, A First Study. Proc. 12th Intl.WWW Conference, Budapest, 2003. http://www.cs.toronto.edu/~mignet/Publications/www2003.pdf

[6] E. Rahm., T. Böhme: XMach-1: A Multi-User Benchmark for XML Data Management. Proc. VLDB workshop Efficiency and Effectiveness of XML Tools, and Techniques (EEXTT2002), Hong Kong, August 2002.

[7] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, R. Busse. XMark: A Benchmark for XML Data Management. Proc. VLDB, Hong Kong, August 2002.

[8] J. Shanmugasundaram, R. Krishnamurthy, I. Tatarinov. A General Technique for Querying XML Documents using a Relational Database System, SIGMOD 2001.

[9] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. SIGMOD 2002.

[10] F. Yergeau, UTF-8, A Transformation Format of ISO 10646. Request for Comments 2279, January 1998.

[11] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. SIGMOD 2001.