Insert-Friendly Hierarchical Numbering Schemes for XML

Timo Böhme[1], Selim Mimaroglu[2], Elizabeth O'Neil[2], Patrick O'Neil[2], Erhard Rahm[1]

[1]University of Leipzig, Germany {boehme, rahm}@informatik.uni-leipzig.de
[2]University of Massachusetts Boston, USA {smimarog, eoneil, poneil}@cs.umb.edu

Abstract. Numbering schemes for labeling XML nodes are the key design element to support generic XML storage and high-efficiency access in relational databases or other reliable stores. In this paper we present a class of hierarchical numbering schemes that combine two prior published schemes [2, 13] by authors of this paper. No schema is required for the data, and streaming access is supported. Our schemes support fast query processing and fast retrieval of document fragments while supporting insertion of new nodes, or sub-trees, at arbitrary positions within a document, without the need to relabel any old nodes. These insertion capabilities are not supported in other numbering schemes that have appeared in the literature. We also present a detailed quantitative analysis to comparatively evaluate the effectiveness and performance of the new numbering schemes.

1 Introduction

Designers of XML database systems are faced with the problem of efficiently representing the tree-like structure of XML documents to efficiently support both query and update operations. One difficulty encountered is that, in addition to the tree structure, the *document order* of XML nodes is meaningful, and both must be represented to support fast access to document fragments. Early work on generic relational storage of XML data [6] showed that representing tree structure exclusively by parent-child relationships results in poor performance in determining ancestor-descendant relationships and performing document reconstruction. The current paper compares two approaches authors previously published independently, DLN [2] and ORDPATH [13], to solve these problems and provide fast ancestry-related queries as well as fast document fragment retrieval. The two approaches propose distinct hierarchical numbering schemes to represent document order and ancestry of XML tree nodes, and provide a natural primary index for clustering node data shredded to relational form in document order. The feature that distinguishes these approaches from prior work of a similar nature was that they were the first to be *Insert-Friendly*, in the sense that nodes and subtrees can be inserted at any point in a document thus represented and be assigned appropriate hierarchical numbering without the need to modify any of the node numberings previously assigned. This capability is important in efficiently representing an XML database that is subject to updates.

Several numbering schemes already existed in the literature [3, 5, 8, 10, 11, 15, 16] prior to the numbering schemes this paper compares. While all of these schemes capture semantics to accelerate query operations, their practical use in most cases is restricted to static XML data, because insert operations require a costly renumbering of existing nodes. Only a few of these prior approaches try to tackle this problem [3, 8, 16], but the approaches in these papers only reduce the need for renumbering in some cases. This is particularly unfortunate in an XML database that provides transactional guarantees since such a renumbering has the hidden cost of locking nodes unrelated to the insert.

In this paper we compare the definitions and key concepts of ORDPATH and DLN, contrast them, and evaluate their pros and cons. The comparison illustrates the key concepts of advanced hierarchical numbering schemes to efficiently support both query and insert operations. We also present a detailed quantitative analysis to comparatively evaluate the effectiveness and performance of the new numbering schemes.

The rest of the paper is organized as follows. In Section 2, we look at related work. Section 3 discusses properties of simple hierarchical numbering, without considering efficient insertions. Section 4.1 classifies solutions for enhancements of hierarchical numbering, and Section 4.2 discusses insertion. Section 5 compares variants identified in the previous section and discusses their application, while Section 6 provides experimental data. Section 7 concludes the paper.

2 Related Work

Several relational mappings for generic storage of XML documents have been proposed. In [6] the tree-like node structure of XML documents is represented as parent-child relationships, but this approach is inefficient for reconstructing documents. The approach in [14] encodes the document tree in binary relations, but also has performance difficulties for reconstructing document fragments. A multidimensional mapping using document id, value and path surrogate is published in [1], but the approach does not deal with update operations and its path coding restricts the number of child elements per node.

A key approach to improve query and retrieval performance is the use of semantically meaningful node-ids when mapping XML data into nodes, either as objects in memory according to the Document Object Model (DOM) or by storing them in rows in a relational database. Several numbering schemes have been proposed using the relational approach. One of the first numbering schemes supporting ancestor-descendant relationships was published in [5], and labeled each tree node with a pair of preorder and postorder position numbers. So for each pair of nodes x and y, x is an ancestor of y if and only if $preorder(x)<preorder(y)$ and $postorder(x)>postorder(y)$. A similar scheme was chosen in [15]. While this numbering scheme is easy to compute and can be used for streamed XML data it is highly inefficient when new nodes are inserted, because each node in preorder traversal coming after the inserted node has to be updated.

The update problem was addressed by the *extended preorder* numbering scheme introduced in [11] and adopted in [3] as *durable node numbers*. They also use a pair of numbers for each node. The first number captures the total order of the nodes within the document like the preorder traversal but leaves an interval between the values of two consecutive nodes. The second number is a range value. As with the preceding scheme the ancestor-descendant relationship between node x and y can be determined from x is an ancestor of y *iff* $order(x) < order(y) \leq order(x) + range(x)$. With the sparse numbering insert operations will not necessarily trigger renumbering of following nodes if the difference of the order value of preceding and following node is larger than the number of inserted nodes. However inserting new subtrees with a substantial number of nodes requires renumbering as well.

In [9] a numbering scheme called *simple continued fraction* (SICF) is proposed. It numbers the nodes from left to right and top-down. Each node number can be expressed as a sequence of integer values – adding an integer per tree level – or a fraction. This approach reduces the update scope,[1] after a node insertion, to the following siblings and their descendants. Furthermore SICF fails if a certain tree depth is reached.

Another approach with left to right and top-down numbering is published in [10]. The so-called *unique element identifiers* (UID) are based on a tree with a fixed fan-out of k. If a node has less than k children virtual nodes are inserted. The UID allows the computation of the parent node label (id) and the label of child i. This approach has two main drawbacks: (1) fixed fan-out is problematic with irregular structured documents, (2) node insertion requires updates of all right siblings and their descendants.

Some of the UID drawbacks were tackled in [8] with the definition of recursive UIDs (rUID). Here the tree is partitioned in local areas allowing different fan-outs and reducing updates after insertion of nodes. However it needs access to the whole tree in order to compute the identifiers, which prevents the streaming of data for insertion.

Theoretical findings for labeling dynamic XML trees are given in [4]. The described schemes determine labels that are persistent during document updates and contain ancestor information. Furthermore lower bounds for the maximum label length are presented. However no sibling order of the XML nodes is maintained, so it is not suitable for general XML document management.

Several solutions for encoding Dewey ids with respect to query processing and lock management are considered in [7]. The authors of this paper present their own approach which is similar to ORDPATH. As an extension they propose compression methods which can be used by implementations of this encoding. The quality of the encodings is measured in terms of initial label length for several documents. Update operations are not considered.

3 [Simple] Hierarchical Numbering

We define a *Simple Hierarchical Numbering* to be a scheme for hierarchical numbering of nodes of an XML tree that *does not* support inserts of new nodes without relabeling. A *Hierarchical Numbering* (not prefixed by "Simple") may or may not provide support for inserts. We define a ***Hierarchical Numbering*** to be one that assigns labels to the nodes of an XML tree by a scheme described as follows. The label of the root node (said to be at level 1) is a simple binary string, which we define here to have an ordinal value "1". A ***Hierarchical Label*** of a lower-level node N (at level k in the tree) is made up of a sequence of k ***Level labels***; a level label of level $m \leq k$ of the node N is a

[1] the set of nodes whose numbers (potentially) have to be updated

binary string that specifies the relative position of the level *m* ancestor node of *N* with respect to its parent at level *m*-1. In the DLN schemes, there is a single 0-bit between successive level labels in a hierarchical label. An initial XML tree load in document order allows us to represent each binary string level label by a simple ordinal number, and the hierarchical label of the node *N* at level *k* can be represented by a concatenation of *k* such ***ordinals***, separated by dots. For example we say that the level 3 node M has label 1.5.9, with three level labels: the initial level label 1 represents the root of the document tree, the second level label 5 specifies that the level 2 ancestor of node *M* is the fifth sibling from the left of children of the root, and the third level label 9 specifies that *M* is the ninth sibling from the left of children of its parent node of level 2. In this freshly loaded XML tree, document order of the nodes corresponds to numerical order of the successive ordinals in a label. We say '1.5.9' < '1.5.10' to indicate that 1.5.9 is lower than 1.5.10 in document order. Note that the inequality '1.5.9' < '1.5.10' is not true in an ASCII form of this representation, but this ordinal form merely indicates the binary representation of level labels that supports the appropriate ordering. Indeed, the binary string level values for successive ordinals at the same level will be of different lengths, and once we describe hierarchical labels for node inserts between existing nodes it will become clear that the level labels resulting will not always be describable as simple ordinals. Nevertheless, we shall use this ordinal representation for examples in all simple examples that follow.

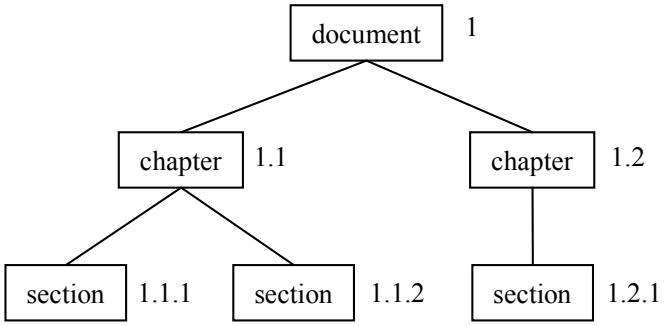

Figure 1. Hierarchical Numbering

Figure 1 shows an example of a simple hierarchical numbering scheme using ordinals for level labels and a dot character as separator. Such a hierarchical numbering scheme looks similar to the Dewey Decimal classification scheme used in libraries, which inspired the XML hierarchical labeling described in [16] by Tatarinov et al.

Recall that a *Simple Hierarchical Numbering* is a scheme for labeling an XML tree that *does not* support inserts of new nodes without relabeling. Even limited in this way, simple hierarchically numbered labels are superior to other proposed schemes for efficient XML query evaluation in several ways. Outstanding properties are these.

- From a given node label one can determine the parent label and the label of all ancestor nodes up to the root node without requiring I/O operations
- The depth of a node within the tree can be determined from its label
- Given two node labels *A* and *B,* one can determine from the labels alone if *B* with regard to *A* is a parent, child, ancestor, descendant, preceding node, following node, preceding-sibling, or following-sibling

Thus a number of XML query operation primitives can be evaluated based on labels alone, without the need for navigating the document tree. Hierarchical numbering permits us to sequentially label document nodes, which is essential when all nodes of a large document need to be loaded in document order. Furthermore the level label bitstring encoding ensures that ordinary byte-string comparison follows XML document order. This property allows adjacent nodes to be clustered in relational storage [9], and allows queries to reconstruct document fragments by range scans. This gives us another property of our scheme.

- By a range scan through the node label index from a given context node label, we can determine the set of preceding, following and descendant nodes
- The level labels of our representations are single (variable-length) values

An important feature of simple hierarchical numbering comes into play when new nodes or subtrees must be inserted. Compared to some other labeling schemes, simple hierarchical numbering restricts the *update scope*, i.e.

the range of nodes potentially to be relabeled, to the sibling nodes that follow the inserted node and their descendants. Of course there might be a great number of sibling nodes in some cases: the Text Centric XML document in the XBench Benchmark [17] has nodes for all words of a dictionary as siblings sitting below the root. Thus the insert-friendly property of our labeling schemes described in Section 4, by which a new node or subtree can be inserted without any rearrangements of existing nodes at all, has significant value.

4 Insert-friendly Hierarchical Numbering

In [16] Tatarinov et al. proposed two approaches to an insert-friendly improvement on simple hierarchical numbering with what they called a Dewey order. The first allocates a fixed number of bytes for each level label, e.g. 1.9 would be represented as 0001 0009 (in hexadecimal blocks of four digits) and 1.10 as 0001 000A. One drawback of this solution is the immense storage overhead, since the largest level label must fit within the range of bytes. To address this in the second approach, a node label is represented as the concatenation of UTF-8 encoded level labels. UTF-8 encodes all 31-bit integer values with a variable number of bytes (between one and six bytes) and a binary (bitstring) comparison of UTF-8 encoded values yields the appropriate numeric order of the values that are represented. However a minimum of one byte per level label is still inefficiently large for levels with only a few siblings. Furthermore, neither of these approaches fully supports insertions without relabeling.

In [2,13] authors of this paper proposed two solutions (ORDPATH, DLN) for hierarchical numberings to label XML nodes providing efficient support for insertions. In this section we classify and compare the ideas from these two solutions: we discuss the details of the hierarchical labelings that support byte-string comparisons of labels to determine document order and show how insertions of new nodes can be performed without relabeling existing nodes.

4.1 Encodings

As explained at the beginning of Section 3, node labels produced by a hierarchical numbering scheme during an initial document-order load can be represented by concatenated sequences of ordinals, each representing level labels: for example the label ‘1.5.9’. Our first goal is to explain how the ordinals representing these labels can determine bitstring level labels so that byte-string comparison of the concatenated sequence of level labels preserves document node sequence: e.g. ‘1.2.9’ < ‘1.2.10’. This labeling can then be used as a primary index to ensure that B-Tree indexing clusters the rows in document order. As a second goal, we would like to minimize the number of bits needed to encode the level labels; however this goal must normally be consonant with the need to accept a document-order one-pass load of nodes of a tree with arbitrary numbers of nodes at each level, without knowing the number of nodes in advance. Finally, a third goal is to support insertions without massive relabelings. In the following subsections we present several encodings from ordinal values to bitstring level labels targeted to the first two goals; we will explain in Section 4.2 how the third goal is met.

4.1.1 Prefix-specified Level label Encoding (Simple ORDPATH)

Following Section 3, the prefix “Simple” in “Simple ORDPATH” means that no effort is made here to support efficient inserts. According to [12] the distribution of XML element fan-out follows a power law, i.e. most elements have only a few children. Therefore the probability for small level labels is much higher than for large level labels, and we can expect a good compression if we use a variable-length encoding for level labels, with smaller and more probable values mapped to shorter bit sequence codes whereas larger values are mapped to longer ones.

In order to be able to uniquely parse concatenated bitstring level labels ORDPATH uses prefix-free encoding[2]. This encoding has to provide the same ordering property as corresponding ordinal level labels to maintain document order. As discussed above, UTF-8 encoding is one possibility, but we can do better. Table 1 shows a trivial example of a variable-length encoding of the values 1-6 with the desired property.

[2] A prefix-free encoding guarantees that for two different values *a* and *b*, the code for *a* is not a bitstring prefix of the code for *b*

Ordinal value	Bitstring value, a prefix-free encoding
1	01
2	10
3	110
4	1110
5	11110
6	111110

Table 1. A trivial prefix code scheme

Since the number of different level labels can become quite large, the ORDPATH scheme doesn't try to maintain a specific encoding for each ordinal value. Instead ordinal values are partitioned into groups of consecutive values called prefix groups. All values of each prefix group are encoded into bitstrings with the same number of bits. We can divide the bitstring-encoded value into a bitstring prefix *L* that identifies the group and the following bitstring *O* that distinguishes the group members. Any sequence of prefix-free codes in binary string order can be used for group prefixes and the length of O is fixed for each group. Table 2 presents one such ORDPATH encoding.

Ordinal level label	Bitstring value, simple ORDPATH			
	L	# of bits in *O*	L with first *O* bitstring	L with last *O* bitstring
1..2	01	1	01 0	01 1
3..6	10	2	10 00	10 11
7..22	110	4	110 0000	110 1111
23..278	1110	8	1110 00000000	1110 11111111
279..4374	11110	12	11110 000000000000	11110 111111111111
4375..69910	111110	16	111110 000…000	111110 111…111
…				

Table 2. A Simple ORDPATH scheme (insertions not supported)

In Table 2, each prefix group has a row. For example, the second row describes a group with prefix *L* of 10 and following *O* bitstrings of 4 bits: 00, 01, 10, and 11. Thus the first such bitstring is 10 00 and the last is 10 11. There are four bitstrings in all, representing ordinal values 3, 4, 5, and 6.

With this encoding, the label 1.2.1 would be represented as 010 011 010 and label 1.3.1 would become 010 1000 010. Notice that the prefix group is different for the level labels 2 and 3, indicated by the *L* codes 01 and 10, and the number of bits that determines the *O* value changes from 1 to 2. It is shown in [13] that the byte-string comparison of node labels with level labels encoded in this way preserves document order. (This is so even though the concatenated bitstrings of an ORDPATH might end in the middle of a terminal byte, extended with 0-bits.) No separator is needed between level label bitstrings since the length of each value *O* is determined by the prefix group, which in turn is specified by its identifier *L*, and if the comparison of two ORDPATHs doesn't differentiate between *L* values the succeeding *O* values will be the same length.

This is the form of encoding used in ORDPATH, except that a proper ORDPATH scheme also allows for arbitrary negative level labels (as shown in section 4.2.1, Table 6) to support inserts of new children to the left of all current children of a node, using *L* values such as 001, 0001, etc. Also, the full ORDPATH scheme reserves the even ordinals as points for future inserts (described in Section 4.2). However, the simplified ORDPATH scheme given above is sufficient to label arbitrary XML documents in document-order loads, with one ordinal for each level. It is important to realize that the prefix-free bitstrings for *L* can be varied, along with the number of bits in the associated *O* bitstrings, so that ordinals at each level will be most efficiently represented according to statistical behavior expected by the designer. For example, in the dictionary example of [17] where a very large number of word nodes sit beneath the root, we might use an *O* bitstring length of 20 for the *L* bitstring 11110, allowing for a million words, yet still using short codes for all the moderate fanouts in the document. On the other hand, a fixed ORDPATH scheme works well across many document types, as shown in Section 6.

4.1.2 Subvalue Encodings (DLN)

Like the ORDPATH scheme, DLN encodes level label ordinals with a varying number of bits. In contrast to ORDPATH, where a prefix determines the bit-length of a bitstring for each ordinal (and thus for each level label in an initial document load), DLN divides a level label bitstring into one or more ***subvalues***; the level label bitstring is thus the concatenation of its subvalue bitstrings. Distinct separator bits are used to separate subvalues of a level label (1-bits) and to separate level labels of a node label (0-bits). The following example shows the structure of a DLN label with subvalues having a length of 3 bits each:

```
001 0 000 1 011 1 101 0 100 1 011     label
L0  | L1              | L2            level labels
S0  | S0  | S1  | S2  | S0  | S1      subvalues
001 | 000   011   101 | 100   011     bitstring level values
```

By using the 0-bit as a level separator, we ensure that the labels of all children of a node *N* come before the siblings following *N* in a binary comparison, a clear requirement for a preorder numbering scheme. For instance if we use 3 bits to encode all subvalues, with 001 representing 1, we would represent 1.3.3 as 001 *0* 011 *0* 011 and 1.3.4 as 001 *0* 011 *0* 100 *1* 000 (note the added subvalue to extend 3, represented as 011, to 4, represented as 100 *1* 000).

For added flexibility, subvalues of different but predetermined length may be employed. In DLN, the length of a subvalue is determined by its position within a level label. For instance a length specification of '1|4|5' means the first subvalue s_0 has a length of 1 bit, subvalue s_1 has a length of 4 bits and each subvalue s_2 and all following subvalues within the corresponding level label are made up of 5 bits. The length specification of subvalues is a parameter than can be adjusted to the document needs. Smaller subvalues may be space-efficient for a small number of children per node, but require many subvalues for a large number of children. Typically, a length specification applies to all nodes of a document or even all documents of the database. In some cases, it can be advantageous to use tailored length specifications on a per-document-level basis.

Given this subvalue structure of a DLN label, we explain below how subvalues are used to encode level labels. Two encodings are differentiated which can be used simultaneously for a given document[3]: (1) A ***Streaming encoding*** where no advance knowledge of the tree structure is assumed, e.g. when the document arrives as streaming data to be loaded in document order; (2) A ***DOM encoding*** where the number of children at each node is known in advance, e.g. because the document is available via DOM. DOM encoding is also applicable if the data is initially loaded in a Streaming encoding, but then reloaded in a DOM encoding for optimally compressed labels. In general we assume that a DOM encoding is not subject to updates, since this would change the structure of the tree. However, we still allow for inserts as a safety measure. The ORDPATH approach introduced in Section 4.1.1 does not assume a known tree structure: it accepts and efficiently loads XML data arriving in document order; it is also able to accept an arbitrary number of future inserts. Various experiments comparing these approaches appear in Section 6.

Streaming encoding (DLN_{Stream})

With streaming data the tree structure is not known in advance. Thus we cannot predetermine the required number of subvalues to handle all sibling nodes. For this case the following algorithm dynamically determines the relevant bitstring of a level label, using an increasing number of subvalues as the ordinal increases. The first child of a parent node uses a level label representing ordinal one, and each new sibling is assigned the succeeding level label (i.e., the level value is incremented.) A new subvalue is appended when the level label to be incremented has left only a single 0-bit. Table 3 illustrates this format, using subvalues of length 4. As explained above, subvalues of varying lengths are also possible.

[3]To be used simultaneously, the DOM and Streaming encodings need to use the same subvalue length specification.

Ordinal	Bitstring level labels, DLN_{Stream}	first bit pattern
1..7	0XXX	0001
8..71	10XX 1 XXXX	1000 1 0000
72..583	110X 1 XXXX 1 XXXX	1100 1 0000 1 0000
584..4679	1110 1 XXXX 1 XXXX 1 XXXX	1110 1 0000 1 0000 1 0000
4680..37447	1111 1 0XXX 1 XXXX 1 XXXX 1 XXXX	…
…		

Table 3. DLN_{Stream}

In Table 3, level labels 1 to 7 are encoded with a single subvalue, level labels 8 to 71 with two subvalues, etc. With additional subvalues the number of nodes that can be encoded increases exponentially, a property that is shared by all other encodings we list here, including ORDPATH. We call this encoding DLN_{Stream}.

Although we do not discuss inserts in detail until Section 4.2, some of the details of the bitstring level labels above are related to supporting inserts after using these labels in the initial load. DLN bitstring level labels must obey the following rules to make inserts possible.

DLN Rule 1. All level labels must contain at least one 1-bit. Otherwise it would not be possible to insert a sibling of the node that sits on its left.
DLN Rule 2. For two level labels *a* and *b*, where *a* is a prefix of *b,* the non-matching right-hand subvalues of *b* must contain at least one 1-bit. Otherwise it would be impossible to insert a node between the two nodes identical except for rightmost level labels *a* and *b*.

Note that only inserts, not deletes, need to pay attention to these rules.

Improving DLN bit usage

Looking at the DLN_{Stream} bit patterns of Table 3 we note that the number of leading 1-bits determines the number of subvalues. If the 1-bit separators were removed, the result would be a scheme strongly reminiscent of ORDPATH encoding. This is true only for an initial load, however, since the correspondence would depart from ORDPATH after a few ad-hoc inserts; furthermore, we cannot remove the 1-bit separators and still use DLN_{DOM} for the initial storage of a document and DLN_{Stream} for later insertions, as we propose below.

However we can improve compression of DLN_{Stream} and still retain the ability to combine it with DLN_{DOM}. In what we call the $DLN_{StreamOpt}$ scheme, when we add a new subvalue to a level label *x* to produce a new label, we do not change the bits of *x*, but instead simply append the subvalue (itself not all 0s) to *x*. We follow the same rule as in DLN_{Stream} for the case of a single 0-bit. This is demonstrated in the following table.

Ordinal	bitstring level label, $DLN_{StreamOpt}$	First level label	last level label
1..7	0XXX	0001	0111
8..22	0111 1 XXXX	0111 1 0001	0111 1 1111
23..86	10XX 1 XXXX	1000 1 0000	1011 1 1111
87..101	1011 1 1111 1 XXXX	1011 1 1111 1 0001	1011 1 1111 1 1111
102..613	110X 1 XXXX 1 XXXX	1100 1 0000 1 0000	1101 1 1111 1 1111
…			

Table 4. $DLN_{StreamOpt}$

Comparing Table 4 to Table 3, we see that starting with two subvalues we can encode more level labels with the same number of bits (86 values in Table 4 compared with 71 in Table 3). The $DLN_{StreamOpt}$ encoding allows for inserts between these labels, although with a somewhat more complicated algorithm than with DLN_{Stream}.

DOM encoding

The DOM encoding also uses several subvalues of a predetermined length as introduced above. However it most efficiently uses the combined subvalue bit range under the assumption that the exact number of children for a specific node is known. The DOM encoding, which we denote DLN_{DOM}, uses as many subvalues as needed to fit all

level labels of the remaining sibling nodes to be encoded within the concatenated bit range. For example, when we know that a node N has n_{remain} = 199 siblings, we can encode the final level label of its labels using a minimum of 8 bits. If the subvalue length is specified as '3|2|3' we have to concatenate 3 subvalues as shown in Table 5. In general, we need a combined length of subvalues l(sv) ≥ $\log_2 (1 + n_{remain})$ where n_{remain} is the number of remaining sibling nodes.

Table 5 illustrates that the number of subvalues can actually decrease for the final sequence of sibling node encodings. Starting from the node with 6 remaining siblings we can encode all labels with the remaining bits of the first two subvalues.

Node	remaining siblings	Bitstring level label, DLN_{DOM}
N	199	000 *1* 00 *1* 001
sibling 1	198	000 *1* 00 *1* 010
sibling 2	197	000 *1* 00 *1* 011
...	...	...
sibling 191	8	110 *1* 00 *1* 000
sibling 192	7	110 *1* 00 *1* 001
sibling 193	6	110 *1* 01
...	...	...
sibling 199	0	111 *1* 11

Table 5. Example of DLN_{DOM} numbering of 199 siblings

The bit usage of DLN_{DOM} is always optimized. If a level label contains trailing subvalues with only 0-bits these subvalues can be removed. This *pruning* has no implication on the label order nor does it impact creation of sibling labels since in DLN_{DOM} we know the number of remaining siblings and calculate the needed number of subvalues accordingly. Figure 2 compares the maximum and average level label length of DLN_{DOM} and $DLN_{StreamOpt}$ (with all subvalues having 3 bits) for various numbers of sibling nodes.

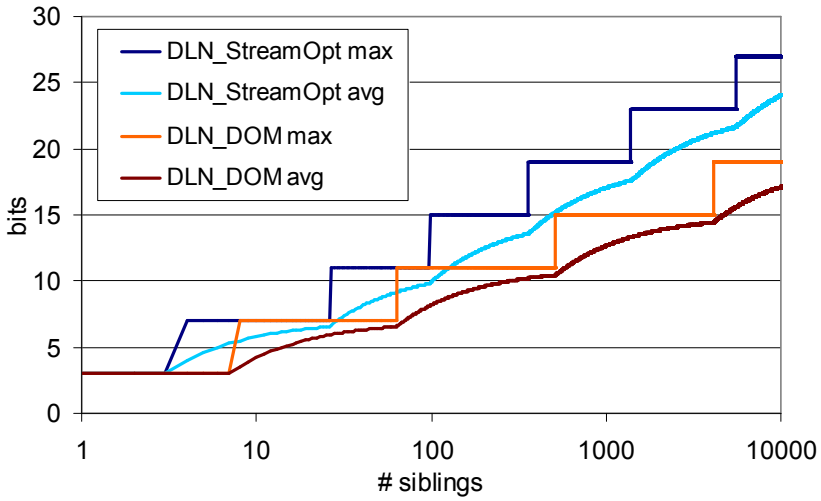

Figure 2: Comparison of DLN_{DOM} and $DLN_{StreamOpt}$ encoding a number of sibling nodes

4.2 Insertion Techniques

Assume that we have already loaded an XML document tree in a relational table (or other store), using one of the node encodings presented in Section 4.1. In this section we explain how to perform insertions of new nodes at arbitrary positions within this tree without the need to relabel a potentially large number of following siblings. Note that the new node inserted may be the root of a subtree. Once this root has been assigned a proper label within the

existing tree, it will be easy to assign labels for all descendents of that root using the corresponding level label encodings for children of an existing node we have introduced.

There are three types of insertions we need to support:

Type 1 Insertions: inserting the last child of a node to the right of all existing siblings

Type 2 Insertions: inserting a child of a node between two existing siblings

Type 2 Insertions: inserting the first child of a node to the left of all existing siblings

Since all encoding schemes from Section 4.1 except DLN_{DOM} allow for adding successive sibling labels to the right of all prior sibling values, we can add a node to the right of all siblings (insertion type 1) without problems. Thus, we need new insertion techniques only to deal with insertion types 2 and 3; we will also introduce a solution for type 1 with DLN_{DOM}.

In all the schemes of this section, inserting new nodes between existing nodes (type 2 insertions) may cause the length of the new nodes' encodings to exceed the length of its siblings' encodings. It is unlikely that this affects performance significantly, but this is discussed further in Section 6. On the other hand, inserting a very large number of new nodes can cause the need to "reload" the tree, or at least all descendents of the most affected parent. This situation is analogous to the need to reload a database table after multiple inserts and deletes have caused the effectiveness of clustering or secondary indexing to deteriorate.

4.2.1 ORDPATH Insertion

ORDPATH [13] supports Type 3 insertions to the left of all existing siblings by generating codes for ordinals with decreasing negative values, as shown in Table 6. To put a sibling to the left of 1.3.1, we can label it 1.3.-1, and the sibling before that can be labeled 1.3.-3, and so on. Note we are using only odd ordinals in these labels; we explain the reason for this below.

The bitstring codes (one for each ordinal, odd or even) for ORDPATH have the format shown in Table 6 in one possible ORDPATH scheme.

Ordinal values	Bitstring for one ordinal, for ORDPATH Std			
	L	# of bits in *O*	First odd-ordinal code	Last odd-ordinal code
…				
[-12,-5]	0001	4	0001 000 1	0001 111 1
[-4, -1]	001	2	001 0 1	001 1 1
[0,1]	01	1	01 1	01 1
[2,5]	10	2	10 0 1	10 1 1
[6,21]	110	4	110 000 1	110 111 1
[22, 277]	1110	8	1110 0000000 1	1110 1111111 1
[278,4373]	11110	12	11110 00000000000 1	11110 11111111111 1
[4374,69909]	111110	16	111110 000…000 1	111110 111…111 1
…				

Table 6. A practical ORDPATH scheme used in Section 6 as ORDPATH "Std"

We now know how to perform Type 1 and Type 3 insertions of children to the right and left of all existing siblings in the ORDPATH scheme. For the remaining Type 2 case, we require that odd ordinals can be used in generating labels of nodes arising from Type 1 insertions (thus in a normal XML tree load) and Type 3 insertions. Given this, we can insert a new node Y between any two siblings of a parent node X (a Type 2 insertion, known as ***careting in***) by creating a component with an even ordinal falling between the (odd) ordinals of the two siblings, then following this even ordinal with a new odd ordinal, usually 1.

As an example, we show how to caret in a sequence of K siblings having parent node with ORDPATH 3.5, the sequence to fall between sibling nodes 3.5.5 and 3.5.7; we do this by providing the new siblings with an even ordinal placeholder 6, thus: 3.5.6.1, 3.5.6.3, 3.5.6.5, ... Here, the value 6 at level 3 (or any even value in any non-terminal ordinal) represents a caret only; that is, it doesn't count as a full level value that increases the depth of the node in the tree. Instead, the level 3 values for the inserted nodes above are, respectively: 6.1, 6.3, and 6.5. However the caret does have the desired effect on ORDPATH order, since binary comparisons 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 an entire subtree with root at 3.5, falling between the sibling nodes 3.5.5 and 3.5.7, using only 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 these siblings.

In interpreting ORDPATH labels, 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. 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 subtree (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 subtree 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 subtree additions to occur, one within another, not at either end. Clearly this is a rarity.

We can consider the last bit of each ORDPATH bitstring encoding to be a delimiter, where 1 and 0 have meanings opposite to DLN: 1 means we have reached the end of the level label, whereas 0 means there is a continuation part to define a level label for a careted-in node. In particular, the construction ensures that all node labels end in an odd ordinal, so a full ORDPATH always has a 1-bit at the end. We 0-fill the rest of the byte at the end of the ORDPATH bitstring, so that even if we only know the byte-length we can locate the final 1-bit within the last byte, and thus the end of the binary ORDPATH.

It will be shown in Section 6 that inserts, random or in runs, produce only logarithmic growth in label length. This can be understood by observing that a first insert between 3.5 and 3.7 yielding a new label 3.6.1 opens up a whole new address space of possible labels in this neighborhood, 3.6.3, etc., all of roughly the same length.

Surprisingly, an ORDPATH scheme can be devised that has 0 bits in some *O* parts. This has the advantage of allowing us to encode level label 1 with the shortest possible bitstring (2 bits), as shown in Table 7. Comparing the bitmap scheme of Table 7 with that of Table 6, we also see that the length of the *O* bitstrings increases much faster. This flexibility of representation for the bitstring length of *O* associated with bitstring *L* allows bitstring labels to optimize length for statistical expectations of sibling numbers, and is an important feature of ORDPATH.

ordinals in group	Bitstring encoding for one ordinal, for special ORDPATH scheme			
	L	# of bits in *O*	First odd-ordinal code	Last odd-ordinal code
…				
[-10,-3]	0001	4	0001 0001	0001 1111
[-2, 0]	001	2	001 01	001 01
[1,1]	01	0	01	01
[2,5]	10	2	10 0 1	10 1 1
[6,21]	110	4	110 000 1	110 111 1
[22, 277]	1110	8	1110 0000000 1	1110 1111111 1
[278,65813]	11110	16	11110 000000000000000 1	11110 111111111111111 1
$65813 + 2^{32}-1$	111110	32	111110 0000000…00000000 1	111110 11111111…1111111 1
…				

Table 7. ORDPATH scheme with 2-bit encoding of 1

Note that the singleton L (for ordinal 1) is 01 in Table 7. This value for L cannot be reduced to a single bit and keep all the normal ORDPATH properties, because the set of L codes need sequences on both sides of this code in binary lexicographic order to allow arbitrary inserts before the first child of a node. In the scheme of Table 7, the pattern 001 11 is not in use, because it is the immediate neighbor of the special 2-bit pattern and has no even-ordinal code between as needed for careting in. The former type of ORDPATH scheme, with slightly longer encoding of 1 (3 bits instead of 2) is more practical for general use.

4.2.2 DLN Subvalues for Careting

The DLN approach [2] to prevent renumbering caused by Type 2 insertions between sibling nodes employs the subvalue approach from chapter 4.1.2. The general idea is that between two sibling nodes with labels *a* and *b* and consecutive[4] level labels, a node can be inserted whose label *c* is composed of *a* appended by one or more subvalues,

[4] Here, consecutive means that, according to the encoding scheme, if level-value of *a* is increased by 1 the result is greater or equal to the level-value of *b*

separated by 1-bits. For instance between two labels a = 0011 and b = 0100, one can insert a new label c = 0011 *1* 0001 which will clearly compare high to a and low to b in left to right bit-by-bit comparison. Furthermore, c will compare high to any child of a, since a child of a will have the form child(a) = 0011 *0* XXXX. The technique of adding subvalues is also used for Type 1 and Type 3 insertions. Note that this argument shows that such an intermediate label always exists, but this is not a demonstration that this insertion algorithm supports logarithmic growth of labels under repeated inserts. We discuss better algorithms below.

DLN_{Stream}

The insertion algorithm of DLN_{Stream} for a Type 2 insertion differentiates two cases. The first case is given when the final level value of a (lv_a) is not a prefix of the final level value of b (lv_b), i.e. lv_a and lv_b differ in at least one bit position. Here the inserted label c is a modified label a where the bitstring x following the differing bit position is incremented according to the DLN_{Stream} algorithm, as follows. Use the successive binary bitstring to x, and also append a subvalue if x had only one 0-bit[5], e.g. a = 0010, b = 0100, c = 0011 *1* 0000. In the other case lv_a is a prefix of lv_b. In order to create label c we take label b and decrement the part y of lv_b which exceeds lv_a. This decrementation of a level value (or bit range) in DLN_{Stream} works analogously to incrementing it. If y had only a single 1-bit before decrementing it we append a subvalue with all bits set to '1' to the decremented level value. We demonstrate this in the following example: DLN_{Stream} decrementation of 0001 results in 0000 *1* 1111. The same decrementation algorithm is used for Type 3 insertions (inserting before the first child) in DLN_{Stream}. For repeated Type 3 insertions this reuse of the DLN patterns among the inserted nodes (at a certain position in the original document) ensures a logarithmically increase of level label length instead of a linear increase, shown experimentally in Section 6.

DLN_{DOM}

DLN_{DOM} can as well be used for inserting new nodes. However its application is only appropriate if the number n of sibling nodes which will be inserted after the current node to be inserted is known. Otherwise DLN_{Stream} is preferable. For Type 1 insertions (appending sibling to a node) with DLN_{DOM} we start with the label a of the existing node. We take the final level value of a (lv_a) and subtract it from the maximum value possible by the bit range of this level (all 1-bits). If the result is below n+1 (bit range insufficient to label all nodes to be inserted) another subvalue is appended. This is repeated until the level values of all nodes to be inserted will fit into the current bit range. After incrementing the resulting value by 1 we have the final level label of the new node label. Examples for Type 1 insertions in DLN_{DOM} are given in the following table.

label of existing node	number n of remaining nodes to be appended	label of appended node (generated with DLN_{DOM})
0001 0 0100	10	0001 0 0101
0001 0 0100	11	0001 0 0100 1 0001
0001 0 1111	0	0001 0 1111 1 0001
0001 0 1111	15	0001 0 1111 1 0000 1 0001

Table 8. Labels for appended sibling nodes using DLN_{DOM}

In an analogous way Type 3 insertions are defined. If the level value of the existing node is below n+2 (the smallest possible value must contain one 1-bit) another subvalue is appended. This is repeated until the level value is large enough. The label of the newly inserted node gets this level value which is decremented by 1. Table 9 provides examples.

[5] if x has no 0-bit at all we don't increment it but append a new subvalue with the right most bit set to '1'

label of existing node	number *n* of remaining nodes to be inserted (to the left)	label of to the left inserted node (generated with DLN_{DOM})
0001 0 0100	2	0001 0 0011
0001 0 0100	3	0001 0 0011 1 1111
0001 0 0001	0	0001 0 0000 1 1111
0001 0 0001	15	0001 0 0000 1 1111 1 1111

Table 9. Labels for nodes inserted before siblings using DLN_{DOM}

For Type 2 insertions between labels *a* and *b* DLN_{DOM} again strives for minimal number of subvalues needed to enable insertion of current node and *n* remaining siblings. This is accomplished as follows. We start with the first subvalue of final level value of *a* (v_a) and final level value of *b* (v_b). If $v_b - v_a < n+1$ we repeatedly add the next subvalues from the corresponding level values to v_a and v_b. If the level values are exhausted subvalues with 0-bits are added. The level label of the newly inserted node gets v_a incremented by 1. Examples of insertion Type 2 are shown in the following table.

Label of existing left node	label of existing right node	number *n* of remaining nodes to be inserted	label of inserted node (generated with DLN_{DOM})
0001 0 0001	0001 0 1000	5	0001 0 0010
0001 0 0001	0001 0 1000	6	0001 0 0001 1 0001
0001 0 0001	0001 0 1000	110	0001 0 0001 1 0001
0001 0 0001	0001 0 1000	111	0001 0 0001 1 0000 1 0001

Table 10. Labels for nodes inserted between siblings (Type 2) using $DLN_{DOM.}$

5 Comparison of ORDPATH and DLN

Clearly ORDPATH and DLN are very similar. Both use the power of prefix codes to generate unique identifiers containing knowledge of the path, and by careful coding of original identifiers for an original load, leave space between any two identifiers for later inserts, even of whole subtrees. This reservation for future inserts costs one bit per level in overall identifier length. That one bit is handled differently in ORDPATH and DLN. In ORDPATH, it is expressed in the required oddness of ordinals used in the original load. The least significant bit of such ORDPATH bitstring codes must be 1. In DLN, level labels are separated by single 0 bits, to allow a 1 bit to mark an extension for careting in.

5.1 Features of Both ORDPATH and DLN

We have already presented the basic features of the two approaches, providing unique labels that contain level information allowing determination of ancestry relationships, including new labels for inserted nodes. In addition, there are more subtle features.

GRDESC(p)

To represent the set of all descendents of *p* as a range of label values (*p*, *q*), it is extremely useful to have a bitstring *q* that sits to the right of all label bitstrings of descendents of *p* but to the left of any label bitstrings of non-descendents on the right of *p*. Clearly *q* cannot be itself a valid node label bitstring, but it can use the bit patterns set up for future inserts. Following [13] we can call *q* "GRDESC(*p*)", for greater-than-descendents of *p*. ORDPATH simply uses the label for *p* modified by replacing the final (odd) ordinal of *p* by the even ordinal that must follow it. For justification, see [13].

In DLN_{Stream} or $DLN_{StreamOpt}$, GRDESC(*p*) = *p* || 1, a single 1 bit appended to *p*. All descendents of *p* have a 0 bit next, and all non-descendents to the right of *p* have higher value among the bits within the length of *p*, or if the same bits as *p* within the length of *p*, then a 1 followed by not all zeroes.

In the simple hierarchical labeling schemes, there is no GRDESC(*p*) algorithm of such a simple form. If the scheme has a highest level lable *p1*, that can be used: GRDESC(p) = *p* || *p1*.

Field Separator
Both ORDPATH and DLN schemes provide a field separator bitstring, an appended bitstring (a certain number of binary zeroes) that ties off the label bitstring much like a 0-byte ties off an ASCII string. For an example of use, consider sorting tuples (x, y) where x is one of these labels and y is anything that sorts by its bitstring value, say a byte. With a field separator bitstring s, the concatenated bitstrings $x \parallel s \parallel y$ will sort properly as (x, y) even though the x bitstrings are of variable length. Another use of s is to hide additional data at the end of a label without perturbing its document position. For ORDPATH Std (Table 6), s = 0000 would work as a field separator if the first prefix in use is 0001. This same field separator s = 0000 would also work for DLN if the first subvalue has 3 bits.

5.2 Specialized ORDPATH Schemes showing ORDPATH-DLN relationship

A particular ORDPATH scheme is based on a set of prefix codes (put in binary string order) and the number of bits in O for each prefix. By choosing these parameters carefully, we can mimic related schemes. The relationship is very close for original-load labels, as you can see by how ORDPATH DLN can be set up to mimic Streaming DLN in the next paragraph. Each full ORDPATH is one bit longer than the corresponding DLN label because of the difference between DLN level label separators and ORDPATH even/odd ordinals. The details of generating labels for careted-in nodes are somewhat different, but in both cases work to avoid undue growth in label length, as shown experimentally in the next section.

ORDPATH DLN
Streaming DLN effectively has prefixes: simply take the bit patterns before the XXX parts in the pattern definitions. Thus a Streaming DLN scheme maps almost perfectly to a particular ORDPATH scheme, as does the optimized version, to another ORDPATH scheme. The subvalue separator bits can be dropped, with only one lost feature, that being the ability to mix these labels with DLN_{DOM} labels in the same document. Consider the DLN scheme of Table 3, which we can call "basic DLN." We can define "basic DLN ORDPATH" as a particular ORDPATH scheme with prefixes {0, 10, 110, …} and associated bit lengths of O as {4, 7, 10, …}. Then, as usual with ORDPATH, only odd ordinals are used in the original load, reducing the effective lengths of O by one, to {3, 6, 9, …} to match the XXX patterns of Table 3. The final bit of an O corresponds to the level label separator bit of DLN. Because of the equal number of these ordinal bits, the same number of usable ordinals are generated (for original loads) by both schemes, in each group, with the same level label length if one DLN separator bit is included. However, other DLN schemes than "basic DLN" provide better statistical label length properties, so the case called "ORDPATH DLN" in the following section on experiments is based on the subvalue length sequence 3|2|4|2|3.

ORDPATH UTF-8
As discussed at the start of Section 4, the order-preserving number compression scheme UTF-8 [18] was used in [16] where it was called the Dewey order. Like DLN, the UTF-8 encoding table has bit patterns with XXX for any sequence of 0s and 1s. For example, hex values 0000 0080 through 0000 07FF are encoded using the pattern 110XXXXX 10XXXXXX. These will include all the patterns starting with 110, so we are free to move the later 10 back to join the starting 110 and use the ORDPATH prefix 11010 to mimic this encoding. We will get ORDPATHs of the same length as the corresponding UTF-8 encodings. However, we are giving up one nice property of UTF-8 codes, namely, that "Character boundaries are easily found from anywhere in an octet string." [18] This property ensures that if one byte of UTF-8 encoded data is corrupted, the rest of the data can be interpreted. This property is much more important in general use than within reliable storage systems. If this were deemed important, then UTF-8 itself should be used. We can view the "ORDPATH UTF-8" experimental results of the next section as standing for actual UTF-8 encodings using ORDPATH-like use of even and odd ordinals.

6 Comparative Experimental Evaluation

A potential drawback of hierarchical numbering schemes compared to other schemes proposed so far is the correlation of depth of a node within the tree and the label length. Therefore an important quality measure when comparing hierarchical numbering schemes is the maximal and average label length for a given document set.

document	max depth	avg depth	max fan-out	avg fan-out	90% fan-out	#nodes
Nasa	8	5,5	2435	2,8	3	530528
Cities	4	3,6	364	5,0	5	21028
Dictionary	8	3,2	163826	3,9	8	1545406
Novel	4	3,9	75	26,9	75	220
Pop. Places	3	2,9	164045	14,0	13	2952811
Religion	6	4,8	289	25,1	44	48259
Shakespeare	6	4,8	434	5,5	10	179689
Sigmod	6	5,4	89	3,7	4	15263
Treebank	36	7,9	56384	2,3	4	2437667
WFB	7	4,9	260	4,1	9	347868
Courses	5	4,0	2112	4,2	7	66735

Table 11. Depth, fan-out and number of nodes of document collection

We evaluated our schemes using the document set already introduced in [2]. For the readers convenience we repeat the document characteristics in Table 11. First we measured the label length for the whole document set for an initial labeling run like if the documents were inserted into a database. Afterwards we tested the increase of label length for random insertions of nodes with and without deletions. Finally we tested the behavior for insertion of a sequence of nodes before the first child and after the last child of a parent.

The encodings tested were

- DLN_DOM: DLN_{DOM} with pruning of 0-subvalues
  DLN_DOM 2|2|3|2|3: (subvalue lengths $s_0 = 2$, $s_1=2$, $s_2=3$, etc.)
  DLN_DOM 3|3|4|3
- DLN_Stream 3|3|4|3
- DLN_StreamOpt 3|3|4|3
- ORDPATH: Std: parameters as shown in Table 6 of Section 4.2.1
- ORDPATH DLN: parameters adjusted to mimic DLN_{Stream} 3|2|4|2|3 as discussed in Section 5.
- ORDPATH UTF8: parameters adjusted to mimic UTF8 encoding as discussed in Section 5.

The length of DLN subvalues were choosen based on evaluations on a range of documents with varying characteristics. For DLN_{DOM} the 2|2|3|2|3 and for DLN_{Stream} the 3|3|4|3 encoding turned out to be the best general settings. Since DLN_{DOM} utilizes the subvalue bit range more efficiently we can use smaller subvalues but encode a comparable number of siblings as DLN_{Stream} with the same number of subvalues. We additionally consider a DLN_{DOM} 3|3|4|3 version because DLN_{DOM} and DLN_{Stream} can only be used together within the same document if they use the same subvalue configuration. ORDPATH DLN was based on another configuration of DLN_{Stream} because here the subvalue separator bits are dropped (cf. section 5.2) implying a smaller penalty for using multiple subvalues.

6.1 Label length after initial labeling

These labeling schemes can be used for simple hierarchical numberings, that is, in labeling initial loads of XML data. Figure 3 shows the resulting maximum label lengths, Figure 4 shows the average label lengths.

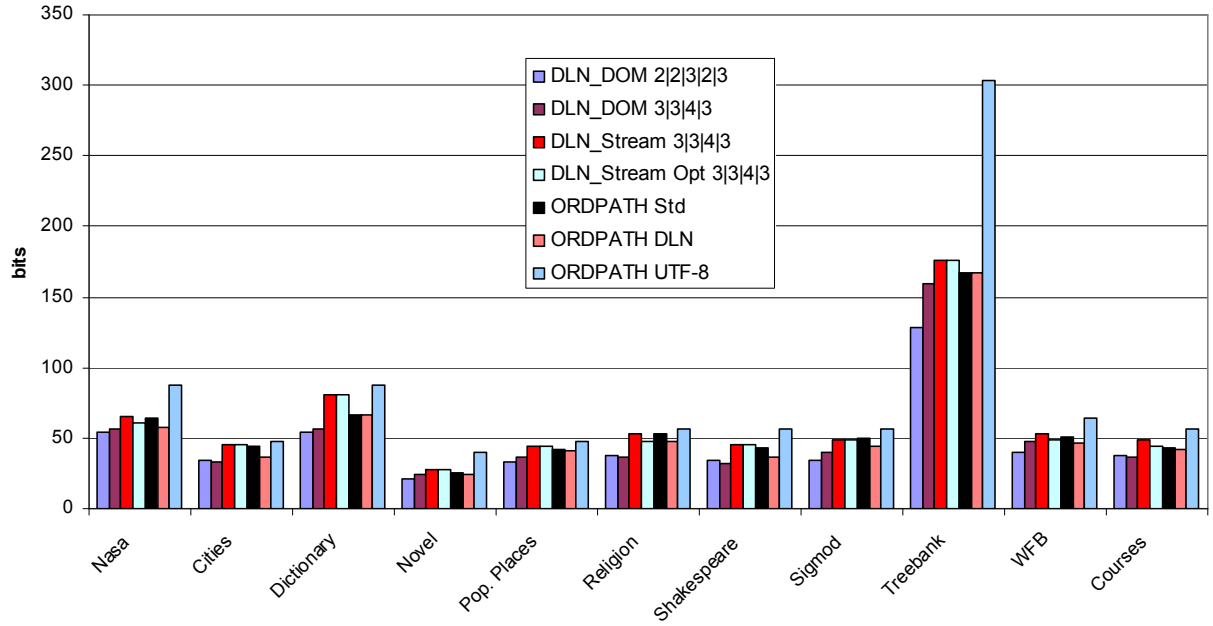

Figure 3. Maximum label length after initial labeling.

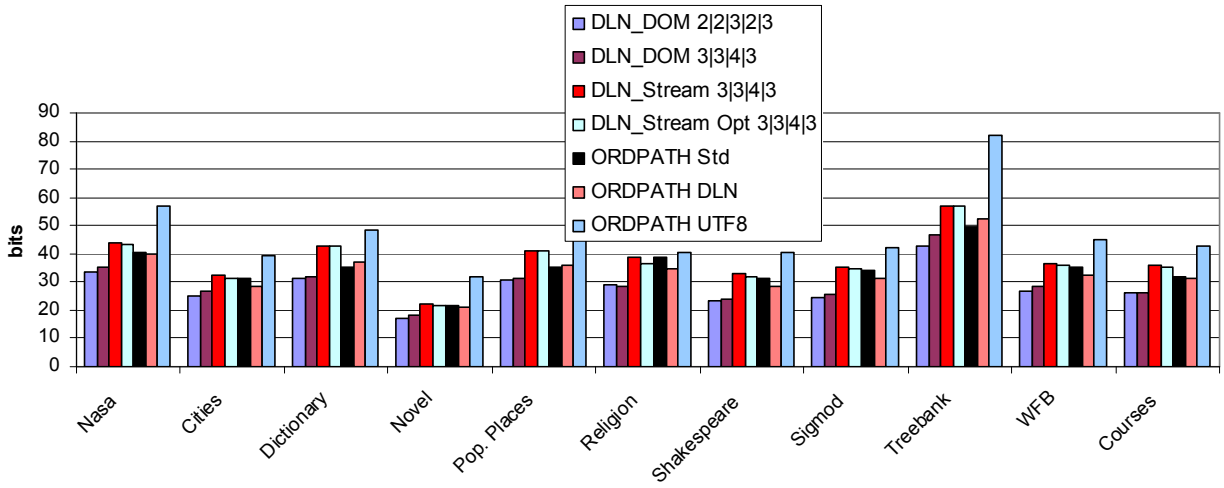

Figure 4. Average label length after initial labeling.

We see that if we ignore the wasteful encoding related to UTF-8, the XML document is more important than the encoding method in causing longer lengths, but even that effect is only moderate, except for the maximum lengths seen in the large and deepest Treebank document (maximum depth is 36, see Table 11). DLN_{DOM} saves up to 30% with a mean of 20% compared to ORDPATH Std, but requires more pre-knowledge. On the other hand DLN_{Stream} produces 3-10% longer labels than ORDPATH Std. The improvement of $DLN_{StreamOpt}$ compared to DLN_{Stream} is relatively small. The maximum label length is improved by 8-10% in four cases; the average label length by only 2%. UTF-8 is clearly outperformed by all encodings.

6.2 Label Length After Random Insertion of Nodes

We now turn to experiments inserting nodes into already-loaded XML data, in particular, the Cities[6] document of about 20,000 nodes. We randomly inserted up to 40 times that number to investigate the growth of label length

[6] One reason for choosing the Cities document for these operations is its homogeneous and relatively flat structure which is typical for a large class of XML documents. This makes it is easy to interpret the increase of label length and to control the

under insert traffic. For insertion a node is randomly chosen and with a probability of 0.9 it will be the adjacent left sibling of the node to be inserted[7]. With a probability of 0.1 the chosen node is the parent of the new node which is inserted with equal probability between, before or after the existing children.

See Figure 5 for data on the maximum lengths and Figure 6 for average lengths. The horizontal axis shows the expansion factor in number of nodes, logarithmically. The logarithm is appropriate because the addition of length opens up an exponential number of new labels.

All DLN encodings here use DLN_{Stream} for insertion. The DLN_DOM results only start from a document which was initially labeled with DLN_{DOM}. Since we only add 1 node at a time it is not sensible to use DLN_{DOM} for insertion here.

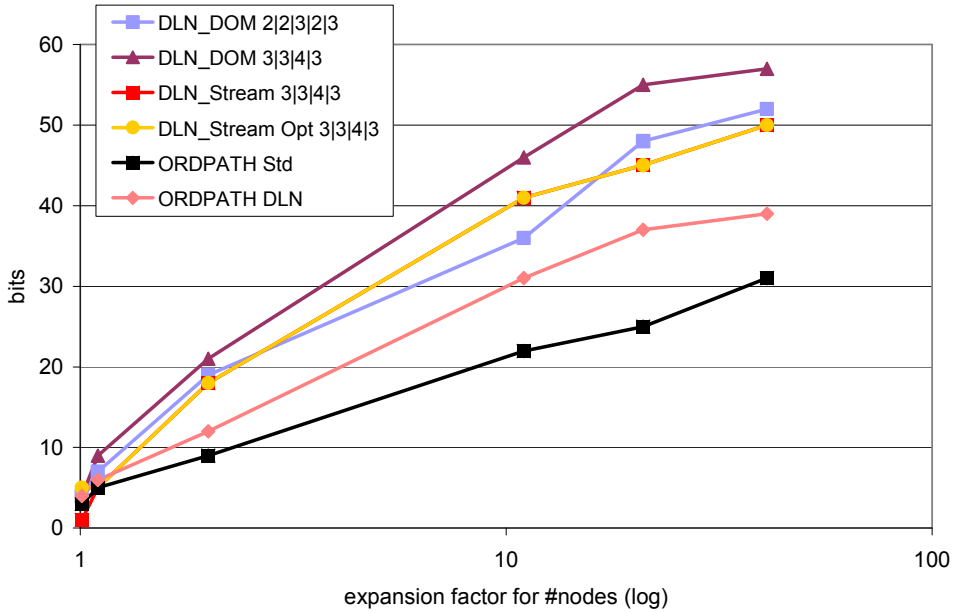

Figure 5. Increase in Maximum Label Length under Random inserts in Cities document.

document depth. Furthermore the size of the Cities document is well suited to evaluate small and very large insertions compared to original document size

[7] With a certain probability, the insertion is done before all siblings, so that all insert positions among siblings have equal probability.

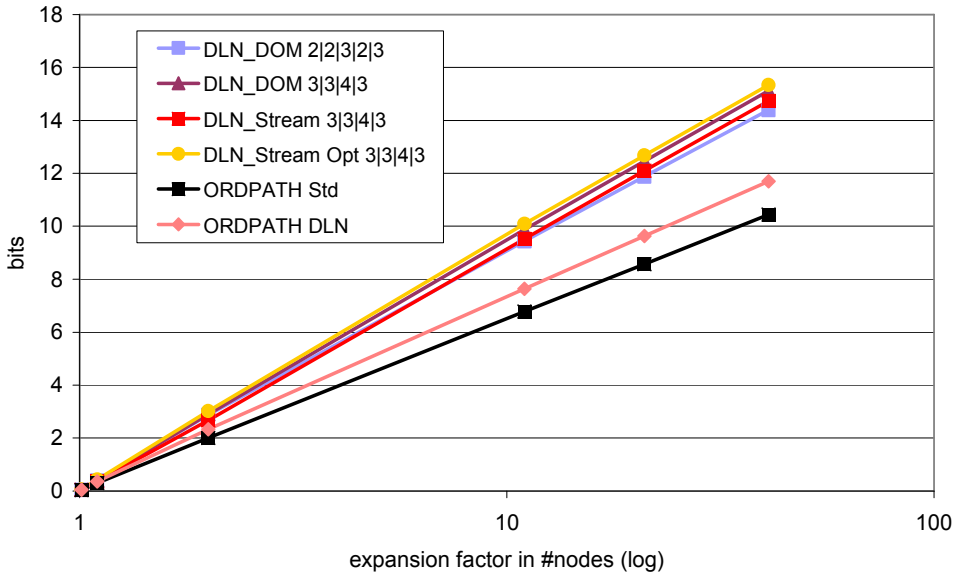

Figure 6. Increase in Average Label Length under Random inserts in Cities document.

This experiment shows that both ORDPATH and DLN are successful in managing the additional bits used for careting in so that the label length grows only logarithmically, not linearly. Naïve algorithms can easily "paint themselves into a corner" and fail in this regard. While ORDPATH has clearly a smaller increase in label length, both in maximum and average, it is starting from a longer length in some cases.

6.3 Label Length after inserting node sequences

In these experiments, sequences of 50-200 new nodes are inserted before first child or after last child. More precisely, a random node is selected and its set of siblings is added to at the beginning or end. Here DLN_{DOM} is used in the DLN_DOM cases.

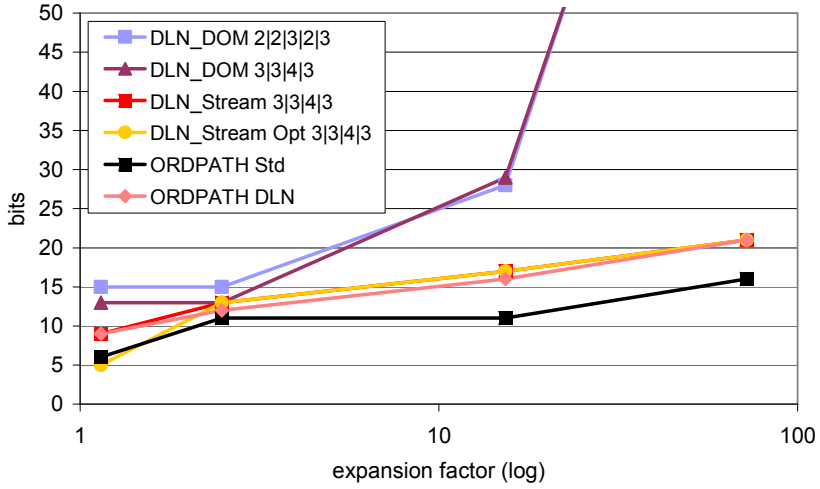

Figure 7. Increase in Maximum Label Length with inserted runs in Cities document

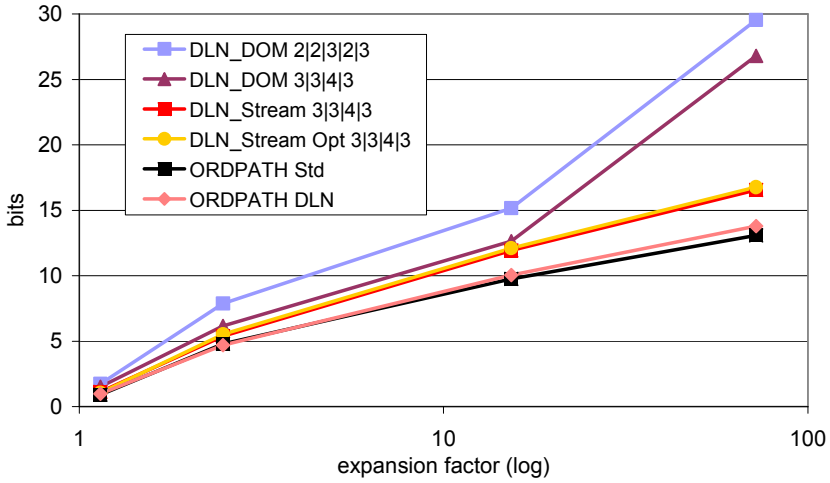

Figure 8. Increase in Average Label Length with inserted runs in Cities document

As in the random inserted nodes experiment, this experiment shows that DLN_{Stream} and ORDPATH both can add nodes at either end without running up beyond logarithmic growth in label length. DLN_DOM gets worse with a higher number of append operations because they were not considered in earlier append operations and therefore no bit ranges were reserved.

Parameter tuning

For documents with regular structure it can be advantageous to use a per-level encoding instead of a per-document encoding (different parameters for each level). For instance for SigmodRecord the DLN_{DOM} encoding has 15% shorter maximum length and 8% shorter average label length.

7 Conclusions

Hierarchical numbering has a number of positive properties making it an interesting candidate for labeling XML nodes. However without modifications it exhibits some disadvantages preventing its adoption. In this paper we have introduced and classified techniques used in different previous work from us that overcomes the drawbacks and enhances hierarchical numbering with document-order capability. Here the most valuable result is the prevention of renumbering of existing nodes after insertion of document fragments.

We have discussed possible variants arising from combinations of the proposed enhancements. Furthermore we compared the variants according to the main quality measure, which in the case of hierarchical numbering is the label length. We showed that both approaches from out earlier work in [2,13] give similar results. Both show logarithmic growth in label length with number of nodes, in cases related to random insertions and insertions of runs of nodes. The ORDPATH approach has advantages when a large number of nodes are inserted. The DLN variant DLN DOM benefits from document structure information known ahead of insert time. Without such pre-knowledge, ORDPATH and DLN are very similar over a wide range of conditions seen in practice. Although they both depend on prefix codes and can be mapped from one to the other for initial-load labels, they are somewhat different in detailed algorithms for label generation for later inserts.

References

1. Bauer, M. G.; Ramsak, F.; Bayer, R.: Multidimensional Mapping and Indexing of XML. In Proc. of German database conference BTW 2003, pp. 305-323, 2003
2. Böhme, T. ; Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In Proc. of CaiSE'04 Workshops, Volume 3 (DIWeb'04), pp. 70-81, 2004
3. Chien, S.; Tsotras, V. J. ; Zaniolo, C. ; Zhang, D. : Storing and Querying Multiversion XML Documents using Durable Node Numbers. In Proc. of the Intern. Conf. on WISE, Japan, pp. 270-279, 2001
4. Cohen, E.; Kaplan, H.; Milo, T.: Labeling Dynamic XML Trees. In Proc. of PODS 2002
5. Dietz, P. F.: Maintaining order in a linked list. In Proc. of the 14th Annual ACM Symposium on Theory of Computing, pp. 122-127, California, 1982

6. Florescu, D.; Kossmann, D.: Storing and Querying XML Data using an RDBMS. In IEEE Data Engineering Bulletin 22(3), 1999
7. Härder, T.; Haustein, M. P.; Mathis, C.; Wagner, M.: Node Labeling Schemes for Dynamic XML Documents Reconsidered. Appears in Data & Knowledge Engineering, Elsevier, 2006
8. Kha, D. D.; Yoshikawa, M.; Uemura, S.: A Structural Numbering Scheme for XML Data. In Chaudhri, A. B. et al. (Eds.): EDBT 2002 Workshops, LNCS 2490, pp. 91-108, Springer-Verlag, 2002
9. Kuckelberg, A.; Krieger, R.: Efficient Structure Oriented Storage of XML Documents Using ORDBMS. In Bressan, S. et al. (Eds.): EEXTT and DIWeb 2002, LNCS 2590, pp. 131-143, Springer-Verlag, 2003
10. Lee, Y. K.; Yoo, S.; Yoon, K.; Berra, P. B.: Index Structures for Structured Documents. Proc. of the 1st ACM International Conference on Digital Libraries, pp. 91-99, 1996
11. Li, Q.; Moon, B.: Indexing and Querying XML Data for Regular Path Expressions. In Proc. of the 27th VLDB Conf., Roma, Italy, 2001
12. Mignet, L.; Barbosa, D.; Veltri, P.: The XML Web: a First Study. In Proc. of the 12th Intern. WWW Conference, Budapest, 2003
13. O'Neil, E.; O'Neil, P.; Pal, S.; Cseri, I.; Schaller, G.; Westbury, N.: ORDPATHs: Insert-Friendly XML Node Labels. ACM SIGMOD Industrial Track, 2004
14. Schmidt, A.; Kersten, M. L.; Windhouwer, M.; Waas, F.: Efficient Relational Storage and Retrieval of XML Documents. In WebDB (Selected Papers) 2000, pp. 137-150, 2000
15. Shimura, T.; Yoshikawa, M.; Uemura, S.: Storage and Retrieval of XML Documents using Object-Relational Databases. In Proc. of the 10th Intern. Conf. on Database and Expert Systems Applications (DEXA'99), LNCS 1677, Springer-Verlag, pp. 206-217, 1999
16. Tatarinov, I.; Viglas, S. D.; Beyer, K.; Shanmugasundaram, J.; Shekita, E.; Zhang, C.: Storing and Querying Ordered XML Using a Relational Database System. Proc. of ACM SIGMOD, pp. 204-215, 2002
17. Yao, B. B.; Ozsu, M. T.; Khandelwal, N.: XBench Benchmark and Performance Testing of XML DBMSs. ICDE 2004: 621-633
18. Yergeau, F.: RFC 2279 - UTF-8, a transformation format of ISO 10646, Jan. 1998, available at http://www.faqs.org/rfcs/rfc3629.html