ELSEVIER
Available online at www.sciencedirect.com
SCIENCE DIRECT®
Data & Knowledge Engineering xxx (2005) xxx–xxx
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Node labeling schemes for dynamic XML documents reconsidered

Theo Härder *, Michael Haustein, Christian Mathis, Markus Wagner

Database and Information Systems, Department of Computer Science, University of Kaiserslautern, D-67663 Kaiserslautern, Germany

Received 30 October 2005; received in revised form 30 October 2005; accepted 30 November 2005

Abstract

We explore suitable node labeling schemes used in collaborative XML DBMSs (XDBMSs, for short) supporting typical XML document processing interfaces. Such schemes have to provide holistic support for essential XDBMS processing steps for declarative as well as navigational query processing and, with the same importance, lock management. In this paper, we evaluate existing range-based and prefix-based labeling schemes, before we propose our own scheme based on DeweyIDs. We experimentally explore its suitability as a general and immutable node labeling mechanism, stress its synergetic potential for query processing and locking, and show how it can be implemented efficiently. Various compression and optimization measures deliver surprising space reductions, frequently reduce the size of storage representation—compared to an already space-efficient encoding scheme—to less than 20–30% in the average and, thus, conclude their practical relevance.

Keywords: Tree node labeling; Dewey order; XML document storage; Huffman codes; Prefix compression

1. Introduction

As XML documents permeate information systems and databases with increasing pace, they are more and more used in a collaborative way. The challenge for database system development is to provide adequate and fine-grained management for these documents enabling efficient and concurrent read and write operations. In essence, this objective postulates the design and management of highly dynamic XML documents. Therefore, future XML DBMSs will be judged according to their ability to achieve high transaction parallelism. Currently, navigational and declarative languages are used to process XML documents. Because they are already available in the form of standards like SAX, DOM, XPath, or XQuery [25], and used as typical XML document processing (XDP) interfaces, XDBMSs should be able to run concurrent transactions supporting all these interfaces simultaneously and, at the same time, guarantee ACID properties [8] for all of them.

* Corresponding author. Tel.: +49 631 205 4030; fax: +49 631 205 3299.
E-mail address: haerder@informatik.uni-kl.de (T. Härder).
URL: http://wwwdvs.informatik.uni-kl.de (T. Härder).

0169-023X/$ - see front matter © 2005 Published by Elsevier B.V.
doi:10.1016/j.datak.2005.11.008

1.1. Desired properties of node labeling schemes

Any node labeling scheme used in XDBMSs as a prerequisite of fine-grained storage and management of XML documents must enable declarative and navigational operations equally well. Such multi-lingual XDP support explicitly means that—starting from a context node—navigational operations of DOM and SAX language models such as *parent*/*first-child*/*last-child*/*previous-sibling*/*next-sibling* must be facilitated and, at the same time, adequate support for processing steps in declarative queries along the 13 axes [26] of the XQuery and XPath 2.0 language model must be guaranteed. It is striking that the earliest proposals for node labeling schemes [5] exclusively focused on *parent*/*child* and *ancestor*/*descendant* support thereby assuming static XML documents. With the upcoming observation that large XML documents are likely to be used in collaborative applications requiring read and write access to them, the aspects of dynamic XML documents enabling arbitrary node insertions and deletions were considered in addition, while the (limited) query evaluation support was preserved by the enhanced dynamic labeling schemes.

As far as declarative query processing is concerned, we assume that the eight axes *parent*/*child*, *ancestor*/*descendant*, *previous-sibling*/*following-sibling*, *previous*/*following* are of particular importance. They are frequently exploited in XML query processing by decomposing complex queries into sequences of operations where such tailored axis operators are employed and chained together to derive the final result. For example, "axis1::name_test1/.../axisn::name_testn" is such an evaluation sequence. For each of these axis operators, a duplicate-free sequence of node labels (preferably in document order, i.e., corresponding to left-most depth-first traversal of the document) together with a name test is assumed as input; the axis operator then derives a duplicate-free sequence of node labels (in document order) where each of the qualified nodes satisfies the name test w.r.t. the specified axis. Note, query processing applied to sufficiently broad classes of XML query types requires the efficient support of all axis operators, potentially multiple times in a single query composed of *n* of such processing steps.

So far, the most important dynamic aspect—the behavior of a node labeling scheme under concurrency control requirements (efficient transactional isolation of readers and writers)—was completely neglected. On the other hand, fine-grained storage and management of XML documents is mandatory for efficient and scalable processing in multi-user XDBMSs. For this reason, we need a holistic view of a node labeling scheme adequate in such environments and have to evaluate its synergetic potential for query processing as well as concurrency control, before we can recommend it for general use.

Before the features of a labeling scheme can be used for query processing, appropriate support for node locking is needed. Fine-grained locking means that all ancestors have to be protected by suitable kinds of intention locks, before the context node—often an inner node of the document tree—can be locked by an R/U/X lock [8]. Because navigational and declarative query processing typically begins—by using an index—with a "jump" inside the tree, locking the entire ancestor chain is a very frequent internal operation. If the transaction decides after some navigation steps that some node has to be updated, access to the entire ancestor chain is again needed to perform appropriate lock conversions. Otherwise, if no index support and no adequate node labeling scheme is available, accessing a specific node would necessarily call for a scan of the entire document. Even in the case that the evaluation of processing steps are performed by exclusive use of index entries (without accessing the document nodes), suitable locks have to protect at least the respective index ranges to provide for repeatable results [20]. Furthermore, relabeling of nodes in transactional environments is not tolerable, because a user may keep node labels at the client-side application context for efficient direct access (using, for example, the DOM interface).

Note, although predicate locking of XQuery statements [26]—and, in the near future, XUpdate-like statements—would be powerful and elegant, its implementation rapidly leads to severe drawbacks such as undecidability problems and the need to acquire large lock granules for simplified predicates—a lesson learned from the (much simpler) relational world [14]. To provide for an acceptable solution, we necessarily have to map XQuery operations to a navigational access model to accomplish fine-granular concurrency control. Such an approach implicitly supports other XDP interfaces mentioned because their operations correspond more or less directly to a navigational access model.

Very important for efficient access to XML tree nodes is a labeling scheme which supports all the navigational operations as well as the evaluation of the main axes for declarative query processing. At the same time, the labeling scheme must facilitate the work of the lock manager. In particular, the set of labels used to identify nodes in a lock protocol must be *immutable* (for the life time of the nodes), must, when inserting new nodes, *preserve the document order*, and must easily reveal *the level* and *the IDs of all ancestor nodes*. Furthermore, the stored document must guarantee the *round-trip property*, that is, the XDBMS must be able to reconstruct the document in its original form. Last, but not least, the labels need a very efficient variable-length representation, because there are frequently millions of nodes in large XML documents (see Table 3).

1.2. Our contribution

We believe that very few of the existing approaches can fulfill the strong requirements outlined above. None of the schemes proposed so far has considered the needs of locking protocols. Furthermore, none has taken the support of navigation into account, which can be optimized together with the physical document mapping.

By surveying the existing node labeling schemes, we identify shortcomings which cannot cover all these features. In contrast, we show that all of these enhanced requirements can be satisfied by the expressiveness of DeweyIDs. To convince ourselves that DeweyIDs are a salient concept which is also implementable, we have developed a native XDBMS prototype called XTC (XML Transaction Coordinator, [9,11]) which exploits them for all tasks mentioned.

In this paper, Section 2 gives a characterization of the range-encoding and prefix-encoding labeling schemes. Our approach based on the idea of Dewey classification and lexicographic order is outlined in Section 3, where we show that all requirements listed in Section 1.1 are satisfied. In Section 4, we explore various methods to efficiently implement DeweyIDs. Section 5 describes extensive empirical experiments and checks various parameters of our DeweyID mapping. Finally, we give experimental results of various optimization approaches, before we summarize our study and wrap up with conclusions.

2. Range-based vs. prefix-based schemes

An XML document is usually represented by an ordered, labeled tree which is defined in the DOM standard [25]. Each node in the tree corresponds to an element, an attribute, or text data[1]; edges between the nodes represent element–subelement or element–attribute relationships. An XML database can be considered as a forest of such trees.

Node labeling was considered a challenging task and has attracted lots of researchers at an early stage [5]. Today, some of the proposals are of historical interest at best, because important requirements, such as support of dynamic schemes, later emerged. For example, bit-vector schemes where all labels have fixed size n and the storage space required for all labels in a document is exactly n^2, cannot cope with the characteristics of large XML documents. Furthermore, a scheme assuming perfectly balanced and static trees can provide extra functionality [15] when certain numbering conventions are observed. Based on a complete k-ary tree possibly filled with virtual nodes, it is easy to calculate from a given ID the ID of its parent, sibling, and (possibly virtual) child nodes. However, such an enforced regular structure comes with a high price: a huge number of IDs must be wasted to balance a highly skewed tree into a complete k-ary tree. If detailed advance knowledge about the XML document structure is available, the parameter k can be adjusted at each level. Hence, metadata consisting of a vector with tailored values k_i per level i [17] could lead to levelwise regular structures (see also Section 4.1). Despite such optimization efforts (levels with varying k), this idea had to be given up for practical applications (see XML document characteristics in Table 3).

Here we concentrate on competitive schemes supporting at least dynamic documents concerning the most important requirements mentioned above. As far as an appropriate scheme for XDBMSs is concerned, we have to observe the modification of such trees relevant for labeling and order preservation of the tree nodes. In a tree, arbitrary insertions and deletions can take place at any position, e.g., using DOM operations, typically resulting in the attachment of new or the removal of existing subtrees. Even renaming of existing nodes

[1] The labeling schemes could be extended to other node types such as namespace or comment in a straightforward way.

is possible [25]. All serious node labeling schemes proposed in the literature [1,4,7,22,27] can be classified into range-based and prefix-based node labeling schemes.

2.1. Range-based schemes

Traditional range-based schemes encode the position of the nodes in the tree by a 3-tuple (DocNo, LeftPos:RightPos, LevelNo). DocNo is the identifier of the document, which we ignore in the following (without any loss of generality), because we concentrate on the labeling of nodes in the same document. The pair of LeftPos (LP) and RightPos (RP) characterizes the range of numbers covered by a node and its subtree; it can be generated by performing a left-most depth-first traversal of the tree (indicated by the node numbers in Fig. 1) and sequentially assigning a monotonically increasing number at each visit of a node. LevelNo (lv) describes the nesting depth of the tree nodes starting with 0 at the root.

Range-based schemes [1,5] can easily determine some of the axis relationships: the *ancestor/descendent* relationship can be directly revealed by comparing the ranges of two nodes: a tree node n_1, $(LP_1{:}RP_1, lv_1)$, is *ancestor* of a tree node n_2, $(LP_2{:}RP_2, lv_2)$, iff $LP_1 < LP_2$ and $RP_1 > RP_2$. Node n_1 is *parent* (*child*) of node n_2, if $lv_1 = lv_2 - 1$ ($lv_1 = lv_2 + 1$) holds, in addition. Furthermore, a simple test is sufficient to determine the *following/preceding* relationship: node n_2 is a *following* (*preceding*) node of n_1, if $LP_2 > RP_1$ ($RP_2 < LP_1$). However, traditional schemes are not expressive enough to figure out the *following-sibling/preceding-sibling* relationship. To cure this shortcoming, an enhanced range-encoding scheme was proposed in [4]: a three-dimensional descriptor $(LP{:}RP, lv, P\_LP)$ additionally includes the parent node's left position (P_LP). With this information, the *following-sibling/preceding-sibling* relationship between two nodes can be concluded: n_1 is a *following-sibling* node of n_2, iff $LP_1 > LP_2$ and $P\_LP_1 = P\_LP_2$. Similarly, n_1 is a *preceding-sibling* node of n_2, iff $LP_1 < LP_2$ and $P\_LP_1 = P\_LP_2$.

How do we gain a range-encoding scheme whose node labels (descriptors) are immutable under arbitrary insertions? An obvious idea is to leave sufficiently large "gaps" in the numbering range upon initial number assignment [7]. For example, we could use instead of 1 an increment of 1000 thereby enabling later node insertions in the tree (see Fig. 1a). In this case, we could, for example, insert after the author node a second author with its subtree using the labeling gap 10,001 to 10,999. This measure would hold off on relabeling nodes in dynamic XML documents, but could not avoid them, e.g., in case of heavy point insertions in a subtree.

Much more serious is the question how we can efficiently figure out the identifiers (labels) of all ancestor nodes. A frequent situation is that an index allows jumps into the document and that a node, in this way, is accessed "out of the blue". If the document itself has to be accessed to determine all ancestors, a kind of backward scan in document order is needed thereby traversing the entire document in the worst case. Assume the node labels $(LP{:}RP, lv, P\_LP)$ stored together with the nodes are additionally organized—having LP as a kind of key—in an index pointing to the physical node locations of the document. Then, P_LP could be used to access the parent node entry in the index and, in turn, P_LP_{parent} to access the parent's parent in the index.

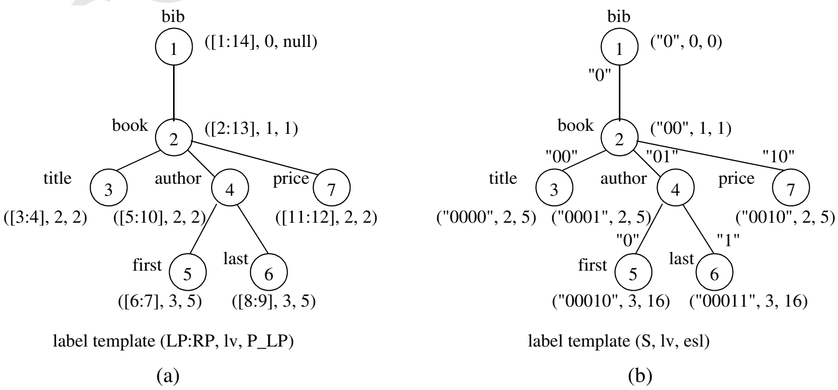

Fig. 1. Examples of enhanced tree node labeling schemes: (a) range-encoding scheme and (b) prefix-encoding scheme.

For deep trees, this still results in a very expensive procedure and seems to be a knock-out criterion for the use of range-encoded labels in XDBMSs. Other procedures such as described in [1] behave in a similar way, because they have to access conceivable ancestor nodes (possibly on disk) and verify their existence, while an ideal scheme should calculate them.

2.2. Prefix-based schemes

Prefix-based schemes directly encode the parent of a node as a prefix of its label using for instance a traversal in document order. The simplest algorithm is the Dewey Decimal Coding (DDC, [6]) frequently used to classify topics in libraries. As a matter of fact, DDC (see Section 3) in its original form introduces a fixed alphabet per level and therefore consumes more bits per node than actually required. This extra cost makes the representation of the DDC scheme easier, because it does not need special separators (in the stored format) to distinguish the tree levels characterized by the label [22]. Furthermore, it makes the scheme more expressive, because each node label can be considered as a kind of index for the entire ancestor path of the node. However, in deep trees these labels may grow very large such that they were considered not "implementable" in XDBMSs.

Traditional prefix-based schemes label each tree node with a unique string S such that (1) S_1 of node n_1 is before S_2 of node n_2 in lexicographic order, iff n_1 is before n_2 in the document order and (2) S_1 is a prefix of S_2, iff n_1 is an ancestor of n_2. As indicated in Fig. 1b, a set of fixed-length (and, therefore, prefix-free) binary strings (edge codes) is assigned—from left to right in lexicographic order—to the outgoing edges of each node. Then we build the string S_c of a node by concatenating the string of the parent node with the string assigned to its incoming edge. A level indicator lv is kept together with this string. If we keep an indicator sl describing the string length of the incoming edge, then we can easily determine the string S_p of the parent node. Given two nodes, such a scheme (S, lv, sl) essentially allows the determination of the eight axis relationships as discussed in Section 2.1.

An enhancement of this traditional prefix-encoding scheme is introduced in [4]. Instead of keeping sl in the three-dimensional node descriptor, the so-called edge string length (esl) is stored in each node label. The values of this parameter are calculated in a complex procedure taking the lengths of all edge codes of all levels in the path to the root into account (see Appendix A). As a consequence, esl can be used to extract the label strings of a node's ancestors.[2] Hence, this opportunity enables us—given the label (S, lv, esl) of any node—to derive the strings (identifiers) of each ancestor node without accessing the document. Note, the lengths of all edge codes are tailored to the fan-out in the individual path to be encoded. The fact that the lengths of these initially assigned edge codes remain constant, is a cornerstone of the stability of this enhanced prefix-encoding scheme. Unfortunately, initial and optimal assignment of these edge codes is susceptible to node insertions and, in turn, to the need to relabel entire paths. Reservation of gaps by providing (overly) long edge codes at each level may dramatically increase the length of S and may not tolerate all insertions, because the document order of the nodes, which is encoded in S, has to be preserved. Hence, the insertion of a second author and its subtree after the author node in Fig. 1b could not use the free edge code "11". Therefore, a reassignment of the edge code "10" or a new assignment of 3-bit edge codes at level 2 to all outgoing edges of book (leaving some room for future insertions) would require the relabeling of some parts in the document which involves complex computations in the entire subtree.

Variations of prefix-free edge codes were explored in [7]. The children of a node, starting from the left, have edge codes "0", "10", "110", etc. with the ith child having edge code $s(i) =$ "$111^{i-1}0$". Hence, the "0" is used as a kind of separator which allows to determine the relative position of a child in the set of siblings and its depth in the tree, when the entire string concatenating the path from the root is checked. For trees with restricted depth, Ref. [7] proposed a more suitable labeling scheme. Again for the edge codes of a sibling set, $s(i)$ for the ith child is defined such that $s(1), s(2), s(3), \ldots = 0, 10, 1100, 1101, 1110, 11110000, \ldots$ This edge code increments the binary number represented by $s(i)$ to obtain $s(i+1)$. If the representation of $s(i)+1$

[2] For typical value distributions in real applications, this approach is impractical, because the representation of an esl would require up to 50 decimal digits (or 167 bits) for deep trees which happen to also exhibit a wide fan-out in some location.

consists of all ones, the code doubles its length by attaching a sequence of zeros. Both schemes for assigning edge codes are not length-restricted, but very expensive in terms of space consumption, if the set of siblings is very large. But the decisive reason for their disqualification is that these codes fail to support order-sensitive insertions.

A variant of a prefix-based labeling scheme is the so-called prime-number labeling scheme [24]. In the top-down prime-number labeling scheme,[3] unique, so-called self-label primes are first assigned to each node. Then the labeling algorithm starts from the root node and assigns the multiplication result of all self-label primes in the ancestor path as a label to each node. Node insertions require the preservation of the document order, which is maintained by order numbers, calculated according to the so-called simultaneous congruence. This may cause the recalculation of the order information for all successor nodes. Ancestor determination can be achieved by a kind of number factorization applied to the label of the context node to gain the individual node numbers along the ancestor path. Although this may be an elegant idea for textbook trees, it is totally inappropriate in real situations.[4] As further optimization, the node's self-label prime can be stored together with the node label such that the parent node label can be computed by a division. Hence, this improvement to derive the parent node label is similar to the *P_LP* idea in [4]; however, it needs in the same way additional index accesses to derive the entire ancestor chain which makes the procedure too slow.

3. Prefix-based schemes reconsidered

An important advantage of prefix-based labeling schemes is their capacity to adjust arbitrary updates in documents. If labels can be of variable size, there is no limitation of the tree growth in breadth and depth. Insertions are particularly simple as long as ordering among descendants is not critical: then new child nodes can be added to the right side of existing nodes without having to relabel them. Hence, labels of variable size enable insertions, while, on the other hand, this variability opens new opportunities of compression. Therefore, we must preserve these benefits while we enhance such schemes to match the new requirements.

3.1. ORDPATH concept

ORDPATH is a hierarchical labeling scheme which implements a prefix-based scheme (similar to the representation in Fig. 2). It is called "insert-friendly XML node labeling" and was first explored in [21]. For example, an ORDPATH label is 1.5.3.3.9, which consists of five so-called divisions (components) separated by dots (in the human readable format). The root node of the document is always labeled by ORDPATH value 1 and consists of only a single division. The children obtain the ORDPATH value of their parent and attach another division whose value increases from left to right. Every division is represented by an ordinal for which a variable-length bit encoding is provided. During the initial load of the tree, only *positive*, *odd* integers are assigned as division values. Counting odd division values of an ORDPATH label is used to determine the level (depth) of the labeled node. Because an ordinal is (theoretically) not restricted in its length, it is obvious that rightmost insertions of new child nodes can be performed at any position of the tree. Leftmost insertions are handled in a similar way by extending the labeling range using negative ordinals. If no labeling space is available when inserting a new node between two existing children, a "careting-in" technique is applied. The label is generated using additional intermediate careting divisions having *even* values. These careting divisions do not count as divisions that increase the encoded depth of the node in the tree. In all cases, relabeling of nodes is avoided, though substantial storage space may by consumed by the carets.

A variation of the DeweyID concept called Dynamic Level Numbering Scheme (DLN) was proposed in [3]. The basic DLN scheme takes advantage of advance knowledge of the document structure and is discussed in Section 4.3. While it supports right-hand insertions after existing siblings, left-hand insertions may quickly

[3] An analogous, but less suitable scheme uses bottom-up prime-number node labeling.

[4] Note, we have experimented with trees having a depth of 37 and a maximal fan-out of several millions. Assume, we would have solved the problem of self-label assignment of unique primes and of number representation. But number factorization during query processing remains a nightmare. Such an approach is definitely impractical.

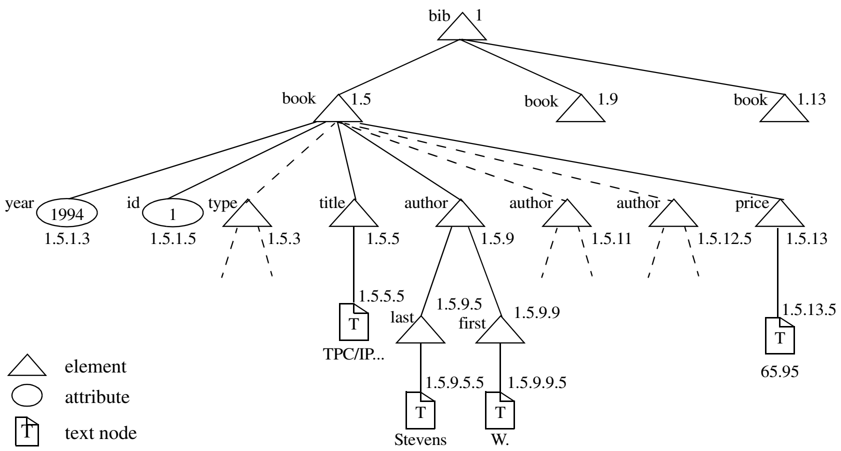

Fig. 2. A sample DOM tree labeled with DeweyIDs.

lead to inflated labels. Compared to ORDPATH, they would need shorter reorganization intervals in case of unfavorable insertion orders.

3.2. Mapping of DeweyIDs to DOM trees

DeweyIDs implement a prefix-based scheme for the labeling of DOM trees, which is also based on the concept of Dewey order characterized by Fig. 2. Conceptually similar to the ORDPATH scheme, our DeweyID scheme refines the Dewey order mapping,[5] provides for gaps in the labeling space, and introduces a kind of overflow mechanism when gaps for new insertions are in short supply. To allow for later node insertions, we introduce a parameter distance, which determines the gap initially left free in the labeling space between neighbor nodes at a given level. Only *odd* division values also used for level identification are assigned during initial document loading and as long as a gap in the labeling space is big enough for inserting a new node. In contrast, *even* division values play a special role as kind of overflow indicator. In Fig. 2, we have chosen a distance value of 4. When assigning at a given level a division to the first child, we always start with *distance + 1*, because division value 1 is reserved for attribute maintenance. When all nodes of the document are loaded—typically bulk-loaded in document order—, their labeling is guided by the following rules:

- Element root node: It always obtains DeweyID 1.
- Element and text nodes: The first node at a level receives the DeweyID of its parent node extended by a division of *distance + 1*. If a node *N* is inserted after the last node *L* at a given level, DeweyID of *L* is assigned to *N* where the value of the last division is increased by *distance*.
- Attribute nodes: All attribute nodes of *N* obtain the DeweyID of *N* extended by a division with value 1 indicating the type "attribute" and another division labeling the attribute and its value. If it is the first attribute node of *N*, this division has the value 3. Otherwise, the division receives the division value of the last attribute node of *N* increased by 2. In this case, the distance value does not matter, because the attribute sequence does not affect the semantics of the document. Therefore, new attributes can always be inserted at the end of the attribute list.

After initial loading of our sample document in Fig. 2, we have inserted the nodes of the second author and then of the third author. For $author_2$, we could assign an odd division value 11 resulting in DeweyID 1.5.11. To keep the gap open for arbitrary many insertions, we cannot use 12 as a regular division value for $author_3$.

[5] The memory representation is based on so-called taDOM trees which virtually provide two new node types "attribute root" and "string" which are exclusively used by the XTC lock manager to enhance concurrency [11]. These extensions are neither stored on external storage nor visible at the XML APIs.

Instead, we indicate by an even value for a division that some overflow has happened. Furthermore, we indicate by an additional division the position of the new node by an odd value using the same distance value. Hence, *author*$_3$ receives DeweyID 1.5.12.5 which preserves the document order and allows the correct level identification by counting the odd division values.

Assignment of a DeweyID for a *new last sibling* is similar to the initial loading, if the last level only consists of a single division. Hence, when inserting element node year after price (with DeweyID 1.5.13), addition of the distance value yields 1.5.17. In case, the last level consists of more than one division (due to earlier insertions and deletions), the first division of this level is increased by *distance* − *1* to obtain an odd value, i.e., the successor of 1.5.14.6.5 is 1.5.17.

If a sibling is inserted *before the first existing sibling*, the first division of the last level is halved and, if necessary, ceiled to the next integer or increased by 1 to get an odd division. This measure secures that the "before-and-after gaps" for new nodes remain equal. Hence, inserting a *type* node before *title* would result in DeweyID 1.5.3. In case the first division of the last level is 3, it will be replaced by *2.distance* + *1*, when the next predecessor is inserted, e.g., 1.5.2.5. If the first divisions of the last level are already 2, they have to be adopted unchanged, because smaller division values than 2 are not possible, e.g., the predecessors of 1.5.2.5 are 1.5.2.3, 1.5.2.2.5, 1.5.2.2.3, 1.5.2.2.2.5, and so on.

The remaining case is the insertion of node d_2 *between two existing nodes* d_1 and d_3. Hence, for d_2 we must find a new DeweyID with $d_1 < d_2 < d_3$. Because they are allocated at the same level and have the same parent node, they only differ at the last level (which may consist of arbitrary many even divisions and one odd division, in case a weird insertion history took place at that position in the tree). All common divisions before the first differing division are also equal for the new DeweyID. The first differing division determines the division becoming part of DeweyID for d_2. If possible, we prefer a median division to keep the before-and-after gaps equal. Assume for example, $d_1 = 1.9.5.7.5$ and $d_3 = 1.9.5.7.16.5$, for which the first differing divisions are 5 and 16. Hence, choosing the median odd division results in $d_2 = 1.9.5.7.11$. As another example, if $d_4 = 1.5.6.7.5$ and $d_6 = 1.5.6.7.7$, only even division 6 would fit to satisfy $d_4 < d_5 < d_6$. Remember, we have to recognize the correct level. Hence, with distance value 4, $d_5 = 1.5.6.7.6.5$. The reader is encouraged to construct DeweyIDs for further weird cases.

3.3. Fine-grained access to XML documents

Fast (indexed) access to each node is provided by variants of B*-trees tailored to our requirements of node identification and direct or relative location of any node. Fig. 3a illustrates the physical document structure—consisting of *document index* and *document container* as a set of chained pages—sketching the sample XML document of Fig. 2, which is stored in document order; the key-value pairs within the document index are referencing the first DeweyID stored in each container page. Using a vocabulary, we can compress the actual storage representation of a node and increase the storage utilization on disk. In addition to the storage structure of the actual document, an *element index* is created consisting of a *name directory* with (potentially) all element names occurring in the XML document (Fig. 3b); this name directory often fits into a single page. For each specific element name, in turn, a *node reference index* may be maintained which addresses the corresponding elements using their DeweyIDs. In all cases, variable-length key support is mandatory; additional functionality for prefix compression of DeweyIDs is very effective. Because of reference locality in the B*-trees

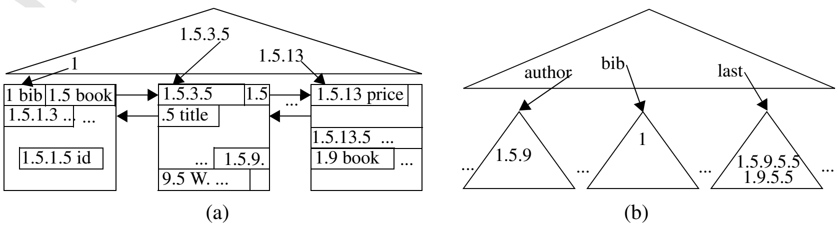

Fig. 3. Document storage using B*-trees: (a) Physical document structure and (b) element index.

while processing XML documents, most of the referenced tree pages (at least the ones of the upper tree layers) are expected to reside in DB buffers—thus reducing external accesses to a minimum.

3.4. Holistic system support of DeweyIDs

Existing DeweyIDs are immutable, that is, they allow the assignment of new IDs without the need to reorganize the IDs of nodes present. A relabeling after weird insertion histories[6] can be preplanned; it is only required, when implementation restrictions are violated, e.g., the max-key length in B*-trees. Comparison of two DeweyIDs allows ordering of the respective nodes in document order. As opposed to competing schemes, DeweyIDs (and ORDPATHs) easily provide the IDs of all ancestors to enable intention locking of all nodes in the path up to the document root without any access to the document itself [10]. For example, the ancestor IDs of 1.5.12.5.2.2.5.9 are 1.5.12.5.2.2.5, 1.5.12.5, 1.5, and 1.

Declarative queries are supported by the efficient evaluation of the eight axes frequently occurring in XPath or XQuery path expressions:

- *parent/child*: Checking whether node d_1 is parent of d_2 only requires a check whether DeweyID of d_1 is a prefix of DeweyID of d_2 and $\text{level}(d_1) = \text{level}(d_2) - 1$ and vice versa.
- *ancestor/descendant*: Checking whether node d_1 is an ancestor of d_2 only requires to check whether DeweyID of d_1 is a prefix of DeweyID of d_2 and vice versa.
- *following-sibling/preceding-sibling*: An element or text node[7] d_1 is a following-sibling of d_2 if $\text{parent}(d_1) = \text{parent}(d_2)$ and $d_1 > d_2$. Similarly, d_1 is a preceding-sibling of d_2 if $\text{parent}(d_1) = \text{parent}(d_2)$ and $d_1 < d_2$.
- *following/preceding*: Node d_1 is in *following* relationship to d_2 if $d_1 > d_2$ and d_1 is not a descendant of d_2, whereas d_1 is in *preceding* relationship to d_2 if $d_1 < d_2$ and d_1 is not an ancestor of d_2.

Using the document index sketched in Fig. 3, the five basic navigational axes *parent*, *previous-sibling*, *following-sibling*, *first-child*, and *last-child*, as specified in DOM [25], may be efficiently evaluated—in the best case, they reside in the page of the given context node *cn*. When accessing the previous sibling *ps* of *cn*, e.g., node 1.9 in Fig. 3, an obvious strategy would be to locate the page of 1.9 requiring a traversal of the document index from the root page to the leaf page where 1.9 is stored. This page is often already present in main memory because of reference locality. From the context node, we check all IDs backwards, following the links between the leaf pages of the index, until we find *ps*—the first ID with the same parent as *cn* and the same level. All IDs skipped along this way were descendants of *ps*. Therefore, the number of pages to be accessed depends on the size of the subtree having *ps* as root. An alternative strategy avoids this unwanted dependency: After the page containing 1.9 is loaded, we inspect the ID *d* of the directly preceding node of 1.9, which is 1.5.13.5. If *ps* exists, *d* must be a descendant of *ps*. With the level information of *cn*, we can infer the ID of *ps*: 1.5. Now a direct access to 1.5 suffices to locate the result. The second strategy ensures independence from the document structure, i.e., the number of descendants between *ps* and *cn* does not matter anymore. Similar search algorithms for the remaining four axes can be found. The *parent* axis, as well as *first-child* and *next-sibling* can be retrieved directly, requiring only a single document index traversal. The *last-child* axis works similar to the *previous-sibling* axis and, therefore, needs two index traversals in the worst case.

Despite of these really useful properties for holistic processing support, it is often claimed that DeweyIDs are not "implementable" because of their size which is primarily influenced by the document depth, the node fan-out, and the distance parameter. High distance values reduce the probability of overflows. Their selection has to be balanced against increased storage space for the representation of DeweyIDs. Nevertheless, DeweyIDs may become quite long, especially in trees with large max-depth values. Therefore, serious efforts are needed to develop a practical solution for them.

[6] For example, point insertions of thousands of nodes between two existing nodes may produce large DeweyIDs. Especially insertions before the currently inserted node may enforce increased use of even division values thereby extending the total length of a DeweyID.

[7] Attribute nodes have no siblings.

4. Efficient encoding of DeweyIDs

Due to the large variance of XML documents in number of levels and, even more, number of elements per level, we cannot design a (big enough) fixed-length storage scheme of DeweyIDs; such a scheme would mean fixed for individual divisions and fixed for the number of maximum allowed repetitions per level. Even if the first sibling at a level has a small division value *distance*, the bulk-loaded millionth sibling would have a value of $10^6 \cdot distance$ (e.g., requiring the representation of $\approx 4 \times 10^6$ as an individual division value using the example in Fig. 2). On the other hand, we have more smaller division values—assigned to the "first" children of a node—than larger ones constructed for children inserted later. Of course, there are definitely more "first" children. Therefore, we urgently need adaptivity for our storage scheme.

For the sake of space economy and flexibility, the storage scheme must be dynamic, variable, and effective in each aspect and, at the same time, it must be very efficient in storage usage, encoding/decoding, and value comparison. As far as space is concerned, we need a bit-level encoding scheme, which achieves very efficient representation of small values, while it is reasonably space-efficient for very large values. To allow for marginal optimization, we assume that field length = 0 and division value = 0 do not occur in our encoding units such that we can use "0" to improve the encoding. The critical question is how can we provide for such a scheme? Therefore, we will explore the solution space for efficient encodings of division values in the following.

We discuss the encoding of division values O_i at the bit level to allow for the minimum storage space possible. To enable integration into a system context, e.g., the use of DeweyIDs consisting of a variable number of O_i's as keys or references in B*-trees, they have to be aligned to and compared with each other using variable-length byte structures, that is, a field of typically 1-byte length prefixes the DeweyID encoding. Here we concentrate on the storage consumption of single division values.

4.1. Static schemes using advance knowledge

Advance knowledge of the maximum number of siblings per document level (*msl*) in static documents leads to the most simple encoding scheme using per level a fixed encoding unit. In this restricted case, the length information El_i per level i can be factored out and kept as metadata. Encoding length $El_i = \text{ceil}(\log_2 msl(i))$ corresponds to the storage space needed for a division representation. If we assume that division value 0 does not occur and we have encountered $msl(0) = 1$, $msl(1) = 8$, and $msl(2) = 35$ in an analysis phase before storing the document, we yield $El_0 = 1$, $El_1 = 3$, and $El_2 = 6$. Hence, DeweyID 1.7.11 is encoded by the concatenation of the three codes for the individual division values resulting in 0 110 001010. Obviously, such encodings at the division level and, in turn, at the DeweyID level are order preserving. Direct bit-level (or byte-level) comparison of a shorter encoding E_1 with the corresponding prefix P_2 of a longer encoding E_2 always delivers correct results. In case $E_1 = P_2$, then E_2 is larger than E_1. Otherwise, the comparison result of E_1 and P_2 decides.

Despite of the strong assumptions, this encoding scheme results in minimal storage usage only, if the document tree is well balanced, that is, (nearly) $msl(i)$ siblings occur under each node of level l_{i-1}. Note, if the document tree is skewed, e.g., $msl(i)$ siblings only occur under a single node of level l_{i-1}, other encoding schemes may deliver better results. Nevertheless, this scheme represents the theoretical minimum if *msl* division values can appear in a given division at a given level (see Appendix B, where we have contrasted these results with a single fixed (maximal) encoding unit per document).

4.2. Use of length fields

If advance knowledge is not available or the number of siblings at a given level is strongly varying (possibly after later insertions), storing of division values O_i can take advantage of variable-length representations. This could be achieved in the simplest case by attaching a fixed-length field L_f representing the actual length of O_i. If we always choose minimal L_f values, we can exploit a kind of *range expansion* by assigning codes to O_i according to the following pattern sketched for $l_f = 2$:

$$00\ 0 \equiv 1, \quad 00\ 1 \equiv 2, \quad 01\ 00 \equiv 3, \quad \ldots,$$
$$01\ 11 \equiv 6, \quad 10\ 000 \equiv 7, \quad 10\ 001 \equiv 8, \quad \ldots$$

Using this schemes provides order-preserving codes for division values. However, what is an adequate length value l_f for L_f? Because $L_f \leqslant 2^{l_f}$, each division value is limited by

$$O_i \leqslant \sum_{j=1}^{L_f} 2^j = 2^{L_f+1} - 2.$$

Most division values are expected to be rather small (<100), but some of them could reach >10^9. While for the former example value $L_f = 7$ and $l_f = 3$ would be sufficient, the latter would require $L_f \geqslant 30$ and $l_f \geqslant 5$. Furthermore, whatever reasonable value for l_f is chosen, it is not space optimal and additionally introduces an implementation restriction. For this reason, we should make the length indicator itself of variable length, especially to improve the encoding of small values.

A first approach makes the length indicator variable without storing explicit length information for it. The variable length L_{O_i} can be coded in a way that it can grow in a stepwise manner without any limit [13]. For code unit $u = 3$ bit, we assign length codes for L_{O_i} in the following way: $000 \equiv 1$, $001 \equiv 2$, ..., $110 \equiv 7$, $111\ 000 \equiv 8$, $111\ 001 \equiv 9$, etc. Obviously, the n code units of u bits needed to represent L_{O_i} can be determined such that following inequality holds for $n = 1, 2, \ldots$:

$$(n-1) \cdot (2^u - 1) = L_{n-1} < L_{O_i} \leqslant L_n = n \cdot (2^u - 1).$$

To guarantee minimal space consumption, we again exploit range expansion for the representation of division values O_i. Using the following inequality:

$$\sum_{j=1}^{L_{n-1}} 2^j = 2^{L_{n-1}+1} - 2 < O_i \leqslant \sum_{j=1}^{L_{O_i}} 2^j \leqslant \sum_{j=1}^{L_n} 2^j = 2^{L_n+1} - 2,$$

we can calculate for a given O_i the required length and, in turn, the entire division length $El_i = n \cdot u + L_{O_i}$. Our evaluation in Appendix B however reveals that we pay the reduction of space overhead for small values with a significant increase for large values.

Therefore, we developed a second approach to make the length information variable. The idea is to spend a fixed-length field LL_f of length ll_f to describe the actual length of L_{v_i} resulting in an entry $LL_f|L_{v_i}|O_i$. Length ll_f of LL_f allows the representation of values n between

$$1 \leqslant n \leqslant 2^{ll_f},$$

which can be used to code values for L_{v_i} which, in turn, determine the length of the O_i representation. Both, for L_{v_i} and O_i, it is advisable to apply range expansion which guarantees for minimal code length and correct division comparison. Using $ll_f = 2$, this double range expansion works as follows:

$$\begin{array}{llllll}
00\ 0\ 0 & \equiv 1, & & 00\ 0\ 1 & \equiv 2, \\
00\ 1\ 00 & \equiv 3, & \ldots, & 00\ 1\ 11 & \equiv 6, \\
01\ 00\ 000 & \equiv 7, & \ldots, & 01\ 00\ 111 & \equiv 14, \\
01\ 01\ 0000 & \equiv 15, & \ldots, & 01\ 01\ 1111 & \equiv 30, \\
01\ 10\ 00000 & \equiv 31, & \ldots, & 01\ 10\ 11111 & \equiv 62, \\
01\ 11\ \ldots & & & &
\end{array}$$

To determine the code for a given O_i, we need to find the smallest L_{O_i} such that the following inequality holds:

$$\sum_{k=1}^{L_n} 2^k = 2^{L_n+1} - 2 < O_i \leqslant \sum_{k=1}^{L_{O_i}} 2^k \leqslant \sum_{k=1}^{L_{n+1}} 2^k = 2^{L_{n+1}+1} - 2.$$

Then the smallest n satisfying the inequality

$$2^n - 2 = L_n < L_{O_i} \leqslant L_{n+1} = 2^{n+1} - 2,$$

allows to compute the space needed for a division value O_i by $El_i = ll_f + n + L_{O_i}$.

Table 1
Assigning codes for streaming DLNs

m	Codes for $k = 4$	Value range of O_i
1	0xxx	0–7
2	10xx 1 xxxx	8–71
3	110x 1 xxxx 1 xxxx	72–583
4	1110 1 xxxx 1 xxxx 1 xxxx	584–4679
...	...	...

The fixed length field with $ll_f = 2$ seems to be large enough for most practical cases, because it only exhausts for values of $O_i \geqslant 2^{31} - 1$. Otherwise, $ll_f = 3$ (allowing values of $O_i \leqslant 2^{511} - 2$) has to be chosen. However, both schemes discussed so far carry a penalty for the frequent divisions with small values (see Appendix B). Furthermore for the practical value range considered, $ll_f \geqslant 3$ reserves useless extra bits for length information. In summary, all methods based on length fields are less suitable candidates for division encoding.

4.3. Use of control tokens

The use of control tokens is based on positions which appear in equidistant, potentially level-specific intervals and can, therefore, be determined by some kind of metadata. Based on their use, the simple scheme discussed in Section 4.1 can be extended for the basic DLN [3] to support insertions which may lead to several encoding units per level. For this reason, control tokens in the form of single bits are applied whose metadata are collected before storing the document. Control bit "0" indicates a level transition, while divisions at the same level stemming from later node insertions, e.g., for node 1.7/1, are marked by "1". The same *msl* numbers as in Section 4.1 lead for DLN 1.7.11 to the encoding 0 0 110 0 001010. The reason for the positional use of "0" and "1" becomes clear when we encode DLN 1.7/1.1. The node 1.7/1 inserted as sibling after node 1.7 results in a DLN 0 0 110 1 000 0 000000. Some indicative values for the DLN space consumption are listed in Appendix B. Note that we assume that *msl* is known such that each division can be represented by a single encoding unit. If skewed sets of disjoint siblings occur at a level, these values may be misleading and far from being optimal. On the other hand, the DLN scheme has to use the fixed encoding units per level. For this reason, they should not be used for cross-comparisons.

The scheme discussed for basic DLN becomes far from optimal if large division values x have to be used for nodes where new nodes are inserted relative to them and are labeled with small division values (e.g., x/1, x/1/1, x/1/2, etc.). Therefore, it is advisable to build division values using smaller encoding units and an expansion mechanism. Such a mechanism is also required for a DLN scheme applied to streamed data (where the *msl* values are not known), because there is no guarantee that a single encoding unit of length k can express all division values present. To reveal the intricacies of encoding approaches based on control token use, we analyze a method proposed in [3] by referring to our running example 1.7.11. In addition to a level separator ("0"), a sequence of m encoding units ($m \geqslant 1$), whose construction is illustrated by Table 1,[8] is needed to express a division value O_i. The first bit occurring with value 0 is used to distinguish the length information (in terms of encoding units of length k) from the encoding of O_i. Seemingly redundant control bits ("1") are used to glue the encoding units of O_i together (so-called glue bits). Obviously, we can drop these control bits without loosing any information for O_i. We can even (repeatedly) attach a new division value O'_i (stemming from an overflow—a later inserted node), because the position after O_i can be computed. A "1" in this position would indicate the existence of an O'_i, whereas a "0" would switch to the next level. Hence, we can attach several division values without changing the level.

However, the subtlety of this encoding enters the stage when it comes to the comparison of two DLNs where overflow division values participate. Assume DLNs 1.7.11 and 1.7/1.1 encoded by means of Table 1 and $k = 4$. Then we yield

[8] Division value 0 enables node insertions before a node labeled by division value 1.

0000 0 0110 0 1000 1 1010
0000 0 0110 1 0000 0 0000

Due to the control token use we achieve the correct comparison result 1.7.11 < 1.7/1.1. If we would drop the control tokens in the chosen encoding of O_i, we could not guarantee anymore that we compare control tokens with each other. This, however, may be a prerequisite to obtain correct comparison results when the division encoding is composed of multiple encoding units [2].

For the analysis of this encoding scheme, we define the following relationships:

$$\sum_{j=1}^{m-1} 2^{j(k-1)} - 1 < O_i \leqslant \sum_{j=1}^{m} 2^{j(k-1)} - 1 \quad \text{for } m \geqslant 1,$$

and $m = \text{ceil}((\text{ceil}\log_2 O_i)/(k-1))$. Then the resulting storage space consumption for O_i is $El_i = m(k+1)$ bits.

Using our DeweyID labeling scheme, we do not need a separate mechanism to indicate overflow divisions or level transitions, because the distinction is made by odd and even values which are chosen to be order preserving. Hence, we could use an optimized DLN encoding scheme for our DeweyIDs by dropping the glue bits in the scheme of Table 1 and by abandoning the control tokens. The resulting encoding of DeweyIDs 1.7.11 and 1.8.3.3 (we assume that 1.9.xx is already taken) is more economical and allows correct and fast comparisons (at the byte level):

0000 0110 1000 1010
0000 0111 0010 0010

This encoding method can be analyzed using the same calculation formulas for O_i and m, but resulting in $El_i = m \cdot k$ bits. See Appendix B for some indicative values for their space requirements.

4.4. Use of separators

As opposed to control tokens, separators are characterized by the value of a bit sequence. Based on such separators, an encoding approach of this class is using a k-based digital representation where the length of the encoding unit is determined by $m = \log_2(k+1)$. The idea is to reserve an m-bit code to represent the separator ".", while a sequence of m-bit codes is interpreted as a number with base k. For example, $k = 3$ delivers the following codes: 00: "0", 01: "1", 10: "2", 11: ".". Hence, 1.7.11 is encoded by $E_1 = 01\ 11\ 10\ 01\ 11\ 01\ 00\ 10$ which reads $(1 \times 3^0) \times (2 \times 3^1 + 1 \times 3^0) \times (1 \times 3^2 + 0 \times 3^1 + 2 \times 3^0)$. Our evaluation in Appendix B compares the conceivable candidates for k. While $k = 1$ delivers a "funny" and very inefficient encoding, $k = 3$ and $k = 7$ may be appropriate for specific value distributions. Ref. [27] claims that $k = 3$ is superior to other Dewey encodings. However, a k-based digital representation for small division values is rather space-consuming and therefore does not provide an optimal solution.

On the other hand, such schemes embody a definite disadvantage: fast bit- or byte-level comparison—a core operation for query processing—is not possible. While, due to their positional use, control tokens preserve comparability, separators do not. Assume $E_2 = 01\ 11\ 10\ 01\ 11\ 10\ 01$ as the encoding for 1.7.7. Then the comparison delivers $E_1 < E_2$ while 1.7.11 > 1.7.7, although in this case separator values and data values were compared with each other. Hence, algorithms have to regard separators and division lengths (to be detected by checking separators) to provide correct comparisons.

4.5. Use of prefix-free codes

Prefix-free codes can be adjusted to the value distributions of the divisions used for DeweyIDs. Hence, they offer an extra degree of freedom for optimization. Using pairs $C_i|O_i$ to represent division values, the idea of Huffman trees can be applied to determine prefix-free codes C_i and to assign to them a specific length for O, e.g., via Table 2. Direct comparability of encoded division values can be always guaranteed by proper assignment of codes and value ranges. To compare the space consumption of Huffman codes with the other schemes

Table 2
H1: Assigning codes to L_i fields

Code C_i	L_i	Value range of O_i
0	3	1–7
100	4	8–23
101	6	24–87
1100	8	88–343
1101	12	344–4439
11100	16	4440–69,975
11101	20	69,976–1,118,551
11110	24	1,118,552–17,895,767
11111	31	17,895,768–2,165,379,414

TL	C_0	E_0	C_1	E_1	. . .	C_k	E_k

Fig. 4. DeweyID template.

considered, we have checked several codes where the superior one H1 delivering encodings $\geqslant$4 bit is contained in Table 2 and H2 in Table 5.

4.6. Encoding DeweyIDs

We have designed an overall template for a DeweyID where each division consists of a $C_i|E_i$ pair as illustrated in Fig. 4. TL of fixed length contains the total length in bytes of the actual DeweyID, belongs to the externally stored DeweyID format, and is kept in a resp. entry of the B*-tree managing the collection of DeweyIDs on external storage. A given encoded DeweyID is decoded as follows: As soon as a code given in Table 2 is matched while scanning the field C_0, the associated length information is used (assume code 101 in row 3) to extract the E_0 value contained in the subsequent 6 bits. Encoding is performed by assigning 000000 to the first value 24 and 111111 to the last value 87 of the related range. Therefore, if we have extracted 001010, we can decode it to value 34. Then we scan field C_1 and so on, until E_k is reached. Because the actual k is not explicitly stored, TL helps to determine the proper end of the DeweyID. Encoding is accomplished the other way around. Assume the encoding E_i of a division O_i with decimal value 11. Hence, the second row in Table 2 delivers $C_i = 100$ and $L_i = 4$. Because 11 is the fourth value of range 8–23, we yield an encoding of 0011, which is composed to the $C_i|E_i$ encoding of 1000011.

Because DeweyIDs are stored and managed in byte-structured sequences in B*-trees, storing a bit-encoded DeweyID in a byte structure needs some alignment measure. By using Table 2, DeweyID 1.7.11, for example, results in the bit sequence 00010111.1000011 where we have inserted a dot to indicate the byte boundary for improved clarity. Because the last byte is incomplete, it is padded by zeros.[9] Hence, the TL value is 2 (bytes) and the stored DeweyID is 00010111.10000110. When Huffman codes are assigned in ascending order (Table 2), the encoded values and the entire DeweyIDs are order preserving. Hence efficient byte-level (prefix) comparisons can be applied to determine the order of two DeweyIDs.

5. Empirical evaluation of DeweyIDs

Efficient encoding and variable-length representation of DeweyIDs is a prerequisite in XDBMSs. To assess their capability in a "holistic" sense, we have implemented XTC [11] as a prototype XDBMS with multi-lingual XML interfaces and concurrent transaction processing. Hence, we were able to study the DeweyID behavior in a real system context. The benefits of DeweyIDs for locking protocols in collaborative

[9] Because value 000 is not used, padded zeros can be distinguished from encoded values. In the following, we exclude TL, typically a single byte, from our size considerations.

Table 3
DOM characteristics of the XML documents considered

	File name (.xml)	Description	Size (bytes)
1	treebank	Encoded DB of English records of Wall Street Journal	86,082,517
2	nasa	Astronomical data	25,050,288
3	psd7003	DB of protein sequences	716,853,016
4	SwissProt	DB of protein sequences	114,820,211
5	dblp	Computer Science Index	284,994,162
6	customer	Customers from TPC-H benchmark	515,660
7	ebay	Ebay auction data	35,562
8	lineitem	Line items from TPC-H benchmark	32,295,475
9	mondial-3.0	Geographical DB of diverse sources	1,784,825
10	orders	Orders from TPC-H Benchmark	5,378,845
11	uwm	Courses of a University Website	2,337,521

	File name	Number of nodes			Depth		Fan-out of nodes	
		Element	Text	Attribute	Max.	∅	Max.	∅
1	treebank	2,437,666	1,391,845	1	37	8.44	56,385	1.58
2	nasa	476,646	303,676	56,317	9	6.08	2435	1.76
3	psd7003	21,305,818	15,955,109	1,290,647	8	5.68	262,529	1.81
4	SwissProt	2,977,031	2,013,844	2,189,859	6	4.07	50,000	2.41
5	dblp	6,662,623	6,013,355	1,375,832	7	3.39	649,080	2.11
6	customer	13,501	12,000	1	4	3.41	1501	1.89
7	ebay	156	107	0	6	4.26	12	1.90
8	lineitem	1,022,976	962,800	1	4	3.45	60,176	1.94
9	mondial-3.0	22,423	7467	47,423	6	4.15	955	3.45
10	orders	150,001	135,000	1	4	3.42	15,001	1.90
11	uwm	66,729	40,234	6	6	4.37	2112	1.91

environments with navigational and declarative access to XML documents were reported in [10]. Here we concentrate on the consumption and optimization of storage space of DeweyIDs needed for fine-grained representation of XML documents and indexing support and refine our preliminary study of [12]. A critical question is "what are representative documents, especially, concerning their depth?". An empirical study [18] gathered about 200,000 XML trees worldwide where 99% have less than 8 levels, i.e., less than depth 8. Almost all of the remaining 1% documents range between 8 and 30. Only a tiny fraction of the documents gathered has more than 30 levels.[10] For this reason, we have empirically explored a variety of 11 XML documents [19] listed in Table 3, which roughly fit into this statistical distribution. Because of the wide spectrum of structural properties, these documents provide an "acid test" for any labeling scheme.

5.1. Consumption of storage space

Given the benefits of DeweyIDs summarized in Section 3.4, the most important question is: at which cost can these processing services be provided? Because XML documents may be very large, they are fetched to memory in small fractions only as needed. If possible, most of the document processing should be performed on indexes and reference lists (similar to TID processing in relational systems) to reduce access to external storage as far as possible. Hence, the most critical cost factor is the size of the DeweyIDs and, because of their immense variation, the average size (∅-size) per document, which is defined as storage consumption of all DeweyIDs of a document divided by the number of all its nodes (element/text/attribute). Of course, ∅-size is strongly influenced by the document characteristics. However, the selection of the distance parameter—critical for later node insertions and the avoidance of division overflows—also largely determines the ∅-size. While the document characteristics are "there", the appropriate choice of distance is a design decision and should somehow reflect the later modification activity.

[10] The maximum depth of 135 found was due to an erroneous translation from html [21].

In our experiments, we assigned the DeweyIDs (according to the rules in Section 3.2) during the bulk-load of the documents thereby using Huffman code H1 (see Table 2). A slight optimization is already included in the results of Fig. 5. Because all DeweyIDs start with "1.", we do not explicitly store this division thereby saving 4 bits for every document node. We have systematically varied the distance parameter of a division from 2 (where almost no inserts are expected) to 256 (to characterize the ∅-size growth beyond the range of practical interest). For clarity, our presentation in Fig. 5 is restricted to the most demanding documents and, as a contrast, to the document customer.xml which more or less represents a relational structure in XML format. We assume that the interesting range of the distance parameter, depending on the update activity anticipated, is less than 32 in practical applications. Note, ∅-size is surprisingly small for (the full storage of) the encoded DeweyIDs. In particular, if we restrict the design space to distance $d \leqslant 32$, we can come up for 10 documents with ∅-sizes of 3–9 bytes which is comparable to TID encodings in relational DBMSs. Even in the exceptional case of document 1 (max. depth 37, max. fan-out 56,385) ∅-size remains under 12 bytes.

Dependent on max-/∅-depth, we group the ∅-size results of Fig. 5 into three classes: As expected, document 1 with characteristic values (37/8.44) clearly represents the "loser" in terms of space consumption. Note, the absolutely minimal length of a DeweyID at level 37 is 18 bytes (with our H1 encoding). Documents 2 and 3 (with (9/6.08) and (8/5.68)) are in the "middle" class, whereas very economical solutions are provided by the remaining documents with an ∅-depth of 3 or 4.

We do not want to keep the maximum lengths (max-sizes) of DeweyIDs secret which occur in our experiments. This property is only relevant if we consider implementation restrictions for variable-length keys or references in the XDBMS. For example, if the B*-tree implementation has a hypothetical restriction for the entry length (say 128 bytes), then a violation by a longer DeweyID would imply a reorganization/relabeling of the document. As illustrated in Table 4 for selected values of documents from the three "size" classes, the max-size behavior is reasonable and can be captured by flexible implementation mechanisms.

5.2. *Influence of distance parameter*

Fig. 6 visualizes for all sample documents the average fraction of the ∅-size caused by the distance parameter. If we define the measure DistanceInfluence per document as $\mathrm{DI}(\mathrm{doc\#}, d) = (\varnothing\text{-size@dist}(d) - \varnothing\text{-size@dist}(2))/\varnothing\text{-size@dist}(2)$, we can immediately calculate some of these factors using Table 4. While distance dominates the ∅-size for large values, e.g., $\mathrm{DI}(1, 256) = 1.39$ (or 139%), this relationship is more reasonable for distance $d \leqslant 32$. For example, applied to documents 1, 2, and 5, we yield $\mathrm{DI}(1, 32) = 0.73$, $\mathrm{DI}(2, 32) = 0.64$, and $\mathrm{DI}(5, 32) = 0.33$.

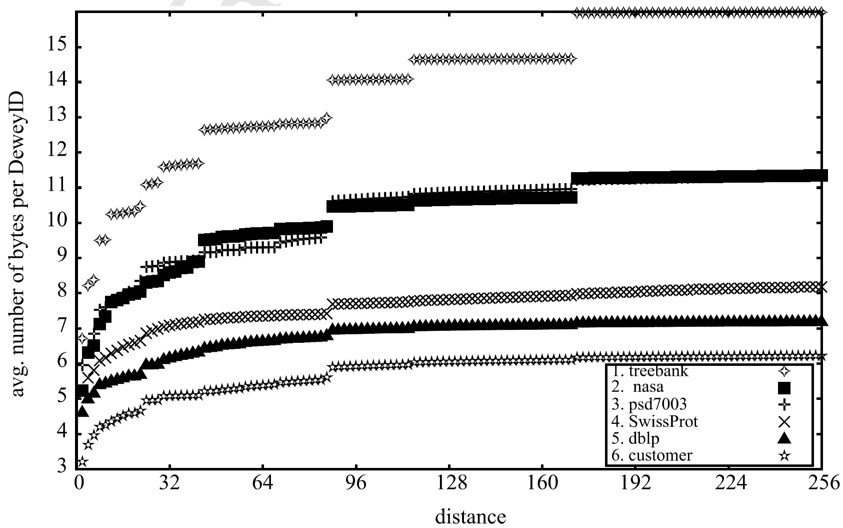

Fig. 5. ∅-size of DeweyIDs grouped by the document's ∅-depth.

Table 4
Comparison of ∅-sizes to max-sizes

Document	∅-size			Max-size		
	dist(2)	dist(32)	dist(256)	dist(2)	dist(32)	dist(256)
1. treebank	6.67	11.57	15.94	22	46	72
2. nasa	5.19	8.54	11.30	8	13	18
3. psd7003	5.61	8.84	11.30	8	13	17
4. SwissProt	5.10	7.04	8.14	8	11	13
5. dblp	4.58	6.12	7.16	7	10	13
6. customer	3.17	5.04	6.19	4	6	7

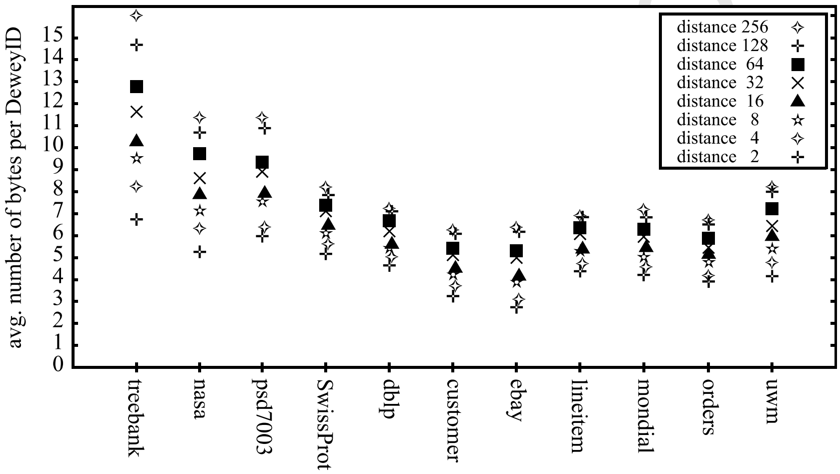

Fig. 6. Influence of the distance parameter.

Hence, the deeper the XML documents are, the more critical is the appropriate selection of distance *d*. If documents are bulk-loaded and experience less modifications, $d = 2$ is the right choice. However, frequent updates need some serious considerations to reduce the danger of "gap overflows" while limiting space consumption. An overflow lengthens the DeweyIDs in the entire subtree and, if several of them in the same "tree area" accumulate even division values in some DeweyID, the first one violating the implementation restrictions on key length provokes a reorganization run (limited to a particular subtree would ease this situation). Thus, optimal assignment of the DeweyID parameters is complex and could be greatly supported by a physical structure advisor which could use our findings.

5.3. *Prefix compression*

Another way to relieve this problem may be found in further optimization measures. A hint may be given by Fig. 3 where the DeweyIDs representing variable-length keys occur in document order in the pages of the document container (document index). This tight sequence of DeweyIDs lends itself to prefix compression across the entire physical document. Even in the node reference indexes (element index) is a great deal of sortedness of the DeweyIDs used here as references to the resp. element nodes. Hence, we have applied prefix compression to the DeweyIDs in both types of structures. For example, in each container page of size 8K, the first DeweyID is stored in the uncompressed format, while for subsequent DeweyIDs only the matching prefix length PL—aligned to byte boundaries—is stored in a 1-byte field and attached to the uncompressed remainder. This prefix compression works in such an effective way that we were surprised about the space reduction achieved. The results for the average number of bytes needed for entries (PL + remainder) are illustrated in Figs. 7 and 8 which can be immediately compared with Fig. 6.

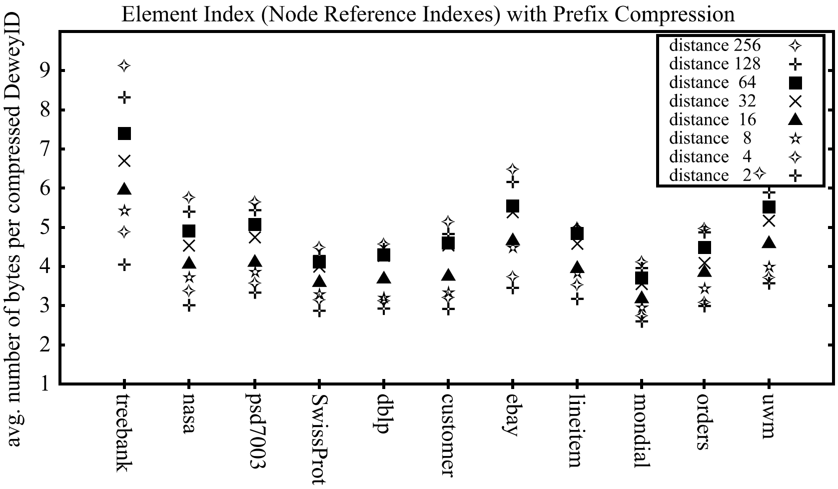

Fig. 7. ∅-size and DistanceInfluence in the prefix-compressed element index.

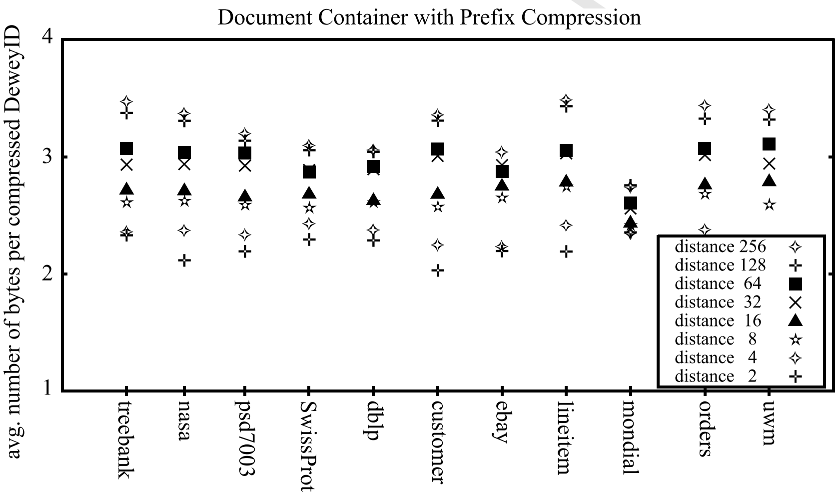

Fig. 8. ∅-size and DistanceInfluence in the prefix-compressed document.

Prefix compression in the node reference indexes applies to DeweyIDs whose element nodes have the same element name. Although they are ordered, they are rather sparse, because they do not occur in the same paths. Nevertheless, this optimization measure is very effective. As illustrated in Fig. 7, we obtain ∅-comp-size@-dist(2) in the range of 3–4 bytes. With the exception of treebank (because of its extraordinary depth), we can estimate ∅-comp-size in the range of 3–6 bytes per DeweyID for all documents and distance values evaluated which often corresponds to a reduction of more than 40%. For this reason, it is safe to say that space consumption of compressed DeweyIDs in node reference lists is absolutely comparable to that of TID lists[11] and in many cases even better.

Because of the lexicographic order in the document container, prefix compression is most effective and gains a reduction (∅-comp-size@dist(d)/∅-size@dist(d)) to less than ≈0.35 (35%) of the uncompressed size; this corresponds to a saving of more than 200%. As a rule of thumb, we obtain ∅-comp-size in the range of 2–3.5 bytes per DeweyID for all documents and distance values evaluated.

[11] Typical sizes of tuple identifiers (TIDs) in relational DBMSs vary from 4 to 6 bytes.

Table 5
H2: Assigning codes to L_i fields

Code	L_i	Value range of O_i
0	7	1–127
10	14	128–16,511
110	21	16,512–2,113,663
1110	28	2,113,664–270,549,119
1111	36	270,549,120 ≈ 2^{37}

5.4. Optimization by tailoring Huffman codes to value distributions

The codes of Table 2 are only an example used for our experiments. They can be constructed using a Huffman tree thereby adjusting the code lengths to the anticipated O_i length distributions. For this reason, we are able to achieve the optimal assignment of code lengths/O_i length distributions, if the latter are known in advance or are collected in an analyzing run or a by a representative sample before bulk-loading of XML documents. By default, we expect the larger numbers of divisions in the smaller value ranges of O_i and use this heuristics for the Huffman codes and length assignments. Obviously, the lion share of optimization was achieved by prefix compression of DeweyIDs in the document structure and the element index. Nevertheless, to explore this opportunity for further DeweyID optimization, we perform an analysis phase before loading of documents. Hence, we can figure out the distributions and frequencies of the division values, which we use to derive a Huffman code tailored to the document.

Table 5 contains Huffman code H2 which was optimized for treebank and, at the same time, for its use under prefix compression. Hence, we counted the frequencies of the different division values to decide on the assignment of codes and length values. One important observation was that prefix compression cuts divisions in the path closer to the root often containing smaller values. Hence, smaller codes are not so critical anymore. Another observation was equally important. Because the DeweyIDs are stored, indicated by TL, in byte structures and the compressed DeweyID is also aligned to full bytes, it is important to avoid padding with zeros (see Section 4.6). Therefore, H2 was designed in a way that each division already observes byte boundaries.

By comparing Fig. 8 with Fig. 9, the additional space saving can be immediately determined. Because we compare (PL + remainder) with a one-byte entry for PL, we can state that the remainder for ∅-comp-size never needs more than 2 bytes and that it is a good rule of thumb to expect a one-byte remainder for ∅-comp-size@dist($d \leqslant 32$). As an example, concerning the treebank document, we achieve a reduction (∅-comp-size@dist($d \leqslant 32$)/∅-size@dist($d \leqslant 32$)) to less than 0.2–0.3 (20–30%) of the uncompressed size. In

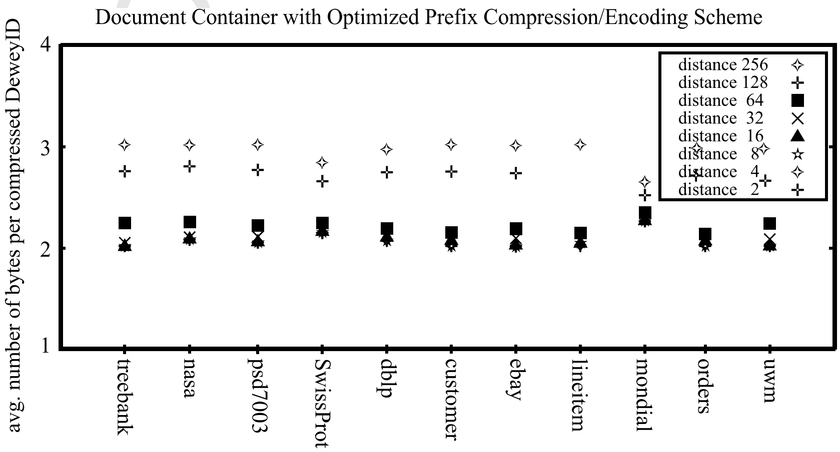

Fig. 9. ∅-size and DistanceInfluence in the optimized document.

our experiments, this further optimization does not much influence the node reference index. Some marginal improvements could be achieved by adjusting the encoding scheme to the distance parameter.

In summary, prefix compression on DeweyIDs works very well and supports the clustering effects of our storage structures on disk. Therefore, it also greatly reduces the number of page fetches needed to reconstruct (parts of) the XML documents or to fetch DeweyID lists from the element index. Furthermore, our Huffman encodings preserve the direct comparability of DeweyIDs at the byte level. This comparison property is very important, because axes operations in path processing steps frequently need to compare (long) lists of DeweyIDs. In experiments we have revealed that DeweyID comparisons at the byte level are 60–100 times faster than those at the bit level, for example, needed in case of separator use [23].

6. Conclusions

In this paper, we classified existing node labeling schemes and analyzed them in the light of new XML processing requirements mainly coming from multi-lingual interfaces (support of declarative and navigational access) and collaborative transactional modifications (requiring kind of hierarchical lock protocols). With the advent such additional XML processing functionality, early node labeling schemes turned out to be less useful, because often balanced document structures or read-only declarative access were assumed. Even sophisticated enhancements in range-encoded or prefix-encoded schemes were unable to repair the defects under the new requirements.

We introduced a particular and dynamic mapping of the lexicographic Dewey order to the nodes of XML document trees which can guarantee immutable node labels and very effective locking support. We refined this mapping to the concept of DeweyIDs and showed that they best satisfy the support of all enhanced XML processing needs. We believe that the use of DeweyIDs is of paramount importance for the lock protocol overhead and, in turn, for the entire performance of concurrency control in XML trees. All ancestor node IDs and most other IDs needed for locking navigation steps can be easily derived from them (using indexes and Dewey order) without traversing the document itself. Queries specified by declarative languages are assumed to be frequently processed via indexes, which will require a large number of direct jumps. On the other hand, DeweyIDs allow structural joins and set-theoretic operations such that they become more useful than TIDs in relational DBMSs. This behavior is achieved because DeweyIDs themselves essentially represent a kind of path index reference which can be effectively used in many processing steps during query evaluation on XML documents [16].

The proposition often raised in the literature that DeweyIDs are not ‘‘implementable’’ triggered a broad empirical study to ‘‘prove’’ the opposite. Based on a suitable physical document structure and element index, DeweyIDs are used as variable-length keys and references. We have shown that DeweyIDs and their divisions can be efficiently encoded by several methods. We preferred a method based on Huffman codes because of its flexibility and optimization potential.

While encoded DeweyIDs are manageable—consuming 3–11 bytes for all documents and distance values considered with the exception of treebank—, our optimization based on prefix compression delivered surprising results: as a rule of thumb, we need an $\emptyset$-comp-size in the range of 2–3.5 bytes and 3–6 bytes per DeweyID for all documents and distance values evaluated for the document container and element index, respectively. Tayloring the Huffman codes to the value distributions of the divisions and adjusting them to the byte structure of the index, further reduced storage overhead and the influence of the distance parameter. In particular for the document container, we achieved an optimized $\emptyset$-comp-size of 2–3 bytes for all documents considered.

Acknowledgments

We want to thank Timo Böhme and the anonymous referees for their helpful hints to improve the readability of this paper. The support of Andreas Bühmann while formatting the final version is appreciated.

Appendix A. Calculation of the edge string label

As shown in Section 2.2, the label string S_c of node c is a concatenation of edge codes, where each code is taken from a prefix-free set B_n of binary strings (assigned to the outgoing edges of a node n). Therefore, S_c can be written as $S_c = s_0 s_1 \ldots s_k$, where s_0 is the edge code for the root node and s_k is the edge code for node c. As a result, S_c contains the labels of all its ancestors as prefixes. To calculate these labels, we simply have to infer the lengths e_j of each edge code s_j $(0 \leqslant j \leqslant k)$ in S_c, and truncate S_c at the derived positions. Because the number of a node's children can vary heavily, the size of each B_n, and therefore the length of distinct edge codes, may be different.[12] To solve this problem, the length information of each e_j is encoded into the *esl*.

The encoding works analogous to the conversion of natural numbers from base b to base 10. Consider, e.g., the number $x_3 = 210_3$ (of base 3). We can calculate its value to base 10 by $x_{10} = 2 \times 3^2 + 1 \times 3^1 + 0 \times 3^0 = 21_{10}$. Conversely, given length k of x, we can infer the value of each digit d_i on position i of $x_3 = d_0 d_1 d_2$ with the following formula:

$$d_i = \left\lfloor \frac{x_{10}}{3^{(k-1)-i}} \right\rfloor \bmod 3.$$

Calculating an *esl*, base b is set to the maximum length of the longest edge code occurring in the document, increased by one. For example, in the document of Fig. 1b, b is 3. With the node's depth k and the length of each edge code e_j, the *esl* can be computed by the formula

$$esl = e_0 \cdot b^{k-1} + e_1 \cdot b^{k-2} + \cdots + e_{k-1} \cdot b^0.$$

Therefore, the *esl* of node n_6 in Fig. 1b is

$$esl_6 = 1 \times 3^{3-1} + 2 \times 3^{3-2} + 1 \times 3^0 = 16.$$

If we want to recalculate the edge code lengths of the ancestors for n_6, we can use the same formula as shown above:

$$e_i = \left\lfloor \frac{esl}{b^{(k-1)-i}} \right\rfloor \bmod b.$$

This results in the lengths $e_0 = 1$, $e_1 = 2$, and $e_2 = 1$.

In practice, the edge string labels may get very large. The length of an edge code is bounded by $e_i = |s_i| \leqslant \log \Delta$, when Δ denotes the maximum fan-out of a node in the document. Therefore, if a node has 10^6 children, $b = 20$. With a node depth of 37, we would need an *esl* with approximately 50 decimal digits.

Appendix B. Comparison of space consumption for encodings

For the methods introduced in Section 4, we have calculated in Table B.2—using relevant parameter values—the storage space needed for indicative division values of DeweyIDs. Furthermore, we have listed the size limit for each encoding method, which indicates that some method/parameter combinations are not eligible for practical applications. The tradeoff between the encodings for very small and very large values varies among the methods, while no method gains in all division sizes. The quantities for the static schemes only serve for comparison reasons. While the choice of a fixed maximum encoding unit for a document embodies the worst case, the optimum msl assignment for O_i characterizes the lower boundary which dynamic schemes strive for. Methods based on length fields with stepwise growth and such using control tokens (optimized DLN codes) deliver comparable results, whereas methods using separators do not convince due to space consumption (and lack of direct comparability of divisions and DeweyIDs). The best results for the compared division values are highlighted by bold style. While some of the eligible methods provide for reasonable storage cost efficiency, we prefer the methods based on Huffman codes, because they can be adjusted to division value distributions and tailored to optimal representation of small values. H1 corresponds to the code given in

[12] In general, when *short* labels are desired, this proposition is true. However, it is possible, though a waste of space, to construct edge codes, which mutually have the same length.

Table B.1
H3: Assigning codes to L_i fields

Code	L_i	Value range of O_i
0	3	1–8
10	6	9–72
110	9	73–584
1110	12	585–4680
11110	15	...

Table B.2
Lengths of division encodings (*ee*: explicitly extendable)

Encoding methods		EL_i for division value O_i					Limit
		7	2^7	2^{14}	2^{21}	2^{28}	
Static schemes with advance knowledge							
Fixed maximum (28)		28	28	28	28	28	2^{28}
Optimum *msl* for O_i		3	7	14	21	28	*ee*
Length fields							
Fixed length	$l_i = 4$	7	11	18	–	–	2^{16}
	$l_i = 5$	8	12	19	26	33	2^{32}
	$l_i = 6$	9	13	20	27	34	2^{64}
Stepwise growth	$u = 2$	5	13	24	34	48	–
	$u = 3$	6	**10**	20	30	40	–
	$u = 4$	7	11	18	29	36	–
Variable length	$ll_i = 2$	7	12	19	27	34	$\approx 2^{31}$
	$ll_i = 3$	8	13	20	28	35	$\approx 2^{511}$
Control tokens							
Optimum *msl* for O_i, skewless docs only		4	8	15	22	29	*ee*
Streaming DLN codes	$k = 3$	7	15	27	43	55	–
	$k = 4$	**4**	14	24	34	49	–
	$k = 5$	5	11	23	35	41	–
Optimum DLN codes for DeweyIDs	$k = 3$	6	12	21	33	42	–
	$k = 4$	**4**	12	20	28	40	–
	$k = 5$	5	**10**	20	30	35	–
Separators							
k-based digits	$k = 1$	8	$2^7 + 1$	$2^{14} + 1$	$2^{21} + 1$	$2^{28} + 1$	–
	$k = 3$	6	12	20	30	36	–
	$k = 7$	9	12	18	27	33	–
Prefix-free codes							
Huffman code	H1	**4**	12	21	29	36	*ee*
	H2	8	16	**16**	**24**	**32**	*ee*

Table 2. As shown in Table 5, Huffman code H2 can be tailored to favor large and very large values. In Section 4.4 we have optimized the DLN scheme of Table 1 where only the length delimiter was kept. Due to DeweyID use we could drop all control tokens. A closer look at this scheme reveals that it is identical to a Huffman code H3 with a fixed length assignment scheme illustrated in Table B.1 for $k = 4$. Hence, our Huffman schemes are superior, because they have an additional degree of freedom, which enables a tailored length assignment adjusted to the distribution of division values.

References

[1] S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, D. Srivastava, Structural joins: a primitive for efficient xml query pattern matching, in: Proc. 18th Int. Conf. on Data Engineering (ICDE), San Jose Calif., 2002, pp. 141–152.

[2] M.W. Blasgen, R.G. Casey, K.P. Eswaran, An encoding method for multifield sorting and indexing, Communications of the ACM 20 (11) (1977) 874–876.
[3] T. Böhme, E. Rahm, Supporting efficient streaming and insertion of XML data in RDBMS, in: Proc. 3rd Int. Workshop Data Integration over the Web (DIWeb), Riga, Latvia, 2004, pp. 70–81.
[4] Q. Chen, A. Lim, K.W. Ong, J.Q. Tang, Indexing XML documents for XPath query processing in external memory, Data Knowledge Eng, in press.
[5] V. Christophides, D. Plexousakis, M. Scholl, S. Tourtounis, On labeling schemes for the semantic Web, in: Proc. 12th Int. WWW Conf., Budapest, 2003, pp. 544–555.
[6] M. Dewey, Dewey Decimal Classification System, http://www.mtsu.edu/vvesper/dewey.html.
[7] E. Cohen, H. Kaplan, T. Milo, Labeling dynamic XML trees, in: Proc. PODS, 2002, pp. 271–281.
[8] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, Los Altos, CA, 1993.
[9] M. Haustein, Fine-granular transaction isolation in native XML database management systems, Ph.D. Thesis, University of Kaiserslautern, 2005 (in German).
[10] M. Haustein, T. Härder, Adjustable transaction isolation in XML database management systems, in: Proc. 2nd Int. Database Symp., Toronto, CanadaLNCS 3186, Springer, Berlin, 2004, pp. 173–188.
[11] M. Haustein, T. Härder: Fine-grained management of natively stored XML documents, submitted for publication. Available from: <http://wwwdvs.informatik.uni-kl.de/pubs/p2005.html>.
[12] M. Haustein, T. Härder, C. Mathis, M. Wagner, DeweyIDs—The key to fine-grained management of XML documents, in: Proc. 20th Brasilian Symposium on Databases (SBBD), Uberlandia, Brasil, 2005, pp. 85–99.
[13] M. Jacobssen, Implementation of compressed bit-vector indexes, in: P.A. Samet (Ed.), Proc. EURO IFIP79, North Holland, Amsterdam, 1979, pp. 561–566.
[14] J.R. Jordan, J. Banerjee, R.B. Batman, Precision locks, in: Proc. ACM SIGMOD Conf., Ann Arbor, Michigan, 1981, pp. 143–147.
[15] Y.K. Lee, S. Yoo, K. Yoon, B.P. Berra, Index structures for structured documents, in: Proc. 1st ACM Int. Conf. on Digital Libraries, 1996, pp. 91–99.
[16] C. Mathis, T. Härder, M. Haustein, Supporting path processing steps of XML queries by DeweyIDs, submitted for publication. Available from: <http://wwwdvs.informatik.uni-kl.de/pubs/p2005.html>.
[17] W. Meier, eXist: An open source native XML database, in: Web, Web-Services, and Database Systems 2002LNCS 2593, Springer, Berlin, 2002, pp. 169–183.
[18] L. Mignet, D. Barbosa, P. Veltri, The XML Web: a first study, in: Proc. 12th Int. WWW Conf., Budapest, 2003. Available from: <http://www.cs.toronto.edu/mignet/Publications/www2003.pdf>.
[19] G. Miklau: XML Data Repository, <http://www.cs.washington.edu/research/xmldatasets/>.
[20] C. Mohan: ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-Tree indexes, in: Proc. VLDB Conf., 1990, pp. 392–405.
[21] P.E. O'Neil, E.J. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury, ORDPATHs: Insert-friendly XML node labels, in: Proc. SIGMOD Conf., 2004. pp. 903–908.
[22] I. Tatarinov et al., Storing and querying ordered XML using a relational database system, in: Proc. SIGMOD Conf., 2002, pp. 204–215.
[23] M. Wagner, Storage structures for native XML database management systems, Master Thesis, University of Kaiserslautern, 2005 (in German). Available from: <http://wwwdvs.informatik.uni-kl.de/pubs/DAsPAs/Wag05.DA.html>.
[24] X. Wu, M.-L. Lee, W. Hsu, A prime number labeling scheme for dynamic ordered XML trees, in: Proc. 20th Int. Conf. on Data Engineering (ICDE), 2004, pp. 66–78.
[25] W3C Recommendations, <http://www.w3c.org/>, 2004.
[26] XQuery 1.0: An XML Query Language, W3C Working Draft, October 2004.
[27] J.X. Yu, D. Luo, X. Meng, H. Lu, Dynamically Updating XML Data: Numbering Scheme Revisited, World Wide Web: Internet and Web Information Systems 8 (2005) 5–26.

Theo Härder obtained his Ph.D. degree in Computer Science from the TU Darmstadt in 1975. In 1976, he spent a post-doctoral year at the IBM Research Lab in San Jose and joined the project System R. In 1978, he was associate professor for Computer Science at the TU Darmstadt. As a full professor, he is leading the research group DBIS at the TU Kaiserslautern since 1980. He is the recipient of the Konrad Zuse Medal (2001) and the Alwin Walther Medal (2004) and obtained the Honorary Doctoral Degree from the Computer Science Dept. of the University of Oldenburg in 2002. Theo Härder's research interests are in all areas of database and information systems—in particular, DBMS architecture, transaction systems, information integration, and Web information systems. He is author/coauthor of 7 textbooks and of more than 200 scientific contributions with >100 peer-reviewed conference papers and >50 journal publications. His professional services include numerous positions as chairman of the GI-Fachbereich "Databases and Information Systems", conference/program chairs and program committee member, editor-in-chief of Informatik—Forschung und Entwicklung (Springer), associate editor of Information Systems (Elsevier), World Wide Web (Kluver), and Transactions on Database Systems (ACM). He served as a DFG (German Research Foundation) expert and was chairman of the Center for Computed-based Engineering Systems at the University of Kaiserslautern, member of two joint collaborative DFG research projects DFG (SFB 124, SFB 501), and co-coordinator of the National DFG Research Program "Object Bases for Experts".

Michael Haustein studied computer science at the University of Kaiserslautern from 1995 to 2002. That followed a scientific staff membership at the chair of Prof. Härder to the end of 2005, where he designed and implemented most parts of the native XML database management system XTC and finished his doctoral examination procedure. In 2006, he changed to SAP AG.

Christian Mathis studied Computer Science from 1998 to 2004 at the University of Kaiserslautern. Since 2004 he is a Ph.D. student at the DBIS research group lead by Prof. Härder. In the XTC project (XML Transaction Coordinator, http://www.xtc-project.de) he explores XML query processing.

Markus Wagner studied Computer Science at the University of Kaiserslautern from 1998 to 2005. He just finished his diploma thesis about storage structures in native XML database systems and reimplemented the storage structures of the native XML database management system XTC. His interests are in the area of information systems and data-intensive applications.