DeweyIDs—The Key to Fine-Grained Management of XML Documents

Michael P. Haustein, Theo Härder, Christian Mathis, Markus Wagner

University of Kaiserslautern, 67653 Kaiserslautern, Germany

{haustein,haerder,mathis,m_wagner}@informatik.uni-kl.de

Abstract. *Because XML documents tend to be very large and are more and more collaboratively processed, their fine-grained storage and management is a must for which, in turn, a flexible tree representation is mandatory. Performance requirements dictate efficient query and update processing in multi-user environments. For this reason, three aspects are of particular importance: index support to directly access each internal document node if needed, navigation along the parent, child, and sibling axes, selective and direct locking of minimal document granules. The secret to effectively accelerate all of them are DeweyIDs. They identify the tree nodes, avoid relabeling of them even under heavy node insertions and deletions, and allow, at the same time, the derivation of all ancestor node IDs without accessing the document. In this paper, we explore the concept of DeweyIDs, refine the ORDPATH addressing scheme, illustrate its implementation, and give an exhaustive performance evaluation of its practical use.*

1 Motivation

Because messages are data and have to be managed in the same way as database data, XML DBMSs (XDBMSs for short) are rapidly evolving to seamlessly support XML applications which dramatically grow in number and complexity and need to process increasing data volumes under tight schedules. Furthermore, collaborative applications often require concurrent read as well as write access to such XML data [17].

Although the language layers of XDBMSs typically provide declarative interfaces such as XQuery and XPath to process XML documents, their requests have to be mapped to procedural operators at the access and storage layers to achieve efficient and direct access of document nodes. On the other hand, standardized XML interfaces such as DOM [17] enable direct requests using navigational operators. Without index support, for example frequent scans of the entire document would make response times intolerable. Hence, directly locating internal nodes of an XML document is the key to fast query processing. Furthermore, multi-user access needs effective and minimum-granule locking of tree nodes. Otherwise, collaborative (and concurrent) processing would often be blocked although no read/write or write/write conflicts are present. Of course, predicate locking of XQuery statements [18]—and, in the near future, XUpdate 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. To provide for an acceptable solution, we necessarily have to map XQuery operations onto node accesses to accomplish fine-granular concurrency control. Such an approach implicitly supports other interfaces like DOM and SAX [17], because their operations correspond more or less directly to navigational accesses.

Most influential for efficient access to and locking of the XML tree nodes is a suitable node labeling scheme for which several candidates have been proposed in the literature [15]. 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*. We believe that very few of the existing approaches—classified into range- and prefix-based schemes [5, 14]—can fulfill these strong requirements. Here, we explore a scheme supporting efficient insertion and compression while providing the so-called Dewey order (defined by the Dewey Decimal Classification System) described in [3]. Conceptually similar to the ORDPATH scheme [11], our scheme refines the mapping and solves practical problems of the implementation. Furthermore, we illustrate its use in the XDBMS context and summarize the results of an extensive empirical evaluation.

In Section 2, we outline our storage model for XML documents, called taDOM model, which is implemented in our XTC prototype (XML Transaction Coordinator, [8]), introduce the labeling of nodes using DeweyIDs, and illustrate how they are used for indexing, navigation, and locking. Section 3 discusses the initial allocation of DeweyIDs and their maintenance under insertions and deletions. In Section 4, we illuminate the use of DeweyIDs in B*-trees and their implementation details. A substantial empirical evaluation of DeweyID storage consumption is given in Section 5, before we summarize our results and conclude in Section 6.

2 System Aspects of XTC

2.1 taDom Storage Model

Efficient and effective processing and concurrent operations on XML documents are greatly facilitated, if we use a specialized internal representation which improves fine-granular management and locking. While we use DOM trees—containing element, attribute, and text nodes as defined in [17]—for the representation of XML documents on external storage, in our XTC system we have implemented for their memory representation a slight extension, the so-called taDOM storage model illustrated in Figure 1. In contrast to the DOM tree, we do not directly attach attributes to their element node, but introduce separate *attribute roots* which connect the attribute nodes to the resp. elements. String nodes are used to store the actual content of an attribute or a text node. Via the DOM API, this separation enables access of nodes independently of their value. Our representational enhancement does not influence the user operations and their semantics on the XML document, but is solely exploited for optimized lock management. To prove our concepts, we have designed and implemented the XTC system which embodies a multilayered architecture and, most important to our discussion, which offers a native storage structure for XML documents tailored to our objectives. In summary, our storage mechanism provides an extensible file structure as a container of single XML documents such that updates of an XML document (by IUD operations) can be performed on any of its nodes; furthermore, a very high degree of storage occupancy (> 96%) is achieved [8].

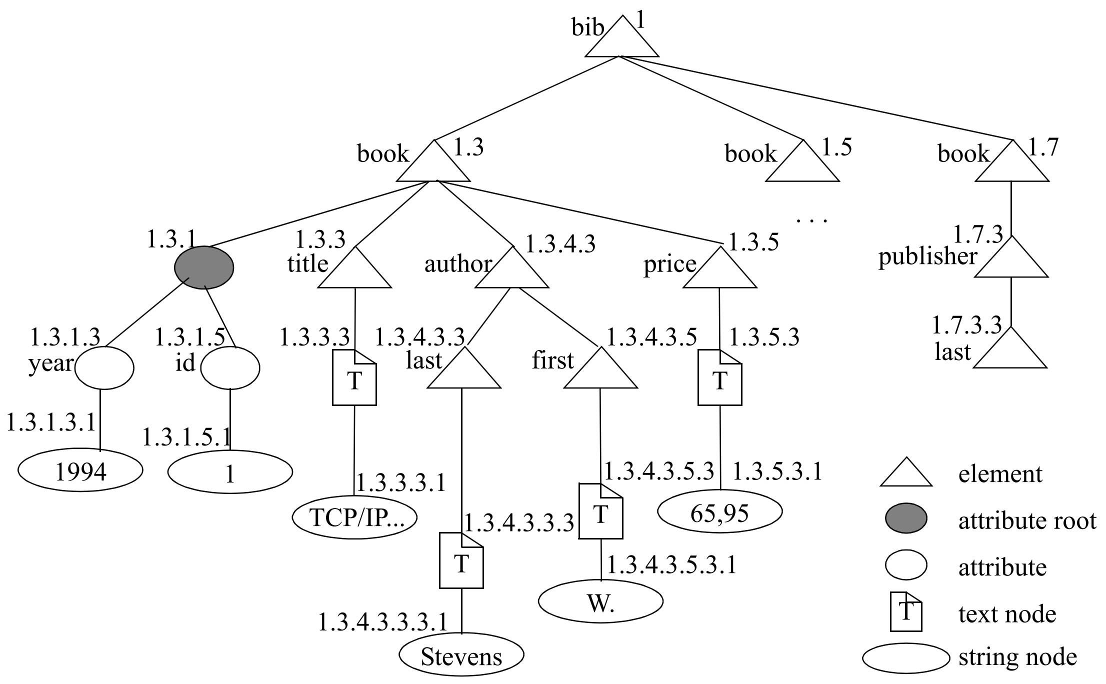

Figure 1. A sample taDOM tree labeled with DeweyIDs

2.2 Essentials of the Access Model

Fast access to and identification of all nodes of an XML document is mandatory to enable effective indexing primarily supporting declarative queries and efficient processing of direct-access methods (e. g., *getElementById()*) as well as navigational methods (e. g., *getNextSibling()*). Our solution is based on the concept of Dewey order. For this reason, we have implemented the related node labeling scheme whose advantages should be illuminated by referring to Figure 1, before we discuss the DeweyID mechanism in detail in Section 3. For example, the DeweyID for *price* is 1.3.5 which consists of three so-called *divisions* separated by dots (in the human readable format). The root node of the document (at level 0) is always labeled by DeweyID 1. The children obtain the DeweyID of their parent and normally attach another division whose value increases in the ordered set of children from left to right. To allow for later node insertions at a given level, we introduce for the assignment of division values a parameter *distance* which determines the gap initially left free in the labeling space. In Figure 1, we have chosen the minimum distance value of 2. Furthermore, assigning at a given level a distance to the first child, we always start with *distance + 1*, thereby reserving division value 1 for attribute roots and string nodes (illustrated for the attribute root of 1.3 with DeweyID 1.3.1). Hence, the mechanism of the Dewey order is quite simple when the IDs are initially assigned, e.g., when all nodes of the document are bulk-loaded. As a result, the lexicographic ordering of the DeweyIDs represents the document order, i.e., the order of a left-most depth-first document traversal.

In the above tree example, the node *author* is inserted later within the gap between the nodes *title* (d_1=1.3.3) and *price* (d_5=1.3.5) and receives DeweyID d_3=1.3.4.3. Note, so far we have only used *odd* values for divisions. If we would use *even* division values in the same way as odd divisions, it is true that we could insert in this situation the author node assigning 1.3.4 to it, but further insertions in this position at this level would be impossible. Therefore, we need a kind of overflow mechanism indicating that the labeling scheme remains at the same level when an odd division value is not available anymore for a gap. Thus, we reserve even division values for that purpose. Hence, $d_1 < d_3 < d_5$ holds in our example thus preserving the document order among the DeweyIDs. Several even division values may consecutively occur in a DeweyID (depending on the insertion history); such a continuous sequence of even values just states that the same node level is kept. Assume the element *second author* is inserted after *author*; then its node is labeled with DeweyID d_4=1.3.4.5. On the other hand, the node of a new element *subtitle* after *title* and before *author* would obtain DeweyID d_2=1.3.4.2.3 (explained in Section 3.2). Note, $d_1 < d_2 < d_3 < d_4 < d_5$ still holds. Because even values are not considered, when the level of a node is determined, for all DeweyIDs (e.g., 1.3.4.3, 1.3.4.5, 1.3.4.2.3) built using the overflow mechanism we obtain level 2. Obviously, the order and ancestor relationships are also preserved. Furthermore, the subtree insertion under node with DeweyID 1.3.4 also reveals that overflows affect the lengths of the DeweyIDs in the entire related subtree.

The salient features of a scheme assigning a DeweyID to each tree node include the following properties: Referring to the DeweyID of a node, we can determine the level of the node in the tree and the DeweyID of the parent node. Hence, we can derive its entire ancestor path up to the document root without accessing the document. By comparing the DeweyIDs of two nodes, we can decide which node appears first in the document's node order. If all sibling nodes are known, we can determine the exact position of the node within the document tree. It is also possible to insert new nodes at arbitrary locations without relabeling existing nodes. In addition, we can rapidly figure out all nodes accessible via the typical XML navigation steps (Section 2.3), if the nodes are stored in document order. However, DeweyIDs may become quite long.

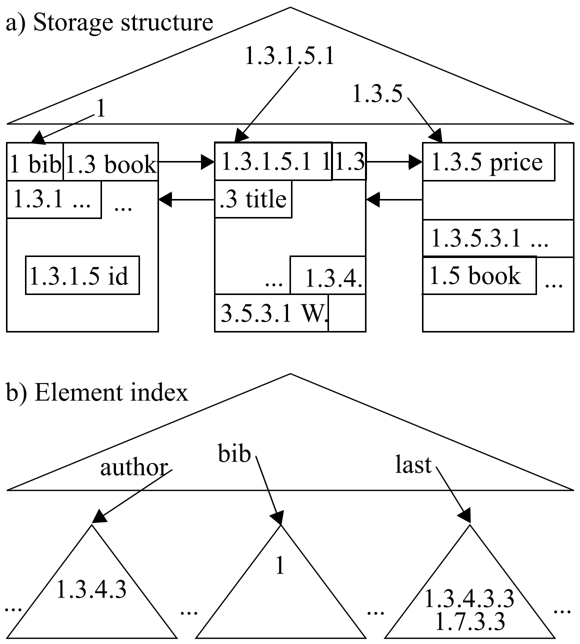

Figure 2. Document storage using B*-trees

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. Figure 2a illustrates the storage structure—consisting of *document index* and *document container* as a set of chained pages—sketching the sample XML document of Figure 1, which is stored in document order; the key-value pairs within the document index are referencing the first DeweyID stored in each container page. 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 (Figure 2b); 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 while processing XML documents, most of the referenced tree pages (at least the ones belonging to the upper tree layers) are expected to reside in DB buffers—thus reducing external accesses to a minimum. As you can see in the next section, these tree-based storage structures are building the fundamentals for very efficient navigational and declarative access to XML documents.

2.3 Supporting Navigation, Declarative Queries, and Lock Management

Typical XML navigation (accessing the parent, previous or next sibling, and first or last child of a given context node) is efficiently supported by the DeweyID addressing algorithm and the B*-trees. The siblings of a context node may reside in leaf pages located "far away" from each other. But using the document index, the pages containing the siblings can be rapidly located. At best, the corresponding objects reside in the page of the given context node cn. When accessing the previous sibling *ps* of *cn*, e.g., of node 1.5 in Figure 2, an obvious strategy would be to locate the page of 1.5 requiring a traversal of the document index from the root page to the leaf page where 1.5 is stored. This page is often already present in main memory because of reference locality. Hence, we inspect the ID *d* of the directly preceding node of 1.5 in document order, which is 1.3.5.3.1 in the example. If *ps* exists, *d* must be a descendant of *ps*. With the level information of *cn*, we can infer the ID of *ps*: 1.3. Now a direct access to 1.3 suffices to locate the result. This strategy ensures independence from the document structure, i.e., the number of descendants between *ps* and *cn* does not matter. We found similar search algorithms for the remaining four axes. The *parent* axis, as well as *first-child* and *next-sibling* is 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.

For certain declarative queries, "set-at-a-time" processing can exploit the semantic information carried by DeweyIDs which promises great advantages over the navigational "node-at-a-time" approach. For example, the XPath query *//author/last* on the document in Figure 1 may be evaluated in two steps: At first, element index scans (Figure 2b) return two lists of DeweyIDs, List_A = {1.3.4.3, ...} for all *author* elements, and List_L = {1.3.4.3.3, 1.7.3.3, ...} for all *last* elements, respectively. Then a *structural join* between these two lists is performed using as the join predicate the parent-child relationship, which can easily be deduced from the given DeweyIDs. For example, because the DeweyID a = 1.3.4.3 from List_A is a prefix of b = 1.3.4.3.3 from List_L and the difference between the levels of a and b is one, 1.3.4.3.3 matches the join predicate and has to be added to the result list. For 1.7.3.3, no such match can be found, because List_A does not contain the DeweyID 1.7.3 (which belongs to a *publisher* node). Such structural join algorithms relying on a *range-based labeling scheme* and only focussing on the parent-child and ancestor-descendant relationships, have been proposed recently [1, 3]. Currently, we are adjusting these algorithms to the more flexible DeweyID mechanism. Because they can effectively be applied in algebraic frameworks for declarative query processing, DeweyID-based methods greatly improve selection and join operations and drastically reduce I/O.

High-performance hierarchical lock management on XML data [7] requires an efficient acquisition of locks along complete node paths starting at the context node (on which the actual lock is requested) up to the document root. For performance reasons, accessing the stored document for acquiring a lock must be prevented in any case (e.g., accessing a single context node at level *l* would additionally require *l* document accesses to fetch all predecessor nodes up to the document root at level 0, before they can be locked in an adequate mode). By simply calculating each DeweyID in the ancestor path of a given context DeweyID, lock management can be performed completely independent from the XDBMS storage engine. As a consequence, accessing a single node also requires only a single document access (to get the actually requested node), although a possibly large number of predecessor nodes has to be locked for this operation.

3 Assignment of DeweyIDs

So far, we have motivated that DeweyID order and use is of paramount importance for the efficiency and effectiveness of performance-critical processing tasks in an XDBMS. Therefore, we want to elaborate on a suitable application of the Dewey ordering mechanism to dynamic document trees and its efficient representation as objects in main memory and on external storage. For the DeweyIDs, it is essential to explore their initial assignment when the nodes of the XML documents are (typically bulk-) loaded. In contrast, their behavior under (heavy) node insertions has to be considered, too.

Distance is the prime parameter of the initial DeweyID assignment (while loading the document) which determines the numerical distance between the IDs of two sibling nodes. It is used as a kind of reserving ID space in the labeling scheme enabling the insertion of new nodes without using an overflow mechanism. The value of the distance parameter influences the ID assignment of nodes inserted during document maintenance. An actual distance > 2 between two consecutive sibling IDs enables the allocation of a sibling in between without the need to assign an *even division*. Hence, the larger the actual distance, which may be increased by sibling deletions, the more nodes can be inserted without using even divisions. However, larger distance values require more bits for their representation. On the other hand, small distance values (= 2) immediately enforce the use of additional, even divisions during insertion and, in turn, increase the lengths of DeweyIDs. Of course, for static documents (almost insertion-free) we should always choose the minimum distance size. As a consequence, we may face a design trade-off between distance size and increased use of even divisions depending on the growth and volatility of an XML document. Furthermore, if a maximum length for DeweyIDs is defined (inevitable in a real implementation) and if it is exceeded due to excessive point-like insertions between two initially assigned IDs—a very rare case when reasonable parameters are used—an expensive reassignment of DeweyIDs (relabeling of nodes) may be provoked for the document.

3.1 Initial Document Loading

While a new document is loaded—typically bulk-loaded in document order—, the DeweyIDs for its nodes are dynamically assigned guided by the following rules:

1. Element root node: It always obtains DeweyID 1.
2. Element 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 level, DeweyID of L is assigned to N where the value of the last division is increased by *distance*.
3. Attribute nodes: A node N having at least one attribute, obtains (in taDOM) an attribute root R for which the DeweyID of N extended by a division with value 1 is assigned. The attribute nodes yield the DeweyID of R extended by a division. If it is

the first attribute node of R, this division has the value 3. Otherwise, the division receives the value of the last division of the last attribute node 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.

4. Text nodes: A node containing text is represented in taDOM by a text node and a string node. For text nodes, the same rules apply as for element nodes. The value of an attribute or a text node is stored in a string node. This string node obtains the DeweyID of the text node resp. attribute node, extended by a division with value 1.

To illustrate the effect of these rules, we have applied them to the document fragment of Figure 1 under the assumption that all nodes are bulk-loaded. Table 1 shows the result using a distance value of 8; if appropriate, this value is also used in the examples of the next section.

Table 1: DeweyID assignment using distance 8

	node type	rule	DeweyID
bib	element	1	1
book	element	2	1.9
	attr. root	3	1.9.1
year	attribute	3	1.9.1.3
1994	string	4	1.9.1.3.1
id	attribute	3	1.9.1.5
1	string	4	1.9.1.5.1
title	element	2	1.9.9
	text	4	1.9.9.9
TCP/IP...	string	4	1.9.9.9.1
author	element	2	1.9.17
last	element	2	1.9.17.9
	text	4	1.9.17.9.9
Stevens	string	4	1.9.17.9.9.1
first	element	2	1.9.17.17
	text	4	1.9.17.17.9
W.	string	4	1.9.17.17.9.1
price	element	2	1.9.25
	text	4	1.9.25.9
65.95	string	4	1.9.25.9.1
book	element	2	1.17
book	element	2	1.25
publisher	element	2	1.25.9
last	last	2	1.25.9.9

3.2 Insertion of New Nodes

When new nodes are inserted at arbitrary logical positions, their DeweyIDs must reflect the intended document order as well as position, level, and type of node without enforcing modifications of DeweyIDs already present. For element nodes and text nodes, the same rules apply. In contrast, attribute roots, attribute nodes, and string nodes do not need special consideration by applying rule 3, because order and level properties do not matter.

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 1.9.25), addition of the distance value yields 1.9.33. In case, the last level consists of more than one division (indicated by even values), the first division of this level is increased by *distance-1* to obtain an odd value, i.e., the successor of 1.3.14.6.5 is 1.3.21.

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 1.9.5. 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 predecessor of 1.9.2.2.8.9 is 1.9.2.2.5. In case the first division of the last level is 3, it will be replaced by *2.distance+1* (see Section 2.2). For example, the predecessor of 1.9.3 receives 1.9.2.9.

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, having distance value 8, d_5 = 1.5.6.7.6.9. The reader is encouraged to construct DeweyIDs for further weird cases.

Let us summarize the advantages of the introduced form of ID assignment:

- Existing DeweyIDs allow the assignment of new IDs without the need to reorganize the IDs of nodes present. A relabeling after weird insertion histories[1] is only required, when implementation restrictions are violated, e.g., the max. key length in B*-trees.
- The DeweyID of the parent node can be determined in a very simple way; this is frequently needed, because a jump into the tree requires locking the entire ancestor path.
- Comparison of two DeweyIDs allows ordering of the resp. nodes in document order.
- 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.
- High distance values reduce the probability of overflows. They have 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.

4 Implementation 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 division value *distance*, the bulk-loaded millionth sibling would have a value of 10^6**distance* (e.g., requiring the representation of ~$8*10^6$ as an individual division value using the example in Table 1). 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. The critical question is how can we provide for such a scheme?

[1] 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.

4.1 Encoding Divisions

A division value O needs a variable-length representation which could be achieved in the simplest case by attaching a fixed-length field L_f representing the actual length of O. However, what is an adequate length value l_f for L_f? Because

$L_f < 2^{l_f}$, each division value is limited by $O < 2^{2^{l_f}}$.

Most division values are expected to be rather small (<100), but some of them could reach $>4*10^9$. While for the former example value $L_f = 7$ and $l_f = 3$ would be sufficient, the latter would require $L_f > 32$ and $l_f > 6$. Furthermore, whatever reasonable value for l_f is chosen, it is not space optimal and additionally introduces an implementation restriction.

Hence, we should make the length indicator itself of variable length. A straightforward approach is to spend a fixed-length field LL_f of length ll_f to describe the actual length of L_v resulting in an entry $LL_f|L_v|O$. A length ll_f of LL_f allows the representation of a length value in L_v

$ll_v < 2^{ll_f}$ limiting the length of divisions O to $L_O < 2^{2^{ll_f}}$. and their values to 2^{L_O}.

While $ll_f = 2$ restricts values of $O < 2^{16}$ and is not big enough for the general case, $ll_f = 3$ (allowing values of $O < 2^{256}$) definitely is for all practical applications. However, such a scheme carries the penalty for the frequent divisions with small values. Other approaches considered include Golomb codes and exponential Golomb codes [16] to allow for space-saving representations of O, but with similar disadvantages.

Another encoding approach [19] is using a k-based representation where the length of the encoding unit is determined by $m = \log_2 (k + 1)$. The idea is to reserve one m-bit code to represent the separator ".", while a sequence of m-bit codes is interpreted as a number with base k. An appropriate value is k = 3 delivering the following codes: 00: "0", 01: "1", 10: "2", 11: ".". Hence, 1.7.11 is encoded by 01 11 10 01 11 01 00 10 which reads $1*3^0$. $2*3^1+1*3^0$. $1*3^2+0*3^1+2*3^0$. Other codes with base k are possible. While k = 1 delivers a "funny" and very inefficient encoding, k = 7 may be appropriate for specific value distributions. [19] claims that k = 3 is superior to other Dewey encodings.

TL	C_0	O_0	C_1	O_1	. . .	C_k	O_k

Figure 3. DeweyID template

We hope to beat this encoding by Huffman codes which can be adjusted to the value distributions of the divisions used for DeweyIDs. Therefore, they offer an extra degree of freedom for optimization. We have designed an overall template for a DeweyID as illustrated in Figure 3. 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 respective entry of the B*-tree managing the collection of DeweyIDs on external storage. Each division consists of a C_i/O_i pair where, based on a code table, C_i allows to determine the length of O_i and O_i the actual value of the division.

4.2 Use of Huffman Codes

We use the idea of Huffman trees to determine codes standing for variable lengths for the C_i (without explicit length information). As a prerequisite, the set of C_i values must be prefix free. 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 O_0 value contained in the subsequent 6 bits. Encoding is performed in such a way that 000000 is assigned 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 O_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 of a division O_i with value 13. Hence, the second row in Table 2 delivers code 100 and $C_i = 4$. Because 13 is the sixth value of range 8-23, we yield an encoding of 0101, which is composed to the C_i/O_i encoding of 1000101.

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 can 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.

Table 2. 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	4440-69975
11101	20	69976-1118551
11110	24	1118552-17895767
11111	31	17895768-2165379414

Because DeweyIDs are stored in byte-structured sequences in B*-trees, storing a bit-encoded DeweyID in a byte structure may need a padding of bits for alignment reasons. By using Table 2, DeweyID 1.13.27, for example, results in the bit sequence 00011000.10110100.0011 where we have inserted dots to indicate byte boundaries for improved clarity. Because the last byte is incomplete, it is padded by zeros[2]. Consequently, the TL value is 3 and the stored DeweyID is 00011000.10110100.00110000.

5 Empirical Evaluation

To evaluate the performance of the DeweyID concept and especially that of our implementation, we have explored a variety of XML documents [10] as listed in Table 3. They represent a wide spectrum of different structural properties which were checked w.r.t. space consumption.

5.1 Consumption of Storage Space

In all cases, the DeweyIDs were assigned during bulk-loading where the distance value was systematically varied from the minimum of 2 (where almost no inserts are expected) to 256. The growth of the distance value reflects the probability that the nodes of the entire document are inserted randomly in a step-by-step manner and that even divisions (which represent a kind of overflow handling) should be avoided as far as possible.

Obviously, the number of divisions together with the chosen distance value exert the largest influence on the DeweyID length. Strongly depending on these factors, the

[2] Because value 000 is not used, padded zeros can be distinguished from encoded values.

Table 3. taDOM characteristics of the XML documents considered

file name	description	size (bytes)	number of element nodes	number of attributes	max. depth	∅–depth	max. fanout	∅–fan-out of elems
1) treebank_e.xml	Encoded DB of English records of Wall Street Journal	86082517	2437666	1	38	8.97	56385	2.33
2) psd7003.xml	DB of protein sequences	716853016	21305818	1290647	9	6.2	262527	3.99
3) customer.xml	Customers from TPC-H benchmark	515660	13501	1	5	3.92	1501	8.99
4) ebay.xml	Ebay auction data	35562	156	0	7	4.76	12	5.0
5) lineitem.xml	Line items from TPC-H benchmark	32295475	1022976	1	5	3.96	60176	17.0
6) mondial-3.0.xml	Geographical DB of diverse sources	1784825	22423	47423	8	5.25	955	4.43
7) nasa.xml	Astronomical data	25050288	476646	56317	10	6.62	2435	2.79
8) orders.xml	Orders from TPC-H Benchmark	5378845	150001	1	5	3.93	15001	10.0
9) SwissProt.xml	DB of protein sequences	114820211	2977031	2189859	7	4.9	50000	6.75
10) uwm.xml	Courses of a University Website	2337522	66729	6	7	4.83	2112	4.21

most expressive indicator for the quality of DeweyID encoding is the number of bytes per DeweyID needed in the average (∅–size). This again is essentially determined by the document's average depth and fanout (∅-depth, ∅-fanout). Having this interrelationship in mind, we have collected the most influential document properties summarized in Table 3.

Table 4. **Document classification**

	∅–fanout	f > 6	f > 3	f ≤ 3
∅–depth		high	medium	low
d > 8	high	**?**	?	**1**
d > 4.5	medium	**9**	**2**, 4, 6, 10	**7**
d ≤ 4.5	low	**3**, 5, 8	?	**?**

The ∅–size of DeweyIDs as a function of the distance parameter is shown in Figure 4. To facilitate interpretation, we have coarsely classified our document collection in Table 4 according to the factors ∅-depth and ∅-fanout into classifiers (low, medium, high). Some classes are either of little practical value (low/low) or will not occur in real applications (high/high). For the classes (high/medium) and (low/medium), we did not have representatives, but we can infer on their behavior. Of course, all graphs exhibit the same principal characteristics with strong storage space growth depending on increasing distance values where, however, the ∅-size of DeweyIDs is strongly correlated to the ∅-depth. Although computed for all 10 files, we focus for reasons of space limitations and clarity on 5 graphs in Figure 4 (marked bold in Table 4). Their comparison clearly reveals the strong dependency of ∅-size on ∅–depth of a document. Documents with lower ∅–depth (files 3, 5, and 8) are the clear "winners" in terms of short IDs, whereas the files 2, 4, 6, and 10 form the middle group. ∅–depth of 8.97 is the decisive factor making the IDs of file 1 the "losers" in terms of space consumption. File 9 (medium/high), but almost classified as low depth, is closer to the (low/*) group, whereas file 7 as the representative of (medium/low) is closer to the middle group.

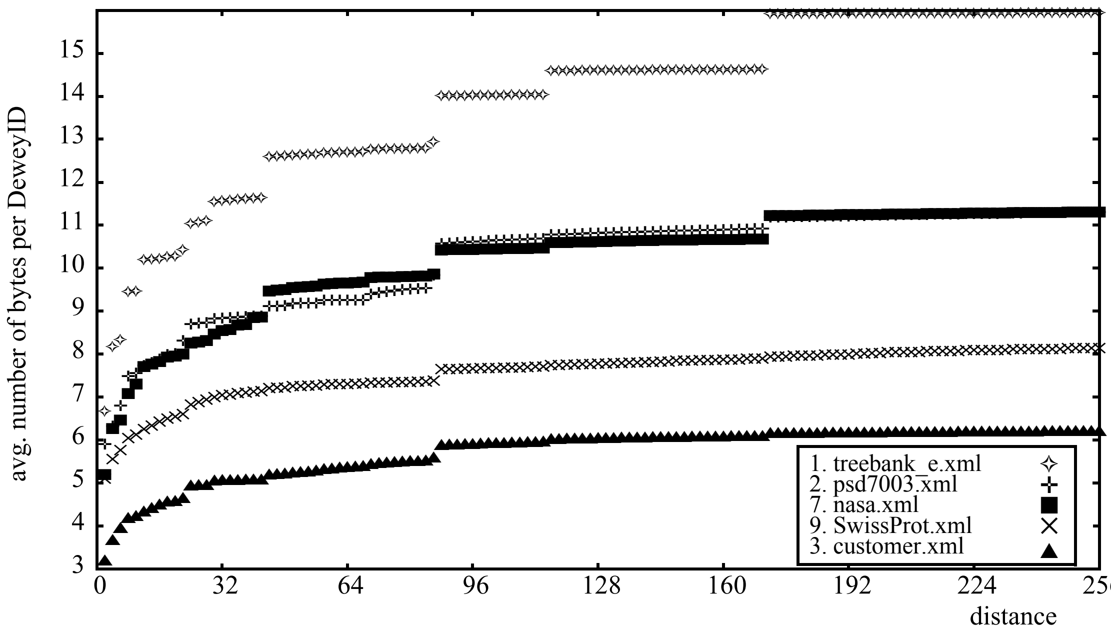

Figure 4. ∅−size of DeweyIDs grouped by the document's ∅−depth

A first space optimization is already included in the ∅-size values of Figure 4. Because all DeweyIDs start with "1.", we don't store this first division on disk and save 4 bits per DeweyID. To estimate the portion of ∅-size due to the *distance* parameter, we refer to a practical design space reasonably restricted by *distance = 32* in Figure 4. In Figure 5, the *average fraction* of the ∅-size caused by the distance parameter (*distance ≤ 32)* on ∅-size is illustrated for all 10 sample files. Note, these ∅-size values are comparable to those anticipated for TID encodings in relational DBMSs. The interesting measures to estimate this *distance* influence are ∅-size@dist(x) and *DistanceInfluence* per file which we have defined as DI(file) = (∅-size@dist(32) – ∅-size@dist(2))/∅-size@dist(2). Applied to files 1, 2, and 8 (the main classification axis), we yield DI(1) = 0.73, DI(2) = 0.49, and DI(8) = 0.39. DI(1) corresponds to the (high/low) case, where the average DeweyID is composed of more divisions, but smaller division values per level. Therefore, larger dis-

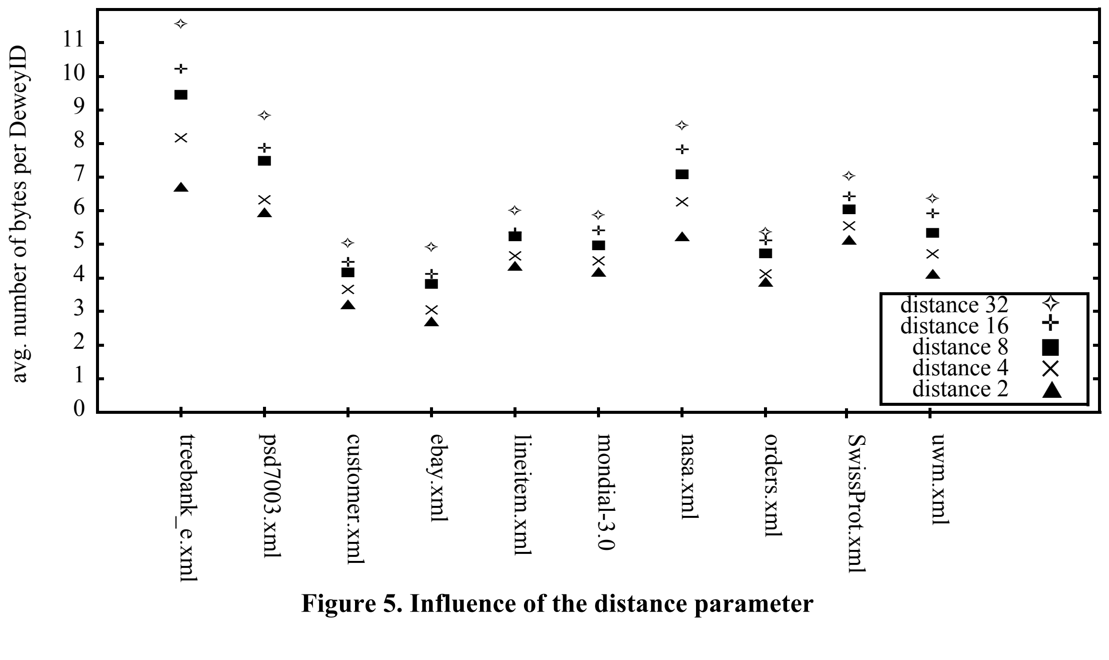

Figure 5. Influence of the distance parameter

tance values have stronger influence on ∅-size (higher DI). In contrast, DI(8) characterizes the (low/high) class with fewer division values per average DeweyID using higher division values. Because—relative to smaller division values—the representation of higher division values is more economical, *DistanceInfluence* is less distinctive (smaller DI). The (medium/medium) class is somewhere in the middle. Note, however, this influence is superposed in all cases by the document's attributes and their different labeling scheme.

XML documents converted from relational tables fall into the (low/high) class, i.e., their DeweyID size is less sensitive to the selection of larger distance values. In contrast, 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.

5.2 Frequency of Reorganizations

Reassignment of DeweyIDs (node relabeling) becomes necessary when the byte representation of any DeweyID exceeds a defined length, for example, the maximally allowed key length of the B*-tree implementation. Large distance values are the prime measure to avoid such undesirable events to the extent possible. To determine how many nodes can at least be inserted at the first level, before such an undesired event occurs, we construct a worst-case scenario which provokes a DeweyID to grow as fast as possible. When the boundary of the mechanism—defined as the difference of the actual DeweyID length after initial loading to the implementation-dependent key-length restriction—is pushed, a reorganization of the label assignment is needed to make room for further insertions. We start with the minimal scenario such that the hypothetical maximal DeweyID sizes (see Figure 6) can be considered as the threshold values to be passed.

The test scenario consists of the root 1 and a child with DeweyID *1.distance+1*; for all cases considered, their space consumption including padding is 2 bytes. The insertion of the sibling always takes place *before* the last inserted one. Hence, using distance value 16 and "halving the gap" as an example, the sequence of assigned DeweyIDs is

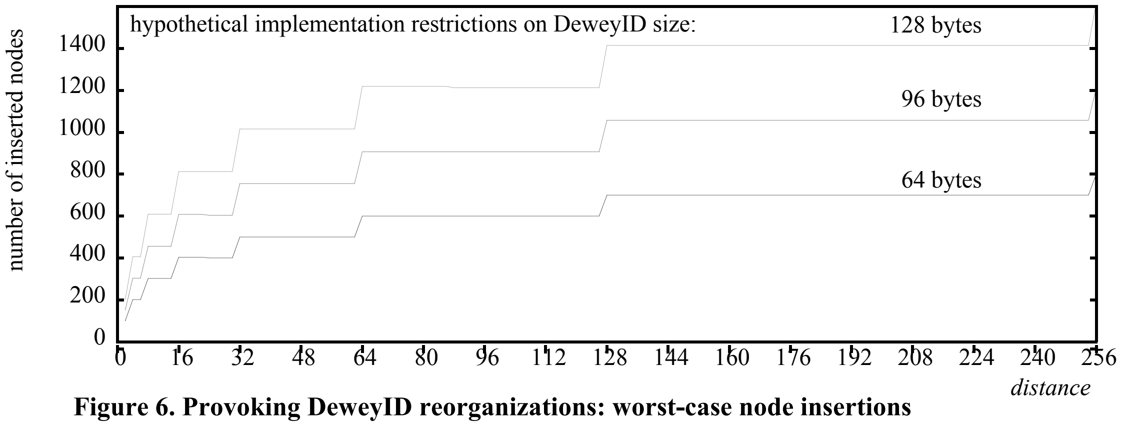

Figure 6. Provoking DeweyID reorganizations: worst-case node insertions

1.17, 1.9, 1.5, 1.3, 1.2.17, 1.2.9, 1.2.5, 1.2.3, 1.2.2.17, ... Therefore, the insertion history resembles the backward-oriented storage of documents. As illustrated in Figure 6, our labeling mechanism is quite stable. For example, using a distance value of 32 and having thresholds of ~(64-2) or ~(128-2) bytes, we can stress-insert >500 resp. >1000 nodes, before relabeling of at least a subtree has to be achieved.

When considering hypothetical implementation restrictions, we have to observe the maximum length (max-size) of a DeweyID occurring in a document which is, of course, strongly dependent on the max depth values (longest paths). In file 1 with max. depth = 38, we obtain ∅-size@dist(2) = 6.67 and ∅-size@dist(32) = 11.57 bytes, whereas the corresponding max-sizes are 22 and 46 bytes. Hence, reorganization frequency depends on the document's max-size, the distance parameter used in the DeweyIDs, and the location of (weird) insertions in the document. In summary, although some care has to be exercised, DeweyIDs are not challenged concerning their practical usability.

6 Conclusions and Outlook

In this paper, we have discussed the need for fast node identification when managing XML documents in databases. For dominant processing tasks such as declarative, index-based query evaluation, tree navigation, and concurrency control, fine-grained access to the documents is indispensable. Thus, efficient and effective node labeling resilient to arbitrary document modifications is of outmost importance. In this way, we have discovered how the Dewey order can be exploited for dynamic XML documents and have tailored the DeweyID mechanism in the lines of [11] to our proven taDOM storage model. An extensive empirical evaluation has explored the solution space for the critical design parameters and has pinpointed its practical usability even under weird application conditions.

So far, fine-grained management of XML documents and its effects on all query processing aspects are hardly discussed in the database literature. This is partly, because some existing systems use relatively coarse storage units [6, 12], and partly, because (almost all) XDBMS focus on query processing and neglect concurrency control at all [9]. In this sense, by elaborating on the DeweyID mechanism we have just found the key to fine-grained management of XML documents in databases. It is obvious that the deeper the document tree, the larger the DeweyID space consumption. But this may be compensated by processing advantages, because such keys carry the structure information of larger paths. As a consequence, the savings for concurrency control and index use correspond to these path lengths.

We strongly believe that the concept of DeweyIDs is tailored to the dichotomy of fast main memory and rather slow external storage devices (according to [3], disks are sequential devices) keeping the voluminous XML data. It enables a large share of XML processing in memory, because the DeweyIDs represent large portions of structure and content information supporting critical paths of query processing and concurrency control (e.g., lock acquisition for ancestors) in main memory and reducing external data access to a minimum. In this respect, it resembles—however, much more complex and effective—the proceeding in flat relational databases where TID lists stored in B*-tree indexes are used in Boolean set operations ($\cap$, $\cup$, –) to reduce the records to be fetched for query evaluation from external devices to an absolute minimum. On the other hand, the precise derivation of the ancestor path without disk access greatly improves locking costs.

There are many other issues that wait to be resolved: For example, we did not say much about the usefulness of optimization features offered. In the XDBMS access layer we currently evaluate the storage of DeweyIDs using prefix compression within the data pages. This physical optimization technique accomplishes an improved utilization of data pages (reducing storage space for documents) and diminishes the probability of XML fragment reorganizations. System-driven self-optimization from a more logical point of view (in contrast to the physical optimization) can be achieved by an analysis run before the actual bulk-loading of the documents. In this analysis phase, the expected average size of DeweyIDs, average document depth and fanout can be discovered and, in turn, used to automatically adjust the *distance* parameter for assigning new DeweyIDs. This adjustment could optionally accept user hints, e.g., the modification frequency for each document. Because of such application-specific DeweyID maintenance, we hope to gain optimal physical management of XML documents in our XDBMS.

References

1. S. Al-Khalifa et al.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. ICDE: 141 (2002)
2. T. Barclay, W. Chong, J. Gray: A Quick Look at SATA Disk Performance CoRR cs.DB/0403021: (2004)
3. N. Bruno, N. Koudas, D. Srivastava: Holistic Twig Joins: Optimal XML Pattern Matching. Proc. SIGMOD Conf.: 310-321 (2002)
4. M. Dewey: Dewey Decimal Classification System. http://www.mtsu.edu/~vvesper/dewey.html
5. E. Cohen, H. Kaplan, T. Milo: Labeling Dynamic XML Trees. PODS 2002: 271-281
6. T. Fiebig et al.: Natix: A Technology Overview. A. Chaudri et al. (eds.). LNCS 2593, Springer, 12-33 (2003)
7. M. Haustein, T. Härder: Optimizing Concurrent XML Processing, submitted (2005)
8. M. Haustein, T. Härder: Fine-Grained Management of Natively Stored XML Documents, submitted (2005), http://wwwdvs.informatik.uni-kl.de/pubs/p2005.html
9. H. V. Jagadish, S. Al-Khalifa, A. Chapman: TIMBER: A native XML database. The VLDB Journal 11(4): 274-291 (2002)
10. G. Miklau: XML Data Repository, http://www.cs.washington.edu/research/xml-datasets
11. P. E. O'Neil et al.: ORDPATHs: Insert-Friendly XML Node Labels. Proc. SIGMOD Conf.: 903-908 (2004)
12. H. Schöning: Tamino—A DBMS designed for XML. Proc. ICDE: 149-154 (2001)
13. A. Schmidt, F. Waas, M. Kersten: XMark: A Benchmark for XML Data Management. Proc. VLDB Conf.: 974-985 (2002)
14. A. Silberstein, H. He, K. Yi, J. Yang: BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. ICDE 2005: 285-296
15. I. Tatarinov et al.: Storing and Querying Ordered XML Using a Relational Database System. Proc. SIGMOD Conf.: 204-215 (2002)
16. J. Teuhola: A Compression Method for Clustered Bit Vectors. Information Processing Letters 7(6): 308-311 (1978)
17. W3C Recommendations. http://www.w3c.org (2004)
18. XQuery 1.0: An XML Query Language. W3C Working Draft (Oct. 2004)
19. J. X. Yu, D. Luo, X. Meng, H. Lu: Dynamically Updating XML Data: Numbering Scheme Revisited, World Wide Web: Internet and Web Inf. Systems, 8, 5-26, 2005