

An Efficient Infrastructure for Native Transactional XML Processing

Michael Haustein, Theo Härder *

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

Abstract

Implementation techniques for relational database management systems (DBMSs) have proven their efficiency and robustness in many existing systems. However, many of these concepts and mechanisms cannot be used when implementing a native XML DBMS (XDBMS) because of substantial differences in the processing properties of natively stored XML documents as compared to relational tables. Therefore, we have to develop new and appropriate techniques with ACID transaction guarantees tailored to the processing characteristics of tree documents and the operations on them.

For this reason, we want to provide for an efficient infrastructure of XDBMSs consisting of tree node addressing and indexing together with fine-grained locking of tree nodes. In this respect, our prime and novel contribution is to reveal the potential of our prefix-based node labeling called DeweyIDs supporting record addressing, indexing, and locking protocols. In this paper, we first sketch our version of prefix-based node labeling and summarize a quantitative study on them. An overview of our layered XDBMS architecture indicates the concepts and functionalities to be reused from relational DBMS implementations. The core part of the paper describes the infrastructural services for XML document storage with compressed DeweyIDs, the principles and methods for navigational and declarative processing of queries, as well as the lock modes and protocols to enable efficient collaboration. Selected empirical experiments evaluate the XTC system performance and support our system assessment.

Key words: Native XML DBMS, Transactional XML processing, XML document storage, Fine-grained locking, Prefix-based node labeling

* Corresponding author. Tel.: +49 631 205 4030; fax: +49 631 205 3299.
Email address: haerder@informatik.uni-kl.de (Theo Härder).
URL: wwwdvs.informatik.uni-kl.de (Theo Härder).

1 Introduction

Implementation techniques for relational DBMSs are well known and have proven their adequacy in many systems for a long time. The so-called data system in the DBMS architecture translates the declarative and set-oriented database (DB) requests to algebraic-based operators and these, in turn, have to be mapped to record-based scans and index accesses provided by the next lower system layer—the access system. Hence, the performance of the set-oriented DB operations is largely determined by the mechanisms and support structures of the access system whose operations must not be unduly hindered by inadequate concurrency control measures. To identify and access relational records, a simple RID (record identifier, also called tuple identifier (TID)) scheme together with appropriate indexes is sufficient in many cases. Because of the hierarchical DB structure of quite simple object types—segments – tables – records (rows)—multi-granularity locking protocols [15] (including extensions for indexes) are appropriate to guarantee minimum blocking delays while processing set-oriented read/write transaction workloads. However, this efficient and balanced interplay of the most important relational processing components cannot be expected when applied to native XML document management, because, in contrast to the relational model, we need to execute a variety of operations on large tree structures thereby following all relationship axes and observing the order among the tree nodes.

For this reason, reuse of relational or object-relational database management systems ((O)RDBMSs) does not provide a satisfactory solution in many cases, because they only manage structured data well. There is no effective and straightforward way for handling XML data. A “brute-force” mapping uses “long fields” or CLOBs where individual and direct access to single XML document nodes (elements or attributes) is not possible. Alternatively, an innumerable number of algorithms [35] maps semi-structured XML data to structured relational database tables and columns (the so-called “shredding”). In any case, there are no specific provisions to process transactions on XML documents and, at the same time, to efficiently provide the ACID properties [15]. Especially isolation of concurrent transactions in RDBMSs is tailored to the relational data model and does not take the semi-structured data model and the typical XML document processing (XDP) interfaces into account. CLOBs or “shredded” mappings of XML documents to relational tables may cause disastrous locking behavior, in particular, if relational systems lock entire pages or even entire tables as their minimal lock granularity.

So far, the focus of XDBMS implementations in most publications is on declarative query processing stated by general XPath or selected XQuery expressions [39]. Often, the efficient evaluation of the 13 relationship axes defined in the XQuery or XPath 2.0 language model is considered the most important XDBMS functionality which has to be supported by adequate range- or prefix-based node labeling schemes [6] and sophisticated indexing [1,5].

1.1 Missing XDBMS support

Two essential aspects are mostly (more often, rather entirely) neglected. First, there are different language models for XDP—already existing in the form of standards [38] like SAX, DOM, XPath, or XQuery—which necessarily lead to multi-lingual XDBMS interfaces. Hence, not only XQuery, but also DOM and SAX statements have to be processed on the same XML documents at the same time.¹ So far, most existing systems execute XPath and XQuery within their database engine where performance and achievable processing parallelism depends on the internal storage model used. The classic XML interfaces DOM and SAX, in turn, are provided, if at all, in the database drivers which means that the entire document has to be locked, reconstructed, and transported to the driver at the client side. A suitable XDBMS approach, however, should guarantee satisfactory performance for all kinds of language models, that is, suitable support for processing steps in declarative queries along the major axes such as *ancestor/descendent*, *previous-sibling/following-sibling*, etc. must be efficiently provided, while—starting from a context node—navigational operations such as *parent/first-child/last-child/previous-sibling/next-sibling* must be processed equally well.

Second, collaborative XDP becomes a "must", because XML documents permeate information systems and databases with increasing pace and are more and more concurrently used by growing numbers of users. Needless to say that the ACID transaction paradigm [19] proven in the relational world has to be observed. Hence, fine-grained isolation of large sets of read and write transactions becomes a prime design objective.

Very important for both kinds of DBMS-internal processing is an adequate node 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. Indeed, after a first implementation, we changed major parts of our XTC prototype system, because we are now convinced that suitable node labeling schemes are the key to fine-grained and transaction-protected management of XML documents.

¹ If multi-lingual XML interfaces should be supported, mapping of XML documents to relational tables and using SQL is no solution.

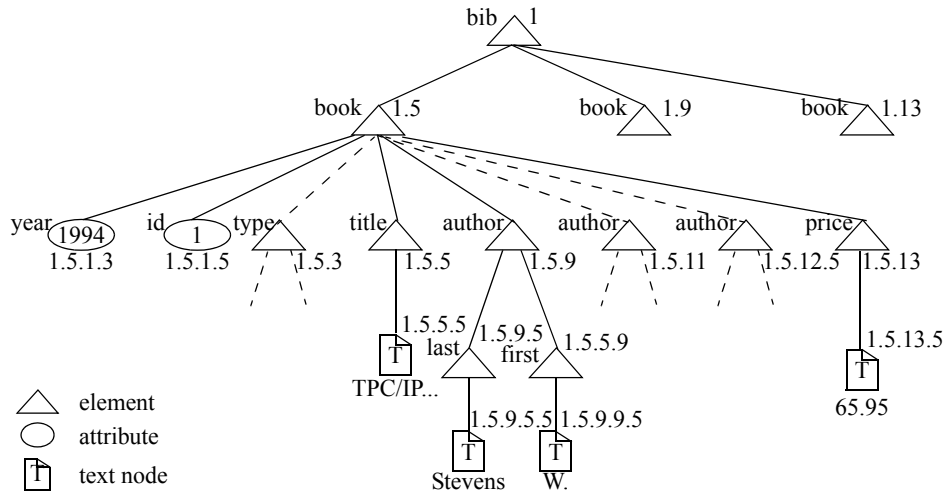


Fig. 1. A sample DOM tree labeled with DeweyIDs

1.2 Our Contribution

We believe that very few of the existing approaches and system implementations fulfill the strong requirements outlined above. None has taken the support of navigation into account or optimized it towards the physical document storage. A strong basis for the efficient execution of navigational and set-oriented operations is a prefix-based node labeling scheme which, on the one hand, allows the evaluation of all relationship axes without accessing the XML document and, on the other hand, allows optimized navigation on our XML document store. Furthermore, none of the XDBMS systems published so far has considered the implementation of fine-grained locking protocols. Therefore, literature on fine-grained locking methods for XML documents is hardly existing [25] or discloses some unfitted concepts [8,13]. Our contribution to isolate concurrent access to XML documents [23] uses locks tailored to the taDOM tree—a data model which extends the DOM tree [38]]—thereby supporting tunable, fine-grained lock granularity and lock escalation. Again, the prefix-based node labeling scheme greatly contributes to the efficiency and effectiveness of the locking protocol.

First, we summarize in Sect. 2 an empirical exploration of DeweyID use for tree node labeling which revealed surprising results. To convince ourselves that DeweyIDs are a salient concept which is also implementable, we have substantially changed our initial XTC implementation (XML Transaction Coordinator [22]) for which an overview is given in Sect. 3. To identify the potential of DeweyIDs, we outline their encoding and use for document storage and indexing in Sect. 4, whereas their expressiveness for navigational and declarative processing steps is sketched in Sect. 5. New lock modes and protocols for fine-grained isolation in XML trees are introduced and their performance is characterized by selected experiments in Sect. 6, before we summarize our findings and wrap up with conclusions.

Query operations	Update operations
– on contents:	– on contents:
<i>Element</i> getElementById (<i>String</i>)	<i>void</i> insertData (<i>Int</i> , <i>String</i>)
<i>NodeList</i> getElementsByTagName (<i>String</i>)	<i>void</i> appendData (<i>String</i>)
<i>boolean</i> hasAttribute (<i>String</i>)	<i>void</i> deleteData (<i>Int</i> , <i>Int</i>)
– on structure (navigation):	– on structure:
<i>NamedNodeMap</i> getAttributes()	<i>Node</i> insertBefore (<i>Node1</i> , <i>Node2</i>)
<i>NodeList</i> getChildNodes ()	<i>Node</i> removeChild (<i>Node</i>)
<i>Node</i> getFirstChild ()	<i>Node</i> replaceChild (<i>Node1</i> , <i>Node2</i>)

Fig. 2. Sample methods for the DOM API

2 Logical Data and Access Model

2.1 DOM Model

Our data model corresponds to that of the DOM standard [38] and is exemplified by the DOM fragment of Fig. 1 (disregard the node labels for the moment). A simple navigational language model for it is characterized by the APIs of DOM2 and DOM3 for which some of the more than 20 methods for querying and updating content and structure are exemplified in Fig. 2. In a multi-lingual XDBMSs, more powerful and declarative language models such as XPath and XQuery [39] are provided in addition. To achieve fine-granular access to document trees, declarative requests have to be translated into sequences of node accesses which again can be described by DOM operations. This mix of operations—especially in a collaborative setting with transactional guarantees—requires very efficient node identification, relationship determination among nodes, and navigation across nodes. For these reasons, an adequate node labeling scheme satisfying all these requirements is of outmost importance in an XDBMS.

2.2 Node Labeling

An intensive comparison of labeling schemes and their empirical evaluation [18] led us to redesign the existing mechanism in XTC based on a straightforward numbering scheme. DOM trees empowered with prefix-based node labels can be considered as an abstract access model much more flexible for XML document processing; it served as a powerful and adaptive structure to be implemented by our storage model in XTC (see Sect. 4.1). The prefix-based scheme for the labeling of tree nodes is based on the concept of Dewey order [9] characterized by Fig. 1. The abstract properties of Dewey order encoding—each label represents the path from the document’s root to the node and the local order w.r.t. the parent node; in addition, sparse numbering facilitates node insertions and deletions—are described in [7,35]. Refining this idea, a number of similar labeling schemes were

proposed which differ in some aspects such as overflow technique for dynamically inserted nodes, attribute node labeling, or encoding mechanism. Examples of such schemes² are DLN [3] or ORDPATH [32] developed for *Microsoft SQL Server*TM. Although similar to them, our scheme is characterized by some distinguishing features and is denoted DeweyIDs [18]; it refines the Dewey order mapping,³ provides for gaps in the labeling space, introduces an overflow mechanism when gaps for new insertions are in short supply, and revises the encoding for storing the IDs into data pages.

2.2.1 Constructing Immutable Node Labels

Because the node labeling scheme is a cornerstone of our processing infrastructure, we repeat essentials of the DeweyID concept for better comprehensibility [18]. Here we only can sketch the assignment method by examples; hence, node label 1.5.9.9.5 consists of five so-called *divisions* separated by dots (in the human readable format). The root node of the document is always labeled by value 1 and consists of only a single division. The children obtain the label of their parent and attach another division whose value increases from left to right. During the initial load of the tree, only *positive, odd* integers are assigned as division values. Counting odd division values of a label is used to determine the level (depth) of the labeled node. 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. 1, 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.

After initial loading of our sample document in Fig. 1, we have inserted the nodes of the second author and then of the third author. For *author₂*, 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 *author₃*. 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, *author₃* receives DeweyID

² Because all of these schemes including ours are adequate and equivalent for internal XML processing tasks, we have proposed the substitutional name stable path labeling identifiers (SPLIDs) for them, if their expressiveness is to be addressed.

³ 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 [14]. These extensions are neither stored on external storage nor visible at the XML APIs.

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 and 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 1.5.14.6.5 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 $2 \cdot \text{distance} + 1$, 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.⁴

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.

⁴ Basically, we could remove this technical restriction. Currently, XTC requires a lower bound of 1 for each division, because all internal operations writing numeric maintenance information into data pages use an interface which provides only 4-byte unsigned integers. This allows a higher number of initial nodes per level (more positive values can be addressed), but on the other hand no negative values at all can be stored.

2.2.2 Expressiveness 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 histories⁵ 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 easily provide the IDs of all ancestors, e.g., to enable intention locking of all nodes in the path up to the document root without any access to the document itself [23]. 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 *parent/child*, *ancestor/descendant*, *following-sibling/preceding-sibling*, *following/preceding* frequently occurring in XPath or XQuery path expressions. Our document representation together with element indexes provided facilitate navigation, as discussed in Sect. 5.1; hence, the five basic navigational axes *parent*, *previous-sibling*, *following-sibling*, *first-child*, and *last-child*, as specified in DOM [38], may be efficiently evaluated.

2.2.3 Empirical Evaluation of DeweyID Use

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.

An empirical study [29] 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.⁶ For this reason, we have empirically explored a variety of 11 XML documents [30] 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.

We have checked different encoding schemes for representing a DeweyID as a se-

⁵ 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.

⁶ The maximum depth of 135 found was due to an erroneous translation from html [32].

Table 1
DOM characteristics of the XML documents considered

file name (.xml)	description		size (bytes)
1	treebank_e	Encoded DB of English records of Wall Street Journal	86082517
2	psd7003	DB of protein sequences	716853016
3	dblp	Computer Science Index	284994162
4	customer	Customers from TPC-H benchmark	515660

file name	number of nodes			depth		fan-out of nodes		
	element	text	attribute	max.	\emptyset	max.	\emptyset	
1	treebank_e	2437666	1391845	1	37	8.44	56385	1.58
2	psd7003	21305818	15955109	1290647	8	5.68	262529	1.81
3	dblp	6662623	6013355	1375832	7	3.39	649080	2.11
4	customer	13501	12000	1	4	3.41	1501	1.89

quence of divisions and have systematically varied the distance parameter from 2 to 256 which definitely covers the reasonable solution space for practical DeweyID use. The winner of the encoding methods was based on Huffman codes primarily because of their ability to optimize the codes to the frequency distributions of the division values occurring in the set of node labels for a document. Representatives of various XML document classes are listed in Table 1 where *treebank* embodies a document of extraordinary depth and *psd7003* one of huge size and medium depth. In contrast, *dblp* represents the class of low-depth documents, whereas *customer* is a conversion of a relational table into an XML document. Some indicative results concerning average and maximum sizes of DeweyIDs occurring at a given distance parameter d (\emptyset -size@dist(d) and max-size@dist(d)) are extracted from [18] and listed in Table 2. Note, reasonable practical choices of the distance parameter value should be limited by dist(16) or dist(32) at the most. At this point we can already state that these results for \emptyset -size are better than expected and come for dist(2) close to TID sizes present in the relational world.

3 Database Architecture and Low-Level Services

So far, we have sketched essential aspects of XML processing at the data and access model level. In this section, we present the XTCserver architecture and the corresponding services to efficiently manage XML documents.

Table 2
Comparison of (uncompressed) \emptyset -sizes to max-sizes

DeweyID size in bytes	\emptyset -size				max-size			
	dist(2)	dist(16)	dist(32)	dist(256)	dist(2)	dist(16)	dist(32)	dist(256)
1. treebank_e	6.67	10.73	11.57	15.94	22	42	46	72
2. psd7003	5.61	8.38	8.84	11.30	8	12	13	17
3. dblp	4.58	6.06	6.12	7.16	7	9	10	13
4. customer	3.17	4.97	5.04	6.19	4	5	6	7

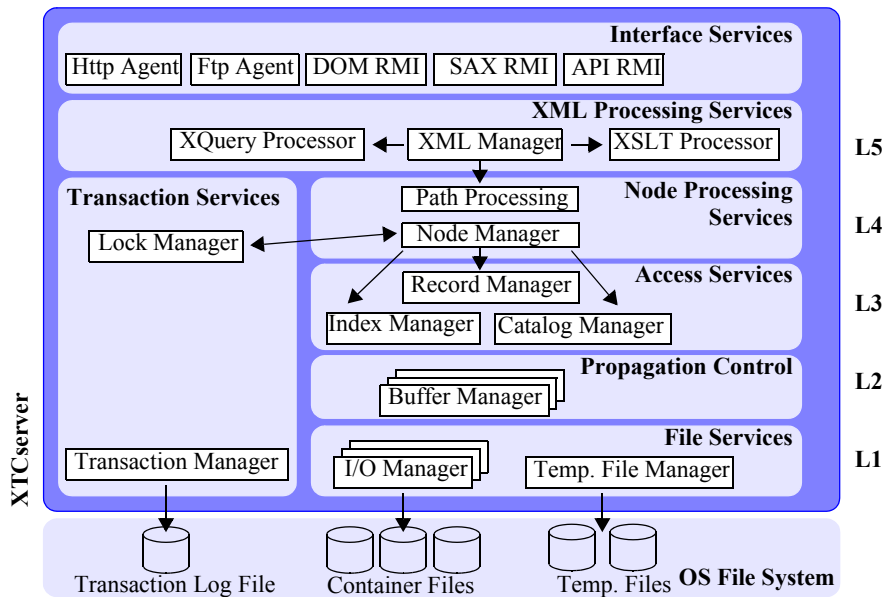


Fig. 3. XTC architecture overview

3.1 XTC Architecture

To implement a centralized database management system, Härder and Reuter have proposed a mapping model consisting of five layers [19]. The design of our XTC-server (depicted in Fig. 3) strictly adheres to this widely used reference architecture. Its mapping hierarchy embodies the major steps of dynamic abstraction from the level of physical storage up to the user interface. At the bottom, the database consists of huge volumes of bits stored on non-volatile storage devices, which are interpreted by the DBMS into meaningful information on which the user can operate. With each level of abstraction (stepping upwards), the objects become more complex, allowing more powerful operations and being constrained by a growing number of integrity rules. The uppermost interfaces both support the XML data model, in our case by providing navigational and declarative data access.

At the layers L1 and L2 responsible for management of external storage and DB buffers, reuse of concepts proven in the relational world is obvious. Hence, we have more or less adopted these DBMS mechanisms. The higher layers need to be adjusted to the specific needs of XML document representations and operations. In particular, the node manager implements the external navigational interface, transforms records from their physical representation within data pages to the external representation of XML nodes, and performs all required steps to isolate the node-based operations from concurrent transactions. The XML manager provides for the interfaces to process XML data, therefore implementing SAX, DOM, XPath, and XQuery.

Transaction management is performed by the *transaction manager*. It provides facilities for logging transaction operations onto disk to enable recovery after a system crash. Furthermore, the transaction manager cooperates with the lock manager which isolates transactional accesses on XML documents using fine-granular locks.

3.2 The Lower XTC Layers

Our storage layer offers an extensible file structure providing containers with varying lengths and configurable block sizes. DB buffers interface them with segments and “fix/unfix page” operations. Because a buffer can be equipped with specific page replacement algorithms, it can be tailored to the anticipated workload behavior. Furthermore, functionality for update propagation and logging & recovery transparent to the upper layers is made available. For example, physiological logging (combined with LSN entries in pages) is used to rollback transactions in case of transaction abort and to guarantee consistency of elementary actions in pages. As another salient feature, this mechanism enables update-in-place page propagation and crash recovery based on a kind of space-saving operation logging [17]. In summary, the layers L1 and L2 provide operations on pages and logging of variable-length entries/records such that well-known and efficient relational implementation methods can be reused.

With these methods the A (atomicity) and D (durability) properties of the ACID paradigm [6] are accomplished. In contrast, efficient I (isolation) and C (schema consistency) properties have to be achieved in the upper layers where objects and operations more and more approach the abstraction levels postulated at the XML APIs.

Collaborative processing of XML documents and, in turn, scalability in multi-user environments require a fine-granular DOM-tree storage representation flexible enough to adjust arbitrary insertions and deletions of subtrees as well as efficient support of query processing and concurrency control. These infrastructural features residing in layers L3 to L5 will be discussed in the next sections.

4 Access Services

Declarative and navigational processing has to be supported by fast indexed access to each document node, location of nodes by DeweyIDs, as well as navigation to parent/child/sibling nodes from the current context node. Our solution to these requirements includes

- indexed access
- maintenance of document order
- variable-length node representation
- representation of long fields.

As sketched in Fig. 3, an efficient implementational framework for accomplishing these requirements is provided in L3 primarily by B- and B*-trees. In XTC, these trees come with a variety of options. While indexed access and order maintenance are intrinsic properties of such trees, some optimization considerations are needed for the remaining issues. Variations of the entry layout for the nodes allow for single-document and multi-document stores, key compression, use of vocabularies, and specialized handling of short documents. Let us focus here on single-document stores.

4.1 Document Store

As illustrated in Fig. 4 by sketching the sample XML document of Fig. 1, a B-tree, the so-called *document index*, with key/pointer pairs (DeweyID+PagePtr) indexes the first node in each page of the *document container* consisting of a set of chained data pages.⁷ The document container preserves order and cluster property of each document; hence, the document index provides clustered document access via DeweyIDs as keys. Using 8K pages, the document index is usually of height 1 or 2. Because of reference locality in the B-tree while processing XML documents, most of the referenced tree pages are expected to reside in DB buffers—thus reducing external accesses to a minimum.

Although similar to the regions index in System RX [2], our approach basically differs in granularity. A DeweyID exactly addresses one node and each node can be directly accessed via the document index. Hence, locking and recovery also benefit from this physical alignment of single nodes. In contrast, system RX divides the entire document into regions (the corresponding papers do not explain what this is exactly based on). A region contains a fragment of the document and, as a

⁷ If we prefix the DeweyIDs with the DocID of the individual documents, we obtain a multi-document store where the documents can be easily partitioned into separate sets of container pages.

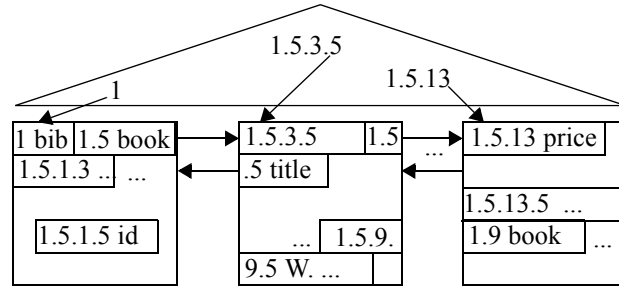


Fig. 4. Document store using B*-trees

consequence, locking and recovery are also constraint to this structure. System RX uses a version-based isolation mechanism which means that every single modification of a node leads to a new version of an entire region which has to be stored on disk. For concurrent document processing, the version-based approach allows each transaction a consistent view of the document without any blocking at any time, but this may cause a possibly large number of concurrently existing versions and distributed (non-clustered) regions at the physical data layer. The cluster property of documents allowing document reconstruction by sequential disk scans can be only maintained using additional reorganization tools which remove unused versions and realign regions (the periodic execution of such tools is also recommended in System RX papers).

Each node of an XML document (independent of its functionality as an element, attribute, or value node) is mapped to a node handled by the node manager, labeled by a DeweyID, and stored into the data pages of a B*-tree. Having in the order of 10^8 nodes and 10^5 pages, document containers need careful optimization considerations. All node formats are of variable length. An element node only consists of a key and a name part, whereas an attribute node has a value part in addition. Similarly, a text node has only a key and a value part. Because the key part consisting of a one-byte field *KL* (key length) and the encoded DeweyID is the Achilles heel of the storage representation (see Table 2 illustrating some DeweyID sizes), it must be reduced in a very efficient way.

4.1.1 Efficient DeweyID Compression

First of all, because all DeweyIDs start with “1.”, we do not need to store it and save 4 bits per DeweyID. Furthermore as visualized in Fig. 4, the document order inherent in the sequence of DeweyIDs lends itself to prefix compression in the key part. In addition, we have adjusted the Huffman encoding scheme to typical distributions of distance values in the DeweyIDs and aligned codes and value representations to byte boundaries.

Obviously, the lion share of optimization was achieved by the prefix compression of DeweyIDs. This is obvious for the document index and the document container, because they directly represent the dense Dewey order. However to a lesser degree,

prefix compression also leads to substantial storage savings in the node-reference indexes, although these indexes are non-dense [18]. Nevertheless, adjustment of the division encoding for its use under prefix compression obtained some further reduction effect. One important observation was that prefix compression cuts divisions in the path closer to the root often containing smaller values. Hence, we tailored the Huffman code in Table 3 to observe this behavior.

Now a DeweyID representation consists of a one-byte field *Lpip* (length of prefix inherited from predecessor) and the actually stored remainder (*Rem*) of the DeweyID. As detailed in [18], we gained an impressive reduction under this optimized prefix compression scheme; of course, the prefix-based labeling scheme lends itself to prefix compression by design. However, we did not expect to obtain such favorable and very stable results across 11 large documents. The average size of DeweyID storage representation (*Lpip+Rem*) for all documents in a wide spectrum of distance values considered varied between $\emptyset\text{-comp-size@dist}(2) \sim 2$ bytes and $\emptyset\text{-comp-size@dist}(256) \sim 3$ bytes. Hence, the standard key part including *KL* consumes 3–4 bytes. The standard name part consists of some type and administration data (1 byte) and a 2-byte field *Vind* used as index to a vocabulary [11] containing all element and attribute names of the system. Because in attribute and text nodes the value part may dominate the entire node representation, we choose the element node with a total of 6–7 bytes to give an indicative value for the standard node layout.

4.1.2 Squeezed Node Layout

Obviously, we do not need a full byte each for *KL* and *Lpip*. Often documents have only a limited number of element and attribute names such that *Vind* is not fully used. On the other hand, a squeezed node layout should not fail due to size restrictions. Therefore, we designed a specific scheme to optimize the storage consumption of these fields thereby observing alignment to byte boundaries. The encoding trick is to use a bit combination which doubles the length of the field (stepwise doubling). Hence, a 4-bit scheme for field *KL* is interpreted as illustrated by our example.

<i>KL</i> value in bytes	code
0	0000
...	...
14	1110
15	11110000
...	...

Using this scheme, we can represent *KL* and *Lpip* in a single byte. Furthermore, the name part only needs 5 bits for type and administration information. Hence, using 2 bytes, we can use 11 bits for *Vind* to index up to $(2^{11} - 1)$ names (without the need

Table 3

H1: Assigning codes to divisions

code	length	value range
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	270,549,120–68,990,025,855

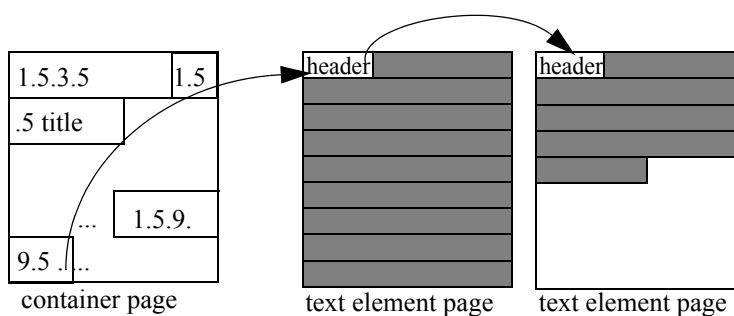


Fig. 5. Inserting long records

to increase the size of the index). As a consequence, we achieve the encoding of an entire element node ($KL+Lpip+Vind+Rem$ and type information) using only 4–5 bytes in the average. Note, Table 2 only contains the average sizes of DeweyIDs. A quick comparison with them confirms that the squeezed node layout dramatically reduces the storage consumption to a fraction of 0.25–0.40 of the original size.

4.1.3 Representation of Long Fields

The value part of a node is materialized up to a parameterized *max-val-size* together with the node as a string (of a given type). For text nodes, the size may exceed *max-val-size*; then it is stored in referenced mode where it is divided in parts each stored into a single page and reachable via a reference from its home page (see also long-field manager in DB2), as illustrated in Fig. 5.

Our record management algorithms guarantee effective and efficient page utilization. In particular, if entire XML documents are stored at a time, we achieve a very high page occupancy, because page splits are minimized.⁸ Even if the structure of these documents is modified later on, we may expect a degree of page filling of more than 96% in most cases.

⁸ Using XML documents ranging up to 100MB with structural accordance to the XMark benchmark documents, we have achieved an average page occupancy of 98%.

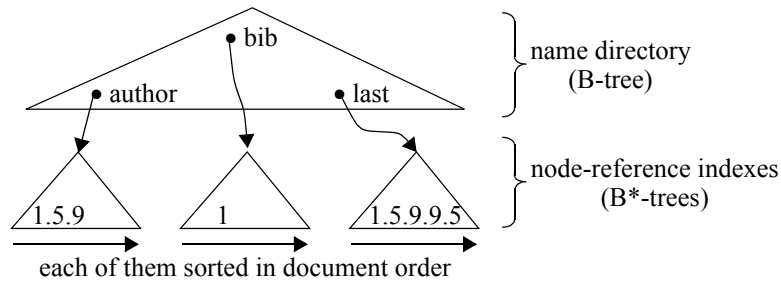


Fig. 6. Organization of the element index

4.2 Element Index

In addition to the document store, an *element index* is created consisting of a *name directory* with (potentially) all element names occurring in the XML document (Fig. 6); 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 again very effective.

In a first version, we implemented the element index in a single B*-tree where each element name was attached with a variable-length and potentially very long list of references (DeweyIDs) indicating the nodes in the document store carrying this element name. Handling and modification of these lists turned out to be clumsy and inefficient. Therefore, we improved the implementation by separating element names and references and managing the variable-length lists in dedicated B*-trees, that is, these lists are stored in doubly-chained leaf pages (similar to the container pages of the document store). In a sense, this is a two-step implementation using the already existing B*-tree. It turned out that many axes operators could take advantage of the indexed DeweyID access to the node references sorted in document order and, therefore, could be accelerated. For example, the location of all nodes having the same element name in the *child/descendent/following/following-sibling* axes of a context node, say 1.21.4711, can be accelerated, that is, the search procedure can locate the start leaf page via the node-reference index and scan the chained leaf pages until all references needed are found, whereas in the initial implementation the corresponding variable-length list had to be scanned from the beginning⁹.

The element indexes are heavily used for the evaluation of declarative XML queries which are typically decomposed into sequences of path processing steps (see Sect. 5.2) where each step corresponds to a structural binary join or, even more complex,

⁹ Together with this reorganization the locking protocol on indexes was improved. Instead of locking the entire path through the B*-tree, we use page latching and free the traversed page as soon as it is observed to be split-free [27]. For this new implementation, a throughput gain of 400% was measured in some applications [36].

to a holistic twig join [4]. Such steps involve axis operators on the nodes of a specified element name (so-called name tests). By accessing the corresponding node reference indexes, we obtain for a given element name ordered lists of DeweyIDs and, if necessary,—by accessing the document store—lists of nodes in document order. Hence, our index mechanism provides all “external” input sets for the join algorithms in document order.

Using our optimized prefix compression scheme, the average compressed DeweyID did not reach the reduction rate of the document store, because the strict document order is not reflected in the node-reference indexes. Nevertheless, with the exception of the extraordinary treebank document, we again obtained impressive results for all documents and distance values considered: \emptyset -comp-size@dist(2) \sim 3 bytes (*treebank* \sim 4 bytes) and \emptyset -comp-size@dist(256) \sim 6 bytes (*treebank* \sim 9 bytes). Hence as a rule of thumb, we may expect as an average entry size ($KL+Lpip+Rem$) for the standard scheme 4–7 bytes and for the squeezed scheme 3–6 bytes.

5 Node Processing and XML Processing Services

Selection and join algorithms based on index access via TID lists together with the use of fine-grained index locking boosted the performance of RDBMSs [17], because they reduced storage access and minimized blocking situations for concurrent transactions as far as possible. Both factors are even more critical in XDBMS.¹⁰ Hence, when designing such a system, we have to consider them very carefully.

5.1 Support of Navigation

Using the document store sketched in Fig. 4, the five basic navigational axes *parent*, *previous-sibling*, *following-sibling*, *first-child*, and *last-child*, as specified in DOM [38], 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. 4, 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 data pages of the document container, until we find *ps*—the first ID with the same parent

¹⁰ As compared to RDBMSs where most transactions typically operate on a larger amount of records within only a few tables, XDBMSs may have to cope with even more blocking situations because of combined index access together with frequent navigation (sibling to sibling, parent to child) between nodes of different types. Therefore, although accessing a larger number of different data types, these conflict situations have to be minimized.

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. Our access 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 an access to 1.5 suffices to locate the result; this is achieved by at most 3–4 page fetches via the document index sketched in Fig. 4. This alternative 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.

5.2 Processing Steps for Query Evaluation

For declarative access via query languages like XQuery, a set-at-a-time processing approach—or more accurately, sequence-at-a-time—and the use of the element index promise in some cases increased performance over a navigational evaluation strategy. Nevertheless, the basic DOM primitives are a fallback solution, if no index support is available. To illuminate the element index use for declarative access, let us consider a simple XQuery predicate that only contains forward and reverse step expressions with name tests: *axis1::name1/.../axisN::nameN*. XQuery contains 13 axes, 8 of which span the four main dimensions in an XML document: *parent-child*, *ancestor-descendant*, *preceding-sibling-following-sibling*, and *preceding-following*. For each axis, we provide index- and hash-based algorithms operating on a duplicate-free input sequence of nodes in document order, producing an output sequence with the same properties, and containing for the specified axis all nodes which passed the name test. Therefore, the evaluation of axes is closed in this group of algorithms and we can freely concatenate them to evaluate path expressions having the referenced structure. Our evaluation strategy follows the idea of structural joins [1] and twig joins [4] adjusted to DeweyIDs, and expanded to support the *preceding-sibling-following-sibling* and *preceding-following* dimensions.

Let us consider the *following-sibling* axis as an example. In Fig. 7, the nodes of the input sequence P, which may be the result of a former path step, are marked in a dark shade. Furthermore, the sequence of nodes F in our document that satisfy the name test for the current evaluation of the *following-sibling* axis carry the letter ‘n’. The DeweyIDs of all nodes having element name ‘n’ are retrieved using the element index. A problem of using the *following-sibling* axis is the possible generation of duplicates. For example, node 1.3.9 qualifies as a *following-sibling* for nodes 1.3.3, 1.3.5, and 1.3.7. Because duplicate removal is an expensive operation, our strategy is to avoid duplicates or remove them as early as possible. The evaluation algorithm

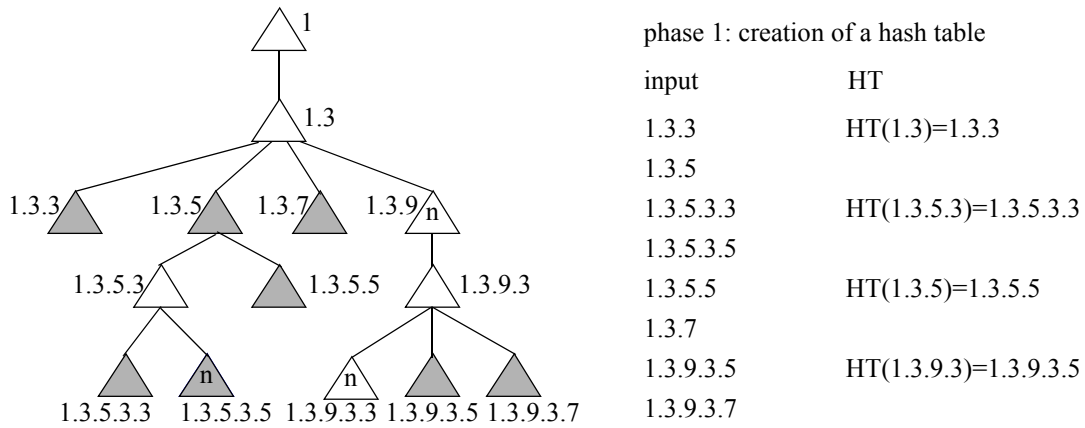


Fig. 7. Following-sibling algorithm

works as follows: In a first phase, input P is processed in document order. For each DeweyID d , a pair (key, value) as $(parent(d), d)$ is added to a hash table HT . If $parent(d)$ is already present in HT , d can be skipped (duplicate removal). Because we process P in document order, only the first sibling among a group of siblings is added to HT . In the second phase, we iterate over F and probe each ID f against HT [16]. If $parent(f)$ is contained in HT , we simply compare whether or not f is a *following-sibling* of $HT(parent(f))$. This comparison can easily be done by looking at the two DeweyIDs. Assume, the parent of $f=1.3.5.3.5$ is contained in HT and f is a *following-sibling* of $HT(1.3.5.3)$, then f will be included into the result sequence. For ID $f=1.3.9.3.3$, this test fails, because f is not a *following-sibling* of $1.3.9.3.5$. F is processed in document order, therefore, the output also obtains this order. Similar evaluation algorithms are provided for all other axes.

Because DeweyIDs carry a kind of “path index” for the labeled node and all nodes in the path to the root node, they provide substantial degrees of freedom for the evaluation of path processing steps. In general, it is possible to choose the *evaluation order* and proceed *top-down*, *bottom-up*, or start somewhere in the *middle* of a query execution plan (QEP). The appropriate strategy choice is strongly dependent on the join selectivities occurring in the individual path processing steps. Furthermore, the join algorithms can be based on indexing or hashing techniques [5,12,26], can be designed as semi-joins or full joins, and can be restricted to binary structural joins or can include additional predicate evaluations in the sense of twig joins. Given 8 types of axis operators, their combination with evaluation order and implementation approach may yield more than 200 individual operators. The set of physical operators implemented is contained in a “tool box” from which the query optimizer residing in layer L5 is supposed to construct from the incoming XML queries optimal QEPs for transactional workloads. This aspect again can be effectively supported by DeweyIDs, because they allow to access and lock minimal granules of XML documents thereby reducing the blocking potential for concurrent transactions to a minimum. Such an locking-aware query processing approach is described in [27,28].

5.3 Query Compilation and Optimization

Currently, only rudimentary functionality exists for layer L5 in XTC. Its prime task is to produce QEPs, i.e., to translate, optimize, and bind the multi-lingual requests—declarative as well as navigational—from the language models to the operations available at the logical access model interface (L4). For DOM and SAX requests, this task is straightforward. In contrast, XQuery or XPath requests will be a great challenge for cost-based optimizers for decades. Remember, for complex languages such as SQL:2003 (simpler than the current standard of XQuery), we have experienced a never-ending research and development history—for 30 years to date—and the present optimizers still are far from perfect. For example, selectivity estimation is much more complex, because more complex cardinality numbers and relationships for nodes in variable-depth subtrees have to be determined or estimated. So far, only conceptual proposals exist, e. g., [11,37], and hardly any empirical work was performed to validate them and to reveal their consequences for the entire system behavior. For example, a statistical guide containing children histograms or typed value histograms for selected nodes in a document tree are proposed in [10,33]; here, XQuery expressions are compiled into purely relational query plans and the cardinality estimates for a complex query are obtained by simulating the resulting relational plan equivalents. Furthermore, all current or future problems to be solved for relational DBMSs [14] will occur in XDBMSs, too.

6 Isolation in XML Databases

Fine-grained concurrency control is of utmost importance for collaborative use of XML documents. Although predicate locking of XQuery and XUpdate-like statements would be powerful and elegant, its implementation rapidly leads to severe drawbacks such as the need to acquire large lock granules, e.g., for predicate evaluations as shown in Fig. 7, and undecidability problems—a lesson learned from the (much simpler) relational world. To provide for a multi-lingual 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 interfaces such as DOM, because their operations correspond more or less directly to a navigational access model.

At the level of the access model, we provide about 20 operations for which a lock manager—transparent to the API—achieves the required transaction isolation by processing locking protocols including acquisition and maintenance of locks as well as protocol optimization through selection of adequate lock granularity, lock depth, i.e., the level in the tree up to which fine-grained locks are set, and lock escalation [23]. To enhance the potential parallelism, the lock manager slightly transforms the DOM tree to a so-called taDOM tree by virtually introducing two

new node types: *AttributeRoot* separates the various attribute nodes from their element node. Instead of locking all attribute nodes separately when `getAttributes()` is invoked, the lock manager obtains the same effect by a single lock on *AttributeRoot*. Hence, such a lock does not affect parallelism, but leads to more effective lock handling and, thus, potentially to better performance. A *string node*, in contrast, is attached to the respective text or attribute node and only contains the value of this node. Because reference to that value requires an explicit invocation of `getValue()` with a preceding lock request, a simple existence test on a text or attribute node avoids locking such nodes.¹¹ Hence, a transaction which is only navigating across such nodes will not be blocked, although a concurrent transaction may have modified them and may still hold exclusive locks on them.

The lock modes depend on the type of access to be performed, for which we have tailored the node lock compatibilities. When an XML document has to be traversed by navigational methods, then the actual navigation paths also need strict isolation. This means, a sequence of method calls must always obtain the same sequence of result nodes. To support this demand, we introduced so-called navigation locks. Furthermore, query access methods also need strict synchronization to accomplish the well-known *repeatable read* property and, in addition, the prevention of phantoms in rare cases [20].

6.1 Node Locks

While traversing or modifying an XML document, a transaction has to acquire a lock in an adequate mode for each node before accessing it. To observe the transaction properties, the granted locks have to be kept until end of transaction to guarantee isolated execution and, in case of a failure, rollback of the effects of all operations within a transaction. An exception can be made only for read locks in lower isolation levels where the user accepts the possibility of inconsistent reads.

Because the nodes in an XML document are organized by a tree structure, the principles of multi-granularity locking schemes can be applied [12]. The method calls of the different APIs used by the XML application are interpreted by the lock manager to select the appropriate intention lock modes for the entire ancestor path, similar to multi-granularity locking in relational environments (SQL). In particular, they prevent a sub-tree s from being locked in a mode incompatible to locks already granted to s or sub-trees of s . On the other hand, it makes perfectly sense to lock an intermediate node n for reads, while in the subtree of n another transaction may perform updates. For this and other reasons, we differentiate the read and write

¹¹ Some concepts such as axes and edge locks are not explained here to avoid too much confusion. The correctness of the above situation is guaranteed by axes locks which are used in a similar way as key-range locks (with some extensions). In this way, index access and querying “nodes which do not exist” can be kept consistent.

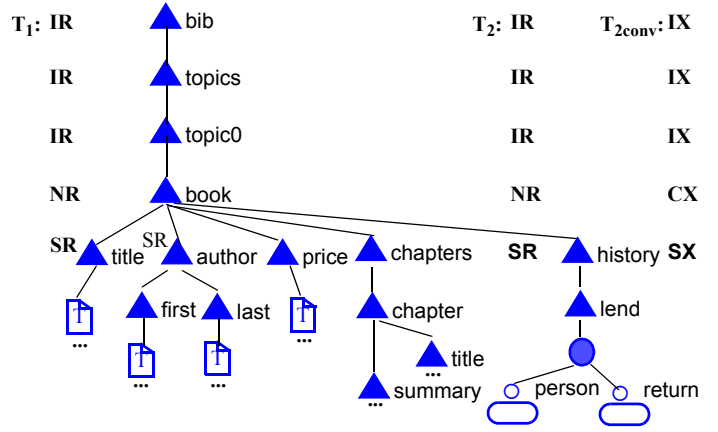
operations resulting in a richer variety of lock modes with (IR, NR, LR, SR) and (IX, CX, SX) modes, respectively. As in the multi-granularity scheme, the SU mode plays a special role, because it permits lock conversion. Fig. 8a gives an overview of the compatibility matrix for our lock modes whose effects can be sketched as follows:

- An IR lock mode (intention read) indicates the intention to read a node (lock modes NR, LR, SR) somewhere in the subtree (equal to the multi-granularity locking approach).
- An NR lock mode (node read) is requested for reading the context node. To isolate such a read access, an IR lock has to be acquired for each node in the ancestor path. Note, the NR mode takes over the role of IR combined with a specialized R, because it only locks the specified node, but not any descendant nodes.
- An LR lock mode (level read) locks the context node together with its direct-child nodes for shared access. For example, evaluating the child axis only requires an LR lock on the context node and not individual NR locks for all child nodes. Similarly, an LR lock requested for an attribute root node, locks all its attributes implicitly (to save lock requests for the `getAttributes()` operation).
- An SR lock mode (subtree read) is requested for the context node c as the root of subtree s to perform read operations on all nodes belonging to s . Hence, the entire subtree is granted for shared access. An SR lock is typically used if s is completely reconstructed, e.g., to be transferred as an XML fragment.
- An IX lock mode (intention exclusive) indicates the intent to perform write operations somewhere in the subtree (similar to the multi-granularity approach), but not on a direct-child node of the node being locked (in contrast to the CX lock).
- A CX lock mode (child exclusive) on context node c indicates the existence of an SX lock on some direct-child nodes and prohibits inconsistent locking states by preventing LR and SR locks. It does not prohibit other CX locks on c , because separate child nodes of c may be exclusively locked by other transactions (the compatibility is then decided on the child nodes themselves).
- An SU lock mode (subtree update option) supports a read operation on context node c with the option to convert the mode for subsequent write access. It can either be converted back to an SR read lock, if the inspection of c shows that no update action is needed, or to an SX lock after all potentially existing read locks of other transactions on c are released. Note, the asymmetry in the compatibility matrix among SU and (IR, NR, LR, SR) prevents granting further read locks on c , thereby enhancing protocol fairness by avoiding transaction starvation.
- To modify the context node c (updating its contents or deleting c and its entire subtree), an SX lock mode (subtree exclusive) is needed for c . It implies a CX lock for its parent node and an IX lock for all other ancestors up to the document root.

Note again, this differing behavior of CX and IX locks is needed to enable compatibility of IX and LR locks and to enforce incompatibility of CX and LR locks.

	-	IR	NR	LR	SR	IX	CX	SU	SX
IR	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	-	-
LR	+	+	+	+	+	+	-	-	-
SR	+	+	+	+	+	-	-	-	-
IX	+	+	+	+	-	+	+	-	-
CX	+	+	+	-	-	+	+	-	-
SU	+	+	+	+	+	-	-	-	-
SX	+	-	-	-	-	-	-	-	-

a) Compatibility matrix



b) Locking example

Fig. 8. DNode locking for the taDOM tree

Fig. 8b represents a cutout of the taDOM tree depicted in Fig. 8 and illustrates the resulting lock protocol using lock depth 4 for the following example: Transaction T_1 (TAqueryBook) uses an index and jumps to the *book* node to read all descendents of the book in document order. It sets an NR lock on *book* and IR locks on all ancestors up to the root. Then, it navigates to the first child and, because lock depth 4 is reached, it places an SR lock on *title*, reads the nodes of the subtree, and proceeds to the *author* node setting again an SR lock. In this situation, T_2 (TAlendandReturn) enters the system, also uses index-based access to the same *book* node, and locks it by NR and its ancestors by IR. Afterwards it forwards to the last child and locks the entire subtree *history* by SR (lock depth 4). Assume it decides to lend this book; then it has to attach an additional subtree *lend'* with attributes *person* and *return* under the *history* element. For this reason, a lock conversion to SX on *history* is needed which is propagated to the root by converting NR on *book* to CX and the remaining IR locks to IX, as shown as T_{2conv} in Fig. 8b.

Although this example is very simple, it reveals a certain kind of complexity to be anticipated in XML lock protocols. Our example also nicely demonstrates the effect of lock depth. If we would have chosen lock depth 3, T_1 would have set an SR lock on *book*. This lock, because incompatible with CX, would have prohibited the lock conversion of T_2 . Hence, fine-grained locking supported by the lock depth parameter enhances concurrency.

6.2 Lock Conversion

The compatibility matrix shown in Fig. 8a describes the compatibility of locks acquired on the same node by separate transactions. If a transaction T already holds a lock and requests a lock in a more restrictive or incomparable mode on the same node, we would have to keep two locks for T on this node. In general, k locks

	-	IR	NR	LR	SR	IX	CX	SU	SX
IR	IR	IR	NR	LR	SR	IX	CX	SU	SX
NR	NR	NR	NR	LR	SR	IX	CX	SU	SX
LR	LR	LR	LR	LR	SR	IX _{NR}	CX _{NR}	SU	SX
SR	SR	SR	SR	SR	SR	IX _{SR}	CX _{SR}	SR	SX
IX	IX	IX	IX	IX _{NR}	IX _{SR}	IX	CX	SX	SX
CX	CX	CX	CX	CX _{NR}	CX _{SR}	CX	CX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

Fig. 9. Lock conversion matrix

per transaction and node are conceivable. This proceeding would require longer lists of granted locks per node and a more complex run-time inspection algorithm checking for lock compatibility. Therefore, we replace all locks of a transaction per node with a single lock in a mode giving sufficient isolation. The corresponding rules are specified by the lock conversion matrix in Fig. 9, which determines the resulting lock for context node c , if a transaction already holds a lock (matrix header row) and requests a further lock (matrix header column) on c . A lock l_1 specified by an additional subscripted lock l_2 (e. g., CX_{NR}) means that the lock on c has to be replaced with l_1 and l_2 has to be acquired on each direct-child node of c .

In particular, conversion of the level locks becomes much more complex; its handling requires specialized locks to preserve the elegance and optimality of the new concept. An example for this conversion is as follows: Assume, a user starts a transaction requesting all child nodes of c which results in acquiring an LR lock on c . LR mode locks c and all direct-child nodes in shared mode. After that, the user wants to delete one of the previously determined child nodes. Therefore, the transaction acquires an SX lock on the corresponding child node and—applying the locking protocol—this requires the acquisition of a CX lock on c which already holds the LR lock. Using rule CX_{NR} specified in Fig. 9, the transaction has to convert the existing LR lock on c to a CX lock and to acquire an NR lock on each direct-child node of c (except the child node which will be locked for deletion by an SX lock).

6.3 Implementation of the Lock Manager

In this section, we consider the concepts implemented to provide lock management for XML documents. The actual lock management is based on a semaphore concept and is realized by so-called semaphore tables. Because we need to synchronize objects of varying types occurring in diverse system layers (e.g., pages and XML-document-related objects such as nodes, edges, and indexes), incomparable lock compatibilities, very short to very long lock durations, as well as differing access

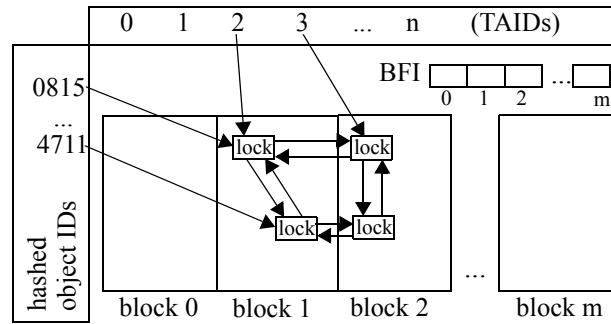


Fig. 10. Semaphore table architecture

frequencies, we decided to provide specialized semaphore tables for them (and not a common one). Size and contents of these tables can be configured. A main memory area is statically allocated at system start-up time where the specific semaphores on objects (identified by unique identifiers) are dynamically maintained. The layout of a semaphore table is described in detail in the following Section 6.3.1. The usage of such semaphore tables in different layers within a native XDBMS is discussed in Section 6.3.2.

6.3.1 Semaphore Table

A semaphore table maintains the semaphores (locks) dynamically acquired for a specified maximum number of transactions and lockable objects (e.g., XML nodes or database buffer pages). Semaphore types (lock modes) are assigned to a semaphore table by using the corresponding compatibility matrices. The acquired semaphores are doubly chained in the semaphore table and maintained for both the responsible transaction and locked object within a lock buffer (see Fig. 10) which makes requests and releases of semaphores very fast. For performance reasons, the lock buffer is additionally subdivided into several blocks, for which the free space information (BFI) is maintained separately. To save memory space and avoid block fragmentation, a bit-level encoded and, hence, compact fixed-size representation of semaphores is used.

Each transaction is mapped by its ID to a column header within the semaphore table whereas the row headers assigned to the objects are determined by a hash function. For a transaction accessing the semaphore table, operations are provided to add or replace semaphores on an object and to remove a semaphore after completing the processing of the locked object.

The row for an object is determined when the first semaphore is acquired for that object. “Collisions” while calculating the rows by our hash function are resolved by choosing the next free row for the object assignment as a collision handling technique. The object ID is also kept in each lock to accelerate the search while tracing an arbitrary transaction lock chain. This requires the ID of an object (e.g.,

an XML node) to be stable for the entire time period the object is locked (even if the object is modified while keeping an exclusive lock). If the last semaphore held on an object is removed (empty chain), the object ID is deleted from the hashed rows. Hence, the row can be reused for future object IDs.

The request of a semaphore requires a compatibility check which is performed by means of the compatibility matrix which is provided at semaphore table initialization. Either the requested semaphore is compatible to all semaphores already held on the object or the requested semaphore is not compatible to one or more semaphores already set. In the former case, the requested semaphore is immediately set on behalf of the invoking transaction, in the latter case, the semaphore is added to the lock chain with state *waiting* and the requesting transaction has to “sleep” until the incompatible semaphores are removed or replaced with compatible modes by the transactions owning them.

Additionally, the pending transaction is reported to the XDBMS transaction manager which maintains a wait-for graph enabling deadlock detection. Such centralized deadlock detection is necessary, because cyclic waiting relationships of transactions can occur across several semaphore tables residing in different XDBMS layers (e.g. T_1 is waiting for T_2 because of a *node* lock request and T_2 is waiting for T_1 because of a *page* fix request).

6.3.2 Lock Management

The actual lock management is performed by the lock manager which initializes a semaphore table to maintain the locks for the XML nodes, the previous- and next-sibling edges, the first- and last-child edges, and the index accesses (e.g., using an element index to get a list of DeweyIDs for an XQuery path evaluation). A further semaphore table is created by each DB buffer manager to maintain read and write fixes for database pages. The principal and most frequent lock management operations are

- allocate a new object: apply the hash function and locate a free row in the semaphore table
- remove an object: release the related row
- assign a semaphore: test compatibility (limited to the existing semaphores in the respective chain)
- replace an existing semaphore: similar to semaphore assignment
- remove a semaphore: check corresponding object chain whether pending transactions can be resumed
- remove a transaction: remove all semaphores from the transaction’s lock chain.

Locks on pages are “short” locks (kept only for the actual page access of an operation), whereas acquisition and release of node and edge locks (which correspond to the locking protocols described in Section 6.1) depend on the isolation levels *none*,

uncommitted, *committed*, *repeatable*, or *serializable* at which a transaction is running. Isolation level *none* means that no node or edge locks at all are requested for individual operations. This mode is used, for example, for internal transactions of the buffer managers (e.g., page accesses to create a savepoint require just a transactional context and no locks on higher semantic objects). At isolation level *uncommitted* only “long” write locks (IX, CX, X) are requested (until transaction commit). Hence, it is possible to access *uncommitted* data because no read locks are requested. Processing only *committed* data requires write locks and “short” read locks (NR, LR, SR, U). Isolation level *repeatable* keeps all locks up to the end of the running transaction and guarantees that multiple read operations on the same object always obtain the same result data within a transaction. *Serializable* avoids, in addition to isolation level *repeatable*, phantom anomalies to happen.

The lock manager provides methods for the acquisition of the different lock types on DB objects (XML nodes, XML node edges, index entries, and database pages) and forwards the lock requests to the adequate semaphore tables. Additionally, a single node lock request for context node *c* can result in further lock requests for nodes along the ancestor path or in lock requests for direct-child nodes of *c* (see the locking protocols described in Section 6.2). These additional lock requests are triggered by the lock manager; the requested nodes are identified by an internal transaction. Furthermore, if a lock-depth parameter is defined (Section 6.1), the lock request on a node below the given lock-depth level must be transformed by the lock manager into a lock request on the first ancestor node located at the specified lock-depth level.

6.4 Optimization of Lock Protocols

So far, we have restricted the discussion to the taDOM2 protocol covering all operations of the standard DOM2. In-depth empirical evaluation of the XTC system behavior suggested some protocol enhancements to increase parallelism. Such efforts are similar to the improvements achieved for the simple R/X protocol in relational DBMSs. However, due to space restrictions, we can only outline the type of optimization in this contribution; details and enhancements of the locking concepts can be found in [21]. Hence, taDOM2+ optimizes certain situations which may occur during lock conversions. Therefore, the four lock modes LRIX/SRIX (level/subtree read intention exclusive) and LRCX/SRCX (level/subtree read child exclusive) are provided in addition. Together with a lock compatibility matrix the related conversion rules have to be derived. The DOM3 standard introduces new operations, e.g., the renaming of inner tree nodes. Therefore, the taDOM3 protocol offers tailored and dedicated lock modes for the enhanced DOM standard and taDOM3+ additionally supports conversion in an optimal way. Because all taDOM* protocols are highly complex—taDOM3+ includes 20 lock modes and three modes for edges together with the related compatibilities and conversion rules—we can-

not just quickly cover them or repeat them for comprehension. We refer the interested reader to [21,26] where we have described and compared the protocols of the taDOM* group, proven their correctness, and run competitor protocols to demonstrate their superiority [21].¹² As for all other protocols referenced, they are not visible to the programmer; in contrast, they are automatically applied by the lock manager when protecting the actual DOM operation or the corresponding operation issued when a higher level request (e.g., XQuery or XPath) is mapped to the operations of the access system.

6.5 Performance Experiments

In a first series of experiments, we consider the basic costs of lock management described so far. It illustrates the tree-specific cost for fine-grained locking. Because we have executed the identical experiments with the simple sequential labeling scheme SEQIDs¹³ and with DeweyIDs, we can precisely compare the lock-related costs and, thus, emphasize the value of prefix-based labeling schemes for lock management. The other way around, our measurements reveal the cost penalty of inappropriate labeling schemes.

For this purpose, we use the *xmlgen* tool of the XMark benchmark project [34] to generate a variety of XML documents consisting of 5,000 up to 25,000 individual XML nodes. The documents are stored in our XDBMS and reconstructed by a client-side DOM application by a consecutive traversal in depth-first order under isolation levels *none*, *committed*, and *repeatable*. To reconstruct the document, the client application requests each node with a separate RMI call. Because of the complete traversal of all database engine layers for each node requested, this proceeding is very inefficient, indeed¹⁴, but it drastically reveals the lock management

¹² By using so-called meta-synchronization, XTC maps the meta-lock requests to the actual locking algorithm which is achieved by the lock manager's interface. Hence, exchanging the lock manager's interface implementation exchanges the system's complete XML locking mechanism. In this way, we could run XTC in our experiments with 12 different lock protocols. At the same time, all experiments were performed on the taDOM storage model optimized for fine-grained management of XML documents.

¹³ In our initial XTC version, we had implemented SEQIDs where node IDs were sequentially assigned as ascending integer values. These identifiers allow for stable node addressing, because newly inserted nodes just obtain a system-wide unique integer ID and their relationship to already existing nodes (sibling, child, etc.) can be determined by their physical position within a data page. However, this crude addressing scheme forces the lock manager to access the stored document for each lock request in order to identify all nodes of the ancestor path of a context node to be locked.

¹⁴ Document or fragment reconstruction is normally performed by locking the entire subtree for shared access, locating the physical address of the corresponding root node, and subsequent page scans to fetch all required nodes.

overhead for single node accesses. Isolation level *committed* certainly provides for higher degrees of concurrency with (potentially) lesser degrees of consistency of shared documents; when used, the programmer accepts a responsibility (because of early lock releases) to achieve full consistency. However, depending on the position of the node to be locked, it may cause much more overhead, because each individual node access acquires short read locks on all nodes along its ancestor path and their immediate release after the node is delivered to the client. In contrast, isolation level *repeatable read* sets long locks until transaction commit and, hence, does not need to repetitively lock and unlock ancestor nodes. In fact, they are already locked due to the depth-first traversal.¹⁵

These expectations are confirmed by the results of the first series of experiments as depicted in Fig. 11a. The potential performance gain of the reduced isolation level *committed* is contrasted by the dramatically increasing lock management overhead due to *repetitive* locking and releasing of locks along the entire ancestor path. While prefix-based labels (e.g., DeweyIDs) deliver all ancestor node IDs by a simple calculation (without access to the document), SEQIDs may cause a severe performance problem (depending on caching provisions), because the identification of ancestor nodes requires search on the document representation on external storage. For these reasons, substantial lock processing time is consumed under the SEQID scheme in mode *committed* ($> 500\%$ of the reconstruction time under isolation level *none* resp. *uncommitted*, i.e., without locking of read accesses), whereas the overhead for *repeatable read* is acceptable ($\sim 25\%$). Because a comparable number of lock requests is processed in the current XTC version (Fig. 11b), the cost saving as compared to the SEQID scheme can be attributed to the expressiveness of the DeweyID scheme. The locking overhead is still visible, but remains, even for the mode *committed*, in an acceptable range of $\sim 20\%$ of the processing costs. The mindful reader may ask why the reconstruction duration without isolation or in *repeatable read* mode is slightly increasing from Fig. 11a to Fig. 11b. This is mainly due to the revised taDOM data model [24] which—implemented in the current XTC version—maintains some more node types to achieve finer isolation granules than the simple DOM tree.

Of course, to guarantee highly consistent documents, *repeatable read* should be used for concurrent transactions. However, its penalty of longer lock durations has to be compensated by effective and fine-granular lock modes which coincides with the objectives of our proposal.

In a second series of experiments (which are based on DeweyIDs), we want to reveal the influence of lock management overhead on typical XML transaction processing in distributed environments. An interesting question was whether or not the high locking overhead for isolation level *committed* can be compensated by its

¹⁵ For locking a tree with n nodes, *repeatable read* acquires exactly n locks, whereas the locking overhead of *committed* is in $O(n^2)$.

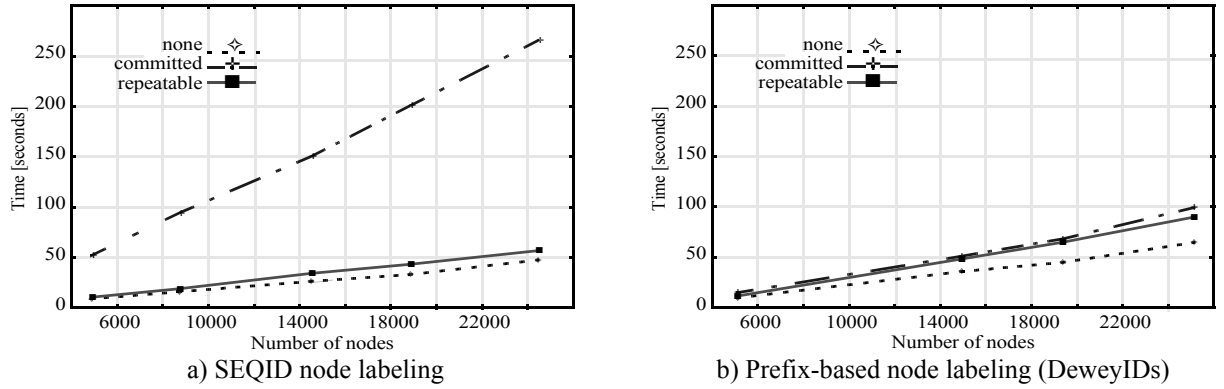


Fig. 11. Document reconstruction time

reduced blocking potential in multi-user mode. Surprisingly, isolation level *committed* seems to be inappropriate for large lock depths and short transactions.

For this scenario, we set up a benchmark with three client applications on separate machines and an XDBMS instance on a fourth machine. Each client is initiated with the same configuration and executes the same mix of transaction types. Each of these clients keeps 20 transactions running, i.e., it immediately starts a new one, if a transaction is finished. Hence, there are 60 transactions running concurrently on the server and processing common access patterns like node-to-node navigation, child- and descendant-axes evaluation, node value modifications, and fragment deletions. Fig. 12a shows the results of this benchmark run. As the most surprising behavior, *uncommitted* isolation leads in lower lock depths to a higher transaction throughput than even isolation level *none*. This can be explained by a closer look to the types of successfully committed transactions (not shown here): In lower lock depths nearly all of the transactions performing write accesses are blocking each other. As a consequence, many read-only transactions (acquiring in contrast to writer transactions no locks in *uncommitted* mode) are committing successfully. Their number is greater than the number of transactions committing without isolation¹⁶, because in the latter case the portion of in this mode non-blocked writing transactions requires much more system resources during processing (data page modifications, forcing log information on disk, etc.). Note, *committed* leads to a fewer number of successful transactions than the stronger isolation level *repeatable*. Although less consistency guarantees are given in mode *committed* to the user, the costs of separate acquisitions and immediate releases of entire lock paths for each operation reduce the transaction throughput. In short transactions, this overhead cannot be amortized by higher concurrency. After we had artificially increased transaction runtimes by programmed delays (in this way simulating human interaction), the transaction throughput meets the expectations of traditional transaction processing (see Fig. 12b): A lower degree of isolation leads to a higher transaction throughput.

¹⁶ Of course, processing transactions without isolation is inapplicable in real systems, because the atomicity property of transactions (in particular the transaction rollback) cannot be guaranteed. Here, we use this mode only for comparisons of lock management overhead.

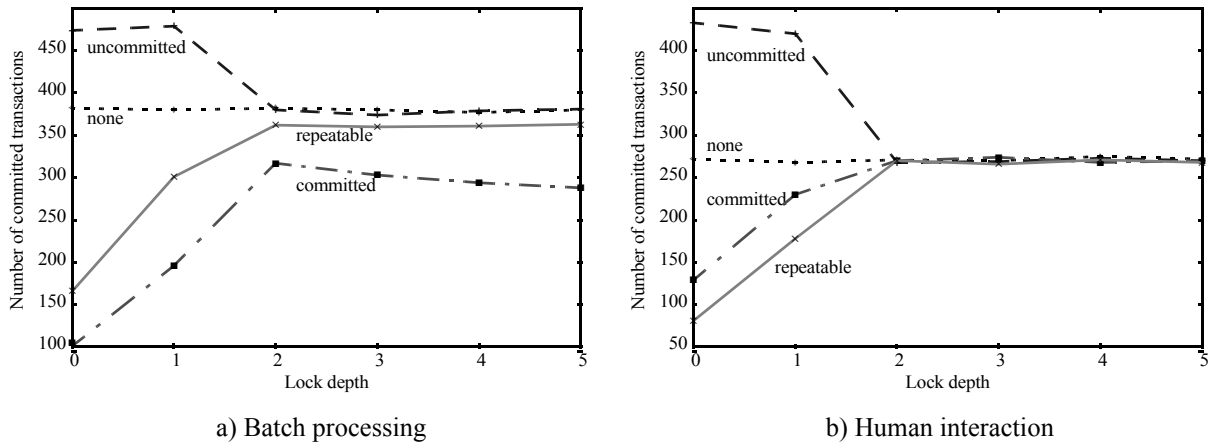


Fig. 12. Influence of isolation levels on transaction throughput

If no access conflicts are occurring (lock depths 2 and higher), the lock management costs do not influence the delayed transactions and transaction throughput is decoupled from the chosen isolation level.

7 Conclusions

In this paper, we primarily presented the infrastructure of our XTCserver, a prototype database system which supports transaction-protected native XML document processing via different language models (multi-lingual XML APIs). XTCserver is strictly designed in accordance to the well-known five-layer database architecture proven for relational DBMS implementations. File management in layer L1 stores any kind of data in block-structured container files whereas propagation control (L2) is implemented by a buffer manager which offers a potentially infinite page-structured address space to the next higher components. At these layers L1 and L2, we have more or less reused the “relational” concepts and adjusted them to the specific needs of XML document representations.

Reuse of functionality and concepts in higher layers is inappropriate in most cases, because internal processing of XML objects is guided by a tree model. The core mechanisms for native XML data storage are implemented in L3 by the access services which consist of the index, catalog, and record managers. All of them gain much more flexibility by a prefix-based node labeling mechanism, the special version of which we designed is called DeweyIDs. Because they remain valid for their lifetime, they can be used to directly access nodes (records) from anywhere, for example, using index structures to invert XML documents or node contents, independent of any document modifications. The node services in L4 again profit from their expressiveness and flexibility in such a way that a large tool box of physical join operators can be provided to be utilized for path processing steps of XML query evaluation. Finally in L5, query translation and optimization has to

build QEPs which select these physical operators depending on join selectivities of the actual documents, predicates of query to be evaluated, etc. in an optimal way.

Access and node services are complemented by lock management which is equally important for XDBMSs enabling collaborative use of XML documents. Therefore, we have designed, optimized, implemented, and evaluated the group of taDOM* protocols. So far, only a few proposals for locking protocols are published; using the idea of meta-synchronization we could implement and compare all of them in our XTC system under identical runtime conditions. In this contest, the taDOM* protocols clearly exhibited their superiority [21,24]. Due to all of our experiments, we believe that DeweyIDs (or equivalent prefix-based labeling schemes) are the key to fine-grained management of XML documents. We have empirically evaluated prefix compression schemes for them and shown that they can be efficiently implemented. Hence, they flexibly allow storing and indexing dynamic XML documents, enable efficient calculation of axes other than ancestor–descendant, improve path processing steps, support fine-grained in-memory-only lock management, etc. in several ways:

- They enhance the expressiveness of document and element indexes. For example, they are able to accelerate various navigation steps in the document itself and, in combination with a structural summary, to upgrade element index references to complete path index references.
- They introduce several new degrees of freedom when designing physical operators for path processing steps. This newly gained flexibility considerably increases the solution space for novel algorithms. In particular, they allow the computation and in-memory checking of all axes relationships required for the evaluation of XQuery and XPath expressions.
- They carry path information for all path nodes up to the root which is indispensable for the work of an XDBMS lock manager. Because of the prevalent index access, which is a necessity of all XML language models, jumps to inner tree nodes occur with very high frequency. Each of these accesses has to be isolated by appropriate intention locks along the entire ancestor path to achieve fine-granular locking. This salient feature supported by DeweyIDs is missing in all other labeling schemes proposed so far.

Future work mainly concentrates on improving and optimizing the functionality in L5. Refinement of DataGuides or statistical summaries is a prerequisite to design cost-based optimizers for XML queries. XTC will help to develop optimization strategies and to empirically evaluate them under realistic conditions in a complete XDBMS environment.

Acknowledgments.

We want 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.

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] K. S. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. M. Lohman, B. Lyle, F. Ozcan, H. Pirahesh, N. Seemann, T. C. Truong, B. van der Linden, B. Vickery, C. Zhang: System RX: One Part Relational, One Part XML. in: Proc. ACM SIGMOD Conf., 2005, pp. 347-358.
- [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] N. Bruno, N. Koudas, D. Srivastava: Holistic twig joins: optimal XML pattern matching. in: Proc. ACM SIGMOD Conf., 2002, pp. 310-321.
- [5] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, C. Zaniolo: Efficient Structural Joins on Indexed XML Documents. in: Proc. VLDB Conf., 2002, pp. 263-274.
- [6] 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.
- [7] E. Cohen, H. Kaplan, T. Milo: Labeling Dynamic XML Trees. in: Proc. ACM PODS Conf., 2002, pp. 271-281.
- [8] S. Dekeyser, J. Hidders: Path Locks for XML Document Collaboration. in: Proc. 3rd Conf. on Web Information Systems Engineering (WISE), Singapore, 2002, pp. 105-114.
- [9] M. Dewey: Dewey Decimal Classification System, <http://www.mtsu.edu/~vvesper/dewey.html>.
- [10] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, J. Siméon: StatiX: making XML count. in: Proc. ACM SIGMOD Conf., 2002, pp. 181-191.
- [11] R. Goldman, J. Widom: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, in: Proc. VLDB Conf., 1997, pp. 436-445.
- [12] G. Gottlob, C. Koch, R. Pichler: Efficient algorithms for processing XPath queries. ACM Trans. Database Syst. 30(2): 444-491 (2005).

- [13] T. Grabs, K. Böhm, H.-J. Schek: XMLTM: Efficient Transaction Management for XML Documents. in: Proc. ACM CIKM Conf., McLean, VA, 2002, pp. 142-152.
- [14] G. Graefe: Dynamic Query Evaluation Plans: Some Course Corrections? IEEE Data Eng. Bull. 23(2): 3-6 (2000).
- [15] J. Gray, A. Reuter: Transaction Processing: Concepts and Techniques, Morgan Kaufmann (1993).
- [16] T. Grust: Accelerating XPath location steps. In: Proc. ACM SIGMOD Conf., 2002, pp. 109-120.
- [17] T. Härder: DBMS Architecture—Still an Open Problem. in: Proc. Datenbanksysteme in Business, Technologie und Web (BTW 2005), LNI P-65, Springer, 2005, pp. 2-28.
- [18] T. Härder, M. Haustein, C. Mathis, M. Wagner: Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data & Knowledge Engineering, 2006.
- [19] T. Härder, A. Reuter: Principles of Transaction-Oriented Database Recovery. ACM Comput. Surv. 15(4): 287-317 (1983).
- [20] M. Haustein: Prevention of Phantoms in XML Database Management Systems Using Value-Based Axis Locks (in German). in: Proc. Berliner XML Tage, 2005, pp. 79-92.
- [21] M. Haustein: Fine-Granular Transaction Isolation in Native XML Database Management Systems (in German). Ph.D. Thesis, University of Kaiserslautern (2005).
- [22] M. Haustein, T. Härder: Fine-Grained Management of Natively Stored XML Documents. Internal Report, Dept. of Computer Science, University of Kaiserslautern, July 2004.
- [23] M. Haustein, T. Härder: Adjustable Transaction Isolation in XML Database Management Systems. in: Proc. 2nd Int. Database Symp., Toronto, Canada, LNCS 3186, Springer, Berlin, 2004, pp. 173-188.
- [24] M. Haustein, T. Härder: Optimizing Concurrent XML Processing. submitted (2006), <http://www.dvs.informatik.uni-kl.de/pubs/p2005.html>.
- [25] S. Helmer, C.-C. Kanne, G. Moerkotte. Evaluating Lock-Based Protocols for Cooperation on XML Documents. SIGMOD Record 33(1): 58-63 (2004).
- [26] H. Jiang, W. Wang, H. Lu, J. Xu Yu, Holistic Twig Joins on Indexed XML Documents. in: Proc. VLDB Conf., 2003, pp. 273-284.
- [27] C. Mathis, T. Härder, M. Haustein: Supporting Path Processing Steps of XML Queries by DeweyIDs. in: Proc. ACM SIGMOD Conf., Chicago, USA (2006).
- [28] C. Mathis, T. Härder: Hash-Based Structural Join Algorithms. in: Proc. DATA'06, Munich, March 2006.
- [29] L. Mignet, D. Barbosa, P. Veltri: The XML Web: a First Study. in: Proc. 12th Int. WWW Conf., Budapest (2003), <http://www.cs.toronto.edu/~mignet/Publications/www2003.pdf>.

- [30] G. Miklau: XML Data Repository, <http://www.cs.washington.edu/research/xmldatasets/>.
- [31] 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.
- [32] 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.
- [33] C. Sartiani: A General Framework for Estimating XML Query Cardinality. in: Proc. DBPL Conf., 2003, pp. 257-277.
- [34] Schmidt, A. et al.: XMark: A Benchmark for XML Data Management. in: Proc. 28th VLDB Conf., 2002, pp. 974-985.
- [35] I. Tatarinov et al.: Storing and Querying Ordered XML Using a Relational Database System, in: Proc. SIGMOD Conf., 2002, pp. 204-215.
- [36] M. Wagner: Storage Structures for native XML Database Management Systems (in German). Master Thesis, University of Kaiserslautern (2005), <http://www.dvs.informatik.uni-kl.de/pubs/DAsPAs/Wag05.DA.html>.
- [37] W. Wang, H. Jiang, H. Lu, J. X. Yu: Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. in: Proc. VLDB Conf., 2004, pp. 240-251.
- [38] W3C Recommendations. <http://www.w3c.org/> (2004).
- [39] XQuery 1.0: An XML Query Language. W3C Working Draft (Oct. 2004).



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.



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* (Kluwer), and *Transactions on Database Systems* (ACM). He served as a DFG (German Research Foundation) ex-

pert 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”.