Storing and Indexing XML Documents Upside Down

Christian Mathis, Theo Härder, Karsten Schmidt
Dept. of Computer Science, University of Kaiserslautern, Germany
{mathis | haerder | kschmidt}@informatik.uni-kl.de

Abstract

XML documents contain substantial redundancy in their structure part, because each path from the root node to a leaf node is explicitly represented and typically large sets of such path instances belong to a path class, i.e., the nodes of the path instances are labeled by the same sequence of element (or attribute) names. To save storage space and I/O cost, we want to get rid of this structural redundancy to the extent possible. While all known methods for the physical representation (storage) of XML documents proceed from the root via the element/attribute hierarchy (internal nodes) down to the leaves (values), we follow an *upside-down* approach which explicitly stores the values and only reconstructs the internal nodes, if needed. The cornerstones for such a solution are suitable node labels and a path synopsis which efficiently represents all path classes of an XML document. As a solution, we propose a compact internal storage format for native XML database systems where the inner structure of the stored documents is virtualized. Because this elementless storage format provides an efficient reconstruction of a document using its path synopsis, all processing properties are preserved and the semantics of navigational and declarative operations of XML languages remains unchanged. Adjusted indexes support the full spectrum of so-called content-and-structure single path queries. Apart from greatly reduced storage consumption, our approach demonstrates its superiority, compared to competing methods, not only for a substantial fraction of those queries, but also for storing, reconstructing, and navigating XML documents.[1]

Keywords. Storage formats, XML indexes, native XML database management systems, elementless XML storage, Path synopsis, Prefix-based node labeling

CR Subject Classification: E.2, H.2.2, H.2.4

1. Motivation

XML models semi-structured data and is becoming the standard for data exchange in many (Web) applications. Because the dramatically growing volumes thereby incurred have to be saved for a long time (for legal and other reasons) and messages are data, too, database systems are a proven technology to persistently store and conveniently manage such data. To avoid conversion, not only messages but also conventional DB data are increasingly kept in native XML format, often resulting in collections of huge XML documents. Furthermore, XML's flexibility, i.e., the ability to change the data mapping (freedom of cardinality determination, handling of varying or non-existing structures, etc.) without too much impact to applications [25], is also a driving factor to enable heterogeneous data stores and to facilitate data integration. For these reasons, XML databases currently get more and more momentum if data flexibility in various forms is a key requirement of the application.

As XML documents permeate information systems and databases with increasing pace, they are also increasingly used in a collaborative or even competitive way [26]. The challenge for database system development is to provide adequate and fine-grained management for these documents enabling efficient and concurrent read and write operations. In essence, this objective postulates the design and management of highly dynamic XML documents. Therefore, effective and storage-saving internal formats for document representation are urgently needed, too, primarily to reduce I/O and to enable caching of larger document fragments in memory.

Currently available relational or object-relational database management systems ((O)RDBMSs) 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 content- and structure-based XML document search as well as selective and direct access to single XML document nodes (elements or attributes) is not possible. Alternatively, a large variety of different storage formats were proposed, which take the structural richness of XML into account. In the past, the so-called "*shredding*" approach created an innumerable number of algorithms mapping semi-structured XML data to structured relational database tables and columns, where the individual XML documents are scattered across multiple relational tables. As an often claimed advantage, such a proceeding may facilitate query processing; when an XML query is mapped to SQL, the relational query processing infrastructure (compiler, optimizer, code generator) can be directly used for query evaluation. While CLOBs prevent maintenance of dynamic documents and multi-user read/write access at all, "shredded" mappings, in particular, of dynamic XML documents imply substantial maintenance overhead and cause disastrous transactional performance behavior, especially, if relational systems lock entire pages or even entire tables as their minimal lock granularity. For these reasons, native and fine-grained XML storage structures addressed in this paper provide a considerable optimization potential for dynamic XML documents and applications in multi-user transactional environments [17].

No matter which kind of storage structure is chosen, all properties of the original document have to be preserved [21]. In particular, when storing an XML document, the *round-trip property* must be

[1] Financial support by the Research Center $(CM)^2$ of the University of Kaiserslautern is acknowledged (http://cmcm.uni-kl.de).

Table 1.Characteristics of XML documents considered

Doc. name	Description	Size in MB	# elem. & attr. nodes	# content nodes	Max. depth	Avg. depth	# vocabulary names	# path classes	Size [KB] of path synopsis	# path instances	Avg. size of content nodes	Huffman compression
lineitem	LineItems from TPC-H benchmark	32.3	1,022,977	962,801	4	3.45	19	17	0.168	962,801	12.5	70.8 %
uniprot	Universal protein resource	1,821.0	81,983,492	53,502,972	7	4.53	89	121	1.328	53,502,972	24.0	76.8 %
dblp	Computer science index	330.0	9,070,558	8,345,289	7	3.39	41	153	1.509	8,345,289	17.0	69.9 %
psd-7003	DB of protein sequences	717.0	22,596,465	17,245,756	8	5.68	70	76	1.002	17,245,756	6.5	74.0 %
nasa	Astronomical data	25.8	532,967	359,993	9	6.08	70	73	0.982	371,593	20.9	64.4 %
treebank	English records of Wall Street Journal	89.5	2,437,667	1,391,845	37	8.44	251	220,894	3,323.0	1,391,846	33.4	75.8 %

guaranteed, that is, the database management system (DBMS) must be able to reconstruct the *original, ordered* document as delivered by a client application.

In recent years, different language models for XML processing—stream-based (SAX), navigational (DOM), or declarative (XPath, XQuery)—were standardized [40, 42] and their complex and rich query expressions have to be efficiently processed on documents of substantial size in any XML Database Management System (XDBMS). Scanning entire documents is certainly an extremely expensive query evaluation strategy and is disastrous in the presence of multi-user read/write transactions. Therefore, various forms of indexing become a prime issue for XML query processing. Performance would be boosted, if entire XML queries or large fractions thereof could be processed where access to (cached) indexes is sufficient and where document reference is avoided at all—as, for example, often achieved in relational DBMSs using B*-tree indexes and TID references. To approach optimal storage and indexing for rich-structured XML documents, detailed knowledge about their characteristics is useful.

1.1 XML document characteristics

An empirical study [31] gathered about 200,000 XML trees worldwide, where 99% have less than 8 levels, i. e., less than depth 8, which should be the primary goal of optimization. Almost all of the remaining 1% documents range between 8–30. Only a tiny fraction of the documents gathered has more than 30 levels. To gain some insight supporting our design idea, we have empirically explored a variety of XML documents [32]; we can only list a summary of the results for selected ones (due to page limitations). The document size is measured in the *plain* format where the XML document is stored in its external representation (i. e., as received by the database server from the client) without any compression technique applied (readable element and attribute names, empty spaces, etc., but without (internal) node labels). The first 5 rows in Table 1 contain a representative subset of all documents, called *reference documents*, and will be considered in the following. These documents range from a uniform XML structure of moderate depth (4)—representing a relational table—to GB-sized documents of rich XML structures and larger depths. As the last row entry, *treebank* is included to show an exotic outlier whose optimization is not in the focus of our approach.

Our goal is to ideally store only the document's content and to virtualize its structure part—of course, without losing any semantics of the document under all operations of the various XML language models. For this purpose, it is helpful to look into the details of typical document structures, which are summarized in Table 1. The number of distinct element and attribute names (vocabulary) gives an indication of the vocabulary size needed to replace the long external names by internal names, so-called VocIDs. All paths from the root to the leaves having the same sequence of element/attribute names form a path class. All path classes of an XML document can be captured in a small main-memory data structure (again with the exception of *treebank*). Such a concise description of the document's structure (see Figure 2) is a prerequisite for effective compression of the document's structure. If we can afford enough memory, structure virtualization can even be applied to *treebank* (see *size of path synopsis* in Table 1). When comparing them to the number of path instances, it becomes obvious that huge redundancy is introduced when all path instances are explicitly stored. In the well-known *dblp* document, for example, one of the dominating path classes */Bib/Paper/Author* currently has ~1,000,000 instances. For our reference documents, the number of path instances per path class ranges between ~5,000 and ~442,000 in the average (for *treebank*, it is only ~6). When considering the average depth (Avg. depth) of them, one immediately gets an impression of the storage redundancy created by such paths or the entire structural document part.

All these empirical results confirm that these structure-determining characteristics are of moderate size and can be kept in main memory to be used for optimization purposes concerning the design of XML document representations with virtualized structure (inner nodes) and search support on them using a variety of indexes.

1.2 Our contribution

The contributions of this paper can be summarized as follows:

- Elementless XML document storage is a novel method for the virtualization of the inner document nodes. It reduces storage overhead and response times for storing and reconstructing XML documents to a much larger extent than standard vocabulary-based approaches.
- We explore the role of SPLIDs [14] as prefix-based node labels for query processing and show that they are

efficiently applicable. Even navigation in virtualized document hierarchies is much better than the corresponding operations on physical document representations.

- We design tailored indexes for content-and-structure (CAS) single path queries. Our implementation supports unique, collective, and generic indexes enabling greater freedom and flexibility for index selection and allocation as well as mapping of queries to existing indexes.
- CAS indexes and virtualized XML documents are orthogonal. We combine both concepts and show how CAS queries can be effectively processed without accessing the documents to be queried and that they outperform any kind of conventional path operator use (e. g., binary structural joins or holistic twig joins).

All proposals considered are implemented in our prototype XDBMS called *XTC* (XML Transaction Coordinator), which we used as a testbed system for the empirical evaluation of benchmarks consisting of navigational and descriptive queries on a variety of large XML documents [32, 37].

In this paper, Section 2 gives a description of the essential concepts forming our elementless XML storage format. The operations supported are characterized in Section 3, before we describe the underlying index structures and possible generalizations in Section 4. Our search model for content-and-structure (CAS) single path queries as well as the way how CAS queries are matched to existing indexes are outlined in Section 5. This search model is assessed by empirical performance measurements, as described in Section 6; it clearly discloses the potential of our approach for response time improvements when navigational or declarative queries are evaluated. In Section 7, we position our contribution and compare it with the abundant and competing supply of related work, before we wrap up with conclusions.

2. Storing XML documents

Space-saving storage formats for XML documents always embody a performance-increasing measure in disk-based environments, because the well-known I/O bottleneck will be relieved in many DB-based application scenarios.

2.1 Applying XML compression

There exists a large body of scientific contributions dealing with XML compression technologies [34, 38] promising enormous gains in storage saving and, at the same time, enabling a kind of query processing (for very simple XPath expressions). However, all these approaches are coarse-granular, i. e., they are applied to the entire document (or file) at a time. Furthermore, they directly compress the *plain*, i. e., "verbose" representation, assume static and file-based application scenarios with single-user operations, are often context-dependent which requires large auxiliary data structures (vocabularies), and provoke potentially substantial compression/decompression overhead [38]. Therefore, these methods are not adequate for dynamic XML structures processed in a multi-user transactional DBMS context and, in turn, cannot be considered as storage formats for fine-grained tree-like structures where each tree node can be addressed and manipulated separately. In our usage scenario, structure had to be isolated from content in an orthogonal way. Because the efficiency of compression methods is highly dependent on the content size, we leave content compression out of consideration. Our reference documents are rather data-centric where content nodes carry short values or strings (see right-hand side of Table 1) and not entire papers or books (the document-centric case) and should be processed by regular DBMS operators. Instead of sophisticated block-based or word-based methods—reasonable for long fields—, cost-effective, character-based and context-free compression schemes like *Huffman* are appropriate and also accomplish homomorphous transformations guaranteeing that compressed and non-compressed documents can be processed by the same operations like indexing, searching, or validating [34]. If we apply a simple Huffman-based compression, we already gain considerable compression ratios varying from ~23% to ~35% on all documents, as listed in column *Huffman compression* of Table 1. Note, it also reduces the storage time up to ~15% compared to uncompressed content, because less I/O is needed to store the document on disk.

In our proposal, the major space-saving effect comes from structure virtualization in XML documents, which is independent of whether or not content compression is applied to the document.

While all known methods for the physical representation of XML documents proceed from the root via the internal nodes down to the leaves, this is reversed by our *upside-down* approach. We only store the leaves (content values) explicitly and reconstruct the internal nodes on demand. The cornerstones for such a solution are suitable node labels and a path synopsis which efficiently represents all path classes of the XML document.

2.2 Node labels

Our node identification mechanism rests on a prefix-based labeling scheme for which a few variations are proposed: OrdPath [35], DeweyID [14], or DLN [5] are adequate and equivalent for our use and are superior to other mechanisms, e.g., range-based labeling schemes. Therefore, we refer to such a scheme (the dot-separated integer lists in Figure 1) and prefer to use the generic acronym SPLID (Stable Path Labeling IDentifier) for it. Here, we can only summarize its abstract properties (please refer to the references for in-detail information): A SPLID contains the SPLID of its parent as a prefix. Therefore, the computation of all ancestors (ancestor path reconstruction) does not require document access (in contrast to range-based numbering schemes [7]). SPLIDs are immutable, i. e., they do not change upon (structural) modifications [35]. Therefore, they support dynamic documents. Furthermore, they facilitate query processing, because their lexicographical order represents the document order (for sorting) and, given two SPLIDs, all XPath axis relations can be easily deduced. SPLIDs support efficient hierarchical locking protocols due to cheap ancestor path reconstruction [16]. Because they lend themselves to prefix compression, SPLIDs can be efficiently stored. The empirical analysis in [14] obtained acceptable prefix-compression results ranging in the average from 3 to 6 bytes for a label. Therefore, we think SPLIDs are comparable to TIDs and can be processed similarly.

2.3 Path synopsis

Our path synopsis (see Figure 2) is an *unordered*[2] structural summary of all (sub)paths of the document. Each node defines a path class which typically corresponds to a considerable number of instances in the document. To distinguish path classes, we number

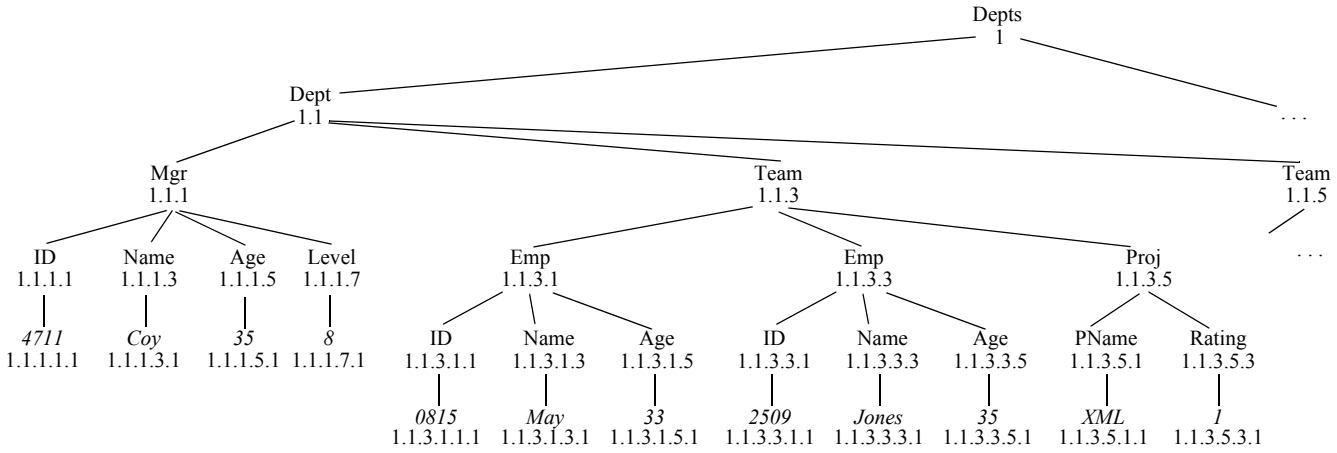

Figure 1. XML fragment labeled with SPLIDs

them with so-called Path Class References (PCRs) that serve as a simple and effective path class encoding[3].

The base idea of a path synopsis is similar to that of a DataGuide which, however, is used as a structure overview for the user, for storing statistical document information, and, thus, enabling query optimization [11]. In addition to that, our primary use of a path synopsis is for structure virtualization, hierarchical locking, and empowering indexing, compact storage, and query processing. It obtains its full expressiveness by the interplay of PCRs and SPLIDs: a SPLID delivers all SPLIDs of its ancestor, while a PCR connected to a SPLID identifies the path class a SPLID-identified node belongs to. Thus, we represent a node reference by a SPLID (describing its unique position in the document) and the PCR of the node. Therefore, it is easy to reconstruct the specific instance of the path class it belongs to. Having an index entry (Key, SPLID, PCR), the entire path instance of the related node can be reconstructed, e.g., (Coy, 1.1.1.3.1, 5) enables to directly compute */Depts/Dept/Mgr/Name/"Coy"* without document access. This usage of the path synopsis indicates its central role in all structural references and operations. To increase its flexibility, we provide indexed access via PCRs and hash access using leaf node names. Additional links between vocabulary IDs (VocIDs) and their occurrences within the path synopsis offer direct entry points for

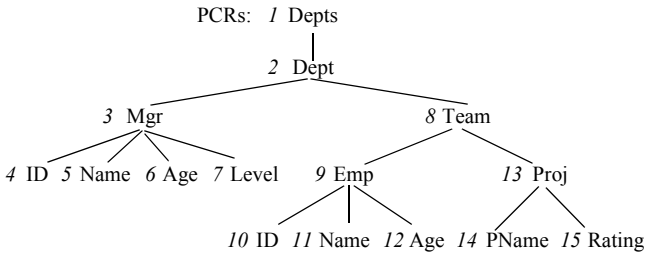

Figure 2. Path synopsis

[2] Because in the following, we are only interested in the path up to the root for a given PCR, the relative order among siblings is not relevant, e. g., all permutations of elements *ID*, *Name*, *Age*, and *Level* as children of *Mgr* may appear in the document.

[3] If a node has an *empty* value in some path class instance, the respective node type must carry a PCR to map the empty value to the correct path.

further navigational steps and matching/searching operations starting at non-leaf nodes. Because order is not important for a path synopsis, evolution of the document—by adding new path classes—only leads to simple adjustments in a path synopsis and, most important, leaves the existing PCRs unaffected.

The size of the synopsis obviously depends on the structural complexity of the document. It usually can be stored using a few consecutive pages on external memory (see Table 1). For fast access, it should reside in a small data structure kept in main memory.

2.4 Elementless document storage

The logical representation of XML documents, as visualized in Figure 1, is frequently used to derive a more or less direct mapping to a physical representation. A standard approach is to replace all element and attribute names by space-saving internal identifiers which are derived from a vocabulary (with VocIDs). Nevertheless, each inner node at least contains the SPLID and some type indicator (element, attribute) besides the VocID, represented as a variable-length entry. Therefore, substantial storage overhead is caused by such a physical document representation.

Our goal is to only store the content part of a document and to save the explicit storage of its structure by *virtualizing* it as depicted in Figure 3, where SPLIDs are only shown for some selected leaf values. This means whenever a reference to the document structure, to an inner node, or to a path is required or an operation is applied to them, the desired concept is recomputed. While preserving all document properties, this has to be accomplished in such a way *as if it would be physically present*.

The key idea is to automatically derive for each value in the leaf nodes the complete path it belongs to up to the root. As discussed in Section 2.3, a SPLID together with a PCR allows to evolve the entire document path to the root thereby extracting the related element names from the path synopsis. Hence, using all leaf nodes and their SPLIDs, the entire document can be efficiently reconstructed in its original form, whenever needed for output. It remains to show that all operations on the virtual document structure such as indexing, look-up, navigation, etc. are equivalent and more efficient as compared to the operations on the document's physical structure.

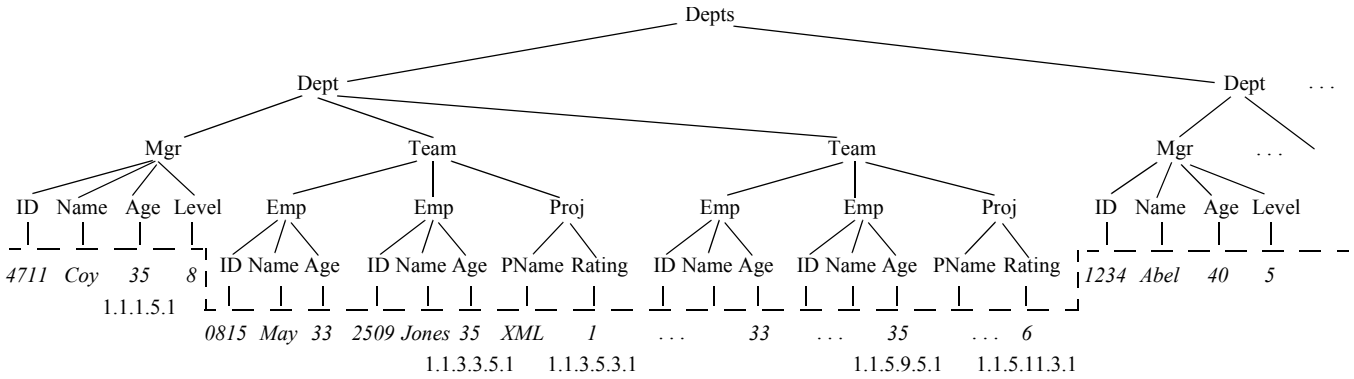

Figure 3. Document represented as Elementless Storage (only nodes below the dashed line are physically stored)

For an XML document, only its content nodes (leaf nodes) are stored in document order using a container as a set of doubly chained pages (called document container). The stored node format is of variable length and is composed of (SPLID, PCR, value). The value part of a node is materialized up to a parameterized *max-val-size* 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. To provide efficient access to the individual nodes via their SPLIDs, we add a document index using SPLIDs as separator keys to the document container. The resulting B*-tree and its split/merge mechanism take care of the storage management in case of modifications in the XML document.

The conceptual physical representation of a fragment of our XML example is illustrated in Figure 4. An essential performance aspect reducing storage overhead and disk accesses is the compact representation of the SPLIDs within a page. Because all SPLIDs of leaf nodes begin and end with a division value '1', we drop both enclosing '1's from each stored SPLID and add them when reconstructing the original SPLID. For the remaining divisions of a SPLID, we apply prefix compression.

Is all this optimization effort worth it? To answer this question, we implemented various storage schemes and measured the space saving for a large collection of documents [15]. Again, here we present only the results for the reference documents which cover the representative spectrum of achievable savings. Figure 5 illustrates the storage consumption of the different approaches and compares it to the *plain* format (100%). Because all nodes are stored as variable-length records, some administrative overhead (admin) is needed for type descriptors, byte alignment, etc. The *full* approach (uncompressed structure and content) labels all nodes with SPLIDs and uses VocIDs (2 bytes) instead of element and attribute names. As opposed to *full*, *pc* applies prefix compression to all SPLIDs which is very effective due to the document order of all nodes (structure and content) which still carry a VocID. Finally, *eless* refers to our elementless storage scheme where only the content nodes (together with prefix-compressed SPLIDs) are stored in the document container (see Figure 4). To reconstruct the related path, a PCR (1 byte) is added to each content node.

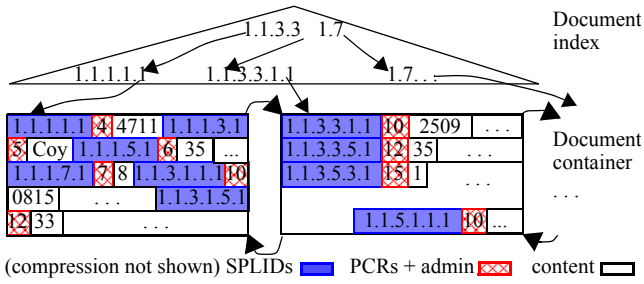

Figure 4. Stored XML document

Here, we focus on the relative saving regarding the structure part only. It is surprising that the *full* approach does not always achieve storage space reduction (*psd7003*); the saving from VocID usage is compensated by the need for node labels. In general, space saving of *full* seems to be less than ~35% compared to *plain*. However, storage gain from *full* to *pc* and *full* to *eless* is substantial. The largest share of this saving is due to prefix compression of the SPLIDs (darker or blue-colored fractions) which reduces the storage space needed for node labels in all cases to less than 25% of its original size. (Note, despite the "obvious length" of SPLIDs, range-based or sequential labeling schemes would consume more storage, because they do not qualify for effective compression). Indicative overall improvement factors (*uniprot*) are ~47% and ~73%, respectively[4]. For our reference documents, these factors range from ~40% to ~52% for *pc* and from ~70% to ~80% for *eless*, respectively. To the best advantage, our novel optimization step (from *pc* to *eless*) accounts for ~50% to ~58%.

3. Processing documents

The gain provided by elementless XML storage structures should not be compensated neither by loss of functionality nor by performance penalties caused by ponderous operations. Therefore, we will show how XML processing is performed on elementless structures and what kind of performance characteristics can be anticipated.

3.1 Storing elementless documents

The combined use of B*-tree, SPLIDs, and path synopsis enables the straightforward creation of the document index: Assume, a client sends an XML document which arrives in document order.

[4] This translates to 465 and 731 Mbytes savings for the structure part. To avoid overloading of the result representation, we do not show the effect of content compression which would reduce the content part (white fraction of the bars) by 23% – 35%.

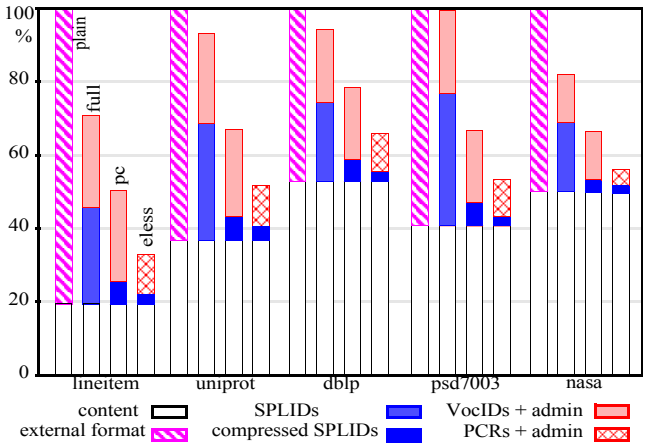

Figure 5. Storage consumption of XML documents

In an analysis phase, essential characteristics including the path synopsis can be determined. Initial loading of the document container is performed in document order thereby assigning (and compressing) the SPLIDs and the PCRs to the value nodes (Figure 4). Hence, the document index can be built bottom up [12].

3.2 Navigational operations

Navigational operations, such as in DOM, can easily be carried out on documents stored in an elementless manner, such as the one in Figure 4. The five navigational primitives *parent* (p), *first child* (fc), *last child* (lc), *previous sibling* (ps), and *next sibling* (ns) require only a single top-down traversal of the document index (typically of height ≤ 2 and 1– 60 leaf pages of 16 KB for moderately sized documents) and a single access to the path synopsis. Therefore, if the index as well as the path synopsis are already present in memory, only a single physical page reference suffices to execute any navigational operation.

Figure 6 contains the algorithm[5] to carry out a navigation originating at a given context node *cn*. The algorithm relies on several functions that are briefly discussed in the following: Given a SPLID, function *anc(int level)* calculates the SPLID of the ancestor node residing at the given level. Applied to an XML node, the function *pcr()* returns its PCR. Functions *splid()*, *level()*, and *parent()* are self-explanatory. The constructor *Node(SPLID id, PCR p)* creates a new XML node instance. Thereby, the constructor queries the structural summary to infer the node's name using *id* and *p*. Index look-up function *lookupMinPrefix(SPLID id)* traverses the document index and returns the node with the smallest SPLID (in document order) having prefix *id*. Likewise, *lookupMaxPrefix(SPLID id)* returns the node with the largest SPLID having *id* as prefix. For example in Figure 4, the former method using *id* = 1.1.3 returns a node with SPLID 1.1.3.1.1.1, whereas the latter method returns a node with SPLID 1.1.3.5.3.1. Note, both methods only need a single top-down traversal. Similarly, *lookupMinPrefixLeft(SPLID id)* and *lookupMaxPrefixRight(SPLID id)* traverse the tree and search for the min/max node having the given prefix *id*. If found, they return the left/right neighbor node. For example, *lookupMinPrefix-*

[5] To keep the presentation simple, documents containing attributes are not considered. The algorithm's extension is, however, straightforward.

Input: ContextNode cn, DocumentIndex I, Direction dir
Output: ResultNode rn

```
01 switch (dir)
02
03 case 'p':
04  rn = Node (cn.splid().anc(cn.level()-1), cn.pcr());
05
06 case 'fc':
07  Node n = I.lookupMinPrefix (cn.splid());
08  if (n == cn)
09   rn = null;
10  else
11   rn = Node (n.splid().anc(cn.level()+1), n.pcr());
12
13 case 'lc':
14  Node n = I.lookupMaxPrefix (cn.splid());
15  if (n == cn)
16    rn = null;
17  else
18    rn = Node (n.splid().anc(cn.level()+1), n.pcr());
19
20 case 'ps':
21   Node n = I.lookupMinPrefixLeft (cn.splid());
22   rn = Node (n.splid().anc(cn.level()), n.pcr());
23   if (rn.parent() != cn.parent())
24     rn = null;
25
26 case 'ns':
27   Node n = I.lookupMaxPrefixRight (cn.splid());
28   rn = Node (n.splid().anc(cn.level()), n.pcr());
29   if (rn.parent() != cn.parent())
30     rn = null;
31 end switch;
32 return rn;
```

Figure 6. Navigation algorithm

Left(1.1.3) returns the node with SPLID 1.1.1.7.1. These two functions also require only a single top-down document index traversal.

A context node's parent and PCR can directly be computed from the SPLID and the PCR in *cn*. During node construction (line 04), an access to the path synopsis is necessary to derive the parent's node name. The first/last child of a context node *cn* is calculated in a similar way: For first (last) child the node *n* with the smallest (largest) SPLID having the SPLID of *cn* as prefix is retrieved via an index look-up (lines 07 and 14). In the subtree rooted at *cn*, *n* is the leftmost (rightmost) descendant leaf node. If *cn* itself is a leaf node (indicated by *cn* = *n* in lines 08 and 15), no child node exists. Therefore, the result is null. Otherwise, starting from *n*, the SPLID of the first (last) child can be calculated by cutting off all divisions up to *cn*'s level increased by one. For example, for the evaluation of the first child on context node *cn* = 1.1.3, *n* is 1.1.3.1.1.1 and, because the first child has to reside at level 4, the resulting node is 1.1.3.1.

The navigation algorithm for *previous sibling* and *next sibling* is similar to *first child* and *last child*. For *previous sibling* (*next sibling*), *n* (lines 21 and 27) is stored in the left-hand (right-hand) neighbor record of the leftmost (rightmost) descendant node in the subtree rooted at the *cn* node. This time, the level of the sibling is equal to the level of the context node (lines 22 and 28). A final action has to check whether the calculated node actually *is a* sibling (lines 23 and 29), because it may also be a node in a completely different subtree which happens to share *cn*'s level. As an example, consider the evaluation of *previous sibling* on context node *cn* = 1.1.3, *n* is 1.1.1.7.1 and, because the previous sibling has to reside at level 3, the resulting node is 1.1.1.

3.3 Modifying elementless XML documents

No matter what kind of language model is used for document modification, it has to be translated into node- or record-at-a-time operations, for which the corresponding DOM operations provide an appropriate example.

The lion's share of the overhead caused by updates of nodes (names or values) or by insertions/deletions of subtrees in the XML document is carried by two valuable structural features: B*-trees and SPLIDs. B*-trees enable logarithmic access time under arbitrary scalability and their split mechanism takes care of storage management and dynamic reorganization. In turn, SPLIDs provide immutable node labeling such that all modification operations can be performed locally.

Referring to Figure 4, a context node *cn* can be determined either by navigation, via references from secondary indexes, or via the document index. To delete *cn*'s descendants (subtree deletion), all records whose SPLID starts with *cn*'s SPLID have to be deleted. For example, in Figure 1, the deletion of the *Mgr* node (1.1.1) results in a deletion of all records in Figure 4 that start with 1.1.1. All records to be removed are stored consecutively. Compared to the *pc* storage format, fewer records and smaller ranges have to be deleted.

During the insertion of subtree *s*, we assign existing PCRs to the values of those paths in *s* that conform to the path synopsis; for all other paths, new PCRs have to be generated and the path synopsis is updated accordingly. As before, insertion affects smaller set of consecutive records, compared to *pc*.

In *pc*, for inner node *n*, the stored document contains a physical record. In *eless* format, *n* is a virtual node. Therefore, all PCRs of leaf nodes in *n*'s subtree (determined by *n*'s SPLID) have to be updated (additionally, the path synopsis has to be altered, if renaming introduces new paths to the document). Therefore, compared to *pc*, renaming of an inner node [40] is the only critical operation in the *eless* format.

4. Indexing XML documents

So far, researchers have designed content [29], path [11, 33] or hybrid indexes [21, 22, 36] with the tree structure of the natively stored XML documents in mind. So the indexes typically delivered node labels for index matches, which then had to be resolved or verified on the document structure. For example, separate matches on a content index (for the value part) and a path index (for the structure part) had to be algorithmically checked whether pairs of matches can be verified on the XML document (for example, by using merge joins). Concerning the evaluation of simple path expressions with content predicates (like *//Team/Proj/[Rating=5]*), hybrid indexes seem to be most valuable. However, the so far proposed structures exhibit several drawbacks. For example, IndexFabric [36], based on *Patricia tries*, stores the entire path (in some encoded form) with each indexed value, contains substantial redundancy, and may need complex path checking in case of descendant axis use. FLUX [22] carries a path signature with each index reference, which is constructed as a *Bloom filter* [4]. When a value qualifies, the related signature is assumed to deliver almost precise path information. However, because false positives may occur, it has to be checked against existing paths in the document which can make XPath query evaluation expensive (see Section 6.2).

The problem of path reconstruction gets more performance critical, when index access methods have to be integrated as operators in a physical XML algebra. The important question is: What is the result of an index access, e. g., an index scan, and how can this result serve as input for further physical operators, such as a holistic twig join? Consider a query *//Team[Proj/Rating=5]* on our sample document. Then, the index structures sketched above can only provide a sequence of *Rating* nodes as an answer. To obtain the requested sequence of *Team* nodes, however, they need to reconstruct their ancestor path which requires expensive accesses to the document. The same is true, if an index access is used to deliver a sequence of intermediate result nodes which are input to further operators. As above, for the evaluation of query *//Team[Proj/Rating=5]//Emp*, the evaluation of the inner path on an index has to return all *Team* nodes to compute the structural join with nodes of name *Emp*.

We want to get rid of path traversals, because we don't have a physical structure part anymore. On the other hand, we have excellent mechanisms—SPLIDs and path synopsis—to quickly reconstruct any ancestor path of a document. Therefore, we can redesign indexing of XML documents for XPath query expressions anew.

4.1 Query types considered

Besides the use of stream-based and navigational languages such as SAX and DOM [40], a number of declarative query languages have been proposed for semi-structured data, e. g., XQuery [42]. To be processed, the operations of all these high-level languages have to be translated into sequences of record-at-a-time navigational operations or, when using secondary indexes, set-at-a-time operations (based on node reference lists), which then refer to physical document structures (see Figure 4). While we have discussed above how navigational processing is achieved on elementless structures, we have to show the viability for more complex (inner) operations.

XPath [41] is a query language that specifies the syntax for path expressions over XML data. Because most of the referenced, declarative XML query languages are essentially based on XPath predicates, optimization of such predicate evaluation is vital for the performance of all these languages. Furthermore, when using suitable indexes, we immediately have to cope with set-at-a-time operations. Hence, we have to show how such operations are executed when referring to our elementless structures.

At the same time, we want to design a new effective and efficient index mechanism for the broad class of *content-and-structure (CAS) single path queries* which includes equality predicates and range predicates built upon comparison operators $\Theta = \{=, <, \leq, >, \geq, \neq\}$ on the values of content nodes or attribute nodes of an XML document (leaf nodes).

Exploring such XPath expressions, we can characterize and evaluate the set of queries to be answered by means of index support. In turn, such powerful index-based operations evaluating single path classes could be used to generate the input of even more complex operations such as structural joins or holistic twig joins [1, 19].

Definition 1: *A simple XPath query expression (XPQ) [22] is formally denoted by p[T], where* $p = e_1t_1e_2t_2...e_k$ *is a path and T is a content comparison predicate. Path p consists of descendant (//) and child (/) edges* e_i*, as well as element and attribute nodes tests or wildcards* t_i*. Edge* e_k *of path p refers to a node the value of which*

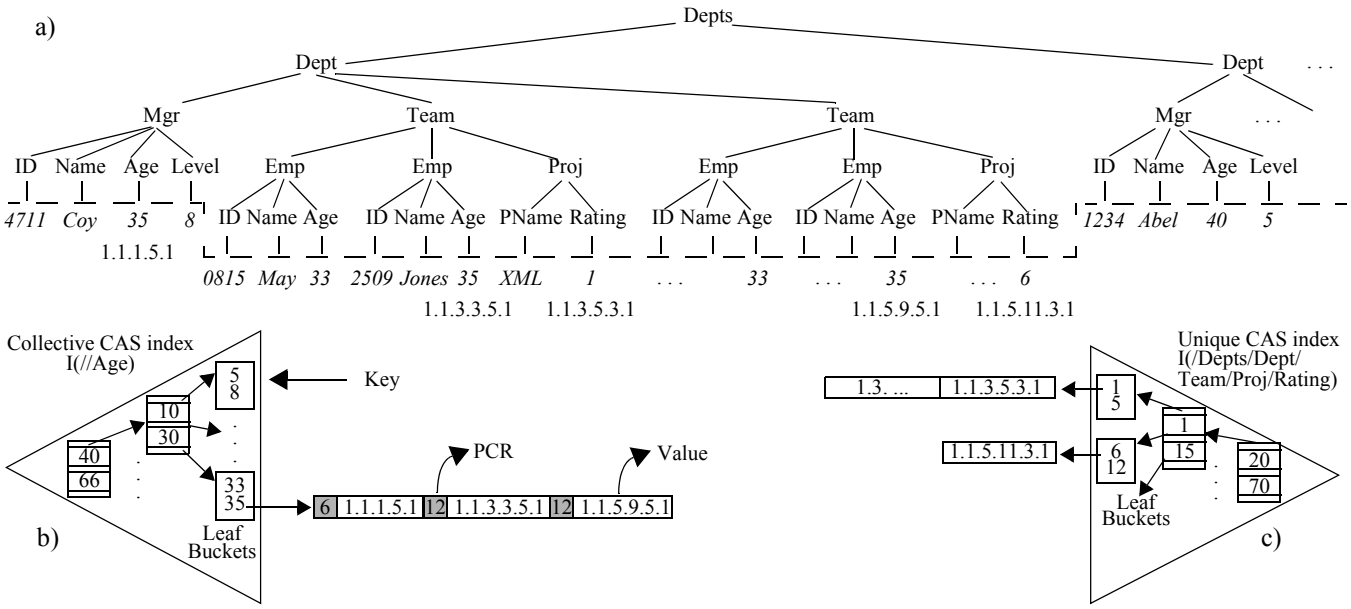

Figure 7. Document search model

can be Θ–compared. Comparison predicate T is of the form C = [t_k Θ v_i] for a simple comparison or R = [v_i $Θ_1$ t_k $Θ_1$ v_j] for a range comparison, where $Θ_1$ is suitably chosen from {<, ≤, >, ≥} and t_k is an indexable element/attribute or an indexable type (e. g., Integer, String, ID, etc.) and v_i, v_j appropriate range boundaries.

Examples Q_1 = *//Dept//Mgr/[Age ≠ 35]* and Q_2 = *//Team//[1 ≤ Integer ≤ 25]* are XPQ expressions on the document in Figure 1.

Note, each XPQ expression has two parts to be matched: a structure part (path predicate) and a content part (value predicate). To evaluate an XPQ on a given data set, we have to determine all matching path instances (possibly of several path classes) and check the related values using the C or R predicate. If no indexes are present, the only search strategy involves a scan over the entire document. When accessing a node, its value can be checked by the predicate part; if a node match occurs, the related PCR is used for fast reconstruction of its path to enable the test of the path predicate. Even when matching indexes are found, a document scan may be the best choice, if either the structure or the value part (or both) have low selectivities. Often, however, path and predicate selectivities enable highly efficient index support, as discussed in the following.

4.2 Defining and creating CAS indexes

To answer XPQ expressions, we provide a hybrid index structure. Therefore, we index all values (of a certain type) in our novel CAS indexes. Because every search tree enables checking of all Θ-based value predicates, B*-trees are the best choice for our base index structure. To overcome the sketched implications for path reconstruction, the records contained in a CAS index consist not only of a key (of a given content type, e. g., Integer, String, etc.) and a value (the occurrences of this key in the document, e. g., a list of node labels) but also of a path class reference (PCR) which denotes the ancestor path to the indexed value (see Figure 7). During the evaluation of an XPQ expression Q on a CAS index, we can then easily decide, if an indexed value matches the structural part of Q. CAS indexes are defined as follows:

Definition 2: *A CAS index on an XML document D is formally denoted as $I_D(p, T)$, where the index path predicate p is a simple XPath query expression with an empty comparison predicate Q (i. e., p is only a structural query), and T is the indexed content type (e. g., Integer, String, etc.)*[6]*. Leaf node n of D is contained in $I_D(p, T)$ as a key if*

- *C1) its parent element is contained in the result of the evaluation of p against D, and if*
- *C_2) n matches the content type of the index definition.*

The keys n in I_D are ordered in ascending order w. r. t. T, while the values (occurrences of n in D) are in document order for one and the same key. Each value carries a PCR.

For example, I_0 = I(*/Depts/Dept/Mgr/Level*, [Integer]) indexes all values of Level in the related path class, that is, for each value v_i occurring for *Level*, a node reference list is maintained (in document order) which stores the SPLIDs for the nodes (records) having v_i as a value. Some sample entries (SPLIDs) in the indexes shown in Figure 7b and c refer to the document leaves (Figure 7a). The attached PCRs enable the reconstruction of the related path instances by means of the path synopsis.

An index $I_D(p, T)$ is created as follows: First, its index path *p* is evaluated[7] against document *D*'s path synopsis, resulting in a list *P* of PCRs that match *p*. Then, *D* is scanned in document order. For each value v_i, we check whether its PCR is contained in *P* (C1 in Definition 2) and whether its type matches *T* (C2). If so, v_i belongs to index *I*. *I* may also be created during document storage on the fly. The correctness of this process—i. e., every indexed content value is on path *p* and the ordering complies with Definition 2—is assured by the path synopsis' consistency and by visiting the indexed nodes in document order (i. e., stable sorting of content values is sufficient).

[6] Note, subscript *D* and type *T* are omitted where non-ambiguous.

[7] The evaluation of a structure query on a path synopsis is not formally defined here. However, it should be intuitively clear.

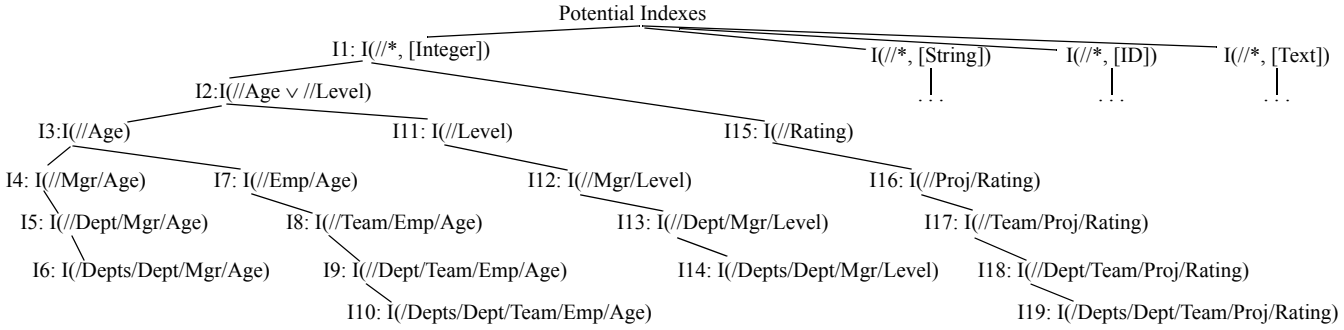

Figure 8. Potential indexes for content (outlined for Integer)

As an example for CAS index creation, consider the index definition I(*//Age*) (Figure 7). The evaluation of the query *//Age* on the path synopsis in Figure 2 returns the two PCRs 6 and 12. During index creation, all leaf nodes having either PCR 6 or 12 are included in I, e.g., nodes 1.1.1.5.1, 1.1.3.1.5.1, ...

Depending on the structure of the path synopsis and the given index path predicate *p*, we can partition the possible indexes into the three classes *unique*, *collective* and *generic*. These classes will be discussed in the next section.

4.3 Unique, collective, and generic indexes

Whenever every record *R* in a CAS index *I* refers to the same PCR, i. e., when *I*'s path predicate corresponds to exactly one path class and therefore exactly qualifies *R*'s ancestor path up to the root, we call the corresponding CAS index a *unique CAS index* (UCI). In this case, the PCR does not need to be stored together with each record, but only once for the whole index (see Figure 7c). A UCI can be used to precisely answer single path queries only on its index path. For example, UCI for *Rating* is equivalent to I(*//Rating*), I(*//Proj/Rating*), ..., I(*/Depts/Dept/Team/Proj/Rating*), because the same set of values qualifies in any case.

As soon as an index path defines more than one path class, e. g., *Age* in Figure 2, indexing such homonyms results in (homogeneous) *collective CAS indexes* (CCI). Accessing the collective CAS index I(*//Age*) (see Figure 7b) directly delivers the result for XPath predicates (*//[Age=35]*), (*/Depts//[Age=35]*), (*/Depts/Dept //[Age=35]*), etc. However, XPath predicates (*//Mgr/[Age=35]*), (*//Emp/[Age=35]*) or (*//Team/Emp/[Age=35]*)—also supported by I(*//Age*)—require some extra effort to remove false positives.

In index definition $I_D(p, T)$, parameters *p* and *T* determine the index' *focus*. We generalize Definition 2 and identify four different index types:

Definition 3: In a *unique* CAS index, all entries have the same PCR, while in a *homogeneous collective* index, the entries may have varying PCRs. For the *heterogeneous collective* CAS index, we generalize *p* to $p = p_1 \vee ... \vee p_i \vee ... \vee p_n$ where the p_i are paths as in Definition 2. A generic CAS index contains all values of a certain type (i. e., p = //*).

On our sample data, I(*//Rating*) is a unique index, I(*//Age*) is a homogeneous collective index, I(//Age ∨ //Level) is a heterogeneous collective index and I(//*, [Integer]) is a generic index over integers. While unique indexes are specialized and can answer queries on a single path class only, the focus widens over collective to generic indexes. Because unique indexes contain records with the same PCR, explicit PCR storage could be omitted to save space. Such a design decision, however, should be supported by a fixed schema, because an insertion of *Emp/Rating* in our sample document would turn the unique index I(*//Rating*) into a collective one.

Homogeneous collective indexes are the standard case. They potentially require some effort to remove false positives. For example, query *//Mgr/[* Age ≤ 35 *]* requires removal of *Emp/Age* entries in I(*//Age*).

Depending on the selectivities of the path classes included in such collective indexes and on the overhead to remove false positives, it can be beneficial to combine as many path classes as possible in CAS indexes. The more frequent an index is accessed, the higher is the locality of reference on the index pages which, in turn, keeps such pages longer periods of time in the DB buffer (memory). Therefore, it could be advantageous to broaden the index use and provide *heterogeneous collective* indexes.

Finally, we can design indexes combining all path classes of a given indexable type, e. g., Integer, String, or Text (where Text implies the use of IR search techniques). Such *generic* indexes are not tailored anymore to a particular CAS query, but drastically reduce the number of indexes needed. In our running example, I(//*, [Integer]) could serve to evaluate such diverse XPath queries as *//[Rating ≤ 3]*, *//[Level ≤ 5]*, or *//Mgr/[Age ≤ 60]*.

To give an impression of the possibilities introduced by CAS indexes, Figure 8 illustrates the spectrum of tailored indexes for all CAS index path predicates having at most one descendant relationship.[8] Because many of them occur in unique paths or path fragments, they are identical. *Rating* and *Level* refer to unique path classes which implies that I11: I(*//Level*) and I15: I(*//Rating*) are identical to I12 – I14 and I16 – I19. This is not true for I3: I(*//Age*), because it is a CCI where two path classes participate. Nevertheless, if we distinguish I(*//Mgr/Age*) and I(*//Emp/Age*), we get two UCIs which directly embody their dependent indexes (I5 – I6) and (I8 – I9). It may also be beneficial to combine unique or collective indexes, e. g., to I2: I(*//Age ∨ //Level*) or even to a generic index I1: I(//*, [Integer]).

[8] Multiple descendant relationships could be easily mapped to them.

5. Path evaluation on CAS indexes

Assume we have an XPQ expression *Q*, a document *D*, and a set of indexes *J*. The two questions arising now are 1.) which set of indexes in *J* can be used to evaluate *Q*, i. e., how is *index matching* done, and 2.) how is the evaluation of *Q* accomplished using an index *I,* which is related to the *search model*? Because the search model is required to clarify how existing indexes are selected, we start the discussion with the second point.

5.1 Answering point and range queries

By Definition 1, XPQ $Q = p[T]$ consists of a content predicate *T* and a path predicate *p*. *Q* is evaluated on index I_D as follows:

1. Path *p* is matched against the path synopsis of document *D*, resulting in a set of PCRs *P*. If *P* is empty, the result is also empty, because the document does not contain any path matching *p*.
2. For content predicate *T*, a point access or a range scan to/over *I* is issued to deliver all records matching *T*.
3. For the PCR of each record *R* delivered by the index access, the set inclusion in *P* is checked. If *P* contains the PCR, record *R* belongs to the final result, because its ancestor path matches path predicate *p*. Thus, this step removes false positives.

For an example, assume we have a collective CAS index I(*//Age*) as shown in Figure 7 and the query Q = *//Dept/Mgr/[Age=35]*. Matching path *//Dept/Mgr/Age* against the document's path synopsis returns a set *P* of exactly one PCR: {6}. A point access to index I results in a sequence of three records, of which only the PCR of the first one (having SPLID 1.1.1.5.1) is contained in *P*. Therefore, all remaining *Age* nodes (false positives) are filtered out.

This search model can be implemented very efficiently. Because B*-trees are search trees, they guide the evaluation of the comparison predicate Θ in T. Our implementation interleaves steps 2 and 3, such that the PCR is immediately matched for each scanned record. At the beginning of Section 4, we argued that previous indexing approaches suffer a performance penalty, if they have to generate answers for queries such as *//Team/[Proj/Rating=5]*, where the ancestor path has to be re-established from the document to generate a requested node sequence S_T. In our proposal, however, the powerful SPLID + PCR construct allows us to compute S_T in main memory, i. e., without document access. This mechanism also solves the mentioned problem to deliver the "right" input nodes to further physical operators (e. g., for query *//Team[Proj/Rating=5]/Emp*, which calculates a structural join after the index access).

So far, only the PCR-based evaluation of a query using a CAS index has been discussed. However, the relationship between the path predicate of the query and the path predicate of the index is not established, yet. Therefore, the question arises whether or not the result is complete. This matter is discussed in the next section.

5.2 Index matching

The initial task of index-based query processing is to find an appropriate set of indexes in *J* based on which XPQ expression *Q* can be evaluated. Let $E_D(Q)$ be the result of the *Q*'s evaluation on document *D* and $E_I(Q)$ the result of its evaluation using index $I \in J$, as outlined in the previous section. Then, there are four possible cases:

Table 2. Frequency of unique/collective indexable elements

Doc. name	# path classes	# indexable elements for UCIs	# indexable elements for CCIs, 2 elem.	# indexable elements for CCIs, >2 elem
lineitem	17	17	0	0
uniprot	121	53	7	10
dblp	153	4	3	25
psd7003	76	42	5	6
nasa	73	32	4	6

1. $E_D(Q) \cap E_I(Q) = \varnothing$: The evaluation on the document and on the index have no common subset. This either means that the query has no result at all or that the index is not applicable to answer the query.
2. $E_D(Q) = E_I(Q)$: The evaluation on the document and the index returns the same result, i. e., the index is applicable without removal of false positives. In this case, the PCR check (Step 3 in Section 5.1) can be omitted.
3. $E_D(Q) \subset E_I(Q)$: In this case, the index contains false positives that make Step 3 necessary.
4. $E_D(Q) \supset E_I(Q)$: The index does not contain all nodes to answer the query, but only a partial result.

The decision of these four cases based on *Q* and *I*'s path predicate *p* alone, i. e., without access to the document, is a difficult problem in the general case. Fortunately, the path synopsis and our PCRs provide a basis to solve this problem in a simple way: The above result-set comparison deciding the four cases shown can be replaced by a PCR-set comparison: $E_D(Q)$ is replaced by the evaluation of *Q*'s structure predicate on path synopsis *PS*, and $E_I(Q)$ is replaced by the evaluation of *I*'s path predicate on *PS*. Both evaluations return a set of PCRs, based on which the above cases can be decided.

As an example, consider the index I = (*//Dept//Age*) and the queries Q_1 = *//[Age<35]* and Q_2 = *//Emp/[Age<35]*. The PCR sets {6,12} for I and Q_1 are equal (case 2) and, therefore, Step 3 can be omitted. For Q_2, the PCR set is {12}. Therefore, Step 3 is required and removes all nodes with PCR 6.

While cases 1 to 3 yield a "positive" result, case 4 signals that the index alone is not sufficient to evaluate the query. However, when multiple (not necessarily unique) CAS indexes qualify, e. g., I(*Depts/Dept/Mgr/Name*) and I(*//Emp/Name*) for query *//Name*, the qualified node reference lists of all matching indexes can be merged to derive the result. If the union of all participating PCR sets in the qualifying indexes is a superset of the query's PCR set, the result is complete (but may contain false positives).

Unique indexes are interesting, because no explicit PCR storage and no post-processing phase is required for them at all, resulting in low storage overhead and fast evaluation algorithms. However, their applicability is restricted only to those cases, when the query exactly matches the path class represented by the index. For example, all queries related to *Rating* can be answered with the unique index I(*//Rating*), e. g., Q = *//Proj/Rating*, Q = *//Team//Rating*, etc.

To illustrate the potential of unique and collective indexes, we have evaluated the synopses for all reference documents. Because the

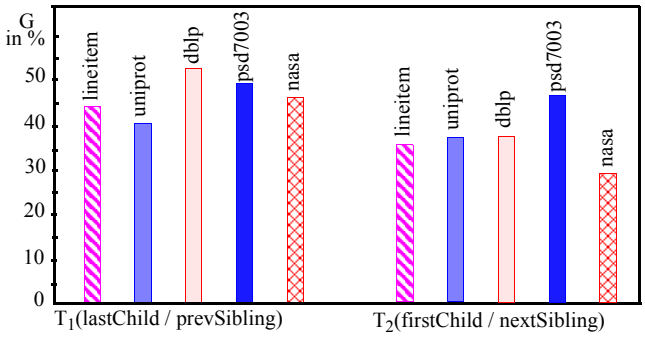

Figure 9. DOM navigation improvements (eless vs. pc)

creation of an index is a careful design decision requiring workload analyses, etc., we can only list the maximum number of indexes to be created as unique or (homogeneous) collective indexes. As indicated by Table 2, the consideration of UCIs is particularly important, because of their high potential numbers and of their simple and flexible evaluation, even in case of descendant relationships.

6. Quantitative results

In Section 2.4, we have explored storage consumption of different physical document representations and have identified their saving potential. Here, we want to focus on the best known storage method and our novel virtualization technique. Therefore, we will reveal improvements for various kinds of operations using elementless document storage (*el*) as compared to the storage format with prefix-compressed SPLIDs (*pc*). All performance measurements were run on an Pentium IV computer (2 x 3.2 GHz CPUs, 1 GB main memory, 80 GB external memory, Java Sun JDK 1.5.0) as the XDBMS server machine. In all experiments, external storage was formatted with a page size of 16 KB. XTC used a cold buffer configured with 4 MB.

6.1 Navigation

In an XDBMS, navigational operations are either directly executed via a given API (e. g., DOM [40]) or by the implementation of physical query operators. Because the execution time of single navigational operations is not very expressive, we have designed a benchmark consisting of two tree walkers. Both walkers T_1 and T_2 start from the root and apply the operations *last_child* / *previous_sibling* (T_1), and *first_child* / *next_sibling* (T_2). In case of *eless*, the root and the inner structure is virtual and has to be computed, while the tree walk is proceeding. Figure 9 shows the substantial gains for the *eless* documents over *pc* documents. In all cases, we achieved improvements of ~40% – 53% for T_1, resp. ~30% – 47%, for T_2. These performance gains are due to less I/O operations and shorter node reconstruction times on the compact *eless* documents.

6.2 Index-supported queries

In the following experiments, we refer to a set of given indexes and want to focus on the speed-up gained by the index use. Because the anticipated results are strongly dependent on the values and selectivities present, our set of reference documents did not allow for simple cross-comparisons. To have better control over the values

Table 3.Indexes

#	Type	Size [MB]	Definition	#PCRs
I_1	PathSynopsis	5927 (B	*summary of all paths*	548 (all)
I_2	Element	6.39	*list of all element nodes*	—
I_3	Content	20.56	*all content nodes*	—
I_4	GCI	22.63	//* [String]	548 *(all)*
I_5	CCI	2.15	//item/location [String]	6
I_6	UCI	0.21	//asia/item/location [String]	1
I_7	FLUX	23.69	//* [String]	—
I_8	CCI	15.40	//text/bold [String]	33
I_9	CCI	16.70	//keyword [String]	99

Table 4.Queries

#	Type	Query	#PCRs
Q_1	Point	//asia/item/[location=“United States“]	1
Q_2	Range	//asia/item/[“C” ≤ location ≤ “G”]	1
Q_3	Range	//text/[“a” ≤ bold/keyword ≤ “Z”]	33
Q_4	Twig	//item[location=“United States”] //text[“c” ≤ bold ≤ “h”]	—

and their distributions and to enable scalability considerations, we constructed a benchmark to provide some insight on how the existence of suitable CAS indexes influences the evaluation performance. We used the XMark framework [37] to evaluate point query Q_1 and range query Q_2 of Table 4 in seven different scenarios (S1-S7) where indexes I_1 to I_7 of Table 3 were exploited. Note, additionally to the index definitions, Table 3 contains the index sizes for a 100MB document and the number of indexed PCRs. All tests were carried out on 4 XMark documents of size 10 MB, 50 MB, 100 MB, and 500 MB. For a direct index comparison, we implemented the FLUX index [22]. In FLUX, each path to a content node v_i is mapped onto a Bloom filter F_i (with entries of 2 bytes in our implementation as in [22]) using the value of an *MD5 digest* on each element name contained in the path. Records $R(v_i, F_i)$ are stored in the FLUX index. To evaluate an XPQ $p[T]$ on FLUX, a suitable scan returns all records $R(v_i, F_i)$ that match T. For query path p, a Bloom filter F_p is computed and matched against the record's Bloom filter F_i. False positives are removed by a reconstruction of the ancestor path from the document. The scenarios for queries Q_1 and Q_2 are:

S1. No CAS/text index is available. In this case, we rely on the holistic twig join operator (TwigStack [6]) to evaluate the structural predicate of the query over the element lists provided by index I_2; the content predicate is evaluated by a node-at-a-time look-up in the document.

S2. A content index (I_3) is available for the document. This index allows to extend the twig join operator to structure predicates (because a content node at the leaf level is the child of its enclosing element). A point query over the content index returns all SPLIDs in document order, indicating where the queried content occurs. For a range query, the resulting SPLIDs are not in document order and have to be sorted to serve as input for the twig join.

S3. A generic CAS index (I_4) is available. The PCR matching algorithm can be used to remove false positives.

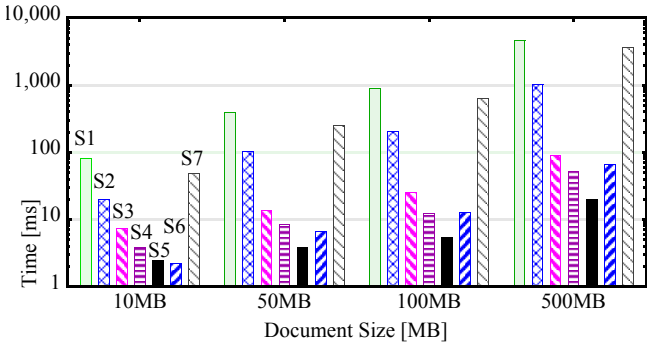

Figure 10. Point query results

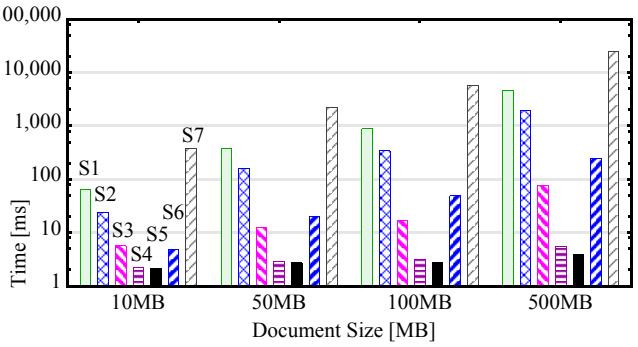

Figure 11. Range query results

S4. A collective CAS index (I_5) is available. Compared to the generic index, a collective one is more focused.

S5. A unique CAS (I_6) index is available. In this case, no false positives can occur.

S6. A FLUX index (I_7) is available. The removal of false positives requiring access to the document to re-establish the ancestor path is *omitted* here.

S7. A FLUX index (I_7) is available. In contrast to S6, false positives are removed by path reconstruction.

The results for our selected point queries and range queries are shown in Figure 10 and Figure 11, respectively. S1 shows the worst performance, because no index was present and the verification of the content predicate required navigational steps (thus implying expensive random I/O). In S2, the content access support from I_3 results in quite promising performance improvements for the twig join. Note, S2 in Figure 11 was deteriorated by an additional sort of the range result delivered by the content index, because the subsequent twig join depends on a sorted input. In S1 and S2, missing or insufficient index support caused linear response time growth w. r. t. document sizes.

S3, S4, and S5 exploit CAS indexes and PCR structure matching, such that joins are not needed anymore. To a large extent, the performance differences can be explained by the varying need to remove false positives from the result set, in particular, in case of a generic index (S3). The range query results in Figure 11 are slightly influenced by a sort of the index output to return a list of references in document order. We added this extra requirement to be in accordance with the scenarios S1 and S2, where the twig join delivered a sorted result. Referring to a UCI, S5 could take advantage of ideal CAS index support which boosted the query performance in both cases by up to two orders of magnitude.

For our comparison with FLUX, we considered scenarios S3, S6, and S7, because the referring indexes contain the same XML values. In Figure 10 and Figure 11, we see that both indexes perform nearly equally good during the retrieval of the indexed records (scenarios S3 and S6). FLUX performs slightly better, because the bitwise Bloom-Filter comparison can be executed a little more efficiently than the PCR set-containment check. However, S6 does not produce an *exact* result set (as S3 does), but a superset (containing false positives due to Bloom filter usage). For example, range query Q_2 on the 500 MB document returns ~29,000 nodes in S6, whereas the correct number of nodes delivered in S3 is only ~400. Therefore, ~28600 false positives are removed in S7. This process is very time consuming; for Q_2, it took ~25 seconds.

In contrast to our index proposal, FLUX [22], A(k)-Index [20], or Kaushik's CAS Index [21] suffer a performance penalty, when they have to return inner nodes for the queried path, as sketched in Section 4. In a second experiment, we want to measure this penalty. FLUX and A(k) propose to retrieve inner elements by navigation, whereas Kaushik et. al. propose to use a structural join (with level restriction). We evaluate query Q_3 on index I_9, thus generating a sequence of keyword nodes. We assume that text nodes are requested as output and, therefore, we 1.) navigate the document two steps up, 2.) retrieve all *text* nodes and do a structural join (matching nodes are grandparents), or 3.) compute them (SPLIDs). The result timings are plotted in Figure 12 (note, the number of returned elements scaled linearly with the document size). Navigation and join-based computation both suffer from document access (navigation even more, because it generates random I/Os to retrieve ancestor paths). In all cases. the in-memory, SPLID-based computation is one order of magnitude faster than the other alternatives.

In a third experiment, we explored the performance behavior when branches of twig queries are evaluated by CAS index access replacing holistic twig matching. For twig query Q_4, scenarios S1 (navigation), S6, and S7 (both FLUX) are left out, because we only want to focus on the speed-up possible by CAS index usage. As before, scenario S2' exploits a text index and extends holistic twig matching on content nodes. In S3', the evaluation of the left twig branch (*//item/[location=„United States"]*) is substituted by an index ac-

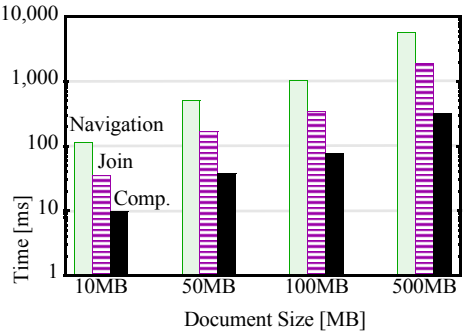

Figure 12. Reconstruction timings

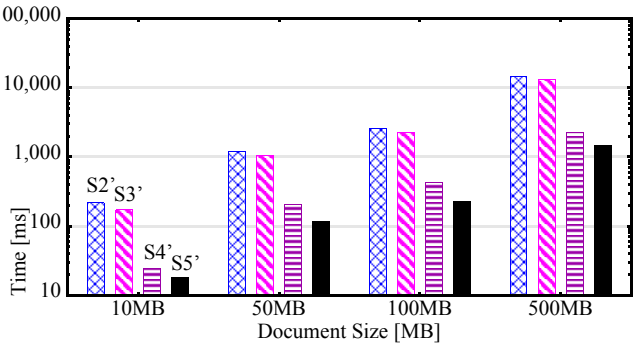

Figure 13. Twig query results

cess using I_4. Scenario S4' does it the other way around and substitutes the other branch of Q_3 (*//text[„c" < bold <„h"]*) by a range access to I_8. Finally in S5', both branches are substituted by appropriate index accesses. As expected, Figure 13 reveals that query performance increases with the opportunities of using suitable indexes. Considering S2' with a plain text index as a baseline, S3' can only be improved marginally: The substituted predicate does not cause too much overhead anyway, because it can occur at most once per item element. In contrast, in S4' a more complex predicate is substituted by access to an index range which translates to a larger performance gain. Substituting both predicates by index accesses obviously achieves the best performance. Hence, maximum CAS index use improved our experiment up to an order of magnitude.

7. Related work

XML document storage strategies can be divided into *shredding* and *native* approaches. While native document stores—like the one presented in this work—are specially tailored to XML, shredding decomposes documents into relational tables, thus enabling XML in any RDBMS.

Similar to RDBMS, all native stores (e. g., the ones of SystemRX [3], Natix [8], eXist [30], and Xindice [39]) define a mapping from an XML document onto records and pages, where the *granule* of XML items stored in a page varies from lists of *neighbor nodes* (eXist and this work), over *subtrees* (Natix, SystemRX) to *complete documents* (Xindice). While all stores support navigation (and, thus, query processing), reconstruction, and modification, there is—to the best of our knowledge—no other native store that eliminates the redundant internal structure of a document, while providing the same external XML processing interfaces. Even the similar approach in [2], using a structural summary and extent lists for each path class, loses document order and, thus, expensive joins are needed to reconstruct the document. This statement also holds for various XML compression techniques (e. g., [34, 38]), which also get rid of the internal structure, but, in contrast, only provide narrow subsets of the mentioned interfaces.

Shredding can also be classified by the granule of XML items used, where *edges* [9], *nodes* [13], and complete *paths* [43] have been considered. Additionally, some techniques rely on a schema (Document Type Definition [10]) to generate the required tables. Apart from the XML storage solution, shredding always comes with some XQuery-to-SQL translation optimized for fast query evaluation. The "nodes" approach is comparable to our *full* storage scheme, where each *elem*ent is explicitly stored. Path-based shredding is somewhat similar to our path synopsis use, because each element "knows" its ancestor path. However, no virtualization techniques have been proposed so far. Furthermore, other XML processing interfaces (than XQuery) have been widely neglected.

A large variety of XML indexes has been published, which can be classified in *content*/*element* [29], *path* [20, 33], and *hybrid/CAS* [22, 22, 36] indexes. While content and element indexes are inverted lists of content and element nodes, path and hybrid indexes support the evaluation of simple path/twig queries (with content predicates). In contrast to our work, most indexes do not 1.) provide for focused index definitions, but index complete documents, leading to high update costs and large index sizes; 2.) cannot cheaply reconstruct inner elements and are therefore hard to integrate into a physical XML algebra; and 3.) do not provide for index maintenance [28], an issue that was excluded here because of space restrictions.

It is interesting to know that we gained for tree-pattern queries using the so-called S^3 algorithm [18] similar results with up to orders-of-magnitude improvements, as presented in Section 6,—essentially enabled by the interplay of SPLIDs and path synopsis. In this paper, query evaluation performance was compared for different path processing and join algorithms. All competitor algorithms were implemented in XTC to provide an identical runtime environment using a full-fledged XDBMS for accurate cross-comparisons: S^3, *Structural Joins*, *TwigStack*, *TJFast*, *Twig²Stack*, and *TwigList* [1, 6, 19, 27]. Unlike all competitor methods, S^3 executed path expressions not directly on the XML document, but first evaluated them against a path-synopsis-like structure, to avoid access to the document to the extent possible. Hence, variations of our idea underlying the S^3 algorithm outperformed any kind of conventional path operator use, achieved stable performance gains and proved their superiority under different benchmarks and in scalability experiments [18].

8. Conclusions

In this paper, we proposed a physical representation of XML documents having virtualized inner structure. Key to this upside-down representation is the use of SPLIDs as node labels and of a suitable path synopsis which both together enable fast computation of structure nodes. Compared to optimized vocabulary-based approaches, our empirical evaluation revealed substantial savings in storage consumption and considerable improvements of processing times for storing and reconstructing XML documents. Even navigation along virtualized document hierarchies delivered positive results. Because the document store is aware of paths, index maintenance detection can be solved in a very efficient way.

Adjusted to this upside-down storage structure, we designed a flexible index mechanism combining content and path indexing. The CAS index use was generalized such that the same implementation can serve as a unique, collective, or generic index. A simple algorithm based on path synopsis use achieves very efficient index matching when a query predicate is to be evaluated. Compared to the so far prevailing application of structural binary joins or holistic twig joins for queries involving structure predicates and content predicates, we achieved dramatic improvements in the order of one magnitude or even two. This is due to the replacement of joins by the use of our specific CAS index supported by SPLIDs for the computation and matching of the document structure.

Future work will focus on a refinement of the CAS evaluation where we plan to develop effective physical algebra operators. Here we want to combine the evaluation results of several separately executed CAS queries to enable general holistic twig joins without the need of join processing. Because optimized use of CAS indexes is sensitive to many parameters (structure, value selectivities, query predicate, etc.), an index advisor could be of great help for such physical algebra operators.

Acknowledgments

The authors are grateful for the hints of the anonymous referees, which improved the readability of this paper.

References

[1] Al-Khalifa, S., Jagadish, H. V., Patel, J. M., Wu, Y., Koudas, N., and Srivastava, D.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. *Proc. Int. Conf. on Data Engineering (ICDE)*: 141-152 (2002)

[2] Arion, A., Bonifati, A., Manolescu, I., and Pugliese, A.: Path Summaries and Path Partitioning in Modern XML Databases. *World Wide Web* 11:1, 117-151 (2008)

[3] Beyer, K. S., Cochrane, R., Josifovski, V., Kleewein, J., Lapis, G., Lohman, G. M., Lyle, R., Özcan, F., Pirahesh, H., Seemann, N., Truong, T. C., Van der Linden, B., Vickery, B., and Zhang, C.: System RX: One Part Relational, One Part XML, *Proc. ACM SIGMOD Conf.*: 374-358 (2005)

[4] Bloom, B. H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13:7, 422-426 (1970)

[5] Böhme, T. and Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. *Proc. 3rd DIWeb Workshop*: 70-81 (2004)

[6] Bruno, N., Koudas, N., and Srivastava, D.: Holistic Twig Joins: Optimal XML Pattern Matching. *Proc. ACM SIGMOD Conf.*: 310-321 (2002)

[7] Christophides, V., Plexousakis, D., Scholl, M., and Tourtounis, S.: On Labeling Schemes for the Semantic Web. *Proc. 12th Int. WWW Conf.*: 544-555 (2003)

[8] Fiebig, T., Helmer, S., Kanne, C.-C., Moerkotte, G., Neumann, J., Schiele, R., and Westmann, T.: Natix: A Technology Overview. *Lecture Notes in Computer Science 2593*: 12-33, Springer (2003)

[9] Florescu, D. and Kossmann, D.: Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin* 22, 27-34 (1999)

[10] Georgiadis, H. and Vassalos, V.: XPath on Steroids: Exploiting Relational Engines for XPath Performance. *Proc. ACM SIGMOD Conf.*: 317-328 (2007)

[11] Goldman, R. and Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. *Proc. Int. Conf. on Very Large Data Bases (VLDB):* 436-445 (1997)

[12] Graefe, G. and Larson, P.-A.: B-Tree Indexes and CPU Caches. *Proc. Int. Conf. on Data Engineering (ICDE)*: 349-358 (2001)

[13] Grust, T., van Keulen, M., and Teubner, J.: Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. *Proc. Int. Conf. on Very Large Data Bases (VLDB):* 524-525 (2003).

[14] Härder, T., Haustein, M. P., Mathis, C., and Wagner, M.: Node Labeling Schemes for Dynamic XML Documents Reconsidered. *Data & Knowl. Engineering* 60:1, 126-149, Elsevier (2007)

[15] Härder, T., Mathis, C., and Schmidt, K.: Comparison of Complete and Elementless Native Storage of XML Documents. *Proc. Int. Database Engineering and Applications Symposium (IDEAS):* 102-113 (2007)

[16] Haustein, M. P. and Härder, T.: An Efficient Infrastructure for Native Transactional XML Processing. *Data & Knowledge Engineering* 61:3, 500-523, Elsevier (2007)

[17] Haustein, M. P. and Härder, T.: Optimizing Lock Protocols for Native XML Processing. *Data & Knowledge Engineering* 65:1, 147-173, Elsevier, (2008)

[18] Izadi, K., Härder, T., and Haghjoo, M.: S^3: Evaluation of Tree-Pattern Queries Supported by Structural Summaries, *Data & Knowledge Engineering* 68-1: 126-145, Elsevier, (2009)

[19] Jiang, H., Wang, W., Lu, H., and Xu Yu, J.: Holistic Twig Joins on Indexed XML Documents. *Proc. Int. Conf. on Very Large Data Bases (VLDB):* 273-284 (2003)

[20] Kaushik, R., Shenoy, P., Bohannon, P., and Gudes, E.: Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. *Proc. Int. Conf. on Data Engineering (ICDE)*: 129-140 (2002)

[21] Kaushik, R., Krishnamurthy, R., Naughton, J. F., and Ramakrishnan, R.: On the Integration of Structure Indexes and Inverted Lists. *Proc. ACM SIGMOD Conf.*: 779-790 (2004)

[22] Li, H.-G., Aghili, S. A., Agrawal, D. and El Abbadi, A.: FLUX: Content and Structure Matching of XPath Queries with Range Predicates. *Proc. Int. XML Database Symposium (XSym*), *Lecture Notes in Computer Science* 4156, 61-76 (2006)

[23] Li, Ch., Ling, T. W., and Hu, M.: Efficient Updates in Dynamic XML Data: From Binary String to Quaternary String. *The VLDB Journal* 17:3, 573-601 (2008)

[24] Liefke, H. and Suciu, D.: XMill: An Efficient Compressor for XML Data. *Proc. ACM SIGMOD Conf.*: 153-164 (2000).

[25] Loeser, H.: XML Storage – It's the Flexibility, Stupid!. Computer Science colloquium, University of Kaiserslautern (2008)

[26] Loeser, H., Nicola, M., and Fitzgerald, J.: Index Challenges in Native XML Database systems. in: *Proc. German National Database Conf. (BTW)*, Münster, Lecture Notes in Informatics, GI-Edition (2009)

[27] Lu. J., Ling, T. W., Chan, C. Y., and Chen, T.: From region encoding to extended Dewey: on efficient processing of XML twig pattern matching. *Proc. Int. Conf. on Very Large Data Bases (VLDB):* 193–204 (2005)

[28] Mathis, C.: Storing, Indexing, and Processing XML Documents in Native XML Database Management Systems. Ph.D. thesis, University of Kaiserslautern (2009)

[29] McHugh, J., Widom, J., Abiteboul, S., Luo, Q., and Rajaraman, A.: Indexing Semistructured Data. Technical report, Stanford University (1998)

[30] Meier, W.: eXist: An Open Source Native XML Database. *Lecture Notes in Computer Science 2593*: 169-183, Springer (2002)

[31] Mignet, L., Barbosa, D., and Veltri, P.: The XML Web: a First Study. *Proc. 12th Int. WWW Conf.*, Budapest (2003). www.cs.toronto.edu/~mignet/Publications/www2003.pdf

[32] Miklau, G.: XML Data Repository, www.cs.washington.edu/research/xmldatasets

[33] Milo, T. and Suciu, D.: Index Structures for Path Expressions. *Proc. Int. Conf. on Database Theory (ICDT)*: 277-295 (1999)

[34] Ng, W., Lam, W. Y., Cheng, J.: Comparative Analysis of XML Compression Technologies. *World Wide Web* 9:1, 5-33 (2006)

[35] O'Neil, P. E., O'Neil, E. J., Pal, S., Cseri, I., Schaller, G., and Westbury, N.: OrdPaths: Insert-Friendly XML Node Labels. *Proc. ACM SIGMOD Conf.*: 903-908 (2004)

[36] Sample, N., Cooper, B. F., Franklin, M. J., Hjaltason, G. R., Shadmon, M., and Cohe, L.: Managing Complex and Varied Data with the IndexFabric(tm). *Proc. Int. Conf. on Data Engineering (ICDE)*: 492-493 (2002)

[37] Schmidt, A. R., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I., and Busse, R.: XMark: A Benchmark for XML Data Management. *Proc. Int. Conf. on Very Large Data Bases (VLDB):* 974-985 (2002)

[38] Skibinski, P. and Swacha, J.: Combining Efficient XML Compression with Query Processing, *Proc. East European Conf. on Advances in Databases and Information Systems (ADBIS)*: 330-342 (2007)

[39] Staken, K.: Xindice 1.1 User Guide (2005)

[40] *W3C Recommendations*. http://www.w3c.org (2004)

[41] XML Path Language (XPath), Version 1.0. *W3C Recommendation* (Nov. 1999)

[42] XQuery 1.0: An XML Query Language. *W3C Recommendation* (Jan. 2007)

[43] Yoshikawa, M., Amagasa, T., Shimura, T., and Uemura, S.: XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet technology (TOIT*) 1:110-141 (2001)

Christian Mathis studied Computer Science from 1998 to 2004 at the University of Kaiserslautern. Since Oct. 2004 he is a Ph. D. student and since Oct. 2007 scientific staff member at the DBIS research group lead by Prof. Härder. In the XTC project (http://wwwlgis.informatik.uni-kl.de/cms/index.php?id=36), he explores XML. query processing.

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 > 130 peer-reviewed conference papers and > 60 journal publications.
His professional services include numerous positions as chairman of the GI-Fachbereich "Databases and Information Systems", conference/program chairs and program committee member, editor-in-chief of Informatik – Forschung und Entwicklung (Springer), associate editor of Information Systems (Elsevier), World Wide Web (Kluver), and Transactions on Database Systems (ACM). He served as a DFG (German Research Foundation) expert and was chairman of the Center for Computed-based Engineering Systems at the University of Kaiserslautern, member of two joint collaborative DFG research projects DFG (SFB 124, SFB 501), and co-coordinator of the National DFG Research Program "Object Bases for Experts".

Karsten Schmidt studied Computer Science from 1999 to 2006 at the Technical University of Ilmenau. Since June 2006 he is a scientific staff member in the XTC project at the DBIS research group lead by Prof. Härder. His main interests are storage structures and indexes for XML documents and adaptivity in database management systems.