**Noname manuscript No.**
(will be inserted by the editor)

# XML Indexing and Storage: Fulfilling the Wish List

**Christian Mathis · Theo Härder · Karsten Schmidt · Sebastian Bächle**

Received: date / Accepted: date

**Abstract** XML Indexing and Storage (XMIS) techniques are crucial for the functionality and the overall performance of an XML database management system (XDBMS). Because of the complexity of XQuery and performance demands of XML query processing, efficient path processing operators – including those for tree-pattern queries (so-called twigs) – are urgently needed for which tailor-made indexes and their flexible use are indispensable. Although XML indexing and storage are standard problems and, of course, manifold approaches have been proposed in the last decade, adaptive and broad-enough solutions for satisfactory query evaluation support of all path processing operators are missing in the XDBMS context. Therefore, we think that it is worthwhile to take a step back and look at the complete picture to derive a salient and holistic solution. To do so, we first compile an XMIS wish list containing what – in our opinion – are essential functional storage and indexing requirements in a modern XDBMS. With these desiderata in mind, we then develop a new XMIS scheme, which – by reconsidering previous work – can be seen as a practical and general approach to XML storage and indexing. Interestingly, by working on both problems at the same time, we can make the storage and index managers live in a kind of symbiotic partnership, because the document store re-uses ideas originally proposed by the indexing community and vice versa. The XMIS scheme is implemented in XTC, an XDBMS used for empirical tests.

**Keywords** XML storage structures · XML indexing · XML query processing · Node labels

C. Mathis · T. Härder · K. Schmidt · S. Bächle
Dept. of Computer Science, University of Kaiserslautern
E-mail: haerder@cs.uni-kl.de

## 1 Motivation

XML models semi-structured data and is the standard for data exchange in many (Web) applications. To avoid conversion, not only messages but also conventional DB data is kept more and more in the XML format, often resulting in huge documents or collections thereof that are managed in specially tailored XML database systems (XDBMSs). Applications exploiting the full expressiveness of XML query languages such as XQuery and its subset XPath, are often confronted with system-dependent limitations. Besides language coverage, query processing style directly determines the XML awareness of query evaluation. For instance, relational backends need to adjust XQuery semantics and syntax to suit the SQL-oriented storage and physical operators available. Because the nested structure part of XML and the content values play likewise important roles for query formulation, numerous XML-aware operators were proposed to enhance query processing. However, most of them only provide partial solutions for subproblems of XML query processing, like, for example, path matching or axis evaluation. Additionally, many proposals make too strong assumptions on document characteristics and the underlying physical storage. Thus, these proposals may pay off for special types of queries, but, in general, quickly suffer from severe performance penalties, making their general application and success questionable.

Taking these deficiencies into account, we focus on a native XDBMS infrastructure without precluding any XML processing approach a priori through a combination of basic and well-known database technologies. Internally, XDBMSs have to implement space-optimized *XML document storage formats*, for which they provide efficient access primitives like navigation and subtree reconstruction, e. g., through a scan. However, to answer declarative queries over these XML stores, e. g., XQuery statements [36], basic access primitives

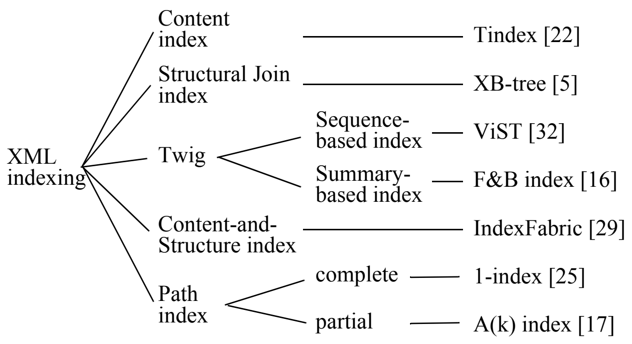
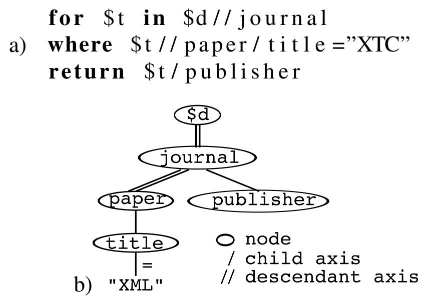


**Fig. 1** Classification of XML indexing approaches



```
   for $t in $d//journal
a) where $t//paper/title="XTC"
   return $t/publisher
```

**Fig. 2** Sample query: a) XQuery notation and b) twig representation

– for example, navigational operations of the DOM standard [32] or streaming operations as defined by the "Simple API for XML (SAX)" [33] – often cannot deliver enough performance to compute a result in sufficiently short time. Therefore, *secondary XML indexes* of different types are essential to guarantee satisfactory performance for at least some important types of queries (e. g., for path queries and tree-pattern queries). In particular, *structure and content* indexes are important, because they effectively support those XML queries, which require structural matching for path expressions or node relationships together with content evaluation for value joins and comparisons. Furthermore, the performance of an XML query processor heavily depends on the potentially expensive operation of *path pattern matching*. This operation occurs frequently, because even multiple paths are often defined in a single query; and it may be expensive, because path evaluation may require physical access to the document, in contrast to almost all other constructs of an XML query language [36], which are evaluated on the output generated by path matchings. The performance of path pattern matching would be boosted, if entire XML queries or large fractions thereof could be processed in a way that access to (cached) indexes is sufficient and document reference is avoided at all as, e. g., often achieved in relational DBMSs using B*-tree indexes and tuple ID (TID) references.

In this paper, Section 2 tries to reveal weaknesses concerning current index approaches and proposes two wish lists for XML document stores and indexes. Section 3 gives a description of the essential concepts forming our XML storage format together with a performance evaluation of the major operations supported. In contrast to the uncontrolled growth of XML indexing approaches (see Fig. 1), Section 4 proposes a minimal set of concepts used for XML indexing and shows that all existing index types can be derived by specializing content-and-structure (CAS) indexes without any performance penalty. We assess the related work in Section 5, where we can – in the light of our wish lists – identify missing functionalities and properties of competitor approaches. Finally, Section 6 wraps up with conclusions.

## 2 Use of Indexes

After more than a decade of XML research, a plethora of solutions have been published on storage and indexing: e. g., storage in Binary Large OBjects (BLOBs), quite a number of mappings to relational tables (aka "shredding"), and various tree-to-page mappings, i. e., "native" formats, among others. Moreover, we know an even more impressive variety of index structures, which are classified in Fig. 1, where only a single reference (standing for numerous proposals) is given per class.

Even the combined functionality of all these proposals does not provide the desirable index support of an XDBMS. For example, an index covering path expression `//a/b/c` cannot cheaply answer a query `//a[b/c]`, which is, however, urgently needed for the processing of complex path expressions. Often, those queries form complex tree patterns, so-called twigs [5] as shown in Fig. 2b. Moreover, most indexes are not *selective*, i. e., they often unnecessarily have to index complete documents, thus leading to high update costs. A third point is the *inflexible clustering* of the indexed data that inherently depends on the index structure and also leads to expensive post-processing operations. And above all, most index proposals are "stand-alone or abstract", because they are not embedded into a *system context* – most notably, a smooth integration with the underlying document store and the node labeling concept is missing.

We briefly reconsider a representative fraction of these approaches by trying to evaluate a small sample query (see Fig. 2), where we refer to an XML document fragment used as a running example. The dotted line (see Fig. 3) separates its structure part (i. e. the inner tree nodes) from its content part (i. e. the leaves carrying the data values). Regarding TID-style processing in XML, we identify some weak points that – to our knowledge – have not been properly addressed so far: The reviewed index proposals fail to provide inner nodes on the indexed path without expensive document access or post-processing.

The XQuery statement depicted in Fig. 2a shall be evaluated on the document in Fig. 3 (referenced by $d). This query can be illustrated as a twig [5] (Fig. 2b) and returns

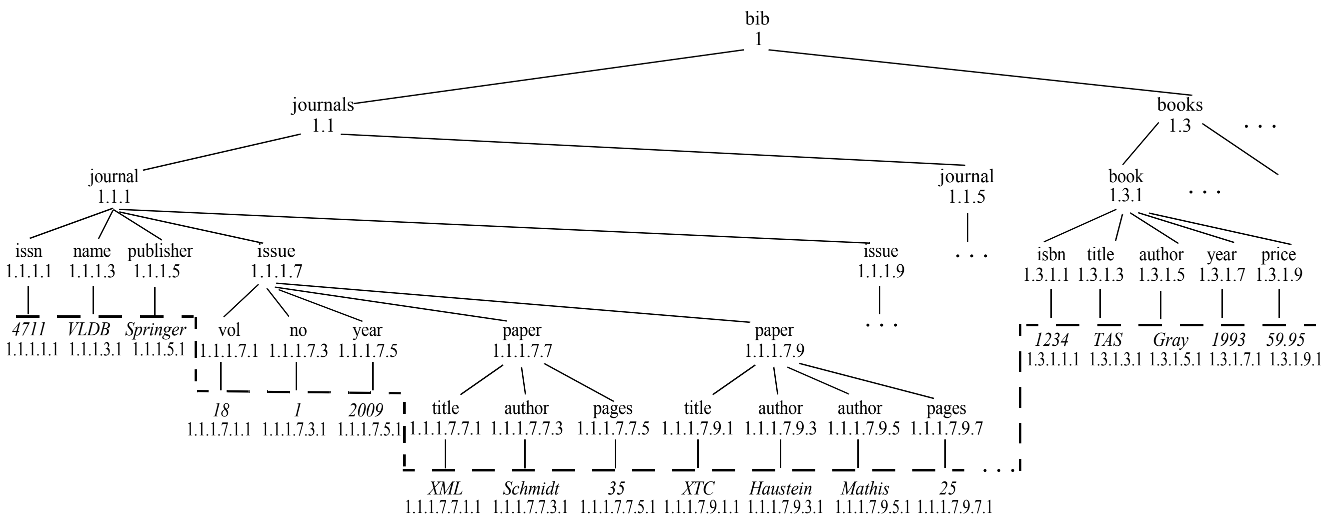


**Fig. 3** XML fragment labeled with DeweyIDs

all publishers whose journals contain a paper with the title "XTC". The output node name and the comparison operator are highlighted. As in the relational world, indexes in XML should be secondary access structures and, therefore, optional. Thus, in our discussion, we successively add different types of indexes to the physical layout of our XML database resulting in the following five alternatives:

*Alt. 1*: *No secondary indexes*: The query has to be evaluated directly on the document store. In native XDBMSs, those document stores [39] can support streaming (SAX) and navigational (DOM) access [32]. Query evaluation over streamed XML has been discussed in various papers, e. g., [3]; navigational processing is the backbone of the XQuery Formal Semantics [7].

*Alt. 2*: *Content index*: In the navigational setting above, our sample query can only be evaluated top-down, i. e., starting by matching the query root *journal*, navigating to the *paper* children, etc., thus, possibly visiting many intermediate nodes that do not lead to an "XTC" title. A content index [23] (inverted list) can alleviate this situation by enabling bottom-up navigational evaluation: The content index returns all occurrences of the string "XTC" in the document, which are used as entry points for the navigational evaluation of the structural part.

*Alt. 3*: *Element index*: These index structures enable a merge-join-based evaluation of twig queries [5] by inverting (inner) XML elements, too. The Holistic Twig Join (HTJ) applied to our example computes the twig matches on a set of sequences, one for the "XTC" content nodes (from the content index) and one for each name test such as *paper* (from the element index).

*Alt. 4*: *Content-and-Structure index*: Our query tree has two subpaths: `//journal[.//paper/title=''XTC'']` and `//journal/publisher`. So-called hybrid or content-and-structure (CAS) indexes [19,20] deliver tailor-made support for the evaluation of the first subpath. Those index structures can directly evaluate simple path expressions consisting of steps on the child (/) or descendant (//) axis and a content predicate. In our sample query, we can substitute the left branch of the query by an index scan operator that delivers all *journal* elements. The remaining subpath can be evaluated using alternatives 1 or 3. In this scenario, the content index would not be required.

*Alt. 5*: *Path index*: The above decomposition leaves the path `//journal/publisher` open for further treatment. Pure path indexes [9,26] support their evaluation: The left branch is answered by a CAS index, the right branch by a path index. We only intersect the intermediate results on the *journal* nodes.

## 2.1 Problems Observed

After this high-level discussion, you may be convinced that the existing techniques sufficiently support the evaluation of branching path queries without further ado. However, a closer look reveals several weak points:

*Alt. 1* and *2* only pay off for extremely low (streaming) or extremely high (navigation) selectivity queries: For streaming, the complete document is accessed, while navigations generate random I/Os and subtree traversals [39] (e. g., on the descendant axis). Nevertheless, a native XDBMS should implement at least *Alt. 1* as a fall-back solution (similar to a relational table scan).

*Alt. 3* replicates the document, because all elements and the content are duplicated in the indexes. This implies high update costs. Furthermore, as our experiments will show, the algorithmic result computation is often by an order of magnitude slower than index combination (see Section 4.3.1).

In *Alt. 4*, the indexes are not expressive enough to directly evaluate `//journal[.//paper/title=''XTC'']`.

The reason is that they fail to efficiently provide the required sequence of (inner) team nodes: IndexFabric [18] and FLUX [20] can only return leaf nodes or the document's root node ("document contains path"). To compute the internal nodes, the ancestor paths have to be looked up implying accesses to the document store. In Kaushik's CAS Index [19], the internal nodes are computed by using structural joins. On all approaches, these additional processing steps may cause substantial performance penalties (see Section 4.3.1).

Indexes for *Alt. 5* run into the same expressivity problem. Furthermore, they impose an intrinsic clustering by partitioning the set of XML elements into extents, where all elements in an extent comply with a specific property on their incoming path(s). For example, in the A(k)-Index [18], all XML elements in an extent have the same *k* "immediate" ancestors (k-bisimilarity). Query processing on descendant axes (//) often requires to merge these extents. In the A(1)-Index on our sample document, entries for `paper/author` and `book/author` are in different extents. A query on `//Name` has to merge these extents resulting in further costs.

Another problem common to *Alt. 3 – 5* is the index focus, which directly influences index size and maintenance cost. In most of the above referenced papers, the authors only argue about the size of the structural summary (that defines extents), but not about the size of the extents themselves. Even in adaptive path indexes [6], all inner elements of the complete document are contained in the extents. The term "adaptivity" in these proposals refers to the refinement or coarsening of extents, but not to the index' extent size adaptation. Indexing complete documents is comparable to indexing all columns of a relational table, which is unnecessary most of the time and leads to unjustified update cost.

In general, XQuery gives application developers so many degrees of freedom that the variety of evaluation strategies for our little example query reveals only the tip of the iceberg. Especially twig queries may always be accompanied by more complex predicates like optional branches, positional predicates, or arbitrary content predicates. The twig in the extended query in Fig. 4, for example, is satisfied if the `paper`'s title is *either* `''XTC''` *or* if there exist more than 3 *author* children. Further, only the first *author* child for each *paper* in a qualified *journal* is returned.

Obviously, storage structures and path processing algorithms, optimized for a too narrow class of queries, can so quickly become useless. Either they do not pay off, because additional processing steps eat up their benefit or they can just not be applied to a given query at all. Consequently, it is questionable to develop a system only for a class of queries assumed to be "typical". Any XDBMS should not only support a narrow class of applications, but should cope with a variety of different workloads (which may be unknown at design time), while delivering satisfying performance. Therefore, we believe that a flexible infrastructure

a)
```
for $t in $d//issue
where $t/paper/title="XTC"
or count($t/paper/author)>3
return
  $t/paper/author[position()=1]$
```

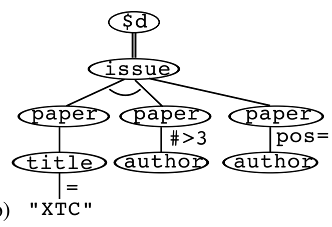
b)


**Fig. 4** Extended sample a) query and b) twig

supporting efficient evaluation strategies for more than just a single type of queries is mandatory to deal with the great variety of queries *and* alternatives to answer them.

Of course, an objective comparison of all approaches shown would be hard to accomplish and is not possible within one paper (actually, this would mean to implement them in one system and to find a fair benchmark). It is even questionable, if such a comparison would be meaningful, because many approaches have a completely different intention and scope (e. g., "shredding" vs. "native"). But, we think it is still worthwhile to reconsider XML storage and indexing, because many of the above approaches concentrate on indexing *or* storage, but not on indexing *and* storage. We therefore can illuminate the opportunities evolving from integrating indexing and storage in a *single scheme*. To concentrate on the integrated use, we exploit well-known XML techniques and DBMS core concepts whenever possible. Consequently, we rely on research that has shown already the flexibility, applicability, and usefulness of each single technique, before we come up with new ideas mostly based on them.

## 2.2 The XMIS Wish List

We start our reconsideration by composing "wish lists" containing what – in our opinion – modern XML applications require from storage and indexing. Based on these desiderata, we then design our integrated XMIS scheme. As the name "wish list" implies, we do not consider the desiderata as normative, but nevertheless assume that they are meaningful for many XML applications. For document stores, we require the following physical properties:[1]

S1 *Round-Trip:* The store should provide some form of round-trip property, that is, it should be able to export stored

[1] To avoid an application-specific focus, these properties are not ranked. For a universal approach, all properties should have the same importance.

documents “without loss”. Some applications require *strict* round-trip, in particular for document-oriented structures such as books or contracts, meaning that the exported result should be byte-wise equal to the originally stored document. For most applications however, a *lax* round-trip is sufficient, where the data model instances (e. g., in the *XML Infoset*) of the stored and the exported version comply. This is typically true for data-oriented applications.

S2 *Storage/Scan*: The store should support fast *storage* and *scan* procedures: Scans are the basis for document and subtree reconstruction, for the implementation of the SAX interface, the fn:data() function, and XQuery result construction. All of them are frequent operations for which performance is mandatory. Fast storage operations are required, because XML is a data interchange format and an XDBMS frequently needs to receive and emit XML documents.

S3 *Navigation*: For the realization of the DOM interface of an XDBMS, navigational operations are required. Furthermore, navigational primitives implement base operators for XML query processing.

S4 *Modification*: With the emerging and recently stabilized XQuery Update Facility [37], all modifications become first class citizens in XDBMSs. To ensure update performance, modifications have to be carried out at a node level (replacing only affected nodes), and not at a document level (i. e., by replacing a complete document with a modified version).

S5 *Documents and Collections*: Documents might come in single instances of large documents or in large collections of small documents. No matter how, the store should efficiently manage them.

S6 *Succinctness*: A space-efficient document store not only saves storage, but also leads to reduced I/O and logging and, thus, better processing performance.

Items S2 and S6 primarily address performance whereas the remaining items target expressive power. You might miss two points in these desiderata that are often associated with XML storage, namely: *XML Schema Validation* and *Versioning* (i. e., keeping a history of modified subtrees in the store). We do not consider these points as physical properties of an XML document store, but rather as logical ones, that can and should be implemented in a layer on top of the physical store.

XML indexes play a major role in evaluating declarative queries over XML data. As already stated, many different types of indexes have been proposed. Here, we only consider path indexing (as opposed to more sophisticated indexes), because path queries frequently occur in declarative XML query languages. Concerning our indexing scheme, we pose the following requirements:

I1 *Optional Use*: As in relational systems, indexes should be redundant, i. e. non-information-bearing access structures. This ensures that indexes can be created on demand to trade query performance with maintenance cost and space consumption.

I2 *Expressiveness*: Indexes should be able to answer path queries supporting the child (/) and descendant (//) axes, name tests, wildcards (*), and one optional structural or content predicate, e. g., `//paper[pages> 25]`. This expressiveness is sufficient enough to answer single path queries directly and branching path queries by index combination, which are the fundamental elements of declarative (X)query processing.

I3 *Selectivity*: Index selectivity, i. e., which paths are actually covered, should be user-defined. It ensures that a set of indexes can be adjusted to document characteristics, query workload, and maintenance overhead.

I4 *Updates*: The index should enable effective updates. Depending on index selectivity, not all document updates lead to index updates. But, efficient mechanisms should exist to discover when an index needs maintenance.

I5 *Applicability*: The test whether an index can be applied for query evaluation should be simple and cost-efficient. This task is similar to that of the previous requirement thereby enabling complexity reduction and facilitating synergies.

I6 *Result Computation*: An index should allow to retrieve *all* elements on an indexed path. If an index exists, e. g., on the content of *author* nodes, it should also allow to return all matching nodes on the paths to the document root, e. g., for query `//paper[author=‘‘Haustein’’]` the *paper* nodes. This is of major importance for the integration into a query engine because, otherwise, expensive reconstruction of internal elements would become necessary for subsequent processing.

I7 *Efficiency*: An efficient physical storage layout, adjusted access algorithms, and effective locality properties are crucial for the eligibility of an index.

I8 *Minimality*: In contrast to the variety of types and storage structures proposed for XML indexing, the set of different index types used in an XDBMS should be minimal and based on the same mechanism to be reused for their implementation. Nevertheless, this minimal index set should provide enough flexibility to cover the functionality of all existing index proposals.

In the next sections, an integrated XML indexing and storage scheme is developed that fulfills the requirements posed in the above two lists. Based on the flexibility expected by XML applications and exploited by query languages, these requirements seem to be inevitable to liberate the isolated use of tailored XML processing techniques. Therefore, a carefully designed toolbox of index and storage techniques is necessary to provide a robust, flexible, and efficient foundation for further optimization. Employing standard techniques[2], this toolbox – our XMIS scheme – is realized in XTC (XML Transaction Coordinator).

## 3 XML Storage

An XML storage scheme is actually a mapping from a hierarchical XML instance, such as our running example in Fig. 3, to a sequence of blocks on external memory. Early approaches did not set a high value on items S2 to S4, because they focused on static XML documents. Nowadays, flexible manipulation of dynamic XML documents and their fine-grained modification in multi-user ACID transactions are indispensable. For this reason, native mappings are prime candidates to realize suitable XML storage structures.

Before we do so, however, we have to introduce three essential ingredients: *node labels*, the *path synopsis*, and *path class references* (PCRs).

### 3.1 Node Labels

In an XML storage scheme, node labels provide for logical references to distinct nodes in an XML tree. Hence, the node labeling scheme is *the key* to fine-grained, effective, and efficient handling of XML documents.

In the first place, a labeling scheme has to guarantee *uniqueness* and *order preservation* of node labels. Moreover, if two node labels are given, the scheme should directly enable *testing of all (important) XPath axes*: all axes relationships should be determined by computation only, i. e., access to the document (on external storage) is not needed. XML documents definitely require *immutable labels* even under heavy updates/insertions to guarantee stable node labels during transaction processing. Therefore, they are a prerequisite for processing dynamic documents as required in S4. Furthermore, a given document node label should enable the reconstruction of *all ancestor labels* without accessing the document. As we will see, cheap ancestor reconstruction is essential for items S6 and I6. Immutability and ancestor label computation have far-reaching consequences to the DBMS-internal processing efficiency: These properties greatly support intention locking for a specific node and all its ancestor nodes on the entire path to the root [15]. They also support efficient path matching, which is crucial for query processing, especially in case of twig queries.

As explored in [12], a node identification mechanism satisfying all these properties rests on a prefix-based labeling scheme for which a few variations are proposed: OrdPaths, DeweyIDs, or DLNs [27] are adequate and equiva-

[2] All variations of document stores and index types are implemented using B*-trees. With code reuse for the base structures, the tree entries only differ in the representation of keys and values.

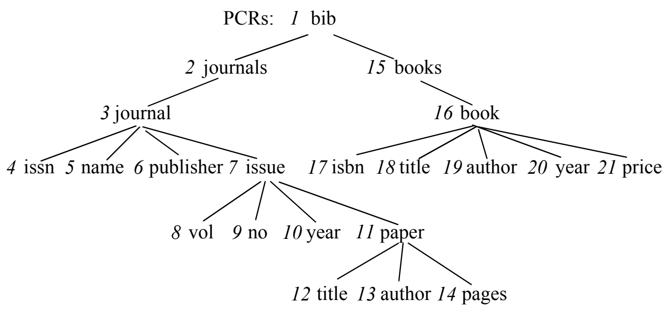


**Fig. 5** Path synopsis

lent for our use. Therefore, we prefer the acronym SPLID (Stable Path Labeling IDentifier) as a generic reference to such prefix-based labeling concepts. A SPLID consisting of *divisions* can be represented (in the human-readable format) by a variable-length, dot-separated integer list as illustrated in Fig. 3. Hence, its size depends on the level of the node labeled. Therefore, a space-optimal implementation is mandatory, which is achieved by XTC using Huffman-encoded divisions. Furthermore, SPLIDs in lexicographical order lend themselves to very effective prefix compression – exploited by XTC for document store and all index reference lists. As a consequence, the SPLID size ranges from 3 to 6 bytes in the average and is, therefore, comparable to that of TIDs.

### 3.2 Path Synopsis and Path Class References

A path synopsis (see Fig. 5) is an *unordered*[3] structural summary of all (sub)paths of the document. Each non-content node belongs to a path class representing all path instances having the same sequence of ancestor labels. To facilitate the use of path classes, we enumerate them with so-called *Path Class References* (PCRs) serving as a simple and effective path class encoding. PCRs are used links from index or document entries to the path class they belong to.

The size of the synopsis depends on the document's structural complexity. It can usually be stored in a single page on external storage. For fast access, it should reside in a small data structure kept in main memory. Note, the path synopsis is conceptually equivalent to a 1-Index [26] or a strong DataGuide [9], and therefore originates from the indexing community. In the path-oriented storage scheme, the synopsis is re-used to optimize document storage.

We want to emphasize the expressiveness of SPLIDs and PCRs: a SPLID delivers all SPLIDs of its ancestor path, while a PCR delivers – by means of the path synopsis – the element and attribute names of the ancestor path. Together,

[3] In the following, we are only interested in the path up to the root for a given PCR. Therefore, the relative order among siblings is not relevant, e. g., all permutations of elements *issn*, *name*, *publisher*, and *issue* as children of *journal* may appear in the document.

they form a kind of coordinate system for the XML document. For example, starting from an arbitrary node whose SPLID and PCR are known, it is easy to reconstruct the complete path instance the node belongs to. For example, text node (“Gray”, 1.3.1.5.1, 19) enables without document access to compute `/bib/books/book/author/‘‘Gray’’`. Because a SPLID contains all its ancestor labels, all SPLIDs along the path to the document root are delivered for free.

Because a path synopsis does not need order, maintenance in case of document evolution (creation of new path classes) or shrinking (deleting the last path instance of a path class) is very simple. New path classes and related PCRs can be added anywhere and existing, but empty path classes do not jeopardize correctness of path synopsis use. Furthermore, hash-based access to the PCRs guarantees efficient evaluation. Providing substantial mapping flexibility, effective lock management support, and also considerable speed-up of query evaluation [13], the combined use of SPLIDs and path synopses/PCRs turned out to be a *key concept* for fulfilling many requirements stated in the wish lists.

## 3.3 Path-Oriented Document Storage

The logical representation of XML documents, as visualized in Fig. 3, can be used to derive a more or less direct mapping to a physical representation. Observing document order (left-most depth-first), a straightforward approach inserts into a B*-tree an entry $< SPLID, D, C >$ for each document node, where *SPLID* is used as key, *D* is a descriptor designating the node type (i. e., element, attribute, text, etc.), and *C* is the encoded content. The content is a byte sequence representing either a text node or, in case of element and attribute nodes, a *VocID* reference (thereby replacing the “long” external names) to a vocabulary entry. We refer to the inner structure of the B*-tree as the *document index* and the leaf pages as the *document container*. This approach is called *naive node-oriented* storage scheme, because *every node* is explicitly stored. A slight variation of the naive approach is the so-called *prefix-compressed node-oriented* storage scheme (or *pc* for short), in which prefix compression on the keys in the B*-tree is enabled, thereby substantially reducing the SPLID space consumption (see Table 2).

Collections of (small) XML documents can be stored by creating a single document with a “virtual” root node for the whole collection. This way, storage and indexing space is shared and memory fragmentation is minimized in case of many small collection documents.

The problem with the *naive* and the *pc* approaches is that they are often “not succinct enough”. Further storage space reductions are possible by consequently removing *structural redundancy*. To measure the structural redundancy, we compared the number of *distinct paths P* in a collection of real

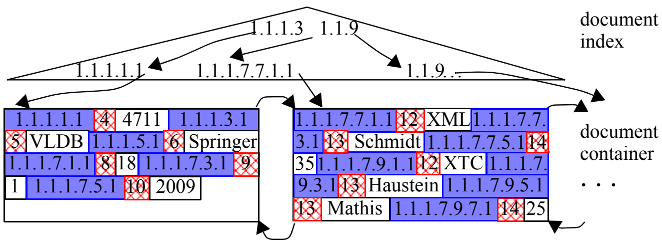


**Fig. 6** XML fragment of Fig. 3 stored in *po* format

world documents taken from [25] with the number of *all paths A* in the same documents. In all documents, *A* was several orders of magnitude larger than *P*. Our sample document also reveals this kind of redundancy: The *paper* subtree occurs multiple times; while the content values differ from subtree to subtree, the structure remains the same.

The basic idea of path-oriented storage denoted by *po* is to avoid to explicitly store the document structure by *virtualizing* it, as depicted in Fig. 3. Only the text nodes below the dashed line need to be stored. If we associate PCRs with these text nodes, it is possible to recompute the inner document structure for every path as explained in Section 3.2. Therefore, whenever a reference to an inner node or a path is required or an operation is applied, the desired node is recomputed using its SPLID and the related PCR together with the path synopsis.

As shown in Fig. 6, the physical *po* representation contains an entry $< SPLID, PCR, D, C >$ for each document’s leaf node, where *SPLID*, *D*, and *C* are defined as for node-oriented storage, and *PCR* refers to the text node’s parent element. Because the SPLIDs are stored in lexicographical order, we effectively apply prefix compression on them.

Again, we reconsider the new storage scheme based on our wish list. Items S1 and S5 can easily be checked off: Because the *po* scheme (similarly to *naive* and *pc*) cannot encode inner element markup, only the lax round-trip property is supported. Obviously, collection management can be smoothly implemented. The remaining “wishes” will be considered in the following.

## 3.4 Performance Evaluation

To substantiate how the remaining wish list items are fulfilled, we will briefly sketch the operational behavior of *po* XML documents. To estimate the performance difference compared to *naive* and *pc* storage, we give relative numbers for the improvements achieved [21]. Our empirical evaluation refers to the well-known XML document collection [25] widely used for cross-comparisons. A summary of their characteristics is listed in Table 1, where data size refers to the *plain* format, i. e., the “external” XML format.

**Table 1** Characteristics of XML data sets used

| Aspect | lineitem | uniprot | dblp | psd7003 | nasa |
|---|---|---|---|---|---|
| Data size (MB) | 32.3 | 1,821.0 | 330.0 | 717.0 | 25.8 |
| Nodes (Mio) | 1.98 | 135.48 | 17.42 | 39.84 | 0.89 |
| Max/avg depth | 4/3.45 | 7/4.53 | 7/3.39 | 8/5.68 | 9/6.08 |

All performance measurements were run on a Xeon 2,66 GHz server with 4 GB main memory and 500 GB external memory and Java Sun JDK 1.6.0. XTC was configured for 8 KB pages and 64 MB buffer.

### 3.4.1 Storage Consumption

We implemented all sketched storage schemes and measured the storage consumption for our reference collection of real world documents. Because all entries have variable length, some administrative overhead (*admin*) is needed for byte alignment. In the *naive* and the *pc* schemes, VocIDs were encoded using 2 bytes. In *po*, a PCR required 1 byte.

As compared to the *plain* format, *naive* as the straightforward internal format typically achieves a storage gain of $\sim 10\%$ to $\sim 30\%$, although the saving from *VocID* usage is partially compensated by the need for node labels. Table 2 summarizes the results of dedicated experiments to determine the space consumption of the XML storage schemes under consideration. Note, we focus on the relative saving regarding the structure part only, because content compression is orthogonal and could be applied to either scheme. Therefore, we have separated the influence factors contributing to the compression of structure entries. First, we consider the influence of the label encoding. (Note, despite the "obvious length" of SPLIDs, range-based or sequential labeling schemes [12] would consume more storage, because they do not lend themselves to compression). For *naive* and *pc*, uncompressed resp. prefix-compressed SPLIDs have to be stored for all nodes in the document, whereas *po* only requires prefix-compressed SPLIDs for the content part. Computing the size reduction from column 3 to column 4 in Table 2 tells us that that prefix compression is very effective; the relative storage space needed for node labels is reduced by a substantial margin, in our case by $\geq 76.8\%$. Nevertheless, the optimization from *pc* to *po* adds a further storage gain of $\sim 40-55\%$ (see column 6 of Table 2).

References and *admin* data are format-dependent. Only content nodes carrying PCRs are stored in *po* format, whereas VocIDs are needed for all element/attribute nodes in the *naive* and *pc* formats. As listed in column 9 of Table 2, the relative gain of *po* compared to *naive/pc* is considerable. Indicative overall improvement factors (*uniprot*) are $\sim 47\%$ for *pc* and $\sim 73\%$ for *po*, respectively[4]. For our reference documents, these factors range from $\sim 40\%$ to $\sim 52\%$ for *pc* and from $\sim 70\%$ to $\sim 80\%$ for *po*, respectively. To the best advantage, our novel optimization step (from *pc* to *po*) accounts for a space reduction of $\sim 50-58\%$.

[4] For *uniprot*, the structure-related saving of the *pc* and *po* formats consists of 465 and 731 Mbytes w.r.t. the *naive* format. Content compression would reduce the content part by 23 – 35%, in addition.

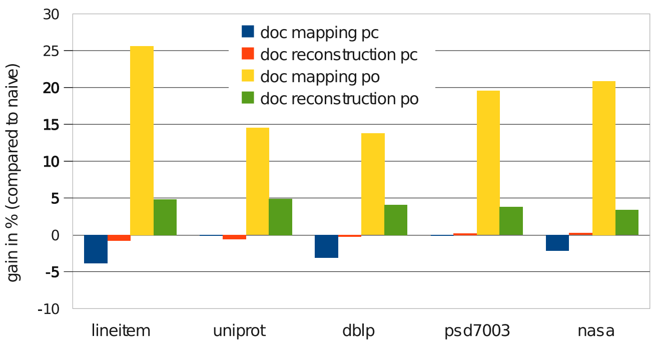


**Fig. 7** Storage / reconstruction gains (po vs. pc)

### 3.4.2 Storage and Reconstruction

Due to space restrictions, we can only sketch the storage and scan algorithms (item S2): For storage, a SAX parser is used to simultaneously create the path synopsis and the necessary B*-tree entries, when leaf nodes are encountered. Again, the index can be built bottom-up [10], because the entries are already sorted on their SPLID used as a key.

Document reconstruction starts at the left-most record $R_1$ in the document container, for which the root-to-leaf path is reconstructed. Then, using SPLID arithmetics, the least-common ancestor $L$ of $R_1$ and its next neighbor $R_2$ is computed. All nodes on $R_2$'s root-to-leaf path below $L$ are computed, and so on. Hence, both storage and reconstruction are scan-based methods with low memory footprint.

A first indicator for efficient processing is the overhead for a document arriving in its external format (*plain*) to transform and store it using the specific internal formats *naive*, *pc* or *po* and, in turn, to reconstruct it again from the physical storage representation. Obviously, these processing times are more or less linearly dependent on the document sizes. Because the sizes of our reference documents differ almost by two orders of magnitude, direct response times are hard to compare. Therefore, we refer to normalized gains for both storage structure optimization w.r.t *naive*. We define them as $G_{po} = (t_{naive} - t_{po})/t_{naive} * 100\%$ and $G_{pc} = (t_{naive} - t_{pc})/t_{naive} * 100\%$, respectively. In Fig. 7, these gains are illustrated for the document mapping resp. document reconstruction times of all documents. The *pc* mapping overhead in both directions is more or less identical to that of the *naive* method, but identifies a negative gain in some cases, i. e., slightly higher mapping costs due to SPLID encoding. However, we can show that storing a virtualized document is substantially faster than using the *naive* (and *pc*) methods ($\sim 14-25\%$), Even *po* document reconstruction is more efficient in

**Table 2** Storage consumption of XML documents in MB

| | | Structure | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Document | Content | SPLIDs | compressed SPLIDs | | | naive/pc | po | gain in % |
| | | (naive) | pc | po | % | | | |
| lineitem | 6.2 | 8.6 | 2.0 | 1.0 | 50.0% | 8.0 | 3.9 | 51.2% |
| uniprot | 668.9 | 582.7 | 117.7 | 70.2 | 40.3% | 433.4 | 214.4 | 50.5% |
| dblp | 174.2 | 71.1 | 16.7 | 8.8 | 47.3% | 65.8 | 33.4 | 49.2% |
| psd7003 | 293.0 | 258.3 | 40.0 | 17.9 | 55.2% | 161.1 | 69.3 | 57.0% |
| nasa | 12.4 | 5.2 | 0.9 | 0.4 | 55.5% | 3.6 | 1.5 | 58.3% |

all cases (~4 − 5%), i. e., the computation overhead of the structure part is more than balanced by I/O saving.

### *3.4.3 Navigation*

Full DOM access (item S3) is provided by the *po* storage, too. The four navigational primitives *first child*, *last child*, *previous sibling*, and *next sibling* require only a single top-down traversal of the document index (typically of height $\leq 2$ and $1-60$ leaf pages of 16 KB) 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 is sufficient to execute any navigational operation. For brevity, we skip a detailed discussion on how these navigational primitives are implemented. They are quite straightforward. In summary, they translate to some B*-tree lookup methods that work on prefix-based key comparisons.

In an XDBMS, navigational operations are either directly executed via a given API (e. g., DOM [32]) or by the implementation of XML query operators. Because execution 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 *first_child / next_sibling* ($T_1$) and *last_child / previous_sibling* ($T_2$). In case of *po*, root and inner nodes are virtual and have to be recomputed during the tree walk.

Fig. 8 shows substantial gains for *po* documents over *pc* documents. In all cases, we achieved improvements, most of them in the range ~7 − 32% for $T_1$, resp. ~9 − 38%, for $T_2$. These gains are due to less I/O operations and shorter node reconstruction times on the compact *po* documents.

### *3.4.4 Updates*

No matter what kind of language model is used for document modification, it has to be translated into node-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 is carried by two structural features: B*-trees and SPLIDs. B*-trees enable logarithmic access time under arbitrary scalability and their split

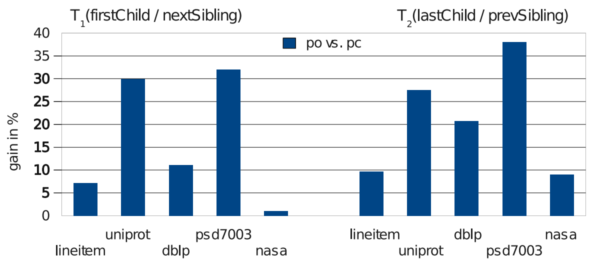


**Fig. 8** DOM navigation gains (po vs. pc)

mechanism takes care of storage management and dynamic reorganization. In turn, SPLIDs provide immutable node labeling such that all modifications can be performed locally.

Referring to Fig. 6, a context node *cn* (stored in the document container) can be located either by navigation within the document, 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 Fig. 3, the deletion of the *journal* node labeled by 1.1.1 results in a deletion of all records in Fig. 6 that start with 1.1.1. Because all entries to be removed are stored consecutively in the B*-tree, fewer entries have to be deleted compared to the *pc* format.

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 a smaller set of consecutive entries as for *pc*.

A *pc* document contains an entry for each inner node *n*. In *po* format, *n* is a virtual node. Thus, compared to *pc*, renaming of an inner node [32] is the only possibly expensive operation in *po*, where all PCRs of leaf nodes in *n*'s subtree (identified by *n*'s SPLID) have to be updated (additionally, the path synopsis has to be altered, if renaming introduces new paths to the document). For a performance evaluation combined with index maintenance, see Section 4.3.2.

## 4 Indexing XML Documents

*Path and tree pattern* queries are essential ingredients in XML query languages and matching such patterns in XML documents is a frequently occurring task. Hence, their evaluation is critical to XDBMS query performance. Because these patterns may be complex and their optimal evaluation may depend on quite a number of parameters, e. g., XPath axes specified, element selectivities present, shape of the document tree, clustering aspects, etc., the spectrum and the richness of the different proposals concerning XML indexing can be hardly overlooked. However, the uncontrolled growth of index types, as stated in Fig. 1, and their hodgepodge of implementation mechanisms is not recommendable for XDBMS use.

From a logical point of view, *element* indexes and *content* indexes are sufficient to evaluate – without scanning or navigating the document – all types of set-oriented requests needed for XQuery/XPath processing. An element/attribute index refers to structure nodes, whereas a content index enables direct document access via text values. These access paths use SPLIDs to refer to the indexed nodes, which are located via the document index. From a performance point of view, however, the sole existence of both index types would be disastrous. Furthermore, large parts of the document had to be often accessed in a *random* node-at-a-time manner, which penalizes performance twice, due to extensive I/Os and unnecessarily large blocking potential to guarantee serializable transactions. Therefore, indexes achieving much more selective document access are mandatory.

As already stated in our wish list, we want to design indexes that can evaluate simple (non-branching) path expressions with an optional content predicate (item I2). Those expressions build the cornerstone of most XPath and XQuery expressions. Our first definition states the shape of the supported queries more precisely:

**Definition 1** *A simple path query expression is formally denoted by $p[T]$, where $p = e_1t_1e_2t_2...e_k$ is a path and $T$ is an optional content comparison predicate or an optional relative path without further predicates. Path $p$ consists of descendant (//) and child (/) edges $e_i$, as well as element and attribute node tests or wildcards $t_i$. If $T$ is a comparison predicate, edge $e_k$ of path $p$ refers to a node the value of which can be $\Theta$-compared. Predicate $T$ is then of the form $C = [t_k \Theta v_i]$ for a simple comparison or $R = [v_i \Theta_1 t_k \Theta_1 v_j]$ for a range comparison, where $\Theta_1$ is suitably chosen from $\{=,<,\leq,>,\geq,!=\}$ 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. If $T$ is a content predicate, we call the query a content-and-structure (CAS) query; otherwise, we call it a plain path query.*

Examples $Q_1$ = `//journals//issue/[year != 2009]` and $Q_2$ = `//paper//[1 ≤ Integer ≤ 5]` are CAS queries on the document in Fig. 3. $Q_3$ = `//book[./price]` and $Q_4$ = `//book/price` are plain path queries with and without a relative path predicate.

Note, our simplified path specification is fully compatible to XPath and can easily be transformed into an XPath expression, where the range comparison is replaced by an XPath-conform *and* conjunction of two value comparisons, and the type check is extended to a type cast. But, to ease the index definition and query examples, in the following, we use our simplified path specification from Definition 1.[5]

## 4.1 XML Index Types Minimized

Most indexing solutions proposed so far (see Section 5.2), only exploit *structure*. To answer CAS queries efficiently, the combined indexing of *values and structure* is necessary. Therefore, we provide a hybrid index that captures content and structure. The resulting access structure is called CAS index and is 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 path query (without a relative path predicate), and $T$ is the indexed content type (e. g., Integer, String, etc.)[6]. For each leaf node $n$ of $D$, an entry is contained in $I_D(p,T)$, iff ($C_1$) the parent element of $n$ 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. An index entry has the form $< \underline{C}, SPLID, PCR >$, where $\underline{C}$ is the content of the indexed leaf node used as key. The keys of $I_D$ are in ascending order w. r. t. $T$, while the SPLIDs (occurrences of $n$ in $D$) are in document order for one and the same key value.*

Obviously, a CAS index is optional, and thus fulfills item I1. It contains all values (of a certain type) that reside on a certain path. Because every search tree enables checking of all $\Theta$-based value predicates, B*-trees are the best choice for our base index structure. As an example, consider $I_0$ = `I(/bib/journals/journal/issue/year, [Integer])` which indexes all values of element *year* in the related path class, i. e., for each value $v_i$ occurring for *year*, a node reference list is maintained (in document order) which stores the SPLIDs for nodes (records) having $v_i$ as a value.

When an index refers to a single path class (or a unique leaf element name), all SPLIDs occurring in the reference lists carry the same PCR. Therefore, the PCR can be factored out as index metadata resulting in homogeneous reference lists (see Fig. 9a). In contrast to such *unique* CAS indexes, *collective* CAS indexes refer to several path classes and, hence, carry multiple PCR values resulting in heterogeneous reference lists (see Fig. 9b).

[5] Although we simplified the path specification for presentation purposes, their XPath equivalent is used to express twig queries as well by simply combining several path specifications (join) and, thereby, allowing their application for twig operators.

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

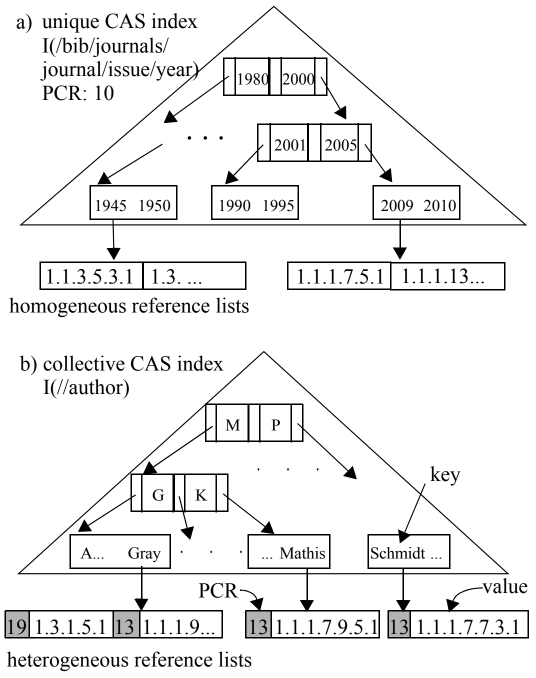


**Fig. 9** CAS indexes

A *content index* is defined by $I_D(T)$, where $D$ is the document and $T$ is the indexed content type. Obviously, CAS indexes are expressive enough to "simulate" content indexes, thus avoiding their separate implementation. If an XML document stores only text values, a *generic* index `//*[text]` would index all its content. If a document carries typed content other than text, we need specific typed content indexes resp. CAS indexes to organize the typed values in B*-trees and to inherit their salient properties such as range search. For example, a CAS index defined by `//year` (with inherited type [integer]) delivers all values under the path class `/bib/journals/journal/issue/year` and the path class `/bib/books/book/year`, whereas `//*[integer]` delivers all integer values in the document, i. e., values of the elements *issn*, *vol*, *no*, etc. in addition.

To support plain path queries by indexes, we again have to slightly modify our CAS indexing scheme. A plain *path index* definition has the form $I_D(p)$, i. e., no content type information is given, where the entries stored in a plain path index have the form $< PCR, SPLID >$. Examples are $p_1 = //journal//paper/author$ or $p_2 = //books/book/author$. In contrast to a CAS index, where only leaf nodes are maintained, a path index inverts inner document nodes which are not physically stored in *po* schemes. However, using path synopsis and index entry, the referenced inner node can be easily recomputed.

Eventually, a special form of a path index leads to an *element index*: $p_3 = //author$ inverts all *author* elements.

In all cases discussed, an indexed value is associated with a list of $< SPLID, PCR >$ references to the related document nodes. Such a combined reference enables together with the path synopsis the reconstruction of the entire path without accessing the document. In particular, an index defined by predicate $p_3$ contains all paths indexed by $p_1$ and $p_2$. Furthermore, as a salient performance feature, their variable-length reference lists can be defined by $< PCR, SPLID >$ for *PCR clustering* or by $< SPLID, PCR >$ for *SPLID clustering* (within potentially very long reference lists).

Note, XTC indexes rely on a single B*-tree-based indexing scheme which is more expressive than all schemes together summarized by Fig. 1 (item I8). Compared to those conventional *content* and *element* indexes, which deliver only the respective node references, we obtain the entire structure information via path synopsis and PCRs for free. This mechanism makes our structure-oriented indexes much more flexible and expressive. CAS indexes are particularly powerful, because a large share of matching queries can be evaluated solely on the index structure. Only when additional nodes are needed for output, access to the disk-based document is inevitable. This is even true when CAS indexes refer to documents in *po* format, i. e. with virtualized structure part. Therefore, indexing SPLIDs and PCRs is *the key* to efficiently implement item I6, i. e., the computation of inner elements on an index result without document access.

## 4.2 Index Maintenance

Upon creation of CAS index $I$, a record including $I$'s index definition $I_D(p,T)$ is inserted into the metadata catalog. This allows us to recompute the set of PCRs for $p$ on the path synopsis of document $D$ at any time. To avoid perpetual evaluation of $p$, we cache the result in the path synopsis to cheaply decide index matching for queries and the need of index update after document modifications [22].

Generally, there are two types of modifications: The first type does not alter the path synopsis while the second one does. Subtree insertions/deletions for the first type can be handled as follows: For each content node $n$ in the affected subtree, its parent's PCR $p$ is inferred from the path synopsis. If any index definition's PCR set in the metadata contains this PCR (hash lookup), the modification is propagated to the corresponding index, because $n$ is contained in that index. Modifications altering the path synopsis trigger a recomputation of PCR set deltas on demand. Then, the same process as described above updates the indexes.

As an example, consider index `I(//author)` on our sample in Fig. 9b, where the PCR set $\{13,19\}$ involved can be derived from the path synopsis (see Fig. 5). If we alter for the first paper depicted the content of the *author* node to "Bächle", we can infer PCR 13 from the affected content node and detect that our index has to be updated. If we add a path `bib/videos/video/author` so far non-existing, we have to alter the path synopsis, resulting in new PCRs (say

22, 23, and 24 (for *author*)); PCR 24 is then added to $I$'s PCR set, before the index is updated as above. Note, because the path synopsis is unordered, we do not need to reassign PCRs at any time, thus, PCRs are stable.

## 4.3 Quantitative Results

In the experiments, we focus on the speed-up gained by 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. Our reference documents used so far cannot be generated and, therefore, do not provide comparable selectivities (for nodes or paths) when predicates are evaluated. Instead, we used generated XMark [31] documents where these properties could be controlled. This fact greatly enabled scalability considerations with comparable structural properties and value distributions. We used the same runtime environment as described in Section 3.4.

### *4.3.1 Index-Supported Queries*

The four queries $Q_1 - Q_4$ of Table 3 – evaluated on documents of sizes 10 MB, 100 MB, 500 MB, and 1GB – represent typical XPath/XQuery expressions and give some insight on how suitable path and CAS indexes influence overall query performance. The reported response times are the average over 10 runs; final result materialization is excluded because it is the same for all evaluation strategies.

Table 3 also shows various indexes for our performance evaluation. To better illustrate storage size and cardinality properties (items I3 and I7) of each index, we included the statistics of the primary storage – document index $D$ – for each document.

Query evaluation on document index $D$ always represents the baseline (item I1). A full scan is required, i.e., each node has to be fetched from disk. Thus, document sizes directly correlate with processing times, where the additional evaluation of simple predicates as in Q2 and Q4 is negligible. Detailed performance figures for $Q_1$,$Q_2$ and $Q_3$,$Q_4$ can be found in Table 4 and 5, respectively.

Twig query $Q_1$ definitely benefits from tailored path indexes $P_1$, $P_2$ because they allow to evaluate the path expression without document access, i.e., their index definitions completely cover the query paths (item I3). Consequently, Fig. 10 reveals runtime improvements in the order of several magnitudes. As $P_2$ perfectly matches the query pattern, index fetches were minimized which leads to optimal processing times (item I7). Similarly, although no content predicate needs to be evaluated, the CAS indexes $C_1$, $C_2$ are also applicable, but slightly slower compared to the path-only variants (item I5). Again, the more specific definition of $C_2$ (i.e., PCR set) pays off.

**Table 4** Runtimes for $Q_1$ and $Q_2$

| Case | | $Q_1$ | | $Q_2$ | |
|---|---|---|---|---|---|
| | doc | fetched | runtime | fetched | runtime |
| $D$ | 10 | 140,463 | 266.1 | 140,463 | 271.1 |
| | 100 | 1,400,423 | 3,032 | 1,400,423 | 3,058 |
| | 500 | 6,924,339 | 15,204 | 6,924,339 | 15,498 |
| | 1000 | 13,373,794 | 29,478 | 13,373,79 | 30,032 |
| $D$,$G$ | 10 | – | – | 0 | 0.8 |
| | 100 | – | – | 143 | 3.9 |
| | 500 | – | – | 93 | 3.1 |
| | 1000 | – | – | 74 | 2.6 |
| $D$,$P_1$ | 10 | 10,858 | 13.6 | 11,577 | 30.6 |
| | 100 | 107,007 | 120.4 | 114,273 | 156.1 |
| | 500 | 529,796 | 608.6 | 566,404 | 482.8 |
| | 1000 | 1,024,635 | 1,181 | 1,095,523 | 1,566 |
| $D$,$P_2$ | 10 | 719 | 6.9 | 1,438 | 23.6 |
| | 100 | 7,266 | 10.3 | 14,532 | 46.6 |
| | 500 | 36,698 | 46.7 | 73,216 | 234.8 |
| | 1000 | 70,888 | 93.9 | 141,776 | 453.3 |
| $D$,$C_1$ | 10 | 7,217 | 14 | 0 | 0.3 |
| | 100 | 73,102 | 126.2 | 94 | 2.9 |
| | 500 | 362,402 | 640.7 | 53 | 1.6 |
| | 1000 | 701,369 | 1,222 | 32 | 1.3 |
| $D$,$C_2$ | 10 | 719 | 13 | 0 | 0.5 |
| | 10 | 7,266 | 16 | 10 | 0.8 |
| | 500 | 36,608 | 69.5 | 4 | 0.3 |
| | 1000 | 70,888 | 136.7 | 4 | 0.3 |

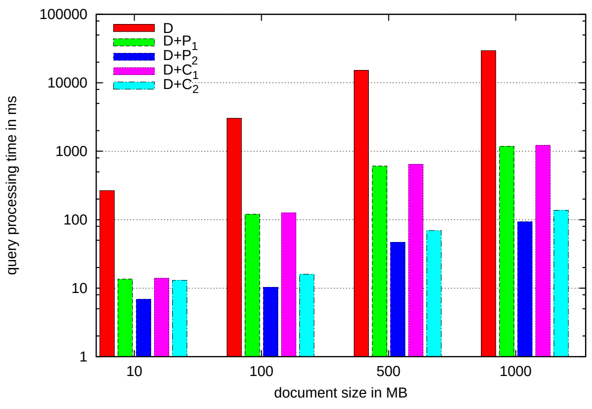


**Fig. 10** Twig query $Q_1$

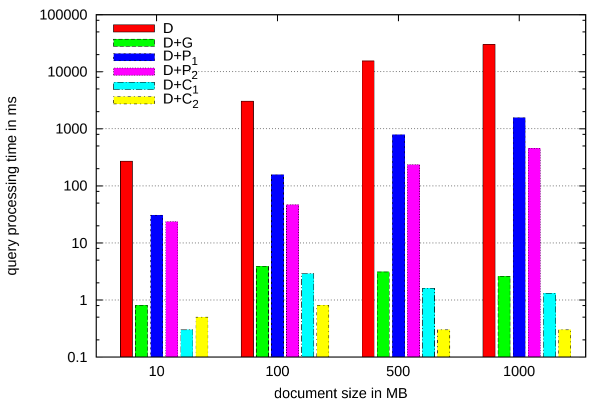


**Fig. 11** Point query $Q_2$

**Table 3** XMark documents, indexes, and queries

| Document Size | | 10M | | 100M | | 500M | | 1000M | |
|---|---|---|---|---|---|---|---|---|---|
| Query | | result size | | result size | | result size | | result size | |
| $Q_1 : //australia//incategory[@category]$ | | 719 | | 7,266 | | 36,608 | | 70,888 | |
| $Q_2 : //australia//incategory[@category = "category432"]$ | | 0 | | 10 | | 4 | | 4 | |
| $Q_3 : //closed\_auctions//closed\_auction/price$ | | 863 | | 8,677 | | 42,900 | | 82,875 | |
| $Q_4 : //closed\_auction/price[text() < 0.2]$ | | 0 | | 23 | | 91 | | 177 | |
| Index | #pcr | card | size | card | size | card | size | card | size |
| **DOCUMENT** $D$ | 548 (all) | 140k | 9.1M | 1.4Mio | 91M | 6.9Mio | 450M | 13.3Mio | 872M |
| **GENERIC CAS** $G : //* \wedge //@*$ | 444 | 139k | 5.1M | 1,38Mio | 39.7M | 6.8Mio | 170M | 13.2Mio | 316M |
| **PATH** $P_1 : //@category$ | 7 | 139k | 139k | 107k | 1.4M | 529k | 7.1M | 1Mio | 13.8M |
| **PATH** $P_2 : //australia//incategory/@category$ | 1 | 719 | 8k | 7266 | 90k | 36k | 483k | 70k | 942k |
| **PATH** $P_3 : //price$ | 1 | 863 | 8k | 8677 | 98k | 43k | 516k | 83k | 999k |
| **CAS** $C_1 : //incategory/@category$ | 6 | 8219 | 172k | 73k | 1.5M | 362k | 7.9M | 701k | 15.4M |
| **CAS** $C_2 : //australia//incategory/@category$ | 1 | 719 | 8k | 7266 | 147k | 36k | 778k | 70k | 1.54M |
| **CAS** $C_3 : //price$ [double] | 1 | 863 | 16k | 8677 | 204k | 43k | 925k | 83k | 1.7M |
| **ELEMENT** $E$ | 514 | 149k | 1.7M | 1.4Mio | 18.3M | 6.9Mio | 91M | 15Mio | 186M |

**Table 5** Runtimes for $Q_3$ and $Q_4$

| Case | | $Q_3$ | | $Q_4$ | |
|---|---|---|---|---|---|
| | doc | fetched | runtime | fetched | runtime |
| $D$ | 10 | 140,463 | 299.9 | 140,463 | 268.4 |
| | 100 | 1,400,423 | 2,896 | 1,400,423 | 3,079 |
| | 500 | 6,924,339 | 14,710 | 6,924,339 | 15,620 |
| | 1000 | 13,373,794 | 28,523 | 13,373,794 | 30,211 |
| $D,E$ | 10 | 863 | 6.1 | 1,726 | 22.6 |
| | 100 | 8,677 | 8.6 | 17,354 | 63 |
| | 500 | 42,900 | 42.2 | 85,800 | 567.1 |
| | 1000 | 82,875 | 85 | 165,750 | 1,103 |
| $D,P_3$ | 10 | 863 | 6.1 | 1,726 | 22.7 |
| | 100 | 8,677 | 8.9 | 17,354 | 66 |
| | 500 | 42,900 | 40.4 | 85,800 | 564,6 |
| | 1000 | 82,875 | 79.9 | 165,750 | 1,098 |
| $D,C_3$ | 10 | 863 | 11 | 2 | 0.5 |
| | 100 | 8,677 | 17.5 | 23 | 0.8 |
| | 500 | 42,900 | 82.3 | 91 | 2.7 |
| | 1000 | 82,875 | 154.6 | 177 | 4.5 |

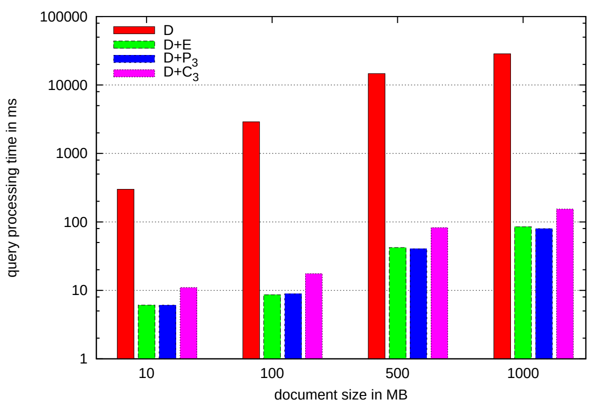


**Fig. 12** Path query $Q_3$

Adding a content predicate, shown in Fig. 11 for point query $Q_2$, led to similar results. Refining the scope of indexes speeds up query execution; in Fig. 11, the CAS index $C_2$ performs best (items I2 and I7) among our index scenarios. To enable comparison with XML content indexes of other systems, we added a generic CAS index $G$. This index $G$ performs quite well, but is slightly slower than the tailored variants while occupying drastically more storage space. Note, query evaluation using path indexes requires additional lookups for each PCR match on document index $G$ resulting in substantially increased processing cost.

The simple path query $Q_3$ is expected to benefit from a path index. The results in Fig. 12 prove that path index $P_3$ performs best, while the larger and "generic" element index $E$ is comparably fast. However, clustering of $E$ heavily benefits from the structure of the XMark documents where only a single path contains the *price* tag. Although CAS index $C_3$ is also applicable, it delivers longer runtimes due to its larger size resulting in a worse buffer behavior.

Query $Q_4$ is a typed range query, where the predicate matches content nodes with a value less than 0.2. Fig. 13 shows that path index $P_3$ is equally fast compared to the "generic" element index $E$, but both require document access for each index node to evaluate the content predicate. The type mismatch also makes all standard (i.e., untyped) content indexes like the generic CAS index $G$ less valuable, because they require a full scan due to their default (string-based) sort order. In contrast, the typed CAS index $C_3$ is perfectly suited to answer the query without any document access at all (item I6) and, therefore, delivers the best result.

Eventually, the number of index fetches directly translates into processing costs. Furthermore, the costs depend on index type and existence of a content predicate. A CAS index delivers nodes ordered by content, whereas a path index delivers nodes in document order, i.e., SPLID-ordered. In case of path index use and predicate evaluation, document access is necessarily leading to random IO. In general, the generic index variants $G$ and $E$ deliver acceptable results, but have poor locality due to their considerably larger scope and size. The fine-tuned path and CAS indexes, tailored to query patterns, are always the fastest alternative. Optional index typing augments the expressiveness of indexes (item I2) and supports typed querying.

### 4.3.2 Document/Index Maintenance

Finally, we checked document updates and index maintenance based on the indexes of Table 3 to illustrate the overhead of the different index types. Additionally, we measured

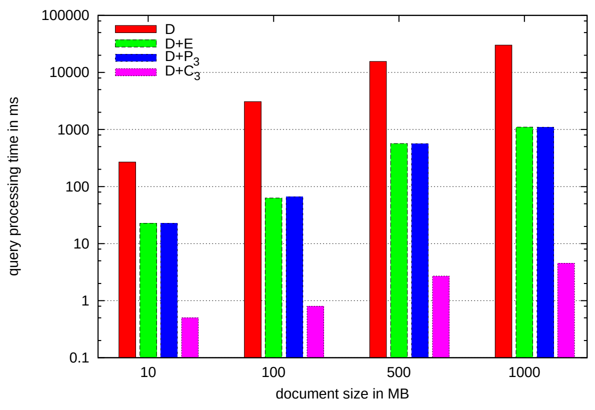


**Fig. 13** Range query $Q_4$

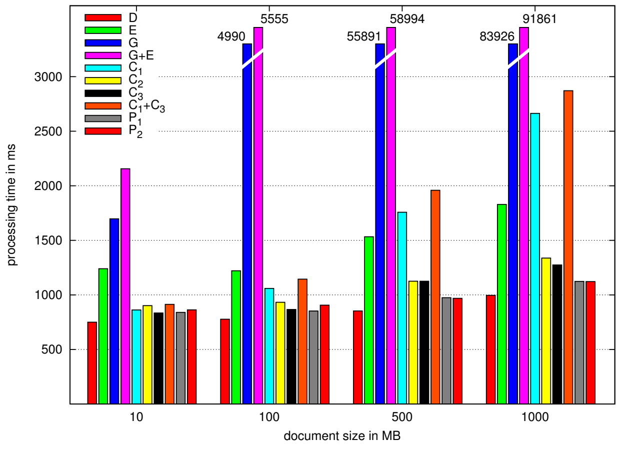


**Fig. 14** Insert performance

the costs of combinations of $G$ and $E$ as well as $C_1$ and $C_3$. The setup $G+E$ is close to typical setups in other approaches where the query engine relies on plain content and element indexes. $C_1+C_3$ exemplifies our view of a good XML index setup: smaller PCR-powered indexes, focused to perfectly support the actual workload.

We measured the time needed for inserting 1000 *item* subtrees, sampled beforehand from a large XMark instance, to have both representative sizes and structural properties for the scenario. We added a *price* element with a random value to each *item* and inserted them at a specified position to ensure that each subtree insertion affects defined indexes.

The processing times for $D$ in Fig. 14 represent the baseline, because the document index is updated multiple times, i.e., for each new leaf node. The cost for updating $D$ is therefore also included in all other timings. Clearly, indexing all content nodes ($G$) is a performance and scalability threat. The random value distribution quickly results in bad buffer hit rates impaired by forced flushes of dirty pages. In contrast, the focused CAS indexes ($C_1$ and $C_2$) scale gracefully due to their smaller sizes. As explained in Section 4.2, PCRs make it easy to decide whether an update has to be propagated to an index or not (item I4). Thus, the affected indexes can be efficiently identified and their maintenance just causes normal B*-tree updates (item I7).

The structure-only indexes $E$, $P_1$ and $P_2$ scaled with growing document sizes. Surprisingly, even the overhead of the element index which inverts every single element node varied only in the range of $\sim 60-80\%$. This pleasent behaviour, however, is only half the truth. The subtrees were sequentially inserted at the same position resulting in densely ascending SPLIDs and thus high locality for updates of SPLID-clustered indexes. With insertions at random positions, we would observe a bad behavior of $E$ similar to $G$, while $P_1$ and $P_2$ retain better locality due to the much smaller size.

At last, the results of $G+E$ and $C_1+C_3$ make the picture complete. As seen in the previous section, generic indexes on all element and content nodes – reflecting the classical setup – provide good support for query processing. But, their maintenance cost is unacceptable. In turn, our PCR-based CAS or path indexes easily achieve the same or better query response times, but with minimal maintenance overhead. Whenever necessary, additional PCR-based indexes with different scopes can be defined or deleted. Like $C_1+C_3$ shows, their overhead simply accumulates and allows to balance query performance and update cost as desired.

## 5 Assessing Competitor Approaches

In the end, we consider some recent storage and indexing approaches w. r. t. our wish list, but first a word on node labeling. Quite a large number of storage and indexing proposals rely on a *range-based* node labeling scheme. Even if *gaps* are left in the labeling space, these proposals sooner or later run into trouble, when the document is modified, because a prohibitively expensive relabeling may be required. Further, the expressiveness of these node labels does not allow to return inner elements from indexes without document access or some structural join, as required by item I6.

### 5.1 Storage

SystemRX [3] stores a document by grouping structurally related nodes into regions, which are then mapped to physical pages. Thereby, the goal is to keep subtrees intact. A special *region index* provides the “glue” to reconstruct the XML tree. The document store of SystemRX fulfills all items of our storage wish list. However, structure virtualization as in our approach is not possible. A detailed comparison between subtree clustering (SystemRX) and sequentially storing neighboring nodes (as in our approach) w. r. t. XML processing is an open research issue. Furthermore, SystemRX provides a similar indexing facility as our scheme, offering optional, selective, and user-defined indexes. However, little to nothing is known about their implementation, the way updates are processed (item I4), and whether inner elements can be reconstructed (item I6).

As in our approach, the authors of [1] suggest a structural summary for storage. They do not use the summary to virtualize the document, i. e., to enable path-oriented storage, but to manage extent lists, one for each path class. Each extent list contains all nodes and values on the corresponding path. Sedna [8] follows the same concept, but uses two extend lists per path class to separate structural from content nodes. As a result, both approaches have implicite path indexes resp. CAS indexes for all path classes available. However, the document order is lost, and expensive joins have to be executed to reconstruct the stored document (item S2). Furthermore, navigation and update handling (items S3 and S4) become more complex and produce random I/O. Sedna tries to improve structure reconstruction with direct and indirect pointers for parent/child and sibling relationships, but, in turn, this leads to higher storage consumption (item S6).

Shredding-based mappings like in MonetDB/XQuery [4] or XRel [38] often use a schema-agnostic concept, which stores elements, attributes, and text nodes in separate tables and answers XML queries through derived relational access plans. MonetDB/XQuery additionally uses the tailored Staircase join operator [11] for faster path processing. As structural relationships are encoded in special label columns of each table, document reconstruction also requires costly joins of the tables containing the shredded information (item S2). Additionally, the labeling schemes used are not stable and updates may require a relabeling of large fractions of a document (item S4). Specialized XML indexes are not required by these approaches, but query processing can be supported with conventional indexes on relevant columns.

The SUCXENT++ scheme [28] is another shredding approach storing only the leaf nodes and a structural summary. But, it does not exploit the salient features of SPLIDs and encodes the document “geometry” using additional columns and tables. Document modifications lead to a re-computation of this geometric data for the complete document, which is prohibitively expensive (item S4).

The storage layout of the eXist system [24] is similar to our node-oriented storage. All nodes are labeled by DLNs [12] and stored in document order in consecutive data pages. To speed-up navigation and direct node access, element nodes at higher document levels are indexed by a conventional B*-tree. Their descendants have to be searched using DLN comparisons in the data store. By default, eXist indexes the labels of all elements and attributes by their name for query processing. Content predicates can be supported with limited CAS indexes of the form `I(//<label>, [<type>])`. Thus, eXist addresses all items S1 – S6, but does not achieve the efficiency and compactness of our path-oriented approach.

Finally, the NoK (Next of Kin) storage approach [39] separates structure from content and encodes the structural part as a string, using a bracket-based notation. The result is a quite succinct representation. But, because node labels are missing, the document can only be indexed using byte offsets (done to keep the relationship between the structural and the textual document part). Such offsets are, however, fatal in case of document updates (item S4).

## 5.2 Indexing

Plain content indexes, such as Lore’s TIndex [23], suffer from the missing ability to answer the structural part of a CAS query. To match the structure, they have to access the document resulting in random access patterns and bad query performance (violating items I6 and I7). However, content indexes can be combined with path matching algorithms, such as holistic twig joins. Nevertheless, our CAS indexes have always outperformed this combination in our experiments.

Structural join indexes, like the XB-Tree [5] or the proposal from [16], supply structural joins and holistic twig joins with input data. The element index employed in the previous section is also a structural join index. In general, join indexes are not very expressive (item I2), because they simply provide posting lists containing XML nodes with some common property (e. g., with the same element name). Furthermore, as we have seen, our path/CAS indexes are always one order of magnitude faster than the algorithmic path matching via twig join and the join index (item I7).

Twig indexes are data structures that can directly evaluate branching path queries (i. e. twigs) with content predicates. Sequence-based indexes [29] transform the document into a string and the query into a pattern. Then, they apply pattern matching to retrieve twig matches. The approach violates item I3, because the complete document is indexed. Furthermore, in a twig query, the branches are order oblivious (twig siblings impose no restrictions), while the sequenced pattern matched against a sequence-based index is not. Therefore, to evaluate a twig, multiple patterns have to be generated, matched, and combined (item I7). Finally, the sequenced document is hard to maintain (item I4).

The disk-based F&B-Index [35] inverts the complete document (item I3) and defines quite a complex mapping to storage (not relying on standard access structures). Furthermore, some very sophisticated access algorithms are required to actually compute a twig result. The question of how such an index can be maintained is still open (item I4).

IndexFabric [30], FLUX [20], and the proposal in [19] are CAS indexes. IndexFabric encodes the paths of a document using a Patricia Trie. FLUX employs a B*-tree to store content values together with a Bloom filter, into which the values path is encoded. Reference [19] relies on a range labeling scheme, inverted lists, and a 1-Index [26]. All three approaches do not provide for user-defined index selectivity (item I3), do not present maintenance algorithms (item I4), and are not able to compute inner elements without document access (item I6). Furthermore, FLUX' Bloom filter is hash-based, and, therefore, causes an unavoidable and expensive false-positive detection for each returned node, which results in an additional document index look-up.

Path indexes come in two styles: complete and partial. An example for the first type is the 1-Index [26]. Basically, the 1-Index consists of a structural summary and, attached to each node of the summary, some extents. As explained before, we also rely on the 1-Index. However, because we can reconstruct inner elements, we do not need to store inner extents (and we also do not need to join/merge extents for query processing). A partial path index addresses the problem of structural complexity: For a highly irregular document, the 1-Index can become quite large (in the worst case, as large as the document). Therefore, partial indexes, like the $A(k)$ index [18] restrict the length of the indexed paths to $k$. However, the $A(k)$ index inverts the complete document, thus violating item I3, and inner elements can also not be computed out of its extents (item I6).

Further sophisticated approaches, such as materialized views for XPath [2], offer more specific query evaluation support for frequent query patterns. However, their maintenance overhead on updates is often painful due to costly re-computations. Furthermore, decisions on index applicability (item I5) become more and more complex, the higher the expressiveness of the language constructs defining the materialized view is. In contrast, our CAS index framework offers scalable, flexible, and adjustable index support as well as low-maintenance and index-matching costs.

## 6 Conclusions

In this paper, we proposed the XMIS approach for physical representation of XML documents with virtualized inner structure and path indexes. It definitely fulfills many meaningful requirements posed by XML applications. Key to XMIS is the use of SPLIDs as node labels and of the path synopsis [14], which both together enable fast computation of structure nodes. This property is consequently exploited to reduce storage space by structure virtualization and to compute inner elements from index results. Actually, because storage and indexing share many concepts and even the same infrastructure (i. e., B*-trees), XMIS is easy to understand and to implement.

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 the document store, we designed a flexible index mechanism for content and path indexing. The 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 substantial 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.

In summary, the overwhelming fraction of known XML storage and indexing proposals claims optimized performance through huge storage redundancy by keeping the structure nodes and by using differing code bases and structures for various indexing approaches. In contrast, we have eliminated storage redundancy to the extent possible by virtualizing the document structure and simplified indexing by reducing it to a single mechanism based on a proven code basis. And our approach is backed by convincing performance results.

## References

1. 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)
2. Balmin, A., Özcan, F., Beyer, K. S., Chochrane, R. J., and Pirahesh, H. A Framework for Using Materialized XPath Views in XML Query Processing. Proc. VLDB: 60-71 (2004)
3. Beyer, K., 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. SIGMOD: 374-358 (2005)
4. Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., and Teubner, J. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. Proc. SIGMOD: 479–490 (2006)
5. Bruno, N., Koudas, N., and Srivastava, D. Holistic twig joins: optimal XML pattern matching. Proc. SIGMOD: 310–321 (2002)
6. Chen, Q., Lim, A., and Ong, K. W. D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data. Proc. SIGMOD: 134-144 (2003)
7. Draper, D., Frankhauser, P., Fernandéz, M., Malhotra, A., Rose, K., Rys, M., Siméon, J., and Wadler, P. XQuery 1.0 and XPath 2.0 Formal Semantics (2004)
8. Fomichev, A., Grinev, M., and Kuznetsov, S. Sedna: A Native XML DBMS. Proc. SOFSEM: 272–281 (2006)
9. Goldman, R. and Widom, J. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. Proc. VLDB: 436-445 (1997)
10. Graefe, G. and Larson, P.-A. B-Tree Indexes and CPU Caches. Proc. ICDE: 349-358 (2001)

11. Grust, T., van Keulen, M., and Teubner, J. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. Proc. VLDB: 524-525 (2003)
12. Härder, T., Haustein, M. P., Mathis, C., and Wagner, M. Node labeling schemes for dynamic XML documents reconsidered. Data & Knowledge Engineering 60:1, 126-149, Elsevier (2007)
13. Härder, T., Mathis, C., and Schmidt, K. Comparison of complete and elementless native storage of XML documents. Proc. IDEAS: 102-113 (2007)
14. Haustein, M. P., Härder, T., Mathis, C., and Wagner, M. DeweyIDs – The Key to Fine-Grained Management of XML Documents. Proc. 20th Brazilian Symposium on Databases: 85-99 (2005)
15. Haustein, M. P. and Härder, T. An efficient infrastructure for native transactional XML processing. Data & Knowledge Engineering 61:3, 500-523 (2007)
16. Jiang, H., Wang, W., Lu, H., and Xu Yu, J. Holistic Twig Joins on Indexed XML Documents. Proc. VLDB: 273-284 (2003)
17. Kaushik, R., Bohannon, P., Naughton, J. F., and Korth, H. F. Covering Indexes for Branching Path Queries. Proc. SIGMOD: 133-144 (2002)
18. Kaushik, R., Shenoy, P., Bohannon, P., and Gudes, E. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. Proc. ICDE: 129-140 (2002)
19. Kaushik, R., Krishnamurthy, R., Naughton, J. F., and Ramakrishnan, R. On the Integration of Structure Indexes and Inverted Lists. Proc. SIGMOD: 779-790 (2004)
20. Li, H.-G., Aghili, S. A., Agrawal, D., and El Abbadi, A. FLUX: Content-and-Structure Matching of XPath Queries with Range Predicates. Proc. XSym: 61-76, LNCS 4156 (2006)
21. Mathis, C. Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems. Ph. D. Thesis, Verlag Dr. Hut (2009)
22. Mathis, C., Härder, T., and Schmidt, K. Storing and Indexing XML Documents Upside Down. Computer Science – Research and Development 24:1-2: 51–68, Springer (2009)
23. McHugh, J. and Abiteboul, S. Lore: A Database Management System for Semistructured Data. SIGMOD Record 26: 54-66 (1997)
24. Meier, W. eXist: An Open Source Native XML Database. Proc. Web, Web-Services, and Database Systems, LNCS 2593: 169-183 (2002)
25. Miklau, G. XML Data Repository. www.cs.washington.edu/ research/xmldatasets
26. Milo, T. and Suciu, D. Index Structures for Path Expressions. Proc. ICDT: 277-295 (1999)
27. O'Neil, P. E., Pal, S., Cseri. I., Schaller, G., and Westbury, N. ORDPATHs: insert-friendly XML node labels. In Proc. SIGMOD: 903–908 (2004)
28. Prakash, S., Bhowmick, S. S., and Madria, S. Efficient Recursive XML Query Processing Using Relational Database Systems. Data and Knowl. Engineering 58:3, 207-242 (2006)
29. Prasad, K. H. and Kumar, P. S. Efficient Indexing and Querying of XML Data Using Modified Prüfer Sequences. Proc. CIKM: 397-404 (2005)
30. Sample, N., Cooper, B. F., Franklin, M. J., Hjaltason, G. R., Shadmon, M., and Cohen, L. Managing Complex and Varied Data with the IndexFabric(tm). Proc. ICDE: 492-493 (2002)
31. Schmidt, A. R., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I., and Busse, R. XMark: A Benchmark for XML Data Management. Proc. VLDB: 974-985 (2002)
32. Document Object Model (DOM) Level 3 Core Specification, W3C Recommendations (Jan. 2004)
33. Brownell, D. SAX2. O'Reilly Media (2002)
34. Wang, H., Park, S., Fan, W., and Yu, P. S. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. Proc. SIGMOD: 110-121 (2003)
35. Wang, W., Jiang, H., Wang, H., Lin, X., Lu, H., and Li, J. Efficient Processing of XML Path Queries Using the Disk-Based F&B-Index. Proc. VLDB: 145-165 (2005)
36. XQuery 1.0: An XML Query Language. W3C Recommendation (Jan. 2007)
37. XQuery Update Facility 1.0. W3C Recommendation (17 March 2011)
38. 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 TOIT 1:1, 110-141 (2001)
39. Zhang, N., Kacholia, V., and Özsu, T.; A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. Proc. ICDE: 54-63 (2004)

Christian Mathis studied Computer Science from 1998 to 2004 at the University of Kaiserslautern. Since Oct. 2004 he was a Ph. D. student and since Oct. 2007 scientific staff member at the DBIS research group lead by Prof. Härder. He received his Ph. D. degree in Computer Science from the University of Kaiserslautern in July 2009. For his Ph.D. thesis, he earned the Dissertation Award of the GI-Fachbereich DBIS for the years 2009 and 2010. In the XTC project, he explored various important aspects of XML query processing. (http://wwwlgis.informatik.uni-kl.de/cms/index.php?id=36)

Theo Härder obtained his Ph. D. degree in Computer Science from the University of 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 University of Darmstadt. As a full professor, he is leading the research group DBIS at the University of 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. Since October 2010, he is senior professor for research at the University of Kaiserslautern. Theo Härder's research interests are in all areas of database and information systems – in particular, DBMS architecture, transaction systems, information integration, and XML database systems. He is author/coauthor of 7 textbooks and of more than 280 scientific contributions with > 160 peer-reviewed conference papers and > 70 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 Computer Science – Research and Development (Springer), associate editor of Information Systems (Elsevier), World Wide Web (Kluver), and Transactions on Database Systems (ACM).

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. He received his Ph. D. degree in Computer Science from the University of Kaiserslautern in September 2011.

Sebastian Bächle studied Computer Science from 2002 to 2007 at the University of Kaiserslautern. Since September 2007 he is a scientific staff member in the XTC project at the DBIS research group lead by Prof. Härder. His main interests are concurrency control in XML documents and operator efficiency in XML database management systems.