Usage-driven Storage Structures for Native XML Databases

Karsten Schmidt and Theo Härder*
Department of Computer Science
University of Kaiserslautern, Germany
{kschmidt,haerder}@informatik.uni-kl.de

ABSTRACT

There are numerous and influential parameters associated with the selection of suitable native storage structures for XML documents within an XML DBMS. Such important parameters are related to node labeling, path synopsis, text compression, document container layout, and indexing. While these storage and compression techniques primarily reduce I/O overhead and space consumption, they imply additional algorithmic costs for encoding/decoding during document processing. We discuss how various storage options favor different usage patterns and how they can be specified beforehand to influence native XML storage options by the anticipated usages.

Keywords

XML database, storage structures, access patterns, workload classification

1. INTRODUCTION

With the dramatically increasing importance of XML and XML-related query languages, several native and non-native storage approaches have emerged. In the past, research often focused on the management of a few isolated documents which are typically very large (up to several GBytes [11]). On the other hand, many real-world applications require DBMSs managing large collections of small to medium sized XML documents (typically less than a few MBytes [1]). In any case, the variety of parameters critical for the performance of XML document representations (e.g., number of nodes, document depths, varying fan-outs, distinct element names, avg. size of text nodes, and document size) will often lead to suboptimal or even bad solutions. For this reason, it is highly recommended to equip the storage manager with the ability to automatically select reasonable or even optimal storage structures for documents.

*This work has been supported by the Rheinland-Pfalz cluster of excellence "Dependable adaptive systems and mathematical modeling" (see www.dasmod.de).

IDEAS08 2008, September 10-12, Coimbra [Portugal]
Editor: Bipin C. DESAI

1.1 Document Usage

Beside traditional workload analysis (offline and online), the initial process of storing an XML document needs to "guess" the prospective usage or requires some help by the user. There are several metrics how to differentiate usages. For instance, by evaluating the share of read and write operations, whereby secondary structures (e.g., indexes) need to be maintained as well. Furthermore, access patterns may serve as distinguishing feature such as access frequency, value range, and data volume processed. Thus, in the first place, we started to classify the workloads present in native XML databases. In doing that, we observed two major kinds of usage patterns for XML documents—collections of small (tiny) documents processed in a document-centric way and single big (huge) documents processed in a data-centric way.

Document-centric Usage
XML database access for document-centric processing primarily retrieves and stores a document as a whole. This is often the case for tiny documents with storage sizes less than a database disk page or for documents having only a small structure part which often results from automatic data transformations into the XML representation, e.g, "binary" data, articles, or books. Although support by structural indexing is possible, the additional I/O of secondary indexes often does not pay off. Text collections and systems exchanging XML documents (Web services, messages) benefit from downloading or retrieving entire documents to process it at the client side. Even if the document is evaluated or modified at the server side, e.g., using XPath or XQuery, tiny documents (with a size less than a (few) disk page(s)) benefit from being processed as a whole. As a result, they fit into a single (or a few) buffer page(s) and only require marginal main-memory space. However, queries to documents requiring search (keyword search) often dominate the workload for these kinds of XML documents and an additional full text index is usually oversized or too expensive.

Data-centric Usage
For example, benchmarks as specified in [15], XML stream processing, and (semi-) structured data sets embedded into XML documents (e.g., relational data) use query languages and rely on index support to optimize selective data access [2]. Often, tiny document fractions, aggregated data, or a few element nodes constitute the final query result. In such cases, exploiting indexes to minimize disk access is essential

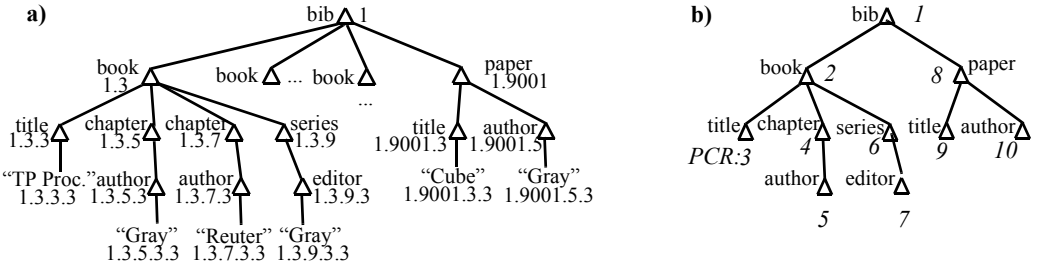


Figure 1: a) SPLID-labeled XML document cut-out and b) related path synopsis with PCRs

for huge documents. Exemplified by a subset of the University of Washington XML document collection [11] and the TPoX benchmark documents [15], we elaborate the characteristics and anticipated workloads for data-centric and document-centric XML processing. This distinction only helps to define indexes and storage parameters, but it does not address the impact of write and read operations on them. Therefore, it is further necessary to cope with their influence on storage structures during transactional query processing.

1.2 Our Contribution

In this contribution, we explore native XML storage configurations and their performance effects for different workloads. For this reason, we identify and analyze typical XML usage patterns to derive heuristics for *what kind of documents may anticipate what kind of workloads*. To determine suitable XML storage structures, we empirically test various hypotheses for distinct usage patterns. Finally, we corroborate or discard them by extensive performance measurements using the XTC system [9], our native XDBMS prototype. In Section 2, we discuss the most important storage-related concepts for XML documents and sketch the implementation of appropriate methods in XTC. In Section 3, we describe how to collect the necessary set of storage parameters in an analysis phase, before we apply the chosen concepts and methods on tailor-made storage structures and empirically evaluate them for well-known sample documents in Section 4. A brief overview of proposed alternatives is given in Section 5, before we wrap up with conclusions.

2. ESSENTIAL CONCEPTS FOR THE STORAGE MANAGER

Currently, we are upgrading XTC [9] step by step towards enhanced adaptivity. For physical document handling, the adaptation primarily rests on the following concepts.

2.1 Node Labeling

As we have learned from many experiments and benchmark runs [7], the node labeling mechanism plays a fundamental role for storage consumption and efficient support of navigational and declarative query evaluation. Moreover, it is the key to processing flexibility and entire internal system performance. We recommend the use of prefix-based labeling schemes such as OrdPaths [13], DLNs [3], or DeweyIDs [4, 7] as sketched in the sample document in Figure 1a. Because any prefix-based scheme is appropriate for our document storage, we use SPLIDs (Stable Path Labeling IDentifiers) as a synonym for all of them.

Space economy and flexibility requires a dynamic, variable, and effective storage scheme to capture tall/flat trees with many/few nodes at a level and a huge fan-out per node or only some children. At the same time, efficiency in storage usage, encoding or decoding, and value comparisons at the bit or byte level must be guaranteed. Huffman codes serve such demands very well [7].

Our specific labeling mechanism enables—with a *dist* parameter used to increment *division* values and to leave gaps in the numbering space between consecutive labels—a kind of adjustment to expected update frequencies. These gaps do not prevent "overflows" of the labeling scheme, they only defer them. Nevertheless, overflows lead to an increased label size and a re-labeling is adviseable at a certain length but not mandatory. A default scheme with minimum *dist*=2 can be overridden if specific knowledge is available. This requires "future knowledge" of updates and, therefore, needs user hints. Adaptivity is confined to observing "label overflows" which could trigger a re-labeling (in selected subtrees) using a more appropriate value for the *dist* parameter. Preanalysis or sampling of a document, however, could reveal characteristics of the structure, average depth, distribution of nodes per level, etc., which are useful for an adjusted SPLID encoding.

2.2 Path Synopsis

XML documents usually have a high degree of redundancy in their structural part, i.e., they contain many paths having identical sequences of element/attribute names (see Table 1). Such similar path instances, e.g., bib/book/chapter/author in Figure 1a, only differ in the leaf values and the order in the document. The structure part of them can be represented as a so-called path class in the path synopsis (see Figure 1b) kept as a kind of metadata. Typical path synopses have only a limited number of element names and path classes and can, therefore, be represented in a small memory-resident data structure. Every node within the path synopsis carries a number called path class reference (PCR), as illustrated for the sample document in Figure 1b. The basic idea of a path synopsis is similar to that of a DataGuide which, however, was introduced as a structural overview for the user, for storing statistical document information, and, thus, enabling query optimization [6]. In contrast, the primary use of a path synopsis is for structural virtualization, concurrency control, and supporting indexing and query processing [9]. To enable the optimization of XPath/XQuery expressions, we maintain a so-called EXsum summary (Elementwise XML summarization), which can capture statistical information of all important query axes related to (the nodes having) the same element name [1]. Moreover, a given synopsis can be compared to existing document synopses to find

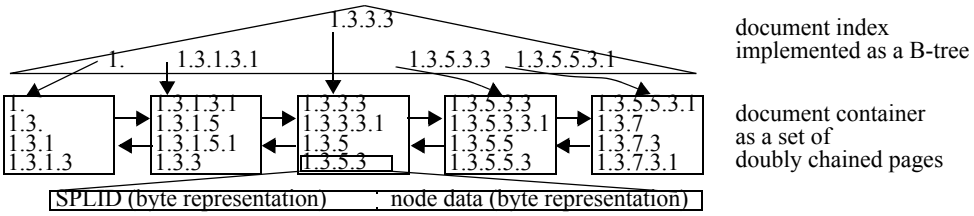


Figure 2: Document store with a B-tree and container pages

documents with similar structure. In our system, a path synopsis obtains its full expressiveness by the interplay of PCRs and SPLIDs: a SPLID delivers all SPLIDs of its ancestors, while a PCR connected to a SPLID identifies the path class it belongs to. Furthermore, the SPLID of a node represents positional information for that node and, in turn, for all its ancestors. Starting from an arbitrary content, attribute, or element node—whose unique position in the document is identified by its SPLID—which is associated with a reference to its path class, it is easy to completely reconstruct the specific instance of the path class it belongs to. For example, with SPLID =1.3.5.3.3 and PCR=5, we can reconstruct the entire path instance with value "Gray" as a leaf using the path synopsis. 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-based access using leaf node names. Additional links between vocabulary IDs (VocIDs —substituting the XML element/attribute names) and their occurrences within the path synopsis offer direct entry points for further navigational steps and matching/searching operations starting at non-leaf nodes.

2.3 Document Storage

Efficient processing of dynamic XML documents requires arbitrary node insertions without re-labeling, maintenance of document order, variable-length node representation, representation of long fields, and indexed access. As sketched in Figure 2, the *document index* enables direct access to a node when its SPLID is given. Together with the *document container*, the document store represents a B*-tree which takes care of external storage mapping and dynamic reorganization of the document structure. Combined with SPLID usage, it embodies our basic implementation to satisfy the above demands efficiently and forms a framework for further specializations. In XTC, this basic structure comes with a variety of options [9] concerning use of vocabularies, materialized or referenced storage of content (in leaf nodes), and, most important, prefixcompressed SPLIDs. As illustrated in Figure 2, the sequence of SPLIDs in document order lends itself to prefix compression as empirically evaluated in [7].

2.4 Structure Virtualization and Content Compression

Two of the main issues to be regarded for XML document compression in databases result from the XML *structure* itself and its *content*. A novel technique of structural virtualization was described and evaluated in [8]. The so-called elementless document storage does not contain any structure nodes (elem. & attr. nodes in Table 1) in its physical representation, i.e., the document container only stores the content (leaf) nodes, each equipped with a SPLID and a PCR. As explained in Section 2.2, it is easy to reconstruct all paths and nodes on demand, e.g., when referenced during the evaluation of an XPath/XQuery expression. It is even possible to perform DOM-based navigation on this virtualized structure.

In case of content compression, we need to preserve the fine-granular document representation (see Figure 2) for declarative and navigational transaction processing and node indexing. Even though, many compression techniques proposed achieve impressive compression rates [12], they cannot be applied for our finegranular purposes. However, when the underlying structures are *document-centric* (an entire book content as a single text node) such compression techniques would pay off, whereas *data-centric* documents typically occurring in DB-based applications do hardly benefit from them. Furthermore, such techniques often rely on large auxiliary dictionaries and, hence, typically provoke substantial compression and decompression overhead. Moreover, they are restricted to static file-based structures and, thus, they enable only single-user access. To avoid undue limitations and overhead of XML processing, compression of single node values seems to be an appropriate and challenging choice. In our compression study, we exclusively focused on single nodes and their data stemming either from text content or attribute values.

To provide some indicative results for the storage of XML documents in different formats, we applied a number of empirical tests in the context of our storage manager where we used—to facilitate comparison—the frequently evaluated set of test documents taken from [11, 14, 15]. For a representative subset, Table 1 assembles essential characteristics of the documents in the "plain" format, i.e., in the verbose XML representation in which they arrive at the DBMS.

Because content compression is orthogonal to the question how the overall document is stored, we explore the compression efficiency to be gained irrespective of how it is used. Indeed, the application of an encoding method could be specified as a storage parameter. Here, we focus on character-based compression using Huffman trees, for which we have performed experiments to explore various types of Huffman encodings. Comparing all experiments, a document-wide fixed Huffman encoding (FH) frequently produced better results than other more sophisticated alternatives. To characterize the potential compression gain for data-centric XML, we have listed the avg. value sizes of content nodes together with a summary of our results in Table 1: Content compression (FH) is fast and delivers considerable compression rates varying from ∼23% to ∼35% on all (large) documents. Note, as long as the compression dictionary's overhead is negligible, 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. This observation emphasizes

Table 1: Characteristics of XML documents considered

Doc name	Description	Size in Mbytes	# elem.& attr. nodes	# voc. names	# path classes	Avg depth	# content nodes	Avg content size	FH compr.	Store time
psd7003	DB of protein sequences	716.0	22,595,465	70	76	5.68	17,245,756	17.0	74.0%	93.4%
lineitem	LineItems from TPC-H benchmark	32.3	1,022,977	19	17	3.45	962,801	6.5	70.8%	93.4%
dblp	Computer science index	330.2	9,070,558	41	153	3.39	8,345,289	20.9	69.9%	94.1%
nasa	Astronomical data	25.8	532,963	70	73	6.08	371,593	33.4	64.4%	84.6%
xmark	XMark benchmark document	11.6	206,130	77	535	5.5	118,141	52.78	77.1%	83.2%
accnt	TPoX benchmark	~0.006	~193	<88	~100	4.7	~127	10.6	98%	
order	account, order, and	~0.002	~81	<139	~83	2.6	0	8.1 (attr)	125%	>130%
secty	security data	~0.006	~52	<64	~53	3.5	~46	92.2	71.6%	

that content compression is advisable only for large documents. In summary, *character-based compression* providing the outlined benefits can be orthogonally applied to the content part without interfering with indexes or other auxiliary structures as well as query evaluation, navigational processing, and reconstruction of original documents (round-trip property).

3. CAPTURING STORAGE OPTIONS

An accurate prediction of future document processing is confined to pre-defined document operations. To enable automatic selection of suitable (if not optimal) storage structures upon XML document arrival, our storage manager relies on information that is accessible beforehand, such as document structure and typical access heuristics.

3.1 Document Characteristics

If the entire document is present for parameter analysis, all desired parameters can be perfectly determined. However in case of stream-based processing, "guessing" or some kind of parameter approximation must suffice. Statistical data may include—besides the degree of document-centric compression behavior—the number of nodes (i.e., element, attribute, and text nodes), maximum depth and average depth, various fan-out ratios, number of distinct element names, number of distinct paths per path class, as well as instances per path class (see Table 1). For these parameters, some dynamic approximations can be derived. Future document usage could be heuristically anticipated to tune fundamental storage parameters (see Section 4).

To enhance storage speed, we explored the potentials of document sampling. For simplification, the head of an XML document is scanned and analyzed, instead of a real sampling technique requiring the analysis of distributed non-connected parts. For our reference documents, Figure 3 shows the proportional estimation error achieved by sampling up to 50% of their size. Surprisingly, our results reveal that, even with only a 1% sample, an error of not more than ∼10% may be expected. Of course, larger sample sizes improve this error margin. Figure 3 also shows that there exist "simply-structured" documents where sampling delivers perfect knowledge of storage parameters even when using only very small samples. As a worst case of our reference documents, nasa exemplifies the sampling and extrapolation of unbalanced documents: its error is bounded to ∼12%. In summary, sampling often delivers parameters accurate enough to plan the physical configuration of an XML document.

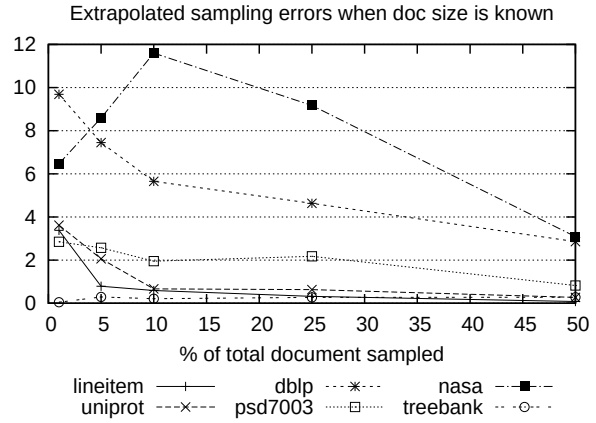


Figure 3: Relative estimation error of sampling

3.2 Document Access

As introduced in Section 1.1, different workloads on XML documents may favor different storage options. Furthermore, we can specify what kind of operations imply certain drawbacks for the chosen set of parameters. First of all, we classify document operations in the following three categories:

- **document-based** – storing and reconstructing complete documents (SAX API), obviously favoring document-centric XML data.
- **index-based** – point and range queries often benefit from indexes (query optimization), however, the penalty of index creation (which implies a document scan) has to be amortized by frequent search operations; for data-centric XML, indexes are essential.
- **fragment-based** – the most complex and varying operations refer to node/subtree lookups, modifications, and deletions; both, document-centric and data-centric documents may be accessed in this way.

Furthermore, large documents typically require selective access when XPath/XQuery predicates are evaluated or subtrees are inserted or modified. In contrast, usually small documents are units of processing, i.e., they are entirely fetched, processed in memory, and, when modified, completely restored to disk. Anticipated operations are specified by a mix of simple queries for searching and modifying data [15]. In contrast, large XML documents vastly benefit from additional indexes when processed as DB objects.

In the following section, we show for various storage models how suitable parameters can be estimated, what storage and processing gain can be achieved as compared to a standard representation, and how various operations scale on theses models. For that, document characteristics serving as a reasonable foundation for decisions are revealed.

4. CHOOSING THE RIGHT STORAGE CONFIGURATION

So far, we have outlined the essential concepts and the sampling of critical parameters for XML document storage. In our empirical study, we focus on the improvement of storage structures which can be chosen by the DBMS for incoming documents. Documents sent by a client arrive in the so-called *Plain* format where the verbose "external" format having long element and attribute names is still present. In a fine-grained native storage structure, all element and attribute names as well as all content values are stored as tree nodes having a unique label, e.g., a SPLID (see Figure 2). A vocabulary is essential for saving document storage space by encoding the element and attribute names, e.g., using one-byte or two-byte integers as VocIDs. It can be represented by a small main-memory data structure (typically keeping only a few hundred names). Such a straightforward document representation, where structure nodes are encoded by a combination of VocIDs and SPLIDs and where content nodes are encoded by text values and SPLIDs, is called the *Standard* model. It serves as the baseline for experiments to show the possible performance gains.

4.1 Standard Storage Configuration

As it is our aim to enable XTC to handle any kind of XML document arriving at the DBMS, the storage manager applies a basic set of storage parameters (including page size, SPLID compression, content compression, and storage model), which normally cannot be changed afterwards. While future access behavior is not considered, all kinds of documents can be stored using the following set of parameters, enabling a maximum degree of flexibility:

- *page size: 16 KB* allows to store large documents, because the addressable storage space is bounded to the maximum page number
- *SPLID compression: off* avoids overhead of compression encoding and offers direct SPLID access (no need to compute predecessors)
- *content compression: off* avoids encoding maintenance and unbiased compression gains
- *full storage model:* does not reqiuire an additional path synopsis for structure virtualization , which may blow up memory consumption when dealing with too many paths; no path encoding required (PCRs)

In the following section, we will show how we can optimize XML storage using this predefined parameter set and how the resulting configurations perform under certain conceivable workloads.

4.2 Improved Storage Configurations

When storing documents using our improved concepts, we accept higher algorithmic costs for the sake of space efficiency. Here, the related path synopsis containing all path classes and PCRs is constructed in memory to primarily support index access and virtualization of the document structure. The question, which secondary element/attribute or text indexes should be provided, is orthogonal to the choice of the native document structure and has to be answered w.r.t. the expected workload; how indexes can be built is discussed in [8].

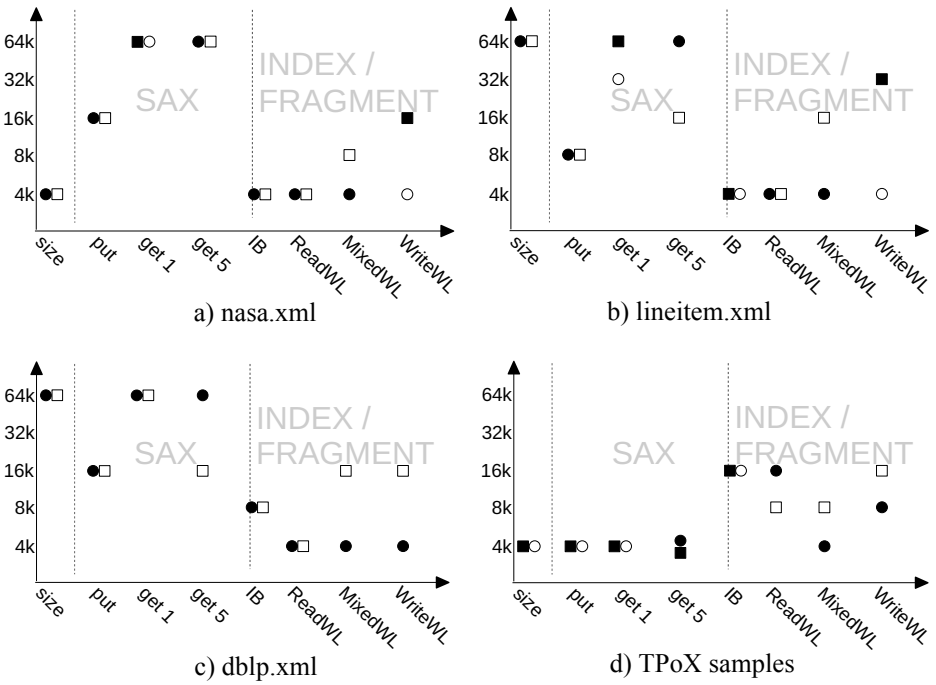
Here, we primarily illustrate which configurations tailored to the parameters of the documents reduce processing time for several scenarios. Due to the given complexity of all measures reducing storage consumption or query processing, we show under which conditions they support specific workload types. An important optimization is the use of optimized SPLIDs, which was applied in all experiments. Note, in all cases neither set-oriented processing (e.g., XQuery) nor node-oriented processing (e.g., DOM) are restrained or impeded.

Storage Configuration and Access Patterns for Single Documents
To illustrate the various storage configurations, we assembled a set of pre-defined configurations. Note, in this work, we confine them to the most promising aspects and combinations, i.e., parameters and configuration combinations not presented here may also improve (some) of the conducted test cases, but their benefit is too small to be evaluated and to be presented in detail.

- *Standard Storage* Configuration (default)
- *Full Storage*, SPLID compression, 4k/8k/16k/32k/64k pages, optional content compression, shared/distinct vocabulary encoding
- *Elementless Storage*, SPLID compression, 4k/8k/16k/32k/64k pages, fixed/adjusted PCR encoding, optional content compression, shared/distinct vocabulary encoding

As introduced in Section 1.1, an XDBMS has to support different access types, while these types not necessarily favor the same storage configuration for an XML document. Therefore, we have pre-defined specific workload scenarios representing different document usages.

- **Storage Space Consumption** – does indirectly matter when processing takes place size – document's space consumption (including path synopsis when stored in elementless mode)
- **SAX-based** – for document-based access; read/write of the entire XML document (SAX API)
  put – storing the XML document once
  get 1 – reading the entire XML document once
  get 5 – reading the entire XML document multiple times (5 times)



Single workload scenarios (e.g., store documents, create indexes, workload mixes) prefer different page sizes and storage models depending on the underlying XML document. It is clearly noticeable that similarly structured documents (e.g., *dblp* and *lineitem*) exhibit similar characteristics. In contrast, the tiny TPoX benchmark documents do more often prefer a full storage model. Thus, it is often not possible to setup one storage configuration supporting all kinds of workloads.

Figure 4: Workload vs. Storage Model benefits

- **IB** – index building costs (only evaluated for element index creation, because other index types, e.g., CAS and path indexes are not available in all storage modes)
- **Index-based or fragment-based**
  ReadWorkload – read-only access; a mixture of XPath expressions and/or DOM navigational steps
  MixedWorkload – a mix of read/write and navigational operations XPath, simple XQuery, DOM-based units of work, updates, and deletions
  WriteWorkload – write-only workload; inserts, updates, and deletes on nodes and/or subtrees

To compare the results of different storage configurations, single documents are assigned to separate storage structures. The storage structure's corresponding schema shown in Figure 2 shows that the SPLIDs occur in document order and that they lend themselves to a very effective *prefix compression* [7]. Hence, space saving achieved by all tests can be primarily contributed to this optimization step. The related optional path synopsis is kept in a separate structure; a number of user-specified content and/or element indexes can be additionally specified and allocated to speed up declarative document access.

All experiments were conducted on a Dual Pentium IV (3.2GHz) with a mirrored disk array of two 80GB disks and 1 GByte of main memory.

In the first part, we evaluate what kind of storage model in combination with varying page sizes leverages certain workload types for certain document types. In Figure 4, we assembled four representative diagrams for various document types. The vertical axis scales the page size used and the other axis identifies the analyzed workloads. The storage models compared (elementless and full storage) are illustrated by dots and rectangles, respectively. A filled symbol means that this storage mode is the best one for such a kind of workload (e.g., diagram in Figure 4a shows that the fastest storing of the *nasa* document is done in elementless mode using 16 KByte pages), whereas an unfilled symbol means that this storage mode is inferior but, when enforced, would suit that page size at best (e.g., diagram in Figure 4a shows that storing the *nasa* document in full storage mode is worse than in elementless, but would be best using 16 KByte pages).

We can further see that disk utilization (labeled *size*) is directly depending on document size and structural complexity. Hence, simply-structured and uniform documents (e.g., *lineitem* and *dblp*) do prefer larger page sizes, whereas complex-structured documents (e.g., *nasa*) or tiny documents (e.g., *TPoX*) better utilize small pages of 4 KBytes [16].

The middle part of each diagram depicts so-called SAX-based workloads for storing and reconstructing XML documents. Document storage (labeled *put*) performs best on medium-sized pages and in elementless mode for large documents, whereas tiny documents prefer the full storage and the smallest page size covering the entire document. For SAX-based reconstruction, small and medium-sized (e.g., *nasa*) documents and very simply-structured documents (e.g., *lineitem*) benefit from the full-storage mode. This benefit becomes a drawback when a document needs to be reconstructed (labeled *get 5*) repeatedly. Here all kinds of documents favor the elementless storage, even though the storage model is almost irrelevant for tiny documents. Index-building costs (labeled *IB*) are a mixture of a SAX scan on an already stored document and the actual index creation. The benchmark results show that simply-structured and small documents favor the full storage mode, whereas all kinds of documents prefer small page sizes.

The right part shows node-based and subtree-based workload scenarios. Read-only workload (labeled *ReadWorkload*)

reads tiny fractions of a document and, therefore, all kinds of documents favor small page sizes. Because every query of such a workload needs to repeatedly access the path synopsis, elementless storage is preferable in all cases. This observation also holds for mixed workload scenarios applying read and write operations on XML documents. But for write-only scenarios (labeled *WriteWorkload*) sometimes (e.g., *nasa* and *lineitem*) the full storage model is preferred, because the additional path synopsis structure in elementless mode needs to be maintained when subtree/node insertions extend it.

Observation for Single Documents
To interpret the results w.r.t the documents, it is obvious that the small *TPoX* documents favor completely different configurations. Furthermore, simple and often flat-structured documents like *dblp* and *lineitem* tend to favor configurations different from those of complex-structured documents like *nasa*, *psd7003*, or *xmark*. However, regarding space and document storage/reconstruction times, there is a strong correlation between them indicating that I/O reduction is the main objective for *document-based* access.

Fragment-based access was compared using several benchmarks having read-only, mixed, and write-only workloads. As known from relational database research, workload shifts may occur in databases and, of course, also in XML databases. Unfortunately, a different set-up is required to maintain optimality. Thus, document storage parameters have to be adjusted to the most frequent workloads or need to become more flexible to enable adjustments even during runtime.

Moreover, a first indicator of scaling properties, CPU load, and I/O throughput was measured by a step-wise increase of the number of repetitions concerning the same workload. In XML data processing, the CPU cost for handling complex XML nodes is higher compared to the handling of a relational record and, therefore, access and processing in XML databases performs differently. That means, simple I/O reduction by using indexes or compression is not that effective to speed up data processing, because the expensive data handling often leads to a high and constant CPU load.

We disclosed further drawbacks resulting from the simplicity of small documents. Let us consider such documents as in [15] having "plain" sizes between 1 and 20 KBytes. They often contain datasets having only a few distinct paths and rarely more than one instance per path class. In such cases, the use of path synopses does not pay off (compare Figure 4d). Due to space limitations, we omit a detailed discussion and simply present our heuristics gained from dedicated experiments. In the average, only 4–5 instances per path class are needed to compensate the additional space consumption of a path synopsis. That means, for the *TPoX* documents, *elementless storage* needs 7%–26% more space than *full storage*.

Fine-Tuning of the Storage
Compression is also applied to encode the *vocabulary* (XML tag and attribute names) for XML documents. In all benchmarks, we used a shared vocabulary for all documents and a fixed encoding size of 2 Bytes allowing 65K different tag names. Although this decision limits the extensibility to capture tag names, all test cases used a fairly small fraction of the possible 65K. In specific cases, it may be preferable to separate vocabularies for each document (or group of documents) and choose encoding sizes of 1 or 3 Bytes, allowing 256 or 16.7 Mio different tag names, respectively. This would lead to an adjusted space consumption and, as a result, to another or even better adapted I/O behavior.

Similar effects can be observed when *PCR encoding* is applied. Usually the path synopsis stays very stable and thus the number of path classes does not vary much during a document's lifetime. However, this observation must not be true for all kind of documents and for the applications operating on them. Hence, PCR encoding must be adaptable to the anticipated processing, too. In our experiments, we therefore used adjusted PCR encodings of $\lceil \frac{Log_2 PathClasses}{8} \rceil$ Bytes.

The optional *content compression* seems to be application-dependent, because only for documents having a large fraction of text (e.g., *nasa* and *xmark*) the I/O reduction is significant and leads to shorter processing times. Especially processing tiny documents (e.g., *TPoX*) is slowed down by the compression overhead. Thus, applications only retrieving and storing large documents (*SAX-based* access) may benefit from content compression, as long as size consumption does not play a role. Further on, internal XML node processing is completely done on SPLIDs and PCRs when possible and, thus, often content access causing decompression is not needed except when the final result is materialized as a sequence of nodes, a single node, or a tree.

Benchmark Findings
While document storage and reconstruction indicates a clear superiority of elementless configurations in terms of space and, in turn, I/O time, selective processing sometimes ruins these performance advantages because of increased encoding/decoding overhead. The initial fixed storage configuration is only advisable when nothing about the expected workload or document characteristics is known in advance. Moreover, the benchmarks have shown that index-based access is more independent from the underlying storage model than SAX-based document processing. Finally, having knowledge about how XML data will be processed can help to optimize the entire parameter set of the storage configuration beforehand, instead of only optimizing storage size or initial store time.

4.3 Extension to Document Collections

Looking at a specific XML usage where a large number of small or tiny documents are stored [15], our so far considered configurations disclose tremendous drawbacks. Given a minimum of one container page per document, most space would be unused. In addition, separately stored auxiliary structures would consume an enormous number of weakly filled pages. To avoid such a low storage occupancy, a trick can be applied to compose a single *artificial* document from a collection of otherwise stand-alone XML structures by adding a common root for all of them. Moreover, the allocation of combined indexes may also save substantial storage space and, at the same time, accelerate the evaluation of queries, primarily due to reference locality. Such compositions are often beneficial if the individual documents stem from *the same domain*. Then, they may share path classes such that the number of different path classes is limited within the composed document. Combined storage of documents with similar structures is one of our main objectives. When user hints are missing, the storage manager tries to dynamically

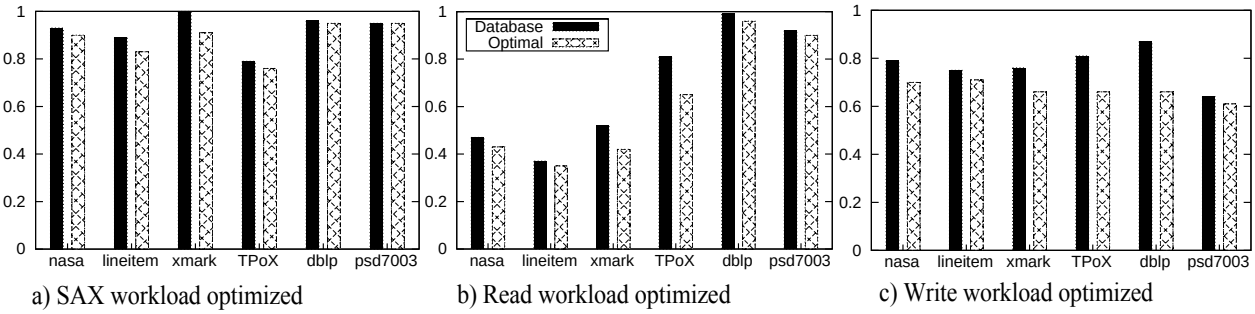


Figure 5: Performance gains for database-chosen and theoretical optimal configuration compared to Standard configuration

assign a new document to an existing collection via path synopsis matching and vocabulary comparison. If no suitable match can be found, a new document is considered dissimilar to all existing collections. To preserve their beneficial storage and processing properties, the best decision in such a case is to store it as a singleton. Hence, for such "compounds", even *elementless* storage is appropriate and according configurations are applicable to fully represent the composed document. To illustrate its effectiveness, we ran several analyses of typical document collections.

We evaluated a bunch of *TPoX* benchmark [15] documents, where our storage manager applied three different path synopses to store 80,000 small documents (1–20KBytes) of three different categories. A simple topdown comparison by path synopsis matching suffices to determine the collection membership. The mapping of path synopses to documents and vice versa can be handled by a simple lookup table in the system catalog. For the *TPoX* documents, the structural information within each group is confined to a maximum of 139 path classes and to 139 VocIDs to encode node names of the document. Including a number of common structural elements, the resulting *TPoX* collection is confined to 295 distinct path classes and a vocabulary of 276 elements. This experiment has shown that, because of the structural similarity of these documents, such database-driven collections remain quite stable and, furthermore, the path synopsis preserves its small size.

4.4 Workload Dependencies

A second group of benchmarks was performed to compare the standard storage performance with a database-chosen configuration—based on heuristics and experimentally evaluated rules—and a theoretically optimal setup. In Figure 5, we have assembled three benchmark results reflecting three different workload patterns. We further weighted the workload as shown in Table 2 to distinguish between typical usage patterns. The workload scenarios introduced above (Figure 4) were combined and weighted. In each diagram, the *standard* configuration is used as reference (y-axis gain is exactly 1). The solid bar shows the performance gains of the *database*-chosen setup. The second bar is the theoretical limit if each workload within a pattern is executed on an optimal configuration on its own; unfortunately that is not possible because most storage parameters cannot be easily changed during runtime. In Figure 5a, we can see that a workload dominated by *SAX-based* operations does not benefit much from an optimized storage configuration. But especially for the *Read*-operation-dominated workload, depicted in Figure 5b, a *database*-chosen setup is competitive and nearly as good as the theoretical optimum. Even the *Write*-dominated workload, depicted in Figure 5c, can be improved by a respectable margin.

5. RELATED WORK

As XML documents are likely to be "verbose", applying appropriate compressions for their physical representation is obviously a good choice. XML structure transformation into relational schemes ("shredding") require complex query translation into SQL. In contrast, native approaches, using suitable labeling schemes, were designed to provide solutions without the need to convert documents between different data models. Prefix-based schemes enabling the evaluations of all XPath axes (see Section 2.1) offer comprehensive support for dynamic documents in a multi-user environment [7].

Concerning XML's variety in size, depth, fan-out, and vocabulary, a fixed storage scheme is often charged with a non-negligible trade-off, often not explicitly dealt with. So far, proposed approaches [5, 10] are storing XML documents separately and "complete", implying that the inner structure is fully represented. DataGuides [6] are used for auxiliary indexes and statistic support, but not, to the best of our knowledge, to tailor XML storage consumption. Tree segmentation, such as used in [5, 10], split documents into subtrees. Such hybrid storage formats regard document structure and content sizes, while connecting the parts with so-called proxy nodes.

In native DBMS-based XML, content compression should

Table 2: Workload weights

Workload	SAX	Read	Write
put	3	1	1
get 1	3	1	1
get 5	3	0	0
read	1	7	2
mixed	0	1	3
write	0	0	3

preserve the structure and should be orthogonal to structure encoding; hence, queryable and non-queryable compressors are not appropriate. The latter ones [12] are only applicable for document interchange or archiving and most queryable compressors focus on read-only scenarios, where document modifications would imply an enormous effort to re-compress changed parts.

6. CONCLUSIONS

In this paper, we primarily discussed important concepts needed to obtain optimal and tailor-made storage configurations for XML documents. We elaborated the potential benefit of XML structure and content compression methods. Heuristics for typical document usages as well as a kind of analysis is needed to identify structure parameters for optimal and, if possible, automatic selection of a storage model. Our performance measures indicate the potential storage saving and operational gain using concepts of adaptivity. Besides hardware limitations, algorithmic costs become more important as operations on documents were repeated very frequently. Here, small documents and documents with a few instances per path class may benefit using a slightly larger storage model. We have shown that heuristics may help to find these "break-even" points to automatically switch storage parameters by the storage manager. Based on three fundamental storage types (full, elementless, and collection), additional encodings for labeling, indexes, and content compression should be adjusted to expected workloads and CPU/memory capabilities.

Future work will explore more sophisticated complex document operations (i.e., modifications) requiring index maintenance in case of additional indexes. Moreover, schema evolution may play a central role in adaptive XML document handling [1] and storage parameters may be automatically adjusted to workload shifts.

7. REFERENCES

[1] Aguiar Moraes Filho, K., and Härder, T.: EXsum - An XML Summarization Framework, *submitted*

[2] Balmin, A., Beyer, K. S., Özcan, F., and Nicola, M.: On the Path to Efficient XML Queries. *Proc. VLDB*: 1117-1128 (2006)

[3] Böhme, T., and Rahm, E. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. *Proc. 3rd Int. Workshop Data Integration over the Web* (DIWeb), Riga, Latvia, 70-81 (2004)

[4] Dewey, M.: Dewey Decimal Classification System. http://www.mtsu.edu/~vvesper/dewey.html

[5] Fiebig, T., Helmer, S., Kanne, C.-G., Moerkotte, G., Neumann, J., Schiele, R., and Westmann, T.: Anatomy of a native XML database management system. *VLDB J.* 11(4): 292-314 (2002)

[6] Goldman, R. and Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. *Proc. VLDB:* 436-445 (1997)

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

[8] Härder, T., Mathis, C., and Schmidt, K.: Comparison of Complete and Elementless Native Storage of XML Documents. *Proc. IDEAS 2007*, Banff Canada, Sept 2007, pp. 102-113

[9] Haustein, M. and Härder, T.: An Efficient Infrastructure for Native Transactional XML Processing. *Data & Knowl. Enineering* 61, 500-532 (2007)

[10] Jagadish, J.V., Al-Khalifa, S., Chapman, A., Lakshmanan, L. V S., Nierman, A., Paparizos, S., Patel, J.M., Srivastava, D., Wiwatwattana, N., Wu, Y., and Yu, C.: TIMBER: A native XML database. *VLDB Journal* 11(4): 274-291 (2002)

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

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

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

[14] 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)

[15] XML Database Benchmark: Transaction Processing over XML (TPoX), http://tpox.sourceforge.net/ (January 2007)

[16] Zhang, N., Özsu, M. T., Aboulnaga, A., and Ilyas, I. F.: XSEED: Accurate and Fast Cardinality Estimation for XPath Queries. *Proc. ICDE 2006*: 61