Comparison of Complete and Elementless Native Storage of XML Documents

Theo Härder Christian Mathis Karsten Schmidt*
University of Kaiserslautern
67663 Kaiserslautern, Germany
Email: {haerder,mathis,kschmidt}@informatik.uni-kl.de

Abstract

Because XML documents tend to be very large, are accessed by declarative and navigational languages, and often are processed in a collaborative way using read/write transactions, their fine-grained storage and management in XML DBMSs is a must for which, in turn, a flexible and space-economic tree representation is mandatory. In this paper, we explore a variety of options to natively store, encode, and compress XML documents thereby preserving the full DBMS processing flexibility on the documents required by the various language models and usage characteristics. Important issues of our empirical study are related to node labeling, document container layout, indexing, as well as structure and content compression. Encoding and compression of XML documents with their complete structure leads to a space consumption of ~40% to ~60% compared to their plain representation, whereas structure virtualization (elementless storage) saves in the average more than 10%, in addition.

1. Motivation

So far, XML research primarily focuses on the management of a few isolated documents which are typically very large (up to several GBytes). Frequently cited examples are available from [15] which reveal huge storage consumption and processing requirements. In many cases [9], [10], storage structures are optimized for specific situations and indexing schemes only support searching (say, based on XPath predicates) within a single document. For general DBMS use, it is mandatory to preserve the full processing flexibility of the "original" documents, while it is highly advisable to provide encoded and suitably compressed storage structures to save storage space and transfer time.

What are the essential characteristics of such XML documents? An empirical study [14] gathered about 200,000 XML trees worldwide where 99% have less than 8 levels, i. e., less than depth 8 which should be the primary goal of optimization. Almost all of the remaining 1% documents range between 8–30. Only a tiny fraction of the documents gathered has more than 30 levels. To gain some insight into the structural parameters, we have empirically explored a variety of XML documents [7], for which we can only list a summary of the results. The document size is measured in the *plain* format where the XML document is stored in its external "verbose" representation without any compression technique applied (readable element and attribute names, empty spaces, etc., but without node labels used for DBMS-internal processing). The entries in Table 1 contain a representative subset of all documents, called *reference documents*, and serve as our test set in the following. These documents range from a uniform XML structure of moderate depth (4)—representing the content of a relational table—to GB-sized documents of rich XML structures and larger depths. As the last entry, *treebank* is included to show an exotic outlier used to determine the reach of our optimization efforts.

These documents are rather data-centric than document-centric, as confirmed by the column 'avg. value size per content node' in Table 1. For issues such as content compression and relative mapping overhead due to node labeling when natively stored, this kind of documents represent bad or even worst cases. As visualized in Figure 3 and 6, most space is consumed by mapping and control information (node labels, administrative data, etc.) rather than content values. In contrast, document-centric XML structures,

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

Table 1: Characteristics of XML documents considered

doc name	description	size in Mbytes	# elem. & attr. nodes	# content nodes	avg. value size per content node	# vocab. names	# path classes	max. depth	avg. depth
line-item	LineItems from TPC-H benchmark	32.3	1,022,977	962,801	12.5	19	17	4	3.45
uni-prot	Universal protein resource	1,821.0	81,983,492	53,502,972	24.0	89	121	7	4.53
dblp	Computer science index	330.0	9,070,558	8,345,289	17.0	41	153	7	3.39
psd-7003	DB of protein sequences	717.0	22,596,465	17,245,756	6.5	70	76	8	5.68
nasa	Astronomical data	25.8	532,967	359,993	20.9	70	73	9	6.08
tree-bank	English records of Wall Street Journal	89.5	2,437,667	1,391,845	33.4	251	220,894	37	8.44

e.g., in digital libraries where a leaf node may host the text of a paper or even a book, have much better overhead/content ratios and would provide much more opportunity especially for content compression.

For the empirical study and all measurements in this paper, we use our prototype DBMS called XTC (XML Transaction Coordinator [8]) which stores and manages XML documents in a native way. To optimize XML storage structures, we describe the most important concepts and options in Section 2. In Section 3, we analyze the storage consumption of formats which store the complete document, i. e., structure and content. In Section 4, we develop a method to virtualize the documents' structure without loosing functionality, before we show that the use of path synopses scales and is a general method to replace the document structure in the storage format. Finally in Section 5, we wrap up with conclusions.

2. Fine-Grained XML Storage

Efficient and effective processing including concurrent read/write operations on XML documents are greatly facilitated, if we use a fine-grained, tree-like internal representation. For this reason, we have implemented in our XTC system an XML tree representation as defined in [20]. In the following, we discuss the core issues of the so-called dynamic DOM storage model exemplified by Figure 1. The structure consists of all inner nodes including the node labels, whereas the leaf nodes together with their labels capture the content of the document.

2.1. Node Labeling

After quite some practical experience, we are convinced that node labeling is the key to efficient management and compression of XML documents. Early requirements included navigational and declarative access of *static* XML documents which put the only focus on the fast evaluation of the 13 axes (parent/child, ancestor/descendant, ...) of the XPath 2.0 and XQuery language models thereby guaranteeing the sequence semantics. Using complete k-ary trees [12] to establish a consecutive numbering scheme enabled direct and very cheap node label computation for the checking of axes predicates, but failed in case of real documents having incomplete structure of considerable breadth and depth, not to mention dynamic documents.

Substantial development effort was spent on labeling schemes supporting dynamic XML documents for which various forms of range-based and prefix-based schemes were proposed [3], [5]. Although equivalent for checking axes predicates, range-based schemes seem to exhibit some inflexibility when extensive document updates (subtree insertions) have to be accommodated. They completely fail if fine-grained document locking has to be supported. When entering inner nodes of the document via indexes, the entire ancestor path up to the root has to be protected by intention locks [13]. The required functionality to determine all ancestor node labels comes for free using prefix-based schemes, whereas range-based schemes need access to the document and/or additional indexes, thus, typically provoking disk accesses. As a consequence, prefix-based schemes are preferable for dynamic documents with multi-user read/write transactions and also for speeding up index-based processing (see Section 4.4).

An intensive comparison of labeling schemes and their empirical evaluation [7] led us to redesign the existing mechanism in XTC based on a straightforward numbering scheme. DOM trees empowered with prefix-based node labels can be considered as an abstract access model much more flexible for XML document processing; it served as a powerful and adap-

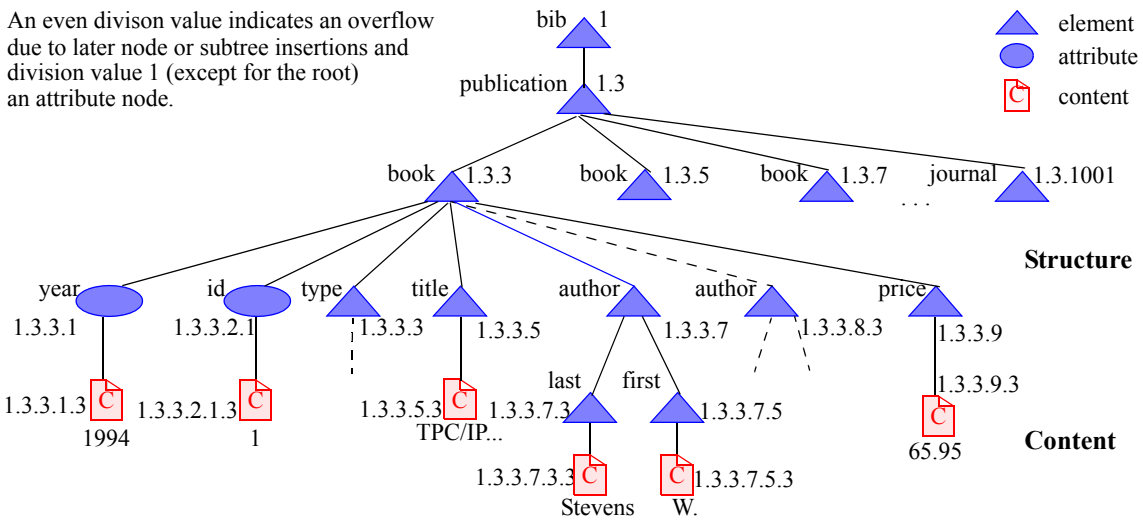

Figure 1: A sample DOM tree labeled with SPLIDs using dist=2

tive structure to be implemented by our storage model in XTC. The prefix-based scheme for the labeling of tree nodes is based on the concept of Dewey order [4] characterized by Figure 1. The abstract properties of Dewey order encoding—each label consists of so-called *divisions* (separated by dots in the external format) and represents the path from the document's root to the node and the local order w. r. t. the parent node; in addition, optional sparse numbering facilitates node insertions and deletions—are described in [3]. Refining this idea, a number of similar labeling schemes were proposed which differ in some aspects such as overflow technique for dynamically inserted nodes, attribute node labeling, or encoding mechanism. Examples of such schemes are DLN [1] or Ordpath [17] developed for Microsoft SQL Server™. Although similar to them, our scheme is characterized by some distinguishing features and is denoted DeweyIDs [7]; it refines the Dewey order mapping: with a *dist* parameter used to increment division values, it leaves gaps in the numbering space between consecutive labels and introduces an overflow mechanism when gaps for new insertions are in short supply—a kind of adjustment to expected update frequencies. Because any prefix-based scheme is appropriate, we use the term SPLID (Stable Path Labeling IDentifier) as synonym for all of them.

Existing SPLIDs are *immutable*, that is, they allow the assignment of new IDs without the need to reorganize the IDs of nodes present. When labels degrade after weird insertion histories[1], relabeling can be preplanned; it is only required, when implementation restrictions are violated, e. g., the max-key length in B*-trees. Comparison of two SPLIDs allows *ordering* of the respective nodes in document order. Furthermore, SPLIDs easily provide the IDs of all ancestors, e.g., to enable intention locking of all nodes in the path up to the document root without any access to the document itself [13]. For example, the ancestor IDs of 1.3.3.7.5.3 are 1.3.3.7.5, 1.3.3.7, 1.3.3, 1.3 and 1.

1. For example, point insertions of thousands of nodes between two existing nodes can be attenuated by the dist parameter, but nevertheless may produce large SPLIDs.

2.2. Physical Node Representation

Having in the order of 10^8 nodes in large XML documents, node encoding needs careful optimization considerations. All node formats (for elements, attributes, or text) are of variable length. Element nodes and attribute nodes only consist of a *key part* and a *name part*, whereas a text node has only a key part and a value part. Because the key part consisting of a one-byte field *KL* (key length) and the encoded SPLID is the Achilles heel of the storage representation (see Figure 1 and 3), it must be reduced very efficiently.

As explored in [7], Huffman codes enable effective and efficient encoding of division values. They consist for each division of a variable-length L_i-code and a binary value O_i stored as (L_i-code | O_i). Using a specific encoding assignment such as in Table 2, a division can be encoded and decoded. Because all SPLIDs start with "1.", we do not need to store it and save 4 bits per SPLID. In addition, we can adjust the Huffman encoding scheme to typical value distributions in the SPLIDs and align codes and value representations to byte boundaries. Hence, this flexi-

Table 2: Assigning codes to divisions

L_i-code	length O_i	value range of O_i
0	3	1 – 7
100	4	8 – 23
101	6	24 – 87
1100	8	88 – 343
1101	12	344 – 4.439
11100	16	4.440 – 69.975
11101	20	69.976 – 1.118.551
11110	24	1.118.552 – 17.895.767
11111	31	17.895.768 – 2.147.483.647

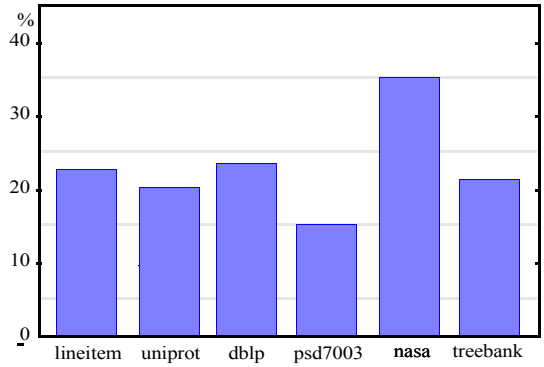

Figure 2: Efficiency of prefix compression

bility enables the tailor-made construction of Huffman codes as illustrated in Table 2.

When traversing and storing XML trees in document order (left-most depth-first order), as visualized in Figure 3, the sequence of SPLIDs lends itself to prefix compression in the key part. To exploit this observation, we designed a prefix-encoded SPLID representation consisting of a one-byte field *Rpip* (reduction of prefix inherited from predecessor) and the actually stored remainder (*Rem*) of the SPLID. Compression is achieved as follows: Within a container page, assume the SPLID sequence 1.3.3.17.33.3, 1.3.3.17.33.5, 1.3.3.17.33.7, 1.3.3.19.3, 1.3.3.19.3.3, ...; then, starting with the first SPLID 1.3.3.17.33.3, we encode the next SPLID by removing a number of divisions from the end to get the common prefix with the current SPLID and add the remainder as a new suffix division sequence: hence, (*Rpip+Rem*) entries in our example look as follows: '-1'+.5, '-1'+.7, '-3'+.19.3, '0'+.3, ... Obviously, this kind of prefix compression achieved the lion's share of space saving. Applied to all SPLIDs in the collection of our reference documents, we obtained the indicative results illustrated in Figure 2. Hence, it is safe to say that prefix compression reduces the space consumed by SPLIDs down to ~25%.

2.3. Document Storage

Document storage is based on variable-length files as document containers whose page sizes varying from 4K to 64K bytes could be configured to the document properties. We allow the assignment of several page types to enable the allocation of pages for documents, indexes, etc. in the same container. Efficient declarative or navigational processing of XML documents requires a fine-granular DOM-tree storage representation which easily preserves the so-called round-trip property when storing and reconstructing the document (i.e., the identical document must be delivered back to the client). Furthermore, it should be flexible enough to adjust arbitrary insertions and deletions of subtrees thereby dynamically balancing the document storage structure. Fast indexed access to each document node, location of nodes by SPLIDs, as well as navigation to parent/child/sibling nodes from the current context node are important demands. As illustrated by Figure 3, we provide an implementation based on B*-trees which maintains the nodes stored in document order and which cares about structural balancing.

No matter what kind of language model is used for document modification, its operations at the storage level have to be translated into node- or record-at-a-time operations. The overwhelming share of the overhead caused by updates of nodes (names or values) or by insertions/deletions of subtrees in the XML document is carried by two valuable structural features: B*-trees and SPLIDs. B*-trees enable logarithmic access time under arbitrary scalability and their split mechanism takes care of storage management and dynamic reorganization. In turn, SPLIDs provide immutable node labeling such that all modification operations can be performed locally.

While indexed access and order maintenance are intrinsic properties of such trees, some additional optimizations are needed. Variations of the entry layout for the nodes allow for single-document and multi-document stores, key compression, use of vocabularies, and specialized handling of short documents. As shown in Figure 3 by sketching the sample XML document of Figure 1, a B-tree, the so-called *document index*, with key/pointer pairs (SPLID+PagePtr) indexes the first node in each page of the *document container* consisting of a set of chained pages. Using sufficiently large pages, the document index is usually of height 1 or 2. Because of reference locality in the B-tree while processing XML documents, most of the referenced

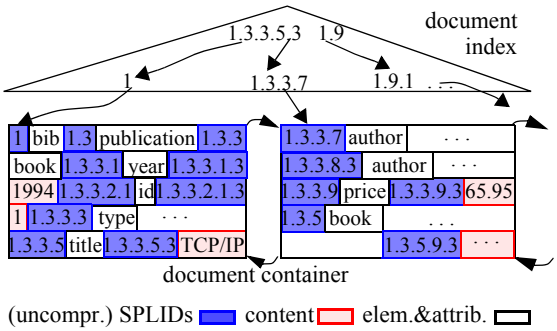

Figure 3: Stored XML document

tree pages are expected to reside in DB buffers—thus reducing external accesses to a minimum.

The value part of a content node is materialized (stored inline) up to a parameterized *max-val-size* together with the node as a string (of given type). When the content size exceeds *max-val-size*, then it is stored in referenced mode where it is divided in parts each stored into a single page and reachable via reference from its home page, as illustrated in Figure 4.

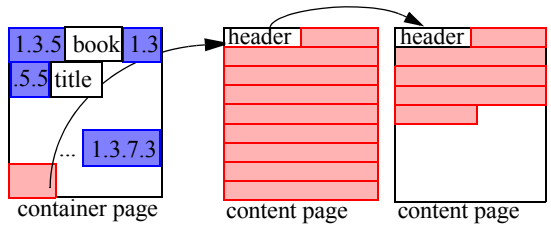

Figure 4: Layout of long fields

2.4. Structure and Content Indexes

In addition to the document store, various indexes may be created, which enable access via structure (element or attribute nodes) or content (values of leaf nodes). An *element index* consists of a *name directory* with (potentially) all element names occurring in the XML document (Figure 5); this name directory often fits into a single page. Each specific element/attribute name refers to the corresponding nodes in the document store using SPLIDs. In case of short reference list, they are materialized in the index; larger lists of references may, in turn, be maintained by a *node reference index* as indicated in Figure 5. Content indexes are created for root-to-leaf paths, e. g., /bib/book/title, and again are implemented as B*-trees keeping for each indexed value a list of SPLIDs as references to the related locations in the document. When processing a

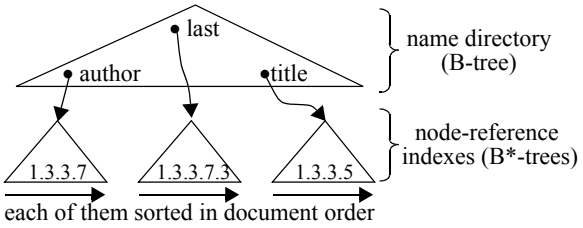

Figure 5: Organization of structure indexes

query, a hit list of SPLIDs is built using one or several indexes. Then, the qualified nodes together with their related path instances are located via the document index (see Figure 3). In all cases, support of variable-length keys and reference lists is mandatory; additional functionality for prefix compression of SPLIDs is again very effective.

2.5. Content Compression

There exists a large body of scientific contributions dealing with XML compression technologies [16], [18] promising enormous gains in storage saving and, at the same time, enabling a kind of query processing (restricted to very simple XPath expressions). However, all these approaches are coarse-granular thereby directly compressing the *plain*, i.e., "verbose" representation, assume static and file-based scenarios with single-user operations, are often context-dependent requiring large auxiliary data structures, and need potentially substantial compression/decompression overhead [19]. Therefore, these methods are not adequate for dynamic XML structures processed in a multi-user transactional DBMS context and, in turn, cannot be considered as candidates for our fine-grained tree-like structures.

Two of the main issues to be regarded for compression of fine-grained XML documents in databases result from the XML structure itself and its content. In contrast to the relational world, where typically column-based compression is used, the storage representation of XML paths and their uncorrelated sequence of element/attribute names complicate "simple" path-based compression algorithms such as XMill [11]. Furthermore, transactional modification applied to XML documents prevents *block-based compression* used by PPM algorithms [18]. Note, it does not seem to be helpful to separate content and structure, only to enable the concatenation of smaller values to larger text blocks and, in this way, to achieve better compression results. Such an approach would involve a complete cycle of de- and re-compression when a specific node

value is modified. Thus, to avoid undue limitations and overhead of XML processing, compression of single node values seems to be an appropriate and challenging choice. Therefore, we exclusively focus on single nodes and their data stemming either from text content or attribute values.

In our view, there exist two practical approaches to such kind of compression. For compressing node-based content, either *word-based* or *character-based compression* algorithms can be applied. For example, our vocabulary for element/attribute names can be considered as a specific word-based compression (applied to the structure part) used in nearly all XML databases. Due to the dynamic values of XML content nodes, it would be hard to keep a word-based dictionary for compression purposes up to date. In addition, such a dictionary would not have size limitations and, therefore, fast lookups in a memory-resident data structure could not be guaranteed. Furthermore, all our reference documents are rather data-centric having relatively short content values where word-based compression methods would cause too much overhead with limited effect. Therefore, we prefer character-based, context-free compression schemes like Huffman which also accomplish homomorphic transformations which guarantees that compressed and non-compressed documents can be processed by the same operations like parsing, searching, or validating. Hence, we provide an efficient and context-free compression/decompression algorithm called *Fixed Huffman* (FH) which seems to work sufficiently effective on data-centric documents, i. e., short node values. On a document basis, we build either a Huffman tree optimized w. r. t. the typical character distribution of the document's domain or, during an analysis run, we collect the character frequencies of the specific document and construct the optimal Huffman tree for it. To adjust for later document modifications, all 256 possible characters are considered.

3. Complete XML Documents

So far, we have outlined the essential concepts used for the optimization of native XML document storage. In our empirical study, we focus on the variability and optimization of storage structures which can be chosen by the DBMS for incoming documents. The question which secondary element/attribute indexes or content 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. Here, we primarily want to illustrate how much storage consumption can be reduced by applying our storage concepts to the documents. As a comparison mark, we use the storage space needed for a document in its textual representation, i. e., a document in the format sent by the client (user) to the DBMS. We denote this as the *plain* format and normalize all results for a given document to this format (consuming 100%). In the following, we distinguish for all formats between the storage space needed for the content part and the structure part. For the collection of our reference documents, the storage consumption of 'plain' is listed in column 3 of Table 1. As illustrated in Figure 6 and presented in the Appendix in greater detail, the relative fraction of the *plain* structure part—as the prime target of our optimization—ranges between ~45% and ~81%.

The *standard* format stands for the normally chosen native XML document storage in DBMSs; it uses our structural framework: For the content part, it stores uncompressed content and SPLIDs and, for the structure part, SPLIDs, "long" VocIDs (2 bytes), and some administrative data. Storage saving as compared to *plain* is not really mind-blowing, because the reduction gained in the structure part by VocID use is partially compensated by SPLID labeling. For the content part, substantially more space is needed in all cases as compared to the *plain* content, because the relative storage space needed for content nodes due to the SPLIDs added is increased by up to ~50%. The highest reduction for *standard* obtained by *lineitem* is ~30%.

The *compressed* format tries to save storage space as much as possible and stores all structure and content nodes with prefix-compressed SPLIDs and with "short" VocIDs (1 byte), because the vocabularies for our reference documents are small (see column 7 in Table 1). Furthermore, it compresses the content nodes using the FH algorithm. In all cases, the content part is smaller than in the *plain* format, although compressed SPLIDs are added to the nodes. As summarized in Figure 6, storage saving becomes remarkable and ranges between ~40% and ~58% depending on the structure and content particularities in the collection of our reference documents.

The content part compression seems to be exhausted, because data-centric documents with relative small value sizes per content node (see col. 6 in Table 1) do not lend themselves to content compression. Hence for further optimizations, we should concentrate on the structure part. Although we have squeezed it as far as possible within the given tree context, it still consumes quite some fraction of the total compressed document, e.g., 168% of the content part of *lineitem*. Therefore, a novel approach to virtualize the structure seems appropriate, thereby reducing the space consumption further without abandoning processing functionality.

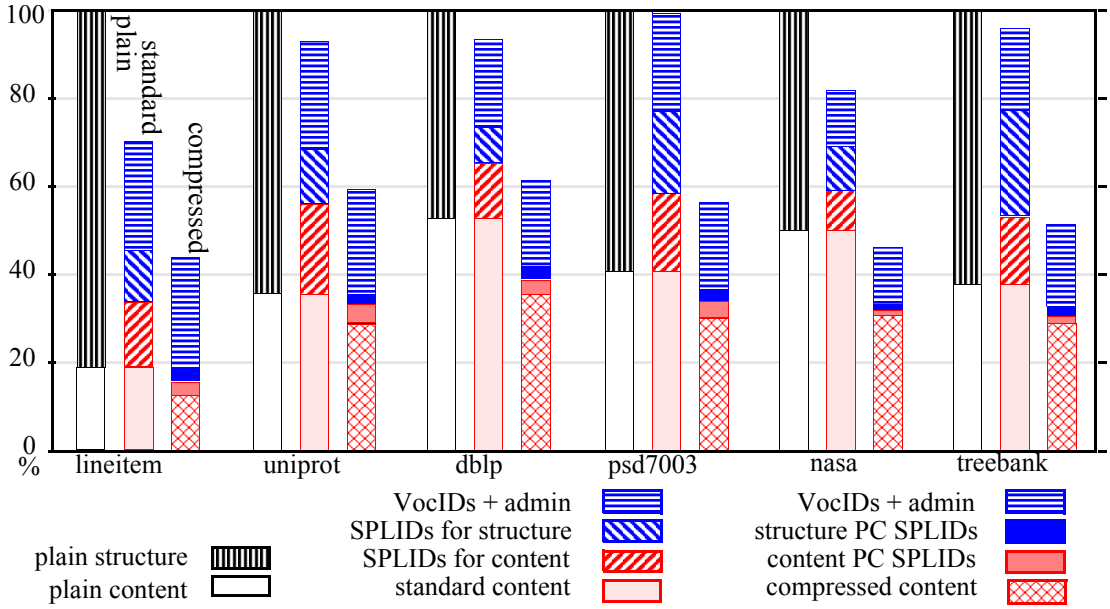

Figure 6: Storage consumption of complete XML documents

4. Virtualizing the XML Structure

So far, compression of structure has used the idea of replacing long external names by so-called VocIDs which index a vocabulary containing all distinct names of elements and attributes. Nevertheless, each structure node had to be explicitly represented and labeled by a (prefix-compressed) SPLID. Obviously with I inner nodes and L leaf nodes, $I > L$ or $I/L > 1$ always holds for all document trees (see Fig.1). To give a rough estimate of the number of structure nodes, we assume a complete tree of height h ($h > 1$) and fan-out n_i at level $1 < i < h$ and consider that each inner node at level $h-1$ has exactly one leaf node assigned at level h. Then, we obtain

$$I = 1 + \sum_{i=2}^{h-1} \left(\prod_{j=2}^{h-1} n_j \right) \quad \text{and} \quad L = \prod_{i=2}^{h-1} n_i \quad (1)$$

Of course, I depends on the specific inner structure of the document. The relationship I/L may be unbounded ($>>$ 1), if many one-way branches occur in the structure part. In case of complete binary structure trees ($n_i = 2$ for $i = 2, ..., h-1$), the well-known relationship $I/L < 2$ holds. To estimate the degree of redundancy present in the structure part, we have listed indicative numbers in Table 1. By looking at columns 4 and 5, we can confirm that I/L is always less than 2. Now consider columns 7 and 8. It immediately becomes clear, that huge repetitions are buried in the names (resp. VocIDs) of the inner nodes and paths. All paths from the root to the leaves having the same sequence of element/attribute names form a *path class*. Thus, each path in the document can be assigned to one of the relatively few distinct path classes. Reflecting these values, dramatic repetition factors become obvious: consider *uniprot*, in the average, each VocID is repeated >921,000 times and each path >442,000 times.

4.1. Path Synopsis

Our key idea is now to capture all path classes of an XML document in a small data structure [6]. Having such a separate structure, we can remove and drop the entire structure part from the physically stored document and, nevertheless, are able to reconstruct each path or the entire document, whenever needed. Note, by providing such an on-demand option, we don't want to sacrifice functionality, but only safe substantial storage space. Again, the *secret* is the SPLID mechanism with which each node carries a shorthand representation of its entire path to the root. The part missing to deliver the complete path information are the attribute/element names (resp. VocIDs) of all ancestor nodes. This task, we will "outsource" to a so-called *path synopsis*.

For this purpose, we have designed a little memory-resident data structure which maintains all path classes of a document. Cyclic-free XML schemata capture all information needed for the path synopsis; otherwise, this data structure can be constructed while

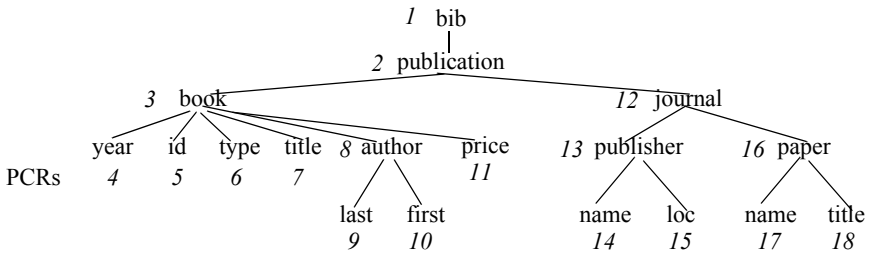

Figure 7: Path synopsis using PCRs to identify paths to the root

the document is stored. To illustrate our approach in Figure 7, we have derived a path synopsis from the document fragment sketched in Figure 1 extended by some additional nodes and path classes. Such a concise description of the document's structure is a prerequisite of effective virtualization of the structure, i.e., for an elementless storage of the document. When comparing them to the number of path instances, it becomes obvious that huge redundancy is introduced when all path instances are explicitly stored. In the popular *dblp* document, for example, one of the dominating path classes /bib/paper/author has ~570,000 instances.

Hence, when matching the right path class with a given SPLID, it is very easy to reconstruct the specific instance of this path class. In a sense, we must associate the value in a document leaf—whose unique position in the document is identified by its SPLID—with a space-saving reference to its path class. Furthermore, when document processing references an inner node—for example, by following an element index for *author* or by setting a lock on a particular *book* for some concurrency control task—, we must be able to rapidly derive the (sub-) path to the root in the virtualized structure. For this reason, by numbering all nodes in the path synopsis, we gain a simple and effective mechanism called path class reference (PCR). Such PCRs are used in the content nodes or in index structures together with SPLIDs serving as a path class encoding.[2]

The sketched usage of the path synopsis indicates its central role as a repository to be used for all structural references and operations. Although it can be stored in a little data structure residing in memory, it should provide indexed access via PCRs and via node names. Another helpful piece of information to be captured in the path synopsis is the number of instances for each path class appearing in the document or other selectivity or fan-out information supporting query optimization. Finally, the path synopsis represents a kind of type structure which may be efficiently used for hierarchical locking protocols on the document structure.

2. If a node has an *empty* value, the respective node type must carry a PCR to map the empty value to the correct path.

4.2. Elementless Document Storage

Using an elementless layout of a natively stored document, we want to get rid of the structure part in a lossless way. For an XML document, only its content nodes are stored in document order using—in a similar way as for the complete document—a container as a set of doubly chained pages. The stored node format is of variable length and is composed of entries of the form (SPLID, PCR, value). Otherwise, as illustrated in Figure 8, its storage format exactly corresponds to the data structure in Figure 3. Again, the resulting B*-tree and its split/merge mechanism together with the SPLID mechanism take care of the storage management and label stability in case of modifications in the XML document.

The *elementless* format only stores the content nodes carrying prefix-compressed SPLIDs and adjusted PCRs together with some administration data (2 bytes). As shown in Figure 8, two aspects increase the content part as compared to *compressed*, a PCR (with administration data) is added to the node format and the effectiveness of our prefix compression may get worse due to the non-dense SPLID sequence. Note, both aspects only may be critical in case of data-centric documents because of the unfavorable ratio between mapping overhead (SPLID + PCR + admin) and the relatively short values in content nodes. Fortunately, both effects have only limited influence that our idea of structure virtualization pays off.

As illustrated in Figure 8, elementless storage has considerably reduced the density of the SPLID sequence, because all SPLIDs of structure nodes were removed. Obviously, the SPLID density of the complete documents caused the excellent results for prefix compression (PC). Therefore, we have checked the influence of this contra-effect to the overall space reduction. As presented in Figure 9 for the non-dense

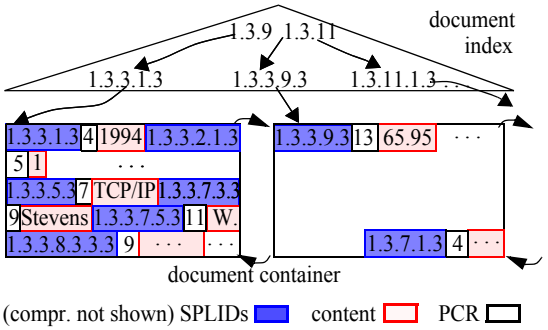

Figure 8: Stored elementless XML document

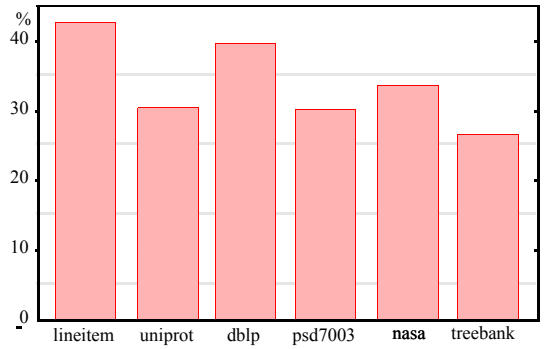

Figure 9: PC in elementless documents

case, prefix-compressed SPLIDs only decrease storage space down to ~35%, as compared to ~25% in the dense case as shown in Figure 2—a trade-off effect that is acceptable.

Figure 10 (and Table 4 in the Appendix) summarize our results gained by structure virtualization. Considering only the compressed SPLIDs and content, the space for our *elementless* reference documents does not consume more storage than the *plain* content. Of course, mapping data PCRs + admin roughly need 10% of the *plain* document size, such that the *plain* content size can not be beaten by *elementless*. However, *elementless* compared to the optimized complete document (Figure 6) is reduced once more by ~10% to ~20% of the *plain* size. Referring to *plain* format, we have achieved a reduction for the optimized document storage down to ~30% to ~50% of the *plain* size. In case of *uniprot*, this saving is 976 MByte. When directly comparing *compressed* and *elementless*, the relative saving ranges from ~15% to ~25%.

One may argue that this favorable space behavior is paid by some extra processing overhead, because coding/decoding and compression/decompression has to be performed for allocating the documents on disk and for (partially) reconstructing them for internal processing or output to the client. However, the compaction effect on the XML structures saves disk I/Os directly proportional to the reduction ratio obtained. Furthermore, it has a beneficial effect on caching effectivity, because the same cache size can host a larger fraction of the XML document in its compacted form. Therefore, there is some potential for them to win the contest also for processing times.

4.3. Comparison of Processing Times

Because the document sizes of our reference collection vary by about two orders of magnitude, the I/O-driven processing times differ by the same ratio—ranging from ~45 secs to ~6600 secs. Therefore, it is unreasonable to compare them directly on an absolute scale. We rather normalize them to the elapsed times needed for *standard* (100%).

The characteristic processing times chosen are the costs of loading and reconstructing a document thereby preserving the round-trip property. Sent by a client in *plain* format, it is stored resp. fetched in the *standard/compressed/elementless* formats on/from disk and delivered in *plain* format to the client. As illustrated in Figure 11, all loading and reconstructing times are less than the resp. times for *standard* (loading=100%) although additional overhead for SPLID encoding and content compression had to be spent. This fact clearly confirms the dominating role of I/O in processing large XML documents. The same observation can be approved when comparing the processing times for both format optimizations with each other. By examining Figure 6 and Figure 10, we can state that the storage saving gained from choosing format *compressed* instead of *standard* resp. *elementless* instead of *compressed* translates in all cases to shorter processing times for fine-grained document storage and reconstruction back to the external format.

4.4. Using Indexes on Elementless Documents

Index support is achieved in the same way as described in Section 2.4. The various types of indexes (for content or elements/attributes) refer to the indexed nodes via SPLIDs. Location of a content node is performed via the document index, as illustrated in Figure 8, where the path instance is then recomputed. The option to include the PCR together with its SPLID in the reference lists, enables query processing often without accessing the document itself, because the path instance can be regained by using the path synopsis directly. In element/attribute indexes, this extended reference format (SPLID+PCR) is mandatory for perfor-

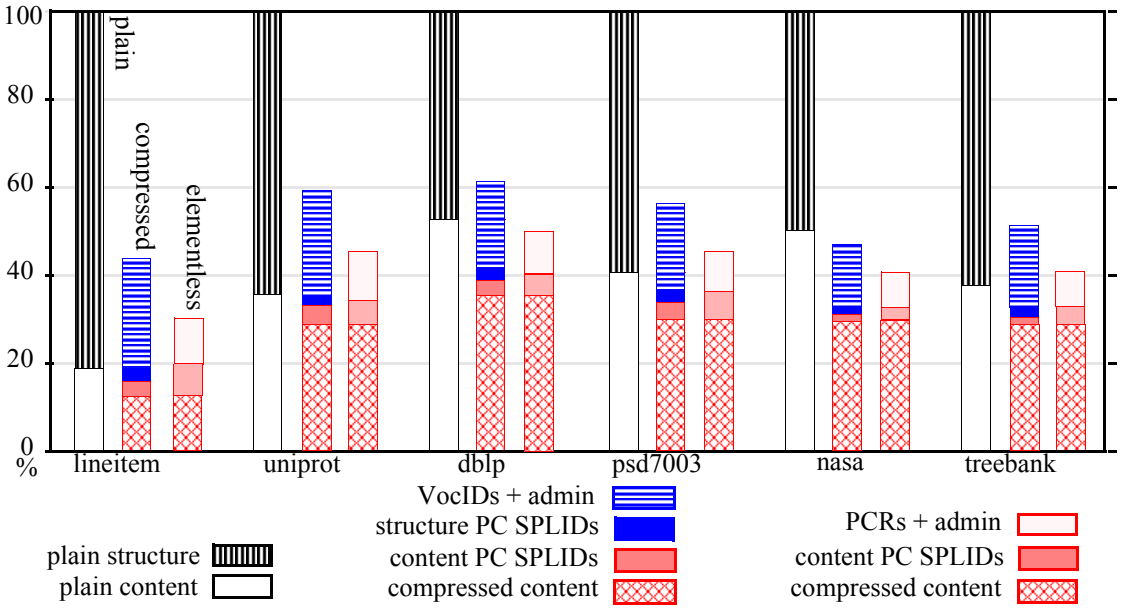

Figure 10: Storage consumption of elementless XML documents

mance reasons, because an algorithmic reconstruction of the virtualized path instance in the non-existing structure part is too troublesome.[3]

In the same way, as we derive the structure or particular path class instances on demand, we can answer content-and-structure (CAS) queries only by using content indexes. Query evaluation on these indexes (B*-trees) delivers a set of SPLIDs together with the PCRs which enable the in-memory reconstruction of the paths (structure) belonging to the indexed values (content). The use of such CAS queries is particularly efficient, because our approach can often avoid expensive structural joins or twig evaluation [2] and can derive the path information from the combined use of SPLIDs and path synopsis. For specific CAS queries supported by content indexes, up to two orders of magnitude response-time reduction compared to traditional approaches were achieved with our XTC prototype DBMS [8].

4.5. Scalability of Path Synopsis Use

So far, we have tacitly assumed that the document's path synopsis is a little data structure that always can be kept in main memory. This is obviously true for most of our reference documents where less than a page is needed for maintaining all path classes. However, this assumption is violated for *treebank*, a kind of exotic outlier. To check the influence of size on path synopsis use, we designed a stress test for its scalability.

Our abstract approach separated structure from content and removed all redundancy from the structure by replacing it by a much more space-economic, however, functionally equivalent path synopsis. The critical question is whether or not the processing and storage benefits will be preserved when the path synopsis grows such that eventually the size of the structure part is reached. As opposed to comparing our collection of fixed-size reference documents under *compressed* and *elementless*, we have created a synthetic document and scaled it under a specific growth pattern thereby measuring the loading and reconstruction times under both storage formats with an initial document size *IDoc*. Starting with a document root, we attached as *child-1* a (sub-) tree of IDoc=26 KB and 154 PCRs. By repeatedly attaching the same tree as *child-i* (i=2, ..., n), we enforced a linear growth of the path synopsis from 1 to 2000 occurrences of *IDoc* (>300,000 PCRs and >50 MB document size) still residing in main memory. The test runs using small growth factors (<20) are disregarded in Figure 12, because extra startup costs (creation of document container and various indexes for document, elements, and IDs) dominated the load and reconstruction cost. Apparently, all our cost measures (ms per *IDoc*) remained more or less constant in the range of >20**IDoc* to 2000**IDoc* as illustrated

3. Reconstruction of path instances had to be accomplished starting from the stored leaves (content nodes). SPLID checking and path synopsis use could then identify the right path instances.

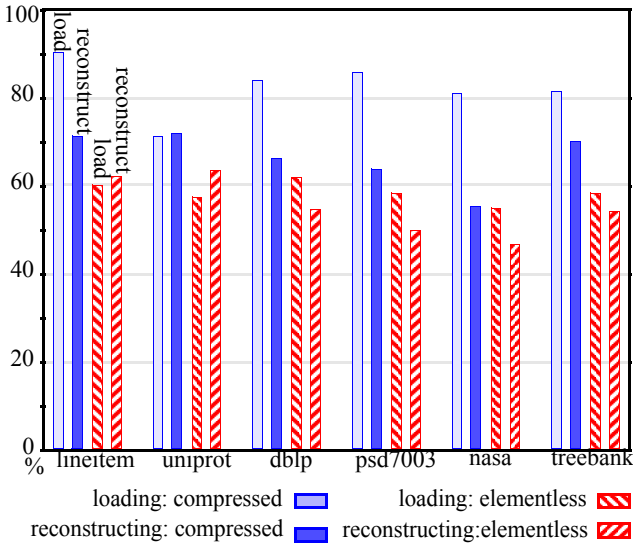

Figure 11: Comparison of processing times

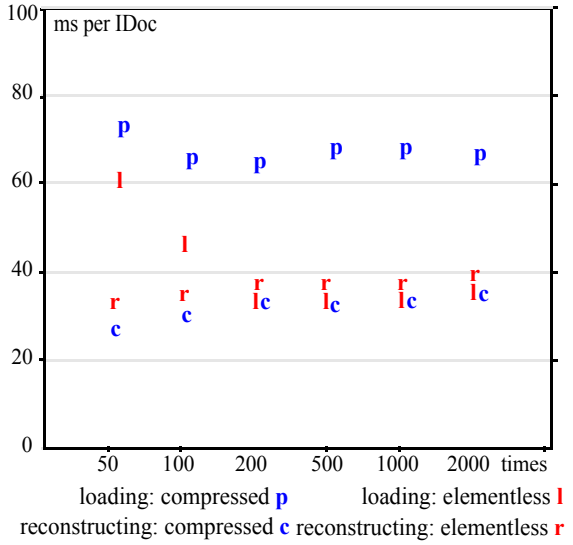

Figure 12: Scalability test for *compressed* and *elementless*

in Figure 12. This result confirms and extends our observations from the measurement of Figure 11 and states the scalability of the path synopsis as long as it can be kept in main memory (where it is designed for).

In this paper, we primarily discussed important concepts needed to obtain fine-grained storage structures for XML documents. Furthermore, we sketched the potential benefit of compression methods applied to content nodes. In a thorough empirical study, we have evaluated the storage consumption of a number of reference documents for the *standard* format and the *compressed* format. Especially, prefix compression used in the *compressed* format contributed to a large extent to the savings achieved. The still substantial storage needs for the documents' structure part gave rise to develop a path synopsis which, together with the SPLID and PCR mechanisms, enabled the design of the *elementless* format. This kind of structure virtualization really achieved impressive optimization results. Furthermore, we could show that the size of the path synopsis is insensitive to the processing times, as long as it can be kept in main memory.

5. Conclusion

What does this saving achieved by structure encoding and content compression mean? For example, assume *uniprot*: the *plain* document (in text format) arriving at the DBMS has 1821 MBytes. A straightforward encoding (VocIDs for the element/attribute names, uncompressed content, added node labels), here denoted as *standard*, results in 1685 MBytes. Our optimizations obtain for *compressed* and *elementless* ~988 and ~845 MBytes, respectively. Using any of these models, all declarative or navigational operations can be applied with the same or improved speed. Even when storing compressed contents, the use of indexes does not pose any problem.

References

[1] 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)

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

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

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

[5] Cohen, E., Kaplan, H., and Milo, T. Labeling Dynamic XML Trees. *Proc. PODS Conf.*: 271-281 (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)

Table 3: Sizes of Plain and Standard

doc name	Plain			Standard			
	total (MB)	structure	content	structure		content	
				SPLIDs	admin	SPLIDs	content
lineitem	32.3	26.1	6.2	3.8	8.0	4.7	6.2
uniprot	1,821.0	1152.0	669.0	204.0	433.0	379.0	669.0
dblp	330.0	156.0	174.0	29.3	65.7	41.7	174.0
psd7003	717.0	424.0	293.0	130.5	161.0	127.5	293.0
nasa	25.8	13.4	12.4	2.6	3.6	2.6	12.4
treebank	89.5	56.0	33.5	21.2	16.4	14.3	33.5

Table 4: Sizes of Compressed and Elementless

doc name	Compressed				Elementless		
	structure		content		content		
	SPLIDs	admin	SPLIDs	content	SPLIDs	PCR+admin	content
lineitem	1.03	7.0	0.96	4.39	1.98	3.84	4.39
uniprot	36.7	356.5	81.3	514.0	117.6	214.0	514.0
dblp	7.5	54.8	9.2	121.7	16.7	33.4	121.7
psd7003	21.4	138.0	18.6	216.9	39.9	69.3	216.9
nasa	0.47	3.1	0.43	8.07	0.9	1.5	8.07
treebank	2.4	14.0	1.4	25.4	3.8	7.0	25.4

[8] Haustein, M. P., and Härder, T. An Efficient Infrastructure for Native Transactional XML Processing, in *Data & Knowledge Engineering*, Elsevier, 2007.

[9] Jagadish, H. 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)

[10] Li, H.-G., Alireza Aghili , S., Agrawal, D., and El Abbadi, A. FLUX: Content and Structure Matching of XPath Queries with Range Predicates. *Proc. XSym*, LNCS 4156, 61-76 (2006))

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

[12] Meier, W. eXist: An Open Source Native XML Database. *Web, Web-Services, and Database Systems*, LNCS 2593, 169-183 (2002)

[13] Mathis, Ch., Härder, T., and Haustein, M. Locking-Aware Structural Join Operators for XML Query Processing, *Proc. SIGMOD Conf.*: 467-478 (2006)

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

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

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

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

[18] Shkarin, D. PPM: One Step to Practicality. *Proc. IEEE Data Compression Conf.*: 202-211 (2002)

[19] Skibinski, P., and Swacha, J. Combining Efficient XML Compression with Query Processing (2007)

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

Appendix

The Tables 3 and 4 contain the numeric results of our empirical evaluation of the different storage formats for natively stored XML documents