An Adaptive Storage Manager for XML Documents

Karsten Schmidt and Theo Härder[1]

AG DBIS, Department of Computer Science
University of Kaiserslautern, Germany

Abstract. Effective and efficient management and manipulation of XML documents requires stable decisions at the time a document enters the XML DBMS to provide for storage structures which are adjusted to the document characteristics and which reflect the future processing needs. While some of the critical parameters require a kind of pre-specification, others can be determined by pre-analysis or sampling of the incoming document or by just making experience-driven "educated guesses". Important parameters are related to node labeling, path synopsis, document container layout, and indexing. In this paper, we discuss approaches to achieve an adaptive behavior of the storage manager to provide tailor-made native XML storage structures to the extent possible.

1 Introduction

An XML storage manager has to consider two cases: a document sent by a client arrives at a time—the so-called block mode, i.e., the document can be pre-parsed and analyzed before a suitable storage structure is chosen—or it arrives at the DBMS interface in stream mode where fragments present, because of their size, have to be allocated in a suitable storage structure on disk before the entire "streamed" document is available for the DBMS. In the latter case, the storage manager must decide—based on imprecise structural information—on the storage structure to be chosen and, at best, can make some educated guesses based on context or sampling information.

So far, observed from a bird's eye view, XML research primarily focuses on the management of a few isolated documents which are typically very large (up to several GByte). Frequently cited examples are available from [13]. In many cases [8, 9], storage structures are optimized for such situations and indexing schemes only support searching (say, based on XPath predicates) within a single document. Industry research and development, however, tells us [1] that, as the real need, DBMS have to manage large numbers of small to medium sized XML documents (typically less than a few MByte). This opposite view concerning size and usage of XML documents is currently captured by some efforts towards the design of an XML Database Benchmark [16]. In such cases, XML indexes most of all serve to filter documents (context) which are searched or scanned in a second step. Of course, all these cases can be handled by default assumptions concerning the characteristics of the final document stored. However, the great variety of possible document param-

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

eters will often lead to suboptimal or even bad solutions. For this reason, it is highly advisable to equip the storage manager with the ability to select reasonable or even optimal storage structures for the physical representation of documents.

To identify suitable storage structures for XML documents, we describe the most important concepts and their critical issues in Section 2, before we sketch the tasks needed to make XML document storage adaptive in Section 3. Some indicative measurements on well-known sample documents support our approach, before we wrap up with conclusions

2 Essential Concepts for the Storage Manager

Currently, we are upgrading XTC [7], our native XML prototype DBMS, step by step towards enhanced adaptivity. As far as physical document handling is concerned, the abilities of our *adaptive storage manager* (ASM, for short) primarily rest on the following concepts.

2.1 Node Labeling

As we have learned from many experiments and benchmark runs [6], the node labeling mechanism plays an essential role for storage consumption and efficient support of navigational and declarative query processing. Moreover, it is key for the flexibility and performance of the entire internal system behavior. We recommend the use of prefix-based labeling schemes as sketched in the sample document in Figure 1a. As its most prominent property, this labeling class equips each node label with its parent node label as a prefix. So far, a number of roughly equivalent prefix-based schemes are proposed in the literature. They not only support all XPath axis operations and hierarchical locking schemes (because all nodes in the ancestor path can be easily derived from a node label), but also dynamic insertions without re-labeling. DeweyIDs [6], as our specific labeling mechanism, enable in addition specialized attribute node mapping and—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. Because any prefix-based scheme such as OrdPaths [14], DeweyIDs, or DLNs[3] is appropriate for our storage manager design, we use the acronym SPLIDs (Stable Path Labeling IDentifiers), as a synonym for all of them.

In Figure 1a, we have applied the minimum "gap" with *dist=2* which is only preferable when insertions of subtrees are rare events. Hence, the insertion of a subtitle node between 1.3.3 and 1.3.5 would immediately cause an overflow and would have been handled by a node label 1.3.4.3. An *even* number indicates an overflow in the labeling mechanism; it always preserves the document order and enables direct comparability of SPLIDs.

An appropriate *dist* parameter cannot be determined while a document is initially stored. Its selection requires "future knowledge" concerning the frequency of expected updates on the document and, therefore, needs some hints from the user. Adaptivity of the storage

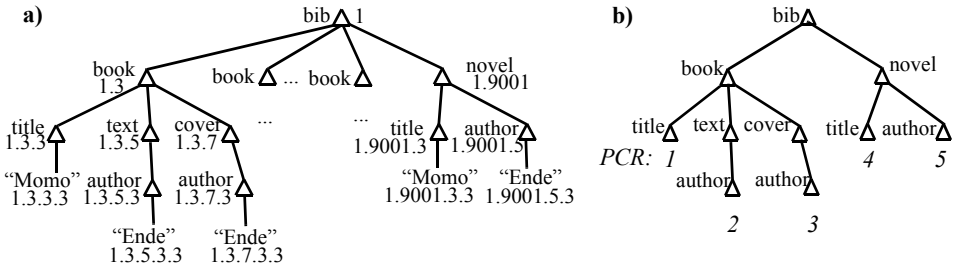

Figure 1 XML document (cut-out) and related path synopsis

manger is confined to observing "label overflows" which could trigger a re-labeling with a more appropriate *dist* size. Pre-analysis or sampling of the document, however, could reveal characteristics of the structure, average depth, distribution of nodes per level, etc. Such information is useful to find optimal encodings for the mapping of the divisions.

Due to the large variance of XML documents in number of levels and, even more, number of elements per level, we cannot design a (big enough) fixed-length storage scheme of SPLIDs. For the sake of space economy and flexibility, the storage scheme must be dynamic, variable, and effective to capture tall/flat trees with many/only a few nodes at a level and a huge fan-out per node or only some children. At the same time, it must be efficient in storage usage, encoding/decoding, and value comparison at the bit/byte level.

Huffman codes[2] can be used as a general mechanism to serve such requirements. For example, the i-th division value can be represented as a pair C_i | O_i where C_i is a prefix-free code used to assign a length value to O_i via a mapping table (or an equivalent Huffman tree). Table 1 illustrates a particular mapping H1 of variable length division values where, in addition, a byte alignment is achieved for each individual division. Because the codes can be freely chosen—regarding the definition of a Huffman tree and a rule preserving comparability—and the assignment of length values in column L_i is independent from them, tailored mappings can be derived for a document. Even in the encoded form, comparison of two SPLIDs or prefixes of them works at a level basis (and decides according to the document order). Therefore, we could refine our encoding mechanism and even choose Huffman encodings per level which could be adjusted to the node distribution of these levels. While such an optimization may save some space at the low percentage range, but may contribute to the implementation complexity, some heuristic encoding rules may be more effective. Frequently, large sets of nodes only occur at levels close to the document root, whereas the fan-out deeper in the tree is typically limited to a few and often to a single node. Here, pre-analysis or sampling

Table 1 H1: Assigning codes to L_i fields

code	L_i	value range of O_i
0	7	1–127
10	14	128–16,511
110	21	16,512–2,113,663
1110	28	2,113,664–270,549,119
1111	36	270,549,120 – ~2^{37}

[2] Despite the claims in [10], so-called quaternary codes or, more generally, use of separators [6] cannot provide fast bit-level comparisons and fail to support direct byte-level comparisons of encoded SPLIDs.

could deliver the level down to which, for example, H1 is applied (and a *dist* parameter is used to provide for gaps), while at deeper levels the most compact form of encoding is chosen, e.g., H2 with code 0 and L_i=3 allows the representation of values 1 – 7 with 4 bits.

Having selected an encoding scheme, continuous self-optimization with feedback learning may result in better adjusted parameters. That is, all documents added to an existing collection in a specific domain and, therefore, implying similar storage and processing properties, may improve encoding gains on future documents allocated in this collection.

2.2 Path Synopsis

All paths from the root to the leaves having the same sequence of element/attribute names form a path class. All path classes of an XML document can be captured in a typically small main-memory data structure called path synopsis. Besides keeping statistical data, it is used to detect path patterns, structure characteristics, or to recommend the creation of indexes. Moreover, a given synopsis can be compared to existing document synopses to find documents with similar structure. Such a path synopsis is illustrated for our sample document in Figure 1b.

An important use, the path synopsis enables the derivation of the entire leaf-to-root path of a content node. For example, when a value in a content index —whose unique position in the document is identified by its DeweyID—is associated with a reference to its path class, it is easy to reconstruct the specific instance of the path class it belongs to. By numbering the path classes in the path synopsis, we achieve an effective path class reference (PCR) serving as a path class encoding. Even an index reference via a DeweyID to a structure node (attribute/element) allows the reconstruction of the referenced node's ancestor path, when a PCR added to the index reference. This usage of the path synopsis indicates its central role in all structural references and operations. To increase its flexibility, it should provide indexed access via PCRs and hash access using leaf node names.

Our path synopsis is restricted to tree structures and does not capture loop-enabled graphs like XSeed [17]. In general, graphs allowing for an economical representation of path recursions, consume more space than an equivalent tree structure, both carrying various kinds of cardinality information.

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 of a node when its DeweyID 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 DeweyID use, it embodies our basic implementation framework to satisfy the above demands effi-

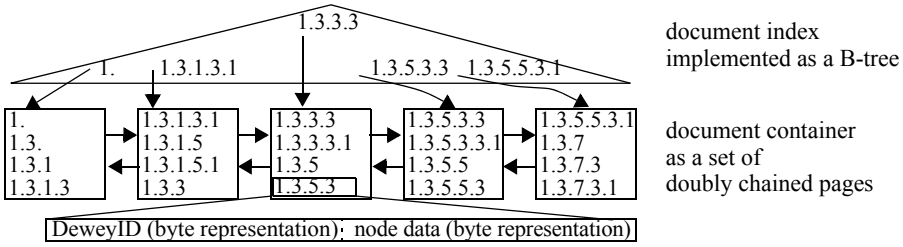

Figure 2 Document store with a B-tree and container pages

ciently. In XTC, this base structure comes with a variety of options [7] concerning use of vocabularies, materialized or referenced storage of content (in leaf nodes), and, most important, prefix-compressed DeweyIDs. As illustrated in Figure 2, the sequence of DeweyIDs in document order lends itself to prefix compression and, indeed, we received impressive results in numerous empirical experiments [6].

2.4 Indexing Structure and Content

So far, XML indexing literature can be classified into content [4], path [5] or hybrid indexes [9]. Typically, these indexes deliver (range-based) node labels for index matches, which then had to be resolved or verified on the document structure. For example, separate matches on a content index for the value part and a path index for the structure part of a content-and-structure (CAS) query had to be algorithmically checked whether pairs of matches can be verified on the XML document (for example, by using structural joins). An improved hybrid index, FLUX [9] carries a path signature with each index reference, which is constructed as a Bloom filter [2]. When a value qualifies, the related signature is assumed to deliver almost precise path information. However, because false positives may occur, it has to be checked against existing paths in the document which can make the evaluation of XML path predicates expensive.

The combined use of DeweyIDs and PCRs dramatically improves index expressiveness and evaluation. Using a B*-tree, we can develop various content indexes and get—by adding PCRs to the DeweyID references—the corresponding path indexes for free. First of all, we could design unique indexes, e.g,, for values of ID attributes or values for the path class /bib/novel/title (see Figure 1b). In the latter case, the index entry (Momo, 1.900.3.3, 4) is sufficient to reconstruct the entire path class instance. Depending on index selectivity and query workload, we could benefit from collective indexes, e.g., //author[char], within a single B*-tree structure. Here, an index entry (Ende, (1.3.5.3.3, 2), (1.3.7.3.3, 3), ..., (1.9001.5.3, 5)) refers to all appearances of "Ende" as author, but, together with the path synopsis, the correct path class instances can be derived. Even more, attribute or element indexes can be designed according to the same principle. So, all occurrences of element name "author" could be found via index entry (author, (1.3.5.3, 2), (1.3.7.3, 3), ..., (1.9001.5, 5)). Here, the PCRs help to reconstruct the paths from inner structure nodes up to the root without document access. Such typically external storage accesses are confined

Table 2 Characteristics of XML documents considered

doc name	description	size in Mbytes	# elem. & attr. nodes	# text nodes	# vocab. names	# path classes	max. depth	avg. depth
line-item	LineItems from TPC-H benchmark	32.3	1,022,977	962,801	19	17	4	3.45
uni-prot	Universal protein resource	1,820.0	81,983,492	53,502,972	89	121	7	4.53
dblp	Computer science index	330.0	9,070,558	8,345,289	41	153	7	3.39
psd-7003	DB of protein sequences	716.0	22,596,465	17,245,756	70	76	8	5.68
nasa	Astronomical data	25.8	532,967	359,993	70	73	9	6.08
tree-bank	English records of Wall Street Journal	86.1	2,437,667	1,391,845	251	220,894	37	8.44

to cases where additional information is needed from the document representation (see Figure 1a). Details of this novel approach to XML document indexes are discussed in [11].

While ASM can perform index creation in parallel to the document storage, the selection of appropriate indexes for an arriving document needs specification by the user. Although a desired property of self-tuning, ASM would be able to automatically create indexes only after a period of workload monitoring and self-observation. Furthermore, approaches concerning query-driven or even predictive index creation would need a sort of probabilistic usage model to estimate future costs and benefits of additional indexes

3 Building Tailor-Made Storage Structures

As already mentioned, ASM has to distinguish two cases: block-mode and stream-mode arrival of (large) XML documents. In any case, an as thorough as possible analysis of the incoming document seems mandatory to accomplish highly optimized storage representations for XML documents. To give an impression of the analysis task needed, we have assembled in Table 2 essential characteristics for a representative subset of the test documents considered [13]. After document analysis, ASM selects a configuration from a number of predefined models, before it starts allocating and filling the corresponding storage objects.

3.1 Analysis Phase

At first, ASM scans the available fragment of the document, in case of block-mode arrival often the entire document, and collects significant document parameters to adjust the initial default parameter setting. Statistical data include number of nodes (i.e., element, attribute, and text nodes), maximum depth and average depth, various fan-out ratios, number of distinct element names, as well as number of distinct paths per path class. Furthermore, the size of text nodes helps to adjust some storage parameters and the size of the document store can be estimated from such statistical data. As discussed in Section 2.1, ASM tries to

find an optimal Huffman encoding for the divisions, possibly restricted to the "critical" document tree levels with large sets of nodes under a common parent.
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 little main-memory data structure (for typically a few hundred names). Therefore, while scanning the document, its vocabulary is incrementally built. Furthermore, the path synopsis containing all path classes and PCRs is derived. Both data structures can be completed on block-mode arrival, whereas a stream-mode document may leave at the end of the analysis phase fragmentary data structures to be completed in later phases.

Another problem is unused space in container pages which is crucial when text nodes have to be allocated. They are materialized in container pages up to a parameterized *max-val-size*. When the size exceeds *max-val-size*; the text is stored in referenced mode possibly divided in parts each stored into a single page and reachable via a reference from its home page, as illustrated in Figure 3. Hence, maximizing page utilization may be achieved by a document-dependent page size optimization. Regarding the document store as an index, these findings can be applied to additional indexes, too.

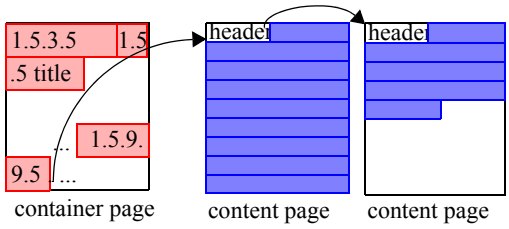

Figure 3 Inserting long records

3.2 Approximating Document Parameters by Sampling

While expensive, a full document analysis always delivers accurate parameters for the configuration phase. To check a conceivable reduction of the analysis effort, we ran some sampling experiments for the documents of Table 2. Sampling only allows to approximate important auxiliary structures and configuration parameters such as vocabulary, path synopsis, text length per element, fan-out (in upper levels), and document depth. In a number of sampling experiments, we have determined—for the parameters applicable—the ranges of the estimation errors to be expected. In Figure 4a, the graphical symbols depict the average estimation error per document, whereas the max/min of the error range correspond to estimations computed when a sampling buffer was filled with 1 resp. 50 MB. These results highlight one of the most fundamental problems in sampling small pieces of documents with heterogeneous or skewed structures. As indicated for *dblp* and *treebank* in Figure 4a, parameters such as vocabulary size and path class may cause the selection of an unfit representation model. When we start building the document with such "wrong guesses", we may get suboptimal structures or may be enforced to revise our design decision. In general, however, the parameters for max/avg depth, average text size, and fan-out are accurate and stable, even for tiny fractions of 1 MB samples. Hence, we can use stable parameters for decisions concerning DeweyID encoding and page size tuning.

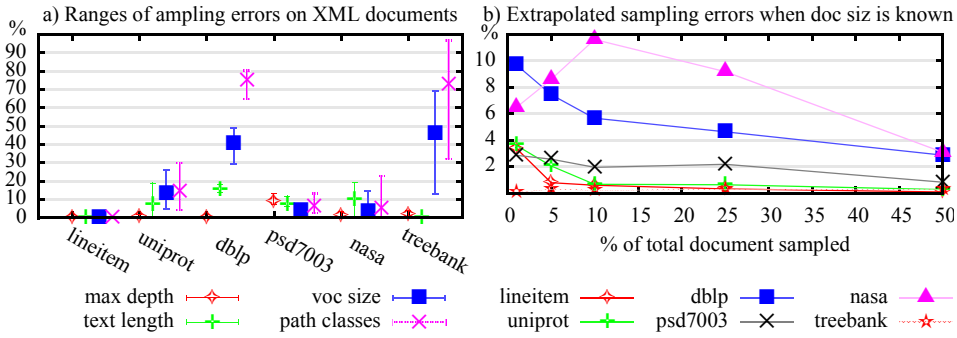

Figure 4 Relative estimation error of sampling

When sampling on block mode or stream mode documents, only its initial fragment can be exploited for reasonable parameter and structure estimations. Sampling inner parts would require jumps into the 'middle of the document' thereby losing location awareness and context information for determining levels, paths, and other structure parameters. Despite the auxiliary information about document size and structure for block-mode documents available, sampling proceeds the same way in both cases. Therefore, outliers for specific parameter values or a skewed document structure may lead for both arrival modes to wrong decisions; only reloading or dynamic reconfiguration may guarantee optimal configurations. Having complete documents and precise analysis available, the mismatch, however, is most likely lower.

Stream-mode documents necessarily enforce ASM to configure storage structures with less than perfect parameter knowledge, as characterized in Figure 4a. Because file size information is available for block-mode documents, extrapolation of some parameters using the size of the entire document is applicable. To show the precision of a sampling step instead of a full scan for size information (number of attribute/element/text nodes), Figure 4b exhibits the relative estimation errors for various sample sizes. 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 4b also shows that there exist "simply structured" documents where sampling delivers perfect knowledge of size parameters even using very small samples. In summary, sampling often delivers in the analysis phase accurate-enough parameters for ASM to plan the physical configuration of an XML document.

3.3 Configuration Phase

To provide a systematic framework for the configuration of XML document representations, we have introduced a number of storage models. Every document is classified into one of the following categories.

- Model 1: A (large) document is stored as a physical stand-alone tree structure in a set of container pages together with a document index according to the schema shown in Figure 2. Its path synopsis is kept in a separate structure and a number of user-specified content and/or element indexes is allocated to speed up declarative document access.
- Model 2: The storage structures of Model 1 are taken, but the layout of the container pages is optimized using an "elementless" representation of the XML document while preserving all document properties. Only the content, i.e., the values of the leaf nodes together with their DeweyIDs and PCRs, is stored in the container pages. The PCRs are used to derive the inner document structure on demand [11].
- Model 3: When a large number of small documents of a given domain, e.g. from Wikipedia, has to be stored, isolated Model 1&2 representations are very space consuming. By adding a new artificial document root, we may store such a collection as a single physical document according to the schema in Figure 2. Having a common path synopsis for this collection, the assignment of new domain-related documents may be done automatically via path synopsis matching. The use of combined indexes may save storage space and, partly due to locality of reference, accelerate the evaluation of queries.
- Model 4: Collections of (small and large) documents with (largely) homogeneous structures are physically stored together as subtrees which, in turn, may be indexed by a B*-tree. The assignment to such a collection is user-specified or can be supported by the path synopses to determine similarities among documents. Homogeneous documents by structure and/or by content may take advantage of common structure/content indexes and gain better storage utilization and query support.
- Model 5: Collections of documents of varying sizes and heterogeneous structures are used in a similar way as in Model 4. Typically, often indexes do not provide search support, but only a kind of filtering. Due to the documents' heterogeneity, query support and reduced storage space may be only achieved by combined indexes and vocabularies rather than by disjoint path synopses to be expected.

So far, we have implemented Models 1 to 3, whereas the remaining models are currently under construction. Their design is more complex, because of the content and structure heterogeneity of the documents and the challenge to achieve reasonable storage utilization with large numbers of small objects.

Note, the different models rather describe a rough, and may be incomplete, classification than a fixed assignment for every incoming document. Indeed inappropriate storage mappings for incoming documents may enforce a classification refinement, that is, further derivation of one or more models or a combination of them is conceivable. However, this is not true for Model 1 and Model 2; they are inherently different that their combination does not seem to be achievable.

3.4 Building Phase

When an appropriate model is identified for an incoming document (either automatically or with user assistance), our ASM, when allocating and filling the storage structures for the document, tries to do as much as possible in parallel. When filling the document con-

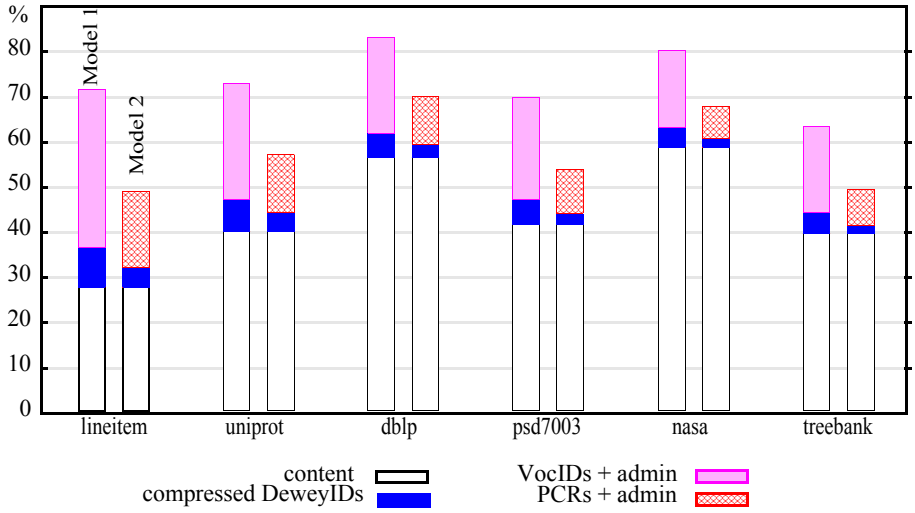

Figure 5 Storage consumption of XML documents

tainer left-to-right in document order, the document index creation can be optimized by a concurrent bottom-up construction. Normally, additional (pre-specified) indexes are built top-down, as the corresponding values and DeweyID references occur in the document being stored. Another option is to collect all indexed values and their node labels, sort them explicitly according to the various index orders, and build the indexes bottom-up.

Concerning the optimization and adaptation issues discussed in Sect 2, ASM has for each of the models quite a number of options to choose from. The models themselves rather reflect application scenarios which are partially overlapping. Especially, Model 1 and 2 are competing as candidates for large isolated documents for which we already performed extensive empirical evaluations. Due to space limitations, we can only focus on storage consumption and show what kind of improvement can be expected from them when compared to a standard approach. In all cases, the document nodes are stored as variable-length records including DeweyIDs, VocIDs, type descriptors, byte alignment, etc. Furthermore, we left the content (text nodes) uncompressed; text or character compression techniques can be orthogonally adopted for further optimization. The results in Figure 5 are normalized w.r.t. Standard using plain DeweyIDs and “long” VocIDs (2 bytes). Model 1 applies prefix compression to DeweyIDs and adjusts VocIDs (1 byte). Model 2 uses only stores the text nodes carrying prefix-compressed DeweyIDs and adjusted PCRs (only 1 byte, if applicable). Compared to Standard, Model 1 saves ~20% – ~40% and Model 2 reaches ~30% – ~50% when considering the entire document. If we regard the structural part only, the saving increases substantially to ~40% – ~61% and ~69% – ~86% for Model 1 and Model 2, respectively.

Model 2 assumes a reasonably small path synopsis available for all processing steps which is true for probably more than 90% of the XML documents [12]. Looking at the parameters

for depth and path classes in Table 2, *treebank*, however, can be characterized as such an exotic outlier, where only Model 1 should be applied.

Model 3 forms a single artificial tree for a collection of XML files by adding an extra level with a common root. It was applied to 527 small Wikipedia documents which we randomly selected from wikipedia.org and where the HTML tags were used as the XML vocabulary. Our tests proved its superiority compared to alternative storage models. Stored as isolated documents, this collection would have created 527 similar path synopses and vocabularies, not to mention the large number of tiny index structures, each of them only filled with a few values and references. Table 3 depicts the differences between Model 1 and Model 3 regarding page utilization and index pages as well as the DeweyID overhead of the added document root. Note, the overhead of the increased DeweyID length (an additional division for every node label) is completely absorbed due to prefix compression.

Table 3 Model 1 vs. Model 3

property	527 single documents	collection
avg. depth	6.29	7.29
max. depth	42	43
document index pages	4,751	2,957
element index pages	42,928	1,504
prefix compression	920 KB	930KB
unused space	180 MB	9.9MB

Finally, there exist no limitations for the coexistence of different models within the same container file. Because every container is divided into pages having a specific page header to describe how the page is accessed, even indexes and BLOB pages may coexist within a single container file. The specific configuration of document to be stored is administrated using a master (XML) document which serves as the system catalog.

4 Conclusions

In this paper, we primarily discussed important concepts needed to obtain optimal and tailor-made storage structures for XML documents. For block-mode and stream-mode document arrival, we sketched the kind of analysis, the selection of a storage model, and finally the creation of the stored document including auxiliary data structures. Preliminary performance measures indicate the potential storage saving and operational gain using our concepts of adaptivity.

Because flexibility is often the main reason to adopt an XML DBMS, we have only considered XML documents with integrated schema information. However, increased integrity requirements may dictate in the near future the support of XML schema and explicit integrity checking against its schema when a document is stored or modified. In general, this implies for the DBMS substantial effort to accomplish integrity control of multiple or evolving schemas or of schemas that include extensibility points [1]. As observed in [15], handling such additional flexibility may become a killer application for XML databases.

References

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

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

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

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

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

[6] 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)

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

[8] 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)

[9] 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)

[10] Li, Ch., Ling, T. W., and Hu, M. Efficient Updates in Dynamic XML Data: From Binary String to Quaternary String. *VLDB Journal:* 16 (2007)

[11] Mathis, C., Härder, T., and Schmidt, K.: Storing and Indexing XML Documents Upside Down, submitted.

[12] 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

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

[14] 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*: 903-908 (2004)

[15] Sedlar, E. Managing Structure in Bits & Pieces: The Killer Use Case for XML. *Proc. SIGMOD*: 818-821 (2005)

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

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