Tailor-Made Native XML Storage Structures

Karsten Schmidt and Theo Härder[1]

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

Abstract. Automatically choosing suitable native storage structures for XML documents arriving at the XML DBMS is a challenging task for its storage manager. While some of the critical parameters require pre-specification, others can be determined by pre-analysis or sampling of the incoming document or by just making experience-driven "educated guesses". 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

The XML data model allows for a great deal of modeling flexibility with which order, optional and multi-valued concepts, as well as deeply nested relationships can be captured. Therefore, the resulting tree structures of natively represented XML documents exhibit large variations in breadth and depth. Furthermore, XML documents may be data-centric having relatively little 'content' stored as short text values or they may be document-centric where a single content node may contain huge portions of text, e.g., an entire book in a digital library.

For these reasons, it is highly advisable to equip the storage manager with the ability to select reasonable or even optimal storage structures for documents. Furthermore, two cases have to be considered: 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 storage structure is chosen—or it arrives at the DBMS interface in stream mode where fragments present, due to 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.

To identify suitable XML storage structures, we describe the most important concepts and their critical issues in Section 2. In Section 3, we explore methods for content compression, before we show in Section 4 how to collect storage parameters in an analysis phase. These concepts and methods are applied and empirically evaluated on tailor-made storage structures for well-known sample documents in Section 5, before we wrap up with conclusions.

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

Y. Ioannidis, B. Novikov and B. Rachev (Eds.): Local Proceedings of ADBIS 2007, pp. 96-106

2 Essential Concepts for the Storage Manager

Currently, we upgrade XTC [5], our native XML DBMS (XDBMS for short), step by step towards enhanced adaptivity. As far as physical document handling is concerned, the abilities of our *adaptive storage manager* (ASM) primarily rest on a few important concepts.

First of all, when storing and processing XML documents in a native way, the node labeling scheme used plays a critical role for many XDBMS tasks. Internal evaluation of navigational (DOM) or declarative requests (XPath, XQuery) as well as concurrency control on large tree structures should be effectively supported to achieve efficient transaction-protected processing or cooperative use of XML documents under a variety of XML language models. Furthermore, indexed access can also take advantage of a suitable node labeling technique [8]. When carefully optimized, storage consumption of documents and related index structures can be limited to an acceptable level. In summary, it is key for the flexibility and performance of the entire internal system behavior. These observations were confirmed by a systematic evaluation with many experiments and benchmark runs [4, 6]. As result of our learning process, we recommend prefix-based labeling schemes. Because any prefix-based scheme such as OrdPaths [12], DLNs [1], or DeweyIDs [4] is appropriate for our document storage, we use SPLIDs (Stable Path Labeling IDentifiers) as synonym for all of them.

Together with prefix-based labels, a so-called path synopsis can provide substantial processing and compression support in an XDBMS. In an XML document, the structure part (element/attribute nodes) typically carries a large amount of redundancy due to the verbose structural description; this is also true when the element/attribute names are replaced by VocIDs used from a vocabulary. The often huge degree of path repetitions (e.g., /bib/book/chapter/author) is not reduced by this standard format of XML document storage. Therefore, we try to get rid of this redundancy while preserving all operational properties of the document representation. 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 [8]. 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.

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 SPLID—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 SPLID 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 [8].

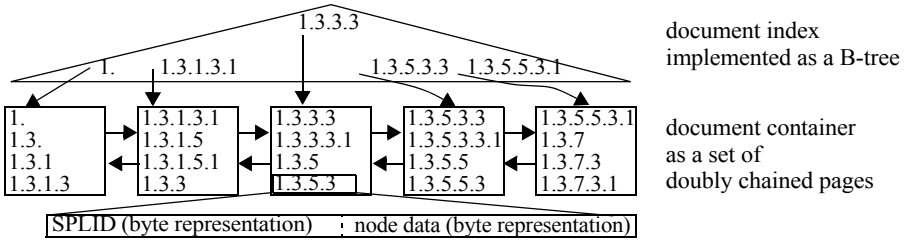

Figure 1 Document store with a B-tree and container pages

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 1, the *document index* enables direct access of 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 use, it embodies our basic implementation framework to satisfy the above demands efficiently. In XTC, this base structure comes with a variety of options [5] concerning use of vocabularies, materialized or referenced storage of content (in leaf nodes), and, most important, prefix-compressed SPLIDs. As illustrated in Figure 1, the sequence of SPLIDs in document order lends itself to prefix compression and, indeed, we received impressive results in numerous empirical experiments [4].

A final issue to be considered when storing XML documents is the benefit of saving storage space by applying suitable compression techniques. Two potential areas where compression can be successfully used are 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 [7]. Furthermore, transactional modification applied to XML documents prevents *block-based compression* used by PPM algorithms [11]. 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. In our compression study, we exclusively focus on single nodes and their data stemming either from text content or attribute values and use character-based compression algorithms.

3 Empirical Results for XML Content Compression

To provide some indicative results for the storage of XML documents in various formats, we have applied a number of empirical tests in the context of our ASM where we used—to facilitate comparison—the frequently evaluated set of test documents taken from [10].

Table 1 Characteristics of XML documents considered

doc name	description	size in Mbytes	# element nodes	# attribute nodes	# content nodes	# vocab. names	# path classes	avg. depth
uni-prot	Universal protein resource	1,820.8	36,195,456	45,788,036	53,502,972	89	121	4.53
tree-bank	English records of Wall Street Journal	89.5	2,437,666	1	1,391,846	251	220,894	8.44
psd-7003	DB of protein sequences	716.0	21,305,818	1,290,647	17,245,756	70	76	5.68
line-item	LineItems from TPC-H benchmark	32.3	1,022,976	1	962,801	19	17	3.45
dblp	Computer science index	330.2	7,529,465	1,541,093	8,345,289	41	153	3.39
nasa	Astronomical data	25.8	476,646	56,317	371,593	70	73	6.08

For a representative subset, Table 1 assembles essential characteristics of the documents in the "gross" format, i.e., as 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 by the ASM. Thus, we restrict ourselves to tests using four encoding methods:

- *Fixed Huffman (M1)*: During a preanalysis run, the character frequencies were collected on a document basis, for which a Huffman tree was then constructed. To adjust for later document modifications, encoding was provided for all 256 possible characters.
- *Flexible Choice (M2)*: Depending on the characteristics of a content node, it is either encoded by a document-wide Fixed Huffman or a tailor-made node-wide Huffman. If chosen, the tailored Huffman tree (typically <40 nodes) is stored together with the encoded node's content.
- *Selective Encoding (M3)*: This method optimizes the runtime of M2 by calculating and applying tailored Huffmans only to longer text/attribute values; smaller text values were encoded by the Fixed Huffman of the document.
- *Domain Encoding (M4)*: Especially applicable to small documents, an overall encoding constructed from a domain-related character distribution base is used to reduce space requirements and to speed up compression time.

M1 uniformly encodes all content nodes of the document, even if compression leads to larger encoded equivalents, for example in case of short values. To have more flexible encoding options by selecting among different algorithms, one or more decision bits are needed for each record to indicate its compression state. However, the savings for multiple encodings may be compensated by such decision bits as shown by our experiments, because often byte alignment consumed additional space. Moreover, each compressed bit string had to be aligned to the next byte boundary to observe the record's byte format. With this increased flexibility, M2 computes a tailored Huffman for every content where applicable. However, this leads to increased computation time and one decision bit for every record to distinguish the type of compression (fixed or tailored). For this reason, M3 tries to reduce these extensive computation times by a threshold heuristically chosen to decide whether or not a tailored Huffman has to be derived. As expected, this leads to shorter

Table 2 Effectiveness of character-based text compression on XML documents

doc name	'Standard' document size in MB	content size in MB	avg. value size per text node	avg. value size per attr. node	avg. value size per content node	compression M1	compression M2	compression M3	time consumption M1
uniprot	1685.0	669.0	37.5	8.3	12.5	76.8%	**76.3**%	76.5%	93.8%
treebank	86.1	33.5	24.0	8	24.0	**75.8**%	76.4%	76.4%	99.9%
psd7003	722.0	293.0	17.9	5.5	17.0	74.0%	**70.7**%	**70.7**%	93.5%
lineitem	22.7	6.2	6.5	8	6.5	**70.8**%	73.9%	73.9%	93.4%
dblp	310,7	174.0	22.1	15.5	20.9	**69.9**%	70.4%	70.4%	94.1%
nasa	21.2	12.4	36.8	14.1	33.4	64.6%	**64.4**%	64.6%	84.6%

compression times thereby retaining acceptable compression ratios. Method M4 applies this idea to several (small) documents to avoid the creation of a per document encoding table which may be larger than the document to be encoded (see Section 5.2).

In the following, all results are compared against *Standard* format where the document nodes are stored as variable-length records including SPLIDs, VocIDs, type descriptors, byte alignment, etc. Our compression experiments only considered content nodes with text or attribute values. To give some coarse hints about the potential compression gain, we have captured the avg. value sizes of content (text and attribute) nodes[2] in Table 2 where our experiments are summarized: M1 is fast and delivers considerable compression ratios varying from ~25% to ~35% on all documents. Note, M1 compression reduces the overall storage time up to ~15% compared to *Standard*, because less I/O is needed to store the corresponding document on disk. M2 and M3 consume substantially more computing time for the compression (not shown in Table 2). Furthermore, they exhibit no or only tiny compression improvements on the kind of documents in our reference collection. For this reason, M1 is our recommended and preferred compression method for large documents.

Looking at the columns 'avg. value length per text/attribute/content node', most of the documents in Table 2 can be classified as "data centric" where a cheap M1-type algorithm seems to be an adequate compression solution. In contrast, so-called "document-centric" XML structures would embody a greater potential for compression which could be exploited by the M2/M3-type algorithms, because larger portions of text fill the content nodes. In our collection of reference documents, only nasa has 'weak' document-centric properties. In truly document-centric XML structures, substantially greater compression gains may be anticipated by applying tailored compression schemes [7, 13].

4 Collecting Document Parameters

Whether or not text compression has to be applied, it is typically pre-specified and may depend on update frequencies as well as availability or cost of resources (storage space, CPU). If compression is desired, ASM can easily decide on a suitable method which may

[2] The effectiveness of compression relies on many factors, not only content size. Although *lineitem* has only small content nodes, they primarily contain digits. This 'narrow' alphabet allows relatively better compression rates as *treebank* containing larger enciphered texts nodes resulting in a 'broader' alphabet.

be selected based on domain-dependent characteristics. For further parameters concerning physical document representation, some dynamic approximations can be derived. Therefore, two cases have to be distinguished: 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 refer to Table 1.

4.1 Dynamic Document Analysis

To enable adaptive decisions, ASM may scans the available fragment of the document, in case of block-mode arrival often the entire document, and collect significant document parameters to adjust an initial default parameter setting. Statistical data may include—besides the degree of document-centric compression behavior—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.

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. 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.

4.2 Approximating Document Parameters by Sampling

To check a conceivable reduction of the analysis effort, we ran some sampling experiments for the documents of Table 1. Sampling only allows to approximate important auxiliary structures and configuration parameters such as vocabulary, path synopsis, content size per node, 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 2a, 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

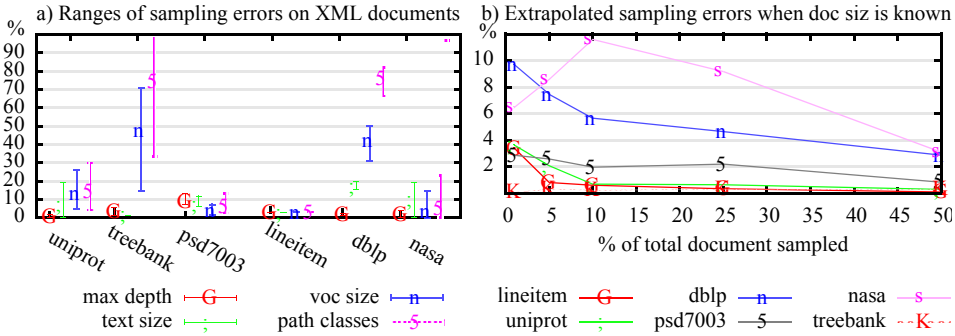

Figure 2 Relative estimation error of sampling

skewed structures. As indicated for *dblp* and *treebank* in Figure 2a, 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 SPLID encoding and page size tuning. Of course, a Fixed Huffman can be derived by sampling, too. Because character distribution and their frequencies typically are domain dependent, nearly optimal encodings can be expected. Stream-mode documents necessarily enforce ASM to configure storage structures with less than perfect parameter knowledge, as characterized in Figure 2a. 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 2b 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 2b also shows that there exist "simply structured" documents where sampling delivers perfect knowledge of size parameters even using very small samples. However, *nasa* exemplifies what happens when sampling and extrapolating unbalanced documents, its error is bounded to ~12%. In summary, sampling often delivers accurate-enough parameters for ASM to plan the physical configuration of an XML document.

5 Building Tailor-Made Storage Structures

So far, we have outlined the essential concepts and the critical parameters for 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 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]. For the support of index access and vir-

tualization of the document index, a path synopsis is provided together with the document container, where applicable. Here, we primarily want to illustrate how much storage consumption can be reduced by deriving configurations tailored to the parameters of the documents. An important optimization is the use of tailored SPLIDs, which was applied in all experiments. Note, in all cases neither set-oriented operations (e.g., XQuery) nor node-oriented navigational operations (e.g., DOM) are restrained or impeded.

5.1 Storing Single Documents

Single large documents are assigned to separate storage structures. The obvious choice denoted as Model 1 is the physical representation of all structure and content nodes in a set of container pages together with a document index. The corresponding schema shown in Figure 1 also reveals that the SPLIDs occur in sort order and that they lend themselves to a very effective prefix compression [4].

In most cases, the set of structure nodes contains a dramatic degree of replication within the path instances of a given path class. Therefore, applying SPLIDs for node labeling in the document, each path instance the node belongs to can be efficiently derived using the path synopsis. This observation leads to the opportunity to virtualize the document structure. For this purpose, we design Model 2 as a modification of Model 1 by optimizing the layout of the container pages in Figure 1 using an "elementless" XML representation while preserving all document properties. Only the values of leaf nodes together with their SPLIDs and PCRs, is stored in the container pages. The PCRs are used to derive the inner document structure on demand [8].

At first, we focus on the minimization of the structure part only and leave the content nodes uncompressed. The results in Figure 3 are normalized w.r.t. the *Standard* format using plain SPLIDs and "long" VocIDs (2 bytes). Model 1 stores all structure and content nodes with prefix-compressed SPLIDs and VocIDs (1 byte). Model 2 only stores the content nodes carrying prefix-compressed SPLIDs 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.

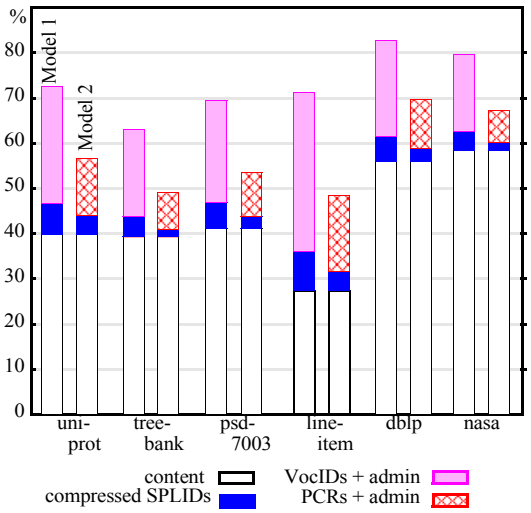

Figure 3 Storage consumption of XML documents

The second group of experiments explores the potential of content compression (discussed in Section 3). Due to space limitations we omit a detailed discussion of the results and restrict ourselves to a summary. Using the simple compression method M1 on our reference collection of documents, the relative share of content nodes is reduced from ~28% – ~59% to ~19% – ~37% as compared to *Standard*. Concerning the entire document, compressed Model 1 reduces the storage consumption to less than 65%, whereas compressed Model 2 reaches less than 50% in all cases. 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 [9]. Looking at the parameter for path classes in Table 1, *treebank*, however, can be characterized as such an exotic outlier (maximum depth of 37), where only Model 1 should be applied.

5.2 Collections of Documents

To handle homogeneous documents that are somehow inter-related, we introduce collection assembling documents which embody similarities w.r.t. structure, domain affiliation, and vocabularies. Model 1 and 2 disclose tremendous drawbacks storing a large number of small documents in separate container pages. In addition, separately stored auxiliary structures would consume an enormous number of weakly filled pages. To avoid such low storage occupancy, small and large documents are physically stored together as subtrees under a virtual root node in a collection which, in turn, may be indexed by a B*-tree. Homogeneous documents by structure and/or by content may take advantage of common structure/content indexes and gain better storage utilization and query support. Our collection idea is endorsed by an analysis of typical documents and varieties in the recently proposed TPoX [14] benchmark which serves as an illustration for our concepts. In general, our collection approach attempts to exploit Model 2 ("elementless") storage and to process distorted documents in an appropriate way. For this reason, we apply, when appropriate, multiple path synopses within a single collection definition, as shown in Figure 4a. Every document, stored according to Model 2, is assigned to the closest matching path synopsis, to a new path synopsis, or, when Model 2 is not smoothly applicable, it is stored according to Model 1, possibly without a path synopsis[3]. Such a dynamic assignment is sketched in Figure 4b while referring to the various TPoX structures. This XML benchmark proposal consists of three different groups of documents: one for *Account* data, one for *Order* data and one for *Security* attachments of customers. Designed for extreme scalability up to PBytes, the smallest benchmark configuration already provides more than 80,000 documents having plain sizes between 1 and 18 KBytes. Even without an existing XML schema, an allocation of new documents would be possible—because of the structural simplicity of these documents—only by path synopsis matching. In the path synopses in Figure 4, simple top-down comparison suffices to determine—using an adequate similarity threshold—the collection membership. For the TPoX documents, the structural information within each group is confined to a maximum of 139 path classes and to 139 VocIDs to en-

[3] Note, a path synopsis is mandatory for elementless structures to regain the internal document structure. It may be helpful to optimize specific operations for documents represented by Model 1, too.

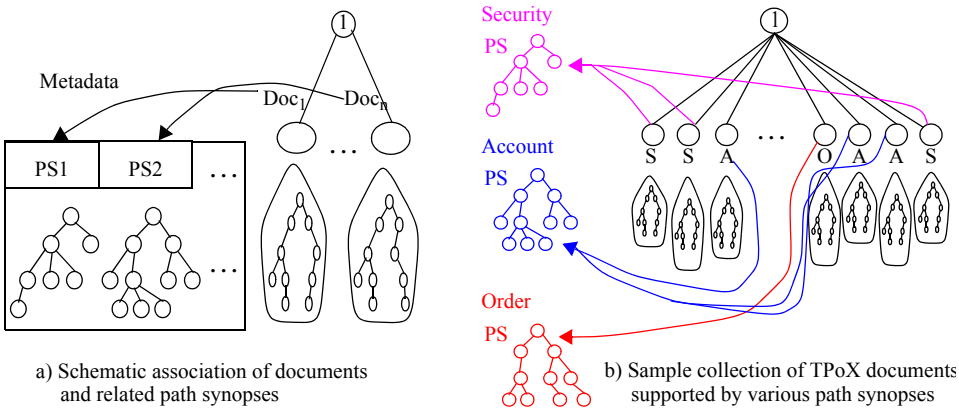

Figure 4 Representing homogeneous collection of documents using path synopses (PS)

code the documents' node names. Including a number of common structural elements, the resulting TPoX collection is confined to 295 distinct path classes and a vocabulary of 276 elements. Obviously, such database-driven collections remain quite stable, because schema evolution may only occur in exceptional cases and, thus, the path synopsis will preserve its reasonably small size. The mapping of path synopses to documents and vice versa can be handled by a simple lookup table in the system catalog.

By combining the physical storage of a collection, it is an adequate design decision to use indexes for holistic query support over all documents. Typically, a collection forms a uniform database and queries refer to all of its documents at the same time. Therefore, if a user would enforce a document with totally incompatible characteristics into an existing collection, existing path synopsis and index accuracy would be inflated by structural information alien to the collection's infrastructure. Thus, our ASM in XTC always tries to determine a suitable existing collection by path synopsis matching and vocabulary comparison. If all matches violate a reasonable threshold, an incoming document is considered *heterogeneous* 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 according to Model 1 or 2.

6 Conclusions

In this paper, we primarily discussed important concepts needed to obtain optimal and tailor-made storage structures for XML documents. Furthermore, we elaborated the potential benefit of compression methods applied to content nodes. For block-mode and stream-mode document arrival, we sketched the kind of analysis 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 such concepts of adaptivity.

What does this saving achieved by structure encoding and content compression mean? For example, assume *uniprot*: the gross document (in text format) arriving at the DBMS has 1820 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 the compressed Model 1 and Model 2 ~1060 and ~825 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 provide any problem. Content-and-structure (CAS) queries are particularly efficient, because our way of processing the queries [8] can often avoid expensive structural joins or twig evaluation 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 [5].

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

[4] 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)

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

[6] Haustein, M. P., Härder, T., Mathis, C., and Wagner, M. DeweyIDs - The Key to Fine-Grained Management of XML Documents, in: Proc. 20th Brasilian Symposium on Databases, Oct. 2005, pp. 85-99.

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

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

[9] 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

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

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

[12] 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)

[13] Toman, V. Syntactical Compression of XML Data. caise04dc.idi.ntnu.no/CRC_CaiseDC/toman.pdf

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

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