First publ. in: Dataspace: the final frontier: 6th British National Conference on Databases, BNCOD 26, Birmingham, UK, July 7 - 9, 2009; proceedings / Alan P. Sexton (ed.). (= LNCS ; 5588) Berlin: Springer, 2009, pp. 158-170

Compacting XML Structures Using a Dynamic Labeling Scheme

Ramez Alkhatib and Marc H. Scholl

University of Konstanz, Box D 188 78457 Konstanz, Germany
{Ramez.Alkhatib,Marc.Scholl}@uni-konstanz.de

Abstract. Due to the growing popularity of XML as a data exchange and storage format, the need to develop efficient techniques for storing and querying XML documents has emerged. A common approach to achieve this is to use labeling techniques. However, their main problem is that they either do not support updating XML data dynamically or impose huge storage requirements. On the other hand, with the verbosity and redundancy problem of XML, which can lead to increased cost for processing XML documents, compaction of XML documents has become an increasingly important research issue. In this paper, we propose an approach called CXDLS combining the strengths of both, labeling and compaction techniques. Our approach exploits repetitive consecutive subtrees and tags for compacting the structure of XML documents by taking advantage of the ORDPATH labeling scheme. In addition it stores the compacted structure and the data values separately. Using our proposed approach, it is possible to support efficient query and update processing on compacted XML documents and to reduce storage space dramatically. Results of a comprehensive performance study are provided to show the advantages of CXDLS.

1 Introduction

XML [1] is becoming widely used for data exchange and manipulation. As a consequence, an increasing number of XML documents need to be managed. Therefore different languages have been proposed to query data from XML documents, among them XQuery and XPath [1] which are currently the popular XML query languages. They are both strongly typed, declarative, and use path expressions to traverse XML data. Therefore, efficiently processing path expressions plays an important role in XML query evaluation. There have been many proposals to manage XML documents. However, two common strategies are available to provide robust storage and efficient query processing. The first is based on labeling schemes that are widely used in XML query processing. These numbers represent the relationships between nodes, playing a crucial role in efficient query processing. However, some labeling schemes [2,3] have the problem that they either do not support updates to XML documents or need huge storage. The second strategy tries to reduce the size of XML documents through compaction techniques [4,5,6] that can improve the query speed by saving scan

time. Unfortunately, the problem in such methods is that they are not able to discover all the redundancy present in the structure of XML, and thus often do not yield the best compression result. Another significant drawback of such methods is that they do not support direct updates or direct querying, i.e. querying a compressed document without decompressing it. Our approach bridges the gaps between labeling schemes and compression technology to bring a solution for management of XML documents that yield performance better than using labeling and compression independently. In our work, we focus on the separation of content from structure of an XML document. It is coupled with an effective method for XML compacting based on the exploitation of the similarity of consecutive tags and subtrees in the structure of the XML documents. Also we use the ORDPATH labeling scheme [7] for gathering sufficient structural information from the compacted XML document. We then store the compacted XML in a way that allows fast access and efficient processing over secondary storage. The main contributions of this paper can be summarized as follows:

- CXDLS avoids unnecessary scans of structures or irrelevant data values by separating the XML structure from the content.
- CXDLS processes queries and updates directly over the compacted XML document by using the labeling technique.
- We evaluate the performance of CXDLS on a variety of XML documents. Our results show that the approach can outperform existing ones significantly.

This paper is organized as follows. Section 2 introduces related research on XML labeling and compacting techniques. In Section 3, our approach for XML compaction and management is described in detail. In Section 4, experimental results are presented with comparisons to other existing approaches in this field. Finally, Section 5 concludes the paper.

2 Related Work

In this section, we consider two main directions for efficient handling XML documents, namely compact representation of XML and labeling schemes.

XML compression. Recently, several algorithms have been proposed for XML compression. One of the first approaches to XML compression was the XMill compressor [8] that separates the content from the XML structure. This separation increases the data similarity in each of them and allows to achieve better data compression rates. The separation is reminiscent of earlier vertical partitioning techniques for relational data, which divide a table into multiple tables defined over subsets of the attributes [9]. This partitioning typically lets queries scan less data and thus improves query performance [10]. However, the separation in XMill is used for XML compression only and it does not allow direct querying of the compressed data. Buneman et al. [5] proposed an approach based on skeleton compression [6], which extends the separation idea of XMill for efficiently querying compressed data. The skeleton approach removes the redundancy of

the document structure by using a technique based on the idea of sharing common subtrees and replacing identical and consecutive branches with one branch and a multiplicity annotation. In this approach, main memory data structures are used for the compressed skeleton, while external memory data structures hold textual contents. This approach still has some drawbacks: First, sometimes the compressed skeleton is still too large to fit into main memory. Second, compressed skeletons will always be scanned in their entirety to identify the relevant data vales.

XML labeling schemes. XML documents are usually modeled as labeled trees, where nodes represent tags or values and edges represent relationships between tags or tags and values. In order to facilitate query processing for XML data, several labeling schemes have been proposed. The basic idea of these schemes is to assign unique codes to the nodes in the tree in such a way that it takes constant time to determine the relationship between any two nodes from the codes. Therefore a good labeling scheme makes it possible to quickly determine the relationships between XML elements as well as to quickly access to the desired data. Existing labeling schemes can be classified into interval labeling and prefix labeling. In interval labeling [2] each node in the tree is labeled with a pair of numbers, with the first one being, e.g., the preorder number of this node and the second being the postorder number. While almost all region labeling schemes support XML query processing efficiently, their main drawback is the processing of dynamic updates of the tree structure. Insertion or deletion of new nodes would require relabeling existing nodes. In prefix labeling [7,3], each element is labeled by its path from the root, so that the label of a parent node is the prefix of the labels of all of its descendants. In schemes of this type, the size of labels is variable and depends on the tree depth. In contrast to interval labeling, the main advantage of prefix based schemes is their dynamics; labels of the existing nodes can remain stable in case of insertions and other updates. Therefore, we aim at preserving the benefits of the prefix labeling schemes and try to avoid their drawbacks. Several methods have been proposed to deal with this problem [11,12]. However the goal of those methods has been to improve the encoding to reduce label length, while our approach (CXDLS) exploits the properties of XML structures for compaction achieving minimal space consumption and it uses the ORDPATH labeling scheme to represent the XML document after the compaction retaining all the desirable features of ORDPATH. The goal of CXDLS is to combine the strengths of labeling and compression technologies. CXDLS bridges the gaps between them to get most benefits and avoid the drawbacks of those approaches yielding performance better than their independent use.

3 Compact Representation of XML

In this section, we introduce the labeling of nodes using the ORDPATH labeling scheme and we shall show their maintenance under insertions and deletions and we give some insight into their internal representation. After that we shall describe the main idea behind our approach for compacting the structure of XML

documents and we illustrate how the ORDPATH labels are used for maintaining relationships between nodes after compaction.

3.1 The Labeling Scheme Used in CXDLS

The ORDPATH labeling scheme [7] is a particular variant of a hierarchical labeling scheme, it is used in Microsoft SQL Server's XML support. It is essentially an enhanced, insert-friendly version of Dewey tree labeling [3] that specifies a procedure for expressing a path as a binary bitstring. It aims to provide efficient insertion at any position of an XML tree, and also supports extremely high performance query plans for native XML queries. ORDPATH assigns node labels of variable length and only uses the positive, odd numbers in the initial labeling, even-numbered and negative integer component values are reserved for later insertions into an existing tree, the fragment root always receives label 1. The n^{th} (n = 1, 2,) child of a parent node labeled p receives label p. (2* n - 1). When an insertion occurs, the negative ordinals support multiple inserts of nodes to the left of a set of existing siblings and the even number between two odd numbers is used as an intermediate node that does not count as a component that increases the depth of the nodes. The new node will be the child of the intermediate node, avoiding the need to relabel existing nodes at any time. An example of an XML document and its ORDPATH labels are given in Figure 1. Internally, ORDPATH labels are not stored as · -separated ordinals but using Unicode-like compact representations called prefix free encoding that is generated to maintain document order and allow cheap and easy node comparisons. This encoding consists of a set of L_i/O_i bitstring pairs: one for each component of the ORDPATH label. Each L_i bitstring specifies the length in bits of the succeeding O_i bitstring. Thus the L_i bitstrings are represented using the form of prefix-free encoding shown in

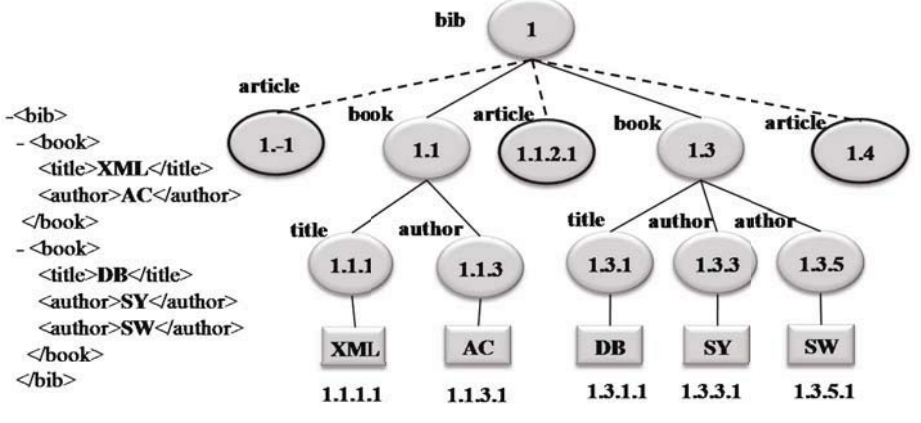

Fig. 1. A simple XML document with insertion of new nodes and corresponding ORDPATH labels

Table 1. L_i/O_i components can specify negative ordinals O_i as well as positive ones. They would be used to encode an ORDPATH label. The binary encoding is produced by locating each component value in the O_i value ranges and appending the corresponding L_i bitstring followed by the corresponding number of bits specifying the offset for the component value from the minimum O_i value for that range. Figure 2 displays the binary encoding of an ORDPATH label.

Table 1. Prefix-Free Encoding of the Bitstrings

Bitstring	L_i	O_i **Value range**
0000001	48	$[-2.8 \times 10^{14}, -4.3 \times 10^{9}]$
0000010	32	$[-4.3 \times 10^{9}, -69977]$
0000011	16	$[-69976, -4441]$
000010	12	$[-4440, -345]$
000011	8	$[-344, -89]$
00010	6	$[-88, -25]$
00011	4	$[-24, -9]$
001	3	$[-8, -1]$
01	3	$[0, 7]$
100	4	$[8, 23]$
101	6	$[24, 87]$
1100	8	$[88, 343]$
1101	12	$[344, 4439]$
11100	16	$[4440, 69975]$
11101	32	$[69976, 4.3 \times 10^{9}]$
11110	48	$[4.3 \times 10^{9}, 2.8 \times 10^{14}]$

In our approach we use the ORDPATH labeling scheme to label the XML documents, where for each element node of the structure is assigned *ORDPATHLB* that is a label obtained by the ORDPATH labeling scheme, while we assign to each text node[1] a text identifier *TID* that is the *ORDPATHLB* of its parent element. We use the ORDPATH labeling scheme because it has nice properties:

First, all relationships between nodes can be inferred from the labels alone. Second, it is easy to determine the order of the nodes. Third, ORDPATH allows the insertion of new nodes at arbitrary positions in the XML tree but nevertheless avoids relabeling existing nodes. Fourth, ORDPATH has an internal representation which is based on a compressed binary form. Finally, ORDPATH allows a faithful representation of the XML documents after compaction.

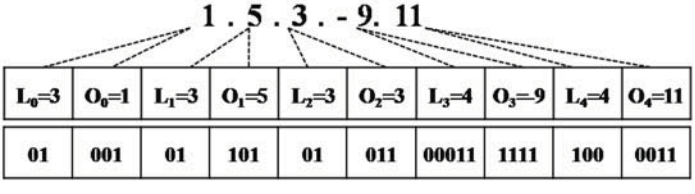

Fig. 2. An example for the binary encoding of an ORDPATH label

[1] For simplicity's sake, attribute nodes are disregarded in this paper,they can be treated like text nodes

3.2 XML Compaction

The basic idea of our approach combines the encoding scheme described in the previous section 3.1 and a new compaction technique to achieve a compact representation of XML documents for efficient management. The handling of navigational aspect of query evaluation, which only needs access to the structure, usually takes a considerable share of the query processing time, while character contents are needed for localized processing. Therefore separation the XML structure from its data values is very useful, because this separation typically lets queries scan less data and avoids unnecessary scanning of structures and thus improves query processing performance. Almost all previously proposed methods used the XML structure or a compressed version of the structure for navigational aspects of queries in main memory. However the XML structure can be very large in complex databases, thus it may not fit in main memory even in its compressed representation.

Our compaction Method helps to remove the redundant, duplicate subtrees and tags in an XML document. It separates the data values from the XML structure that is compacted based on the principle of exploiting the repetitions of similar sibling nodes in the XML structure, where "similar" means: elements with the same tag name. These similar nodes are replaced with one compacted node that is assigned a start label equal to the label of the first node of compacted nodes and an end label equal to the label of the last node of compacted nodes. Another principle is exploiting the repetitions of "identical" subtrees in the XML structure, where "identical" means: consecutive branches of trees which have exactly

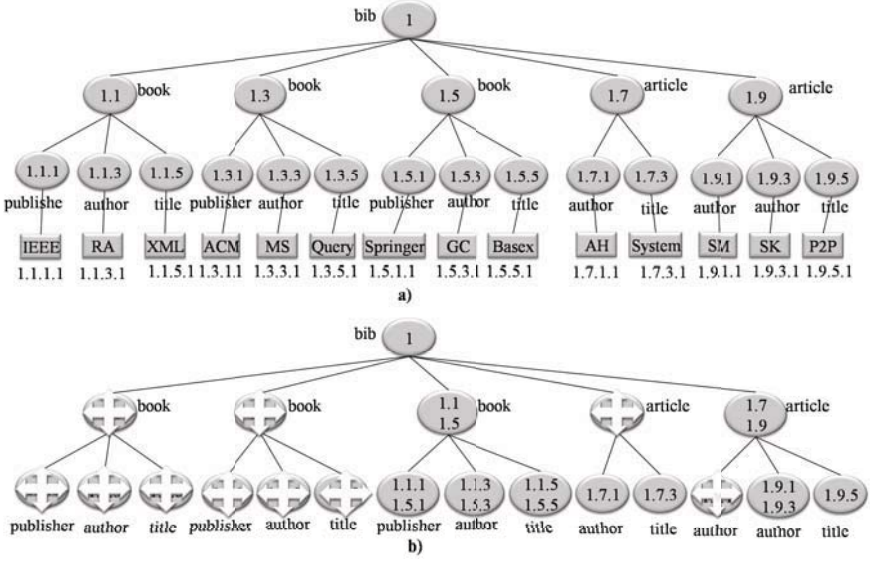

Fig. 3. The XML structure and its compacted form

the same substructure. These identical subtrees are also replaced with one compacted subtree, where each node in the compacted subtree is assigned a start label equal to the label of the node in the first subtree of compacted subtrees and an end label equal to the label of corresponding node in the last subtree of compacted subtrees.

Example 1. For the document of Figure 3(a) the compacted structure is shown in Figure 3(b); observe that the first, second and third subtrees which have the root "book" "1.1" , "1.3" and "1.5", are identical. Therefore, we can replace them by one subtree, where its root is assigned a start label equal to the label of the root in the first compacted subtree i.e "1.1" and an end label equal to the label of root in the third compacted subtree i.e "1.5" so that the new subtree has the root "book" and its label is "1.1, 1.5", and each child node of this subtree is assigned a start label equal to the label of the node in the first compacted subtree and an end label equal to the label of corresponding node in the third compacted subtree, so that the new subtree has child nodes which have the following labels respectively publisher "1.1.1, 1.5.1" author "1.1.3, 1.5.3" title "1.1.5, 1.5.5". Note that the nodes "article" with the labels "1.7" and "1.9" are similar, we also can replace them by one node "article" with label "1.7, 1.9" while the labels of their child node do not change. The process is recursively applied to all repetitive consecutive subtrees and tags in the XML structure. Figure 3(b) displays the compacted XML structure, where the crossed-out nodes will not be stored.

Using these labels we fully maintain all nesting information after the compaction, so the original XML document can be faithfully reconstructed. We consider for example the node "publisher" with the label "start = 1.1.1, end= 1.5.1". We can infer that this compacted node contains other nodes. To get the labels of decompacted nodes, we compare between each component of the start label with its corresponding in the end label to find all the odd numbers falling between them. Then we combine the resulting numbers from first two components with the resulting numbers from the second two components and so on. The process for inferring the labels is as follow: from the first components 1 and 1 it yields 1, from the components 1 and 5 we get the values (1, 3 and 5) and from the last components 1 and 1 it yields 1.We combine 1 with (1, 3, 5) yields "1.1", "1.3", "1.5" and we then combine this result with 1 yields the final decompacted nodes with the labels "1.1.1","1.3.1" and "1.5.1".

3.3 Data Structures

Based on our previous research [13,14] we strongly believe that improving the performance of XML management requires efficient storage structure and access methods. Towards that goal, we store the compacted XML structural information separately from the data information in the storage structure. This separation allows us to improve query processing performance by avoiding unnecessary scans of structures and irrelevant data values. But for query evaluation, we need to maintain the connection between structural information and value information. The ORDPATH labels can be used to reconnect the two parts of

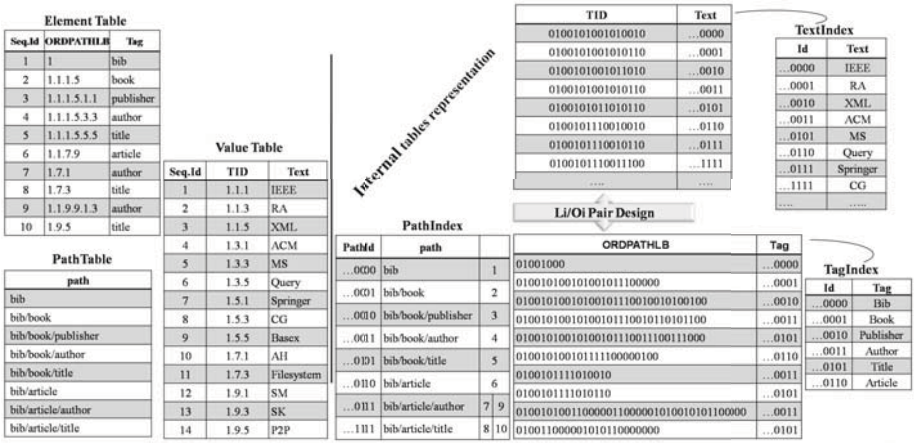

Fig. 4. Storage Structures for compacted XML document

information. Our storage model, shown in Figure 4, contains three tables: Element table, Value table and Path table.

The Element table stores the compacted structure of XML documents, where a sequence identifier, *ORDPATHLB* and tag name are stored for each element node of the compacted XML structure. We must note at this point that an *ORDPATHLB* of compacted node consists of start label and end label, which are stored as one label, where its first component is from the start label and its second component is from the end label and so on alternatingly. For example: the label "start=1.1.3,end= 1.5.3" of the compacted node "author" is stored as one label "1.1.1.5.3.3" in Element table. The tag names are indexed in a hash structures; the index entries are referenced by integer values. Thus each tag name is stored by its integer reference, the sequence identifier is implicitly given by the array position and each *ORDPATHLB* is stored in a byte array in binary form as described in Section 3.1. For example, the encoding of the *ORDPATHLB* "1.9" is the bitstring 010011000001. Because this bitstring is stored in a byte array, note that the last byte is incomplete. Therefore it is padded on the right with zeros to an 8bit boundary. Thus the stored bitstring is 0100110000010000. Note that all *ORDPATHLBs* start with "1.", therefore it is unnecessary to store this component explicitly, and thus we save 5 bits for each node of an XML document.

The Value table keeps all the data values of XML document, where sequence identifier, the own text identifier *TID* (see Section 3.1) and the text contents are stored for each leaf node. The sequence identifier is implicitly given by the array position and the *TID* is stored in a byte array in binary form and the text contents are indexed in hash structures and stored by their integer references.
The Path table stores all distinct paths in the structure of an XML document, where path means the sequence of elements from root node to any element node. The Path table is indexed in a hash structure; the index entries are referenced by

integer values. To achieve better query performance, this index is extended by references to sequence identifier values of the Element table, resulting in an inverted index [15].

3.4 Support for Querying and Updating Compacted XML Structures

CXDLS supports querying and updating of the compacted structure directly and efficiently. We can efficiently process XPath queries, including almost all axis types, node tests and predicates by using different algorithms and indexes such as tag index, value index and path index and also by applying some query optimizations. Moreover the key concept in quickly evaluating a query is using the ORDPATH labels to quickly determine the ancestor-descendant relationship between XML elements as well as fast access to the desired data.

Example 2. For an XPath query containing slash "/" or double slash "//" can be easily answered by performing exact-match or prefix-match on the path index yield pathId(s) and *ORDPATHLB*(s) referenced by the resulting pathId(s). Let the XPath expression to be evaluated be Q1: /BIB/ARTICLE/TITLE. Thus, we only need to perform exact match on the path index yield the pathId = 8 and *ORDPATHLBs* "1.7.3", "1.9.5" referenced by this pathId. For a path expression containing the //-axis, such as Q2: //TITLE, it requires suffix match for TITLE in the path index yield the pathIds 5, 8 and the *ORDPATHLBs* "1.1.1.5.5.5" , "1.7.3" and "1.9.5" referenced by these pathIds. Note that *ORDPATHLB* equal to "1.1.1.5.5.5" is label of compacted node. To show the final results it requires decompaction using the process which we mentioned it in Section 3.2 yield three nodes having the *ORDPATHLBs* "1.1.5", "1.3.5" and "1.5.5". For path expressions containing a wildcard or //-axis in the middle of the path expression, such as Q3: /BIB/*/TITLE or Q4: /BIB//TITLE. In this case, exact-match for BIB and prefix-match for TITLE on the path index yield the same result of Q2.

Example 3. For an XPath query containing predicates, the path expression inside the predicate is rewritten. Child steps are converted to parent steps and descendant steps are converted to ancestor steps and vice versa [16]. This type of queries fits the pattern"path = value". For example, to answer the predicate query such as Q5: /BIB/ARTICLE[TITLE="P2P"], first the path expression /BIB/ARTICLE/TITLE is answered in a way similar to answering Q1 yield the *ORDPATHLBs* "1.7.3 " and "1.9.5 ". Next we check which one has a text value equal to"P2P" using exact match on the Text table. In this case only the *ORDPATHLB* "1.9.5" has data value equal to"P2P". To obtain the ARTICLE in the final result, we find the parent's *ORDPATHLB* of "1.9.5", that is, "1.9", which is easily inferred from the label only.

Example 4. For twig queries, such as Q6:/BIB/ARTICLE[TITLE="P2P"]//XX, we first answer the path expressions /BIB/ARTICLE[TITLE="P2P"] like Q5 then the ancestor-descendant relationship between the resulting nodes and XX can easily be determined by their labels.

Note that CXDLS supports all XPath axes in the same way as in the prefix labeling scheme because it is based on ORDPATH labeling, in which we can easily determine the relative order of nodes, the child-parent and the ancestor-descendant relationships by a byte comparison of two ORDPATH labels. It is important to note that the compacted labels also retain all such features of ORDPATH labels. For example, in Figure 3 article "1.9" is an ancestor of title "1.9.5" since "1.9" is prefix of "1.9.5". Also the compacted book "1.1.1.5" is an ancestor of publisher, author and title having the compacted labels "1.1.1.5.1.1", "1.1.1.5.3.3" and "1.1.1.5.5.5" respectively because book "1.1.1.5" is a prefix. The update behaviors of CXDLS are exactly as in ORDPATH, which guarantees the complete avoidance of relabeling for new insertions in any positions (see Section 3.1). In addition, the updated nodes are compacted, if they fulfill the conditions of compaction mentioned in Section 3.2. For deletions we can just mark as deleted the corresponding nodes in the structure without any relabeling. However since compacted nodes represent set of nodes in the uncompacted structure, we must consider some deletion cases which might require partial decompaction. For example, if we want to delete the node element specified by the XPath expression BIB/BOOK[2] which selects the node BOOK "1.3", the compacted node BOOK "1.1.1.5" needs decompaction to mark the selected node and any descendant nodes below it as deleted.

4 Performance Evaluation

In order to assess CXDLS, we implemented it in Java with JDK 1.6 and ran three sets of experiments (storage, query and update) on a 2.00GHz CPU, 2GB RAM, and 80 GB hard disk running Windows Vista. We carried out the experiments using real and synthetic XML data sets: 11MB, 111MB version of the XMark benchmark [17]. TreeBank [18] and Factbook. Shakespeare is a set of the plays of William Shakespeare marked up in XML for electronic publication. SWISS-PROT is a curated protein sequence database. PART and CUSTOMER are from the TPC-H Relational Database Benchmark [18].TOL is the tree structure of the Tree of Life web project [19] which has high depth XML tree more than 240 levels. We choose these datasets because they have different characteristics. Attributes are omitted for simplicity.

4.1 Consumption of Storage Requirements

We measure storage requirements of compacted structures to determine the influence of our compaction method on reducing the storage requirements. We also compared these results with the storage requirements of the original ORDPATH labeling scheme in [7] and our recent work called CXQU in [13] and different labeling schemes such as Dewey labeling scheme and pre/post [2] used in MonetDB/XQuery [20].In the implementation, we store for each node of the XML structure its label and tag name pointer and we use the same prefix-free encoding (see Table 1) for all mentioned prefix labeling schemes to do a fair

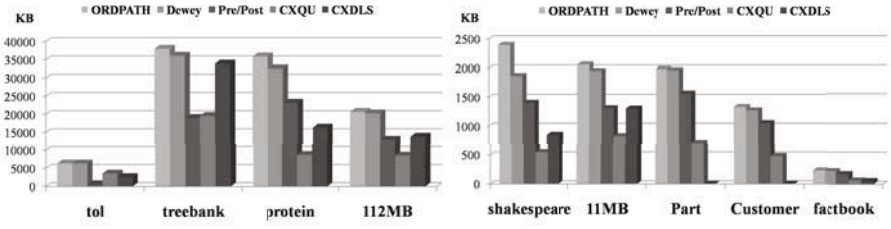

Fig. 5. Comparison of Storage Requirements

comparison between them. Also for pre/post labeling, we do not store the pre labels because they are implicitly given by the array position. The results shown in Figure 5 demonstrate that the success rate of the use of our compaction is very high. An interesting observation was made from the results. The storage requirements are determined by the degree of regularity of the structures of the XML documents. For this reason, it can be observed that the storage requirements are very small for the documents PART and CUSTOMER because they have regular structures. At the same time the storage requirements are still relatively small for other documents that have irregular structure or less regular structure when compared with other approaches,while our recent work CXQU [13] needs less storage requirements for this type of documents. The results from the last experiment show that our compaction method has efficient capabilities to reduce the storage requirements for both regular and irregular structures.

4.2 Query Performance

We investigate the query performance of CXDLS and compare it with the 32-bit version of MonetDB/XQuery 4.26.4 [20] and CXQU [13]. All approaches were experimentally evaluated using a set of queries which is compatible with XPath 1.0 currently supported by our system and comprises different kinds of XPath queries. Table 2 shows these queries for various XML datasets and presents the pure query evaluation times, excluding the times for parsing, compiling and optimizing the queries as well as serialization times. To gain a better insight into the query performance of all approaches, each query was repeated 10 times and the average of them was recorded as final result. These results confirm that CXDLS provides remarkably good query performance.

4.3 Update Performance

To evaluate the update performance, we measured the time for single node insertions at different positions of Hamlet XML document which is a Shakespeare's play. We chosen these documents due to their high compression ratio, thus the update time will be in a worst case. Hamlet has 5 ACTs, we add a new ACT before ACT[1], between ACT[4] and ACT[5], and after ACT[5] using "//ACT" as XPath expression with a predicate that selects the target node. We also measured the time for subtree insertions into different

Table 2. Elapsed time and XPath queries used in the comparison

Queries	MonetDB	CXQU	OUR	Hits
XMARK				
/site/regions/africa/item/description/parlist/listitem	9,30	0,54	0,30	57
/site/closed_auctions/closed_auction/price	7,60	2,39	3,34	975
/site/people/person/name	7,50	5,98	8,05	2550
/site/closed_auctions//emph	5,50	6,38	5,45	1154
/site/regions//item/description	7,30	7,54	5,56	2175
//category/description/parlist/listitem	3,30	1,67	4,58	104
//people//person	4,50	4,24	3,63	2550
//africa//item	4,20	1,08	0,95	55
//africa//item[location = "United States"]	8,50	2,22	3,08	47
/site/regions/asia/item[shipping]/description	11,30	5,65	6,52	200
SHAKESPEARE				
/SHAKESPEARE/A_AND_C/PLAY/ACT/SCENE/SPEECH/SPEAKER	10,50	5,03	3,81	1179
/SHAKESPEARE/TAMING/PLAY//SCENE//SPEAKER	4,20	2,59	3,39	895
//PLAY/ACT/SCENE/SPEECH/LINE/STAGEDIR	57,70	5,0	4,35	618
//PLAY//EPILOGUE/STAGEDIR	5,30	2,82	4,7	5
//PLAY/*/*	30,90	11,32	14,64	1938
//PLAY//SCENE//STAGEDIR	11,20	29,64	22,65	6224
//PLAY/ACT/SCENE/*[2]	64,20	52,61	43,05	748
//PLAY/ACT[2]	8,50	8,54	5,34	37
//LINE[STAGEDIR="Aside"]	77,20	88,52	84,37	208

positions to the SHAKESPEARE XML document. We add a new play in three insertion positions of SHAKESPEARE selected by the XPath expressions: XP1:/SHAKESPEARE/A_AND_C, XP2:/SHAKESPEARE/MERCHANT, XP3:/SHAKESPEARE/WIN_TALE, this new play is inserted as a subtree that consists of 97 element nodes, 73 text nodes and has 6 levels. Figure 6 compares the average insertion times in milliseconds of CXDLS with Monet/XQuery and CXQU and demonstrates that the elapsed time for these updates in CXDLS is similar to CXQU and particularly fast. This result is expected because both use labeling schemes that adapt to updates. The difference of their elapsed time comes from the difference in the time that for locating the update point and generating the new labels takes.

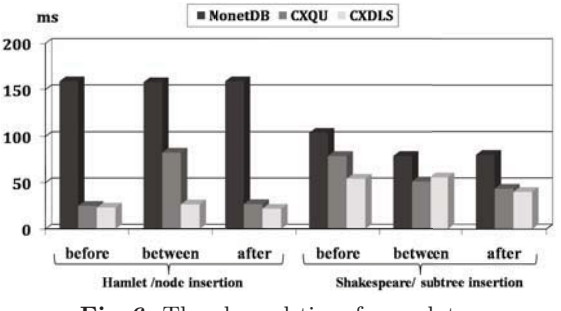

Fig. 6. The elapsed time for updates

5 Conclusions

In this paper, we proposed new approach called CXDLS that compacts the structures of XML documents by exploiting repetitive consecutive subtrees and tags in the structures and supports both update and query processing efficiently. The significant reduction in processing time and storage space is achieved by a combination of XML compacting and node labeling scheme. Our experiments verified that CXDLS improves performance significantly in terms of storage space consumption and query processing and updating execution time. As future work,

we plan to extend CXDLS with more indexing structures and query processing algorithms and we will continue our investigations on variations of binary encoding forms that can further minimize the storage costs. We focused on XPath queries. For the future, we plan to extend our work to evaluating XQuery on compacted XML structures.

References

1. W3C, Extensible Markup Language (XML), XML Path Language (XPath), XQuery 1.0: An XML Query Language, http://www.w3.org/TR/
2. Grust, T.: Accelerating xpath location steps. In: SIGMOD Conference, pp. 109–120 (2002)
3. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered xml using a relational database system. In: SIGMOD Conference, pp. 204–215 (2002)
4. Arion, A., Bonifati, A., Costa, G., D'Aguanno, S., Manolescu, I., Pugliese, A.: Efficient query evaluation over compressed xml data. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., Ferrari, E. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 200–218. Springer, Heidelberg (2004)
5. Buneman, P., Choi, B., Fan, W., Hutchison, R., Mann, R., Viglas, S.: Vectorizing and querying large xml repositories. In: ICDE, pp. 261–272 (2005)
6. Buneman, P., Grohe, M., Koch, C.: Path queries on compressed xml. In: VLDB, pp. 141–152 (2003)
7. O'Neil, P.E., O'Neil, E.J., Pal, S., Cseri, I., Schaller, G., Westbury, N.: Ordpaths: Insert-friendly xml node labels. In: SIGMOD Conference, pp. 903–908 (2004)
8. Liefke, H., Suciu, D.: Xmill: An efficient compressor for xml data. In: SIGMOD Conference, pp. 153–164 (2000)
9. Batory, D.S.: On searching transposed files. ACM Trans. Database Syst. 4(4), 531–544 (1979)
10. Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving relations for cache performance. In: VLDB, pp. 169–180 (2001)
11. Böhme, T., Rahm, E.: Supporting efficient streaming and insertion of xml data in rdbms. In: DIWeb, pp. 70–81 (2004)
12. Härder, T., Haustein, M., Mathis, C., Wagner, M.: Node labeling schemes for dynamic xml documents reconsidered. Data Knowl. Eng. 60(1), 126–149 (2007)
13. Alkhatib, R., Scholl, M.H.: Cxqu: A compact xml storage for efficient query and update processing. In: ICDIM, pp. 605–612 (2008)
14. Alkhatib, R., Scholl, M.H.: Efficient compression and querying of xml repositories. In: DEXA Workshops, pp. 365–369 (2008)
15. Grün, C., Holupirek, A., Kramis, M., Scholl, M.H., Waldvogel, M.: Pushing xpath accelerator to its limits. In: ExpDB (2006)
16. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking forward. In: Chaudhri, A.B., Unland, R., Djeraba, C., Lindner, W. (eds.) EDBT 2002. LNCS, vol. 2490, pp. 109–127. Springer, Heidelberg (2002)
17. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: Xmark: A benchmark for xml data management. In: VLDB, pp. 974–985 (2002)
18. University of Washington, XML Data Repository, http://www.cs.washington.edu/research/xmldatasets
19. Tree of Life web project, TOL, http://tolweb.org/tree/home
20. University of Amsterdam, Monetdb, http://monetdb.cwi.nl/