Improving XML Processing using Adapted Data Structures

Mathias Neumüller* and John N. Wilson**

Department of Computer and Information Sciences
University of Strathclyde in Glasgow, Scotland, U.K.

Abstract. From its origins in document processing, XML has developed into a medium for communicating all kinds of data between applications. More recently, interest has focused on the concept of native XML databases. This paradigm requires that database queries can be resolved by direct searching of XML data structures. Relational databases can be compressed without the loss of direct addressability. A similar approach can be applied to XML data structures. Compression in the relational paradigm is associated with improved performance. We review this approach and show results from the implementation of a prototype compressed DOM. Our research indicates that it is possible to optimise queries over XML structures by choosing appropriate physical representations on the basis of document analysis.

1 Introduction

Emerging standards such as XPath and XQuery are founded on the vision of XML as both a standard for document and data interchange between applications and also as a structure that may need to be queried directly in much the same way as a database system. Whilst the principles of querying hierarchical data structures were developed early in the history of computer science [24], further development of direct querying capability for XML data sources requires close attention to be paid to issues of acceptable performance. Whereas in hierarchical databases the designer had full control over the physical representation and database schema, XML documents are only defined in terms of a common data model and a textual representation. Physical storage varies widely and schema design is often an ad hoc process carried out by designers with skills in an application domain rather than specific database design skills. Emerging native XML databases (NXDs [22]) are designed to make XML applications independent from the physical storage in much the some way as relational databases. They offer tailor-made storage solutions for XML documents and allow access to the data using a standardised interface such as the Document Object Model (DOM). Relational database management systems however have a lead of two decades or so in the development of implementational technology.

* mathias@cis.strath.ac.uk
** jnw@cis.strath.ac.uk

We are exploring the consequences of applying compression directly to appropriate XML data to maximise the use of main memory storage in query processing. The varied nature of XML documents suggests that a range of algorithms are necessary to optimise the performance benefits that can be achieved by efficient internal representation. The aim of our research is to develop a framework that enables the choice of optimally efficient data representation and processing techniques based on the analysis of XML data structures. Due to the inherent performance limitation of external XML representations such as database mappings and textual XML files we are focusing our research on native, memory internal representations.

We start our paper by giving a short review of related work in section 2. We then show how compressed representations in the relational field are used already and how this can be adapted for semistructured data (section 3). Section 4 describes the DDOM system, a prototype of a compressed DOM implementation. We have measured memory consumption and have preliminary results for query performance (section 5). In section 6 we discuss some of the research issues that need to be explored before the compression technology can be applied in the context of a native XML database system.

2 Related Work

Significant research has already been carried out in the area of integrating XML data into relational and object-relational databases. There are many approaches to breaking up XML data into relational tuples, each with its own advantages and disadvantage. Some approaches interpret the XML data as a graph and store it in form of edges and nodes [6]. This approach is very flexible, but requires many joins upon querying or reconstruction. An improvement of this scheme is to store the graph as a set of complete paths and nodes [20]. However, both approaches fail to exploit any regularity that may occur in the XML structure. Other approaches, usually based on object-relational databases break the XML data into bigger, more complex blocks [10]. If similar structures occur in different parts of the document, these can be stored and queried together. However, most of these approaches require the presence of an XML Schema or a DTD in order to generate the necessary complex data types and relations and are thus less flexible. In the case of completely irregular data or loosely defined schemata they lead to the creation of a large number of sparsely populated tables, which is also disadvantageous. There are also hybrid mappings that try to combine advantage of both approaches [17].

In relation-based data structures, data compression is often useful for accelerating performance. Conventional databases usually represent data items as strings [7] although it is also possible to use fixed length pointers to compress domain values [16]. We have been able to exploit this compact representation of relational data [4]. New research in this field shows how to compress the data even further, using advanced compression techniques on the required dictionary

structures [9]. A flexible framework used to apply optimised compression algorithms in the context of query results was described in [3].

Compression in the context of XML is mainly focused on efficient transmission. Research exploiting the structured nature of XML documents was carried out [13], resulting in a serial compression algorithm for XML documents. The performance of this approach exceeds the compression ratio achievable with general-purpose compression algorithms. However, in a database environment content addressability is necessary, therefore such methods are not applicable. Some of the already available native XML databases, like Tamino [19] and Xindice [21], also use compression techniques. However, the compression is used for secondary storage only and typically based on a conventional serial compression algorithm. The binary XML format [25] suggested by the WAP forum uses tokenized representation of the XML tags, but does not apply this approach to the document data.

3 Architecture

The relational database model is founded on the concept that data can be normalised into regular table structures. This is a useful simplification but many applications, especially Internet-based information systems, require the storage and processing of irregular data structures. A data model that supports the representation of semistructured data has the potential to overcome the limitations of relational structures. Despite its widespread use, data centric XML applications tend to suffer from poor performance of the underlying technology since this is based on assumptions of document-centricity rather than data-centricity.

Compression is a central point of our approach to achieve better performance. Our work on relational database systems shows that compression can result in significant performance benefits by moving more of the workload from secondary into primary storage, i.e. from relatively slow disk storage into fast RAM. However, this approach can only be successful if individual elements remain accessible, thus serial, variable length compression algorithms such as LZH [26] are not appropriate. Our current method of compression enables us to compress the data off-line and resolve queries by decompressing only the output data. Dictionary encoding shows good compression ratios for relational data as well as for verbose XML documents, especially if they are machine generated using typically a rather small vocabulary.

3.1 Compressing relational data

The benefit of dictionary-based storage methods is that data can be represented using only minimal bit fields. At the same time direct addressability that is a fundamental requirement of efficient database models is preserved. First we will show how this can be applied in the simpler, relational case. Figure 1 shows some example data about sports clubs. Note that the data is already in third normal form and thus fairly compact. However, by using minimal bit strings

to represent values, the data elements can be stored in an even more compact form. There is, of course, still the overhead of dictionaries needed to convert the tokens, although these can also be compressed [9].

MEMBERSHIPS	
MEMBER	**ACTIVITY**
Miller	Volleyball
Miller	Golf
Smith	Volleyball
Wood	Golf

ACTIVITIES	
ACTIVITY	**LOCATION**
Rugby	Activities room
Volleyball	Gym
Golf	Gym

Fig. 1. Structure of **CLUB** example relations

The relation data would typically be stored in tables with fixed length fields. Thus the table **ACTIVITIES** would have a size of (10+15) characters $\times 3$ tuples $\times 8$ bits/character = 600 bits.

Inspection of the tables suggests that it would be possible to represent the information content in a more compact form by using codes to represent the domain values rather than using the domain values themselves in the relation. The attribute **ACTIVITY** contains three different values therefore in its most compact form, it could be represented as a two bit integer. Since there are only two values in the **LOCATION** attribute, it could be represented as a one bit integer. The compressed integer representation of the relations is shown in Figure 2.

MEMBERSHIPS	
MEMBER	**ACTIVITY**
00	01
00	10
01	01
10	10

ACTIVITIES	
ACTIVITY	**LOCATION**
00	0
01	1
10	1

Fig. 2. Structure of the compressed relations

The effect of representing data in this encoded format is to reduce the space occupied by **ACTIVITIES** to 9 bits and **MEMBERSHIPS** to 16 bits. To this it is necessary to add the dictionary that allows the tokens to be converted to their string equivalents and for the reverse process to take place (Figure 3). Dictionaries can be represented as lists of domain values. In the case of all non-unique attributes the number of entries in the dictionary will be less than the number of tuples in the relation.

ACTIVITY	MEMBER	LOCATION
Rugby	Miller	Activities Room
Volleyball	Smith	Gym
Golf	Wood	

Fig. 3. Dictionary structures

3.2 Compressing semistructured data

One possible representation of this data as XML is shown in part in Figure 4. It can be seen that the complete representation would contain almost the same redundancy that is exploited in the approach shown above for relations. The only reduction in redundancy is achieved by allowing set valued attributes, here two **activities** are stored below one **member** entry. But there also exists further redundancy caused by repetitions in the tag structure. Depending on the degree of flexibility of an associated DTD or XML Schema, the structure may be known in advance. Even if only well-formedness is assumed, the name of any closing tag is guaranteed by the XML syntax and thus redundant.

The structure of the compressed representation is shown in Figure 5. Our implementation of the compression strategy results in the structure being separated from the content. As in the original XML document, the order of individual entries is important as it encodes ancestor/descendant relationships. The tokenized structure representation is very closely related to the textual representation, allowing mixed content, comments and other XML specific data items to be included. References from the structure point to different dictionaries that store the document data. Note that the structure, i.e. the element names, are stored in a global context, whereas the document data is stored in context dependant dictionaries. This allows related information to be kept together. Multiple entries of the same string within one dictionary domain are avoided, thus reducing the redundancy. The compression algorithm currently used for our data structure is applied equally to data and meta-data (tag and attribute names etc). This approach results in significant reduction in the volume of the data stored.

3.3 Querying compressed data

There a two fundamentally different ways to query compressed data sources. The entire data can be decompressed and then queried in the uncompressed domain. Alternatively, one can compress the query and query the compressed data, followed by the decompression of the result set only.

Using the first approach, it is possible to benefit from compression only if the retrieval of the compressed file followed by its decompression takes less time than the retrieval of the larger, uncompressed version. However, the smaller the selectivity of a query, the less efficient this approach will be. In general one would assume that only a small sub-set of the actual data is returned for queries typical of data-centric applications. Thus it seems to be more desirable to transfer the

```
<club>
     ...
     <activities>
         <activity>
             Volleyball
         </activity>
         <location>
             Gym
         </location>
     </activities>
     ...
     <memberships>
         <member>
             Miller
         </member>
         <activity>
             Volleyball
         </activity>
         <activity>
             Golf
         </activity>
     </memberships>
     ...
</club>
```

Fig. 4. The club example data represented in XML

Type	#
Document	-
Element	1
...	
Element	2
Element	3
Text	1
/Element	3
Element	4
Text	1
/Element	4
/Element	2
...	
Element	5
Element	6
Text	1
/Element	6
Element	3
Text	2
/Element	3
Element	3
Text	3
/Element	3
/Element	5
...	
/Element	1
/Document	-

#	Element
1	club
2	activities
3	activity
4	location
5	memberships
6	member

#	Text:activity
1	Rugby
2	Volleyball
3	Golf

#	Text:location
1	Activities room
2	Gym

#	Text:member
1	Miller
2	Smith
3	Wood

Fig. 5. The structure (l.) of the compressed XML document together with the associated dictionaries (r.)

query itself into the compressed domain and only to return and decompress the actual result set.

In the case of dictionary compression this is very easy. The lexemes occurring in the query are sought in the document dictionaries. If no matching entry exists, the query will return no result. If matching tokens exist these in turn are sought in the document structure. Comparisons of these short binary tokens are typically faster than string comparisons. However, using the described architecture without any indexes a linear scan through the structure has to be performed. This may be slower, especially in the semistructured case when long path expressions need to be checked.

4 Implementation

The implementation of the compressed relational system has already been described in [4]. Since our Java implementation of the Dictionary compression based Document Object Model (DDOM) was developed independently of this system, implementation details of the compressed relational systems will be omitted from this paper. The DDOM is based on the architecture described in the previous section. For reasons of technical simplicity it currently supports read-only access on a document once it is parsed or generated. We believe that this limitation is acceptable for the targeted application area, e.g. mining of large, static, scientific data sets. It is not however, a fundamental limitation of the design. Furthermore we do not support persistence at the moment, which would be required in a production system.

One of the major differences between dictionary compression in relational systems and semistructured data is the definition of the dictionary domains. In the relational case this is relatively simple. Every attribute of a given relation is associated with its own domain. All values of this attribute lie within this domain. The only difficulty that can arise is that more than one attribute may share one domain, which can be hard to detect automatically. In the case of XML data the concept of domains is less well defined. As a first criterion we used the node type to distinguish different domains. Element and Text nodes for example exist in different domains. Additionally, those nodes that represent leaves in the DOM tree, e.g. Text nodes, are stored in sub-domains formed by their immediate parent node. In the example document, the values of the Text nodes for "Miller", "Smith" and "Wood" are stored in the domain **Text:member_name** as shown in the right part of Figure 5. Note that this is a technical simplification due to the fact that no type information is available in schema-less XML. Even if schema information is available, domains remain arguable. In the XML version of Shakespeare's plays[1] for example, **TITLE** elements occur in many contexts, for example inside the **PERSONAE**, **ACT** and **PLAY** elements. These form semantically different domains (e.g. *play titles* opposed to *act titles*). However, syntactically there is just one definition of the **TITLE** *type*. In our current approach all titles would be stored in one common dictionary, following the syntactical definition. This is also helps to avoid the creation of too many sparsely populated dictionaries. The problem of identifying which elements belong to which domain can not be resolved automatically in absence of a precise schema. The approach currently used is simple but works well for data-centric documents that usually do not contain highly nested data structures.

The difficulty of separating individual domains also influences the representation of the structure. The structure of a document is stored in form of an array as shown in the left part of Figure 5 with both columns represented by binary number types. The type of an entry is stored in a 8-bit type. Because references to all possible domains can occur in almost every possible context, we so far have been unable to use minimal bit patterns to store the references.

[1] Available at http://www.ibiblio.org/xml/examples/shakespeare

Currently 32-bit integers are used, which represent a significant waste, especially for domains with very low cardinality.

The structure array together with the associated dictionaries contain the entire data of an XML document. No tree of DOM nodes is stored to reduce memory consumption. Only the Document node that contains the structure and the dictionaries exists at any given time. However, the methods supplied by the DOM interface are required to return Node objects. These are generated dynamically and contain only a reference into the structure array. As soon as no further external reference to such a Node object exists, it can be garbage collected. Internally the DDOM uses methods that work directly on the structure array to avoid the overhead of frequent object generation and destruction. Externally however this mechanism is necessary to achieve DOM conformance.

The DOM interface does not support querying directly. Therefore we were required to use an external engine that works on top of this interface to perform XQL queries. However, the method *getElementsByTagName* of the DOM node types Document and Element allows the selection of ancestor nodes by name. Our implementation of this method follows the idea of doing as much work in the compressed domain as possible. Hence the tag name that is passed in as argument is sought in the dictionary containing the element names and its compressed representation is then sought in the corresponding part of the structure array. This is currently done by linear scanning, resulting in a $\mathcal{O}(n)$ runtime behaviour. At least for higher level elements, where n is of significant size this would need to be improved upon using indexes. The current approach does however avoid costly string comparisons and is able to resolve queries for non-existing tagnames early.

5 Performance

Test documents with various sizes were generated from a database used by a domain name server (DNS). The individual entries contain fields for the server name, the four parts of its numeric IP address and up to six parts of its symbolic domain name. In the case of XML (Figure 6) only those parts of the domain name that are present are stored, whereas in the original database null values are used to represent missing entries.

5.1 Memory consumption

Figure 7 shows the memory consumption of several representations of the same XML data as function of the database cardinality. Compressed and uncompressed textual XML documents are compared with different DOM representations. Our DDOM requires less memory than the popular Xerces and Crimson implementations[2], which both required the heap to be enlarged from 64 MB to 256 MB in order to process the largest document in this measurement. However,

[2] Available at http://xml.apache.org/xerces-j and http://xml.apache.org/crimson

even the DDOM representation required still more memory than the textual representation. This is partly caused by the Java implementation as it always uses 16 bit representations for characters whereas the XML file is using 8 bit encoding. Both conventional DOM implementations show a linear growth with database cardinality. By contrast, the DDOM shows sub-linear growth. We use the gzip compressed file size as a practical measure of the document entropy. The graph clearly shows that there is still a large potential for further memory savings.

```
<?xml version="1.0" ?>
<server-LIST>
  <server>
    <HOSTNAME>web0</HOSTNAME>
    <LEVEL0>uk</LEVEL0>
    <LEVEL1>ac</LEVEL1>
    <LEVEL2>strath</LEVEL2>
    <LEVEL3>cis</LEVEL3>
    <LEVEL4>www</LEVEL4>
    <IP0>130</IP0>
    <IP1>159</IP1>
    <IP2>196</IP2>
    <IP3>115</IP3>
  </server>
</server-LIST>
```

Fig. 6. Example entry of the DNS database in XML format

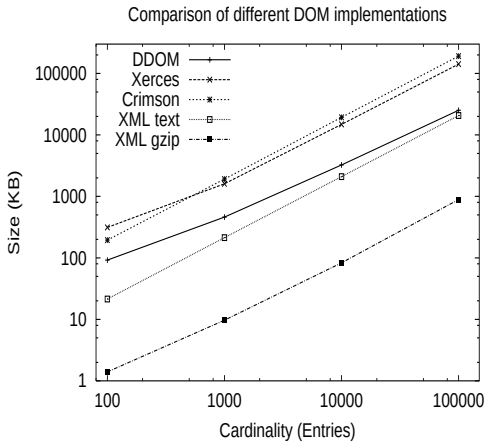


Fig. 7. Memory consumption of different representations of a DNS database (taken from [15])

5.2 Query performance

Measuring the performance of a storage/query system for XML in itself is somewhat complicated. The requirements of different applications vary widely. This is reflected by a growing number of XML query languages [2]. Standard benchmarks for XML databases are just emerging [18]. Since XML is based on the idea of documents, document-centric storage systems are further developed than data-centric ones. Queries in the document-centric domain are typically limited to locating entire documents that match certain requirements or to the execution of relatively simple transformations. Therefore storage is often based on information retrieval systems using full-text indices. In contrast, data-centric applications typically use relational databases as a back end. Queries in such systems are typically aimed at retrieving only a small fraction of a set of documents.

Figure 8 shows some preliminary query results, indicating some of the problems and possibilities in terms of performance of querying XML. Querying for

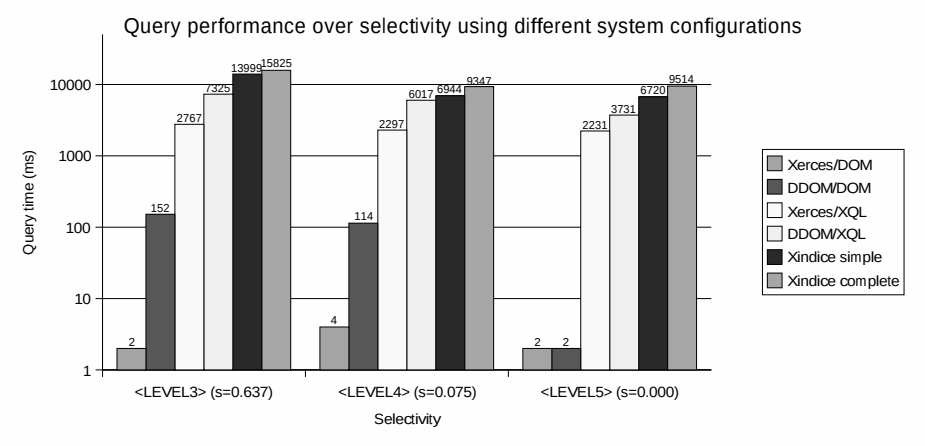


Fig. 8. Query performance of different query and storage systems for queries with different selectivity

the same results using different query and storage strategies and systems shows a wide range of performance variations. The measurements were performed using a single XML document containing 10,000 entries from the DNS database. The query selects elements with a certain name, in this case the elements containing the fourth, fifth and sixth component of the domain names. The selectivity of this query is decreasing as almost all domain names contained in the example document have four parts, but none has six parts. The chosen query may look overly simple, but provides the chance of comparing the DOM interface directly to any higher level query system. Although of limited practical use on its own, it is the most commonly used selection operator in XML, thus preceding almost every other query. We also think that this query can be used to show the variance in performance achievable.

Again the DDOM was compared against the Xerces implementation. The query can be directly resolved by using the *getElementsByTagName*("LEVEL*n*") method. To allow a comparison with a more realistic system, that would be capable of handling more complicated queries, the query was repeated with an XQL query engine on top of the two DOM implementations ("//LEVEL*n*"). The query engine used was taken from the GMD-IPSI XQL Engine[3] implementation and is not optimised for use with either of the implementations and thus restricted to the standard DOM interface. DOM representation, query engine and the proprietary query application run in a single Java virtual machine (JVM) with a default maximum memory space of 64 MB. Finally the measurements were repeated using a native, disk-based XML database, in this case Xindice [21]. No indexes were generated to allow a comparison with the DOM implementations that also do not use indexes. For Xindice, measurements were performed on a

[3] Available at http://xml.darmstadt.gmd.de/xql/

warm cache. Database engine and query application run on a single computer but in different JVMs. The server runs in a JVM with a maximum of 168 MB of memory, the client uses the JVM default setting of 64 MB. Xindice is accessed using its XML:DB interface and supports XPath for querying. The query can be stated in two different ways. The simple version ("LEVELn") just locates elements with the given name, whereas the more complex version ("//LEVELn") searches for these as descendants of the document root. Although these two queries are semantically identical, query times vary measurably.

Four orders of magnitude lie between the limited but direct access to main-memory based DOM representations and the flexible but slow disk-based NXD. However, using a separate query engine on top of the DOM implementations consumes most of this performance advantage, despite the fact that the entire querying process is performed in the computer's main memory. It also becomes apparent that the DDOM implementation suffers from the dynamic object creation as its performance diminishes with higher query selectivity. This limitation is exaggerated by the fact that the external XQL engine performs four complete traversals of the DOM tree using only methods supplied by the Node interface, rather than using the more specialised methods of the Element or Document nodes that are able to perform the required task much faster.

5.3 Limitations

The experiments with external query engines based on the DOM interface showed that this interface severely limits the performance of data-centric applications as shown in Figure 8. This result, together with some new ideas about more efficient, adapted compression algorithms have led to a general overhaul of the developed structure. We are reviewing the representation of the document structure and content and the dynamic node generation mechanism. Partial approaches to these problems are described in the following section.

6 Future research

We plan to exploit the experience gained from the implementation of the compressed relational and semistructured systems for future research in the area of compressed in-memory databases. We anticipate focusing our effort on several issues needed to transform our compressed DOM to a main-memory database for semistructured data. We need a high-performance query system that works with and utilises the advantages of a compressed representation. This will be based on a adaptive architecture that encodes content dependent on its characteristics. This should further help to improve compression ratios. We also need to support queries that run over more than one document at a time. Indexing has not been addressed so far, despite its important influence on query performance. A review of available techniques and analysis of their possible uses seems to be necessary. All these conceptual changes will depend on a different representation of the document structure.

6.1 High-performance query systems

To achieve better query performance we will develop an interface to the compressed data at a lower level than that provided by the DOM. The need to instantiate many nodes of a DOM tree during traversal could be overcome by the use of a cursor to access the data. As in the relational case, higher level, non-procedural query languages should allow for optimisation of the query execution. We will therefore aim to produce an optimised query system that bridges the gap between low-level data representation and high-level query language. The question as to which query language to support is not decided yet. XQuery will become the recommended standard of the W3C but has not yet reached this state yet. The expressive power of other query languages such as XQL or XPath is slightly more limited but may be sufficient for the targeted application area. These alternatives are currently more mature and widely accepted.

6.2 Domain specific compression methods

Currently the compression mechanism is closely coupled with the DDOM implementation and it is hard to separate the compression algorithm from the rest of the implementation. Recently we have separated the dictionary part using a pluggable interface to allow experiments with different dictionary implementations. Experience from the relational architecture has shown that significant memory is taken by the dictionaries. New research in this area [9] has shown that useful improvements in compression can be achieved by using compressed domain dictionaries. It is our goal to separate the compression mechanism used from the external interface to allow experimentation with different implementations.

We further hope to bridge the division between data-centric and document-centric XML by selecting different compression methods for different documents or even fragments of a document. Adapted compression methods are widely used in other information storage systems [27] and should be beneficial for our purposes too. Choosing appropriate compression methods could be done using emerging metrics defined for XML data [11]. An framework for adaptive compression of query results was already described in [3]. This will need to be modified in order to cope with the properties of semistructured data.

6.3 Support for document collections

Although it is currently possible to work with multiple compressed documents in parallel, the advantages of applying the compression techniques across such a collection are quite limited. Compression and querying is handled on a per document level. Especially in the targeted context of data-centric applications, many documents will share a common structure and may share a high degree of common content. Applying the compression technique across such a collection of documents will result in an improved compression ratio compared to the separate compression of individual documents. This is mainly caused by

the relatively smaller overhead of the dictionaries. The merged dictionaries of two similar documents are typically smaller than the size of the two separate dictionaries.

Spanning the compression across several documents becomes even more important for querying than it is for compression. NXDs typically allow documents to be queried collectively. If compression is applied on document level, the compressed form of the same content may have different representations in different documents. Thus a query across several compressed documents would have to be transformed for every document to be queried. This can be avoided if documents that are typically queried together are also stored together in a common compressed domain.

6.4 Indexing

Indexing is an important building block of high performance database operations. However, indexing in the context of XML is complicated by the fact that domains are often unknown and the structure contained within an document may or may not be of importance. So far we have not analysed this issue far enough to give a conclusive estimation of its importance for our research. Value based indexing within a domain could be handled by the dictionary system. Structural indexing however is closely related with its representation. We will aim to close this gap in the future and refer to literature such as [12] and [5] for a first impression.

6.5 Structural representation

Because the current structural representation is the main cause of performance limitation, both in terms of compression ratio and query performance, we will investigate the use of other data structures. At the time of writing, a representation based on the idea of DataGuides as introduced in [8] seems to be most likely. However, unlike the Lore database system [14] for which this technique was developed and which uses DataGuides as a mean to querying, we want to store the document data within the DataGuide. This approach separates attribute domains from each other and allows statistical information to be stored together with the data itself. This can be used to improve compression and also as a structural index for the contained data, consequently further enhancing query performance.

7 Conclusion

The DDOM prototype implementation demonstrates a significant space saving compared with standard DOM implementations. The evidence suggests that the entropy of typical data-centric XML documents is such that considerable savings beyond those we have achieved should be possible. Our implementation of dictionary compression in relational structures indicates that a saving of a factor of

eight in the space occupied by data is achievable without compressing the dictionaries themselves or resorting to space saving strategies for compressing indexes. For XML data structures, the influence of the type of data needs to be analysed more closely. Adaptive techniques may be possible that compress domain specific data more effectively. At the lowest level, we have represented tokens as integers rather than minimal bit-strings and further savings in space may be achieved by optimised data structures. Many of the issues surrounding the efficient handling of data centric XML applications are not solved yet. The compression algorithm we have used needs further refinement and other algorithms need to be tested out and compared with the current technique.

In the relational domain, we have previously demonstrated that our approach to compression provides significant benefits in terms of enhanced query performance. The results we report here suggest that similar benefits can be achieved for querying XML data structures if the queries can be resolved in the compressed domain. We have yet to developed a suitable strategy for querying the compressed XML data directly and expect to be able to make further significant improvements in the performance.

The performance of future Internet-based applications will be determined in part by the ease with which semistructured data can be queried and transferred. The work we describe suggests that compression has the potential for enhancing these processes and accelerating application performance in a resource-hungry environment.

References

[1] Peter M. G. Apers, Paolo Atzeni, et al., editors. *Proceedings of the 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 2001.

[2] Angela Bonifati and Stefano Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1):68–79, 2000.

[3] Zhiyuan Chen and Praveen Seshadri. An algebraic compression framework for query results. In *ICDE 2000*, pages 177–188, San Diego, California, USA, 2000. IEEE Computer Society.

[4] W. Paul Cockshot, Douglas McGregor, and John Wilson. High-performance operations using a compressed database architecture. *Computer J.*, 41(5):283–296, 1998.

[5] Brian Cooper, Neal Sample, Micheal J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In Apers et al. [1], pages 341–350.

[6] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *Data Engineering Bulletin*, 22(3):27–34, Sep 1999.

[7] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Trans. Knowledge Data Eng.*, 4:509–516, 1992.

[8] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In Matthias Jarke, Michael J. Carey, et al., editors, *VLDB 1997*, pages 436–445. Morgan Kaufmann, 1997.

[9] Abu Sayed M. L. Hoque, Douglas R. McGregor, and John N. Wilson. Database compression using of-line dictionary methods. Technical report, Department of

Computer and Information Science, University of Strathclyde, Glasgow, Scotland, UK, 2002.

[10] Meike Klettke and Holger Meyer. XML and object-relational database systems. In Suciu and Vossen [23], pages 151–170.

[11] Meike Klettke, Lars Schneider, and Andreas Heuer. Metrics for XML document collections. In Akmal Chaudri and Rainer Unland, editors, *XMLDM Workshop*, pages 162–176, Prague, Czech Republic, Mar 2002.

[12] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In Apers et al. [1], pages 361–370.

[13] Hartmut Liefke and Dan Suciu. XMill: An efficient compressor for XML data. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD 2000*, volume 29 of *SIGMOD Record*, pages 153–164, 2000.

[14] J. McHugh, S. Abiteboul, et al. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, Sep 1997.

[15] Mathias Neumüller and John N. Wilson. Compact in-memory representation of XML data. Technical report, Department of Computer and Information Science, University of Strathclyde, Glasgow, Scotland, UK, 2002.

[16] P. Pucheral, J.-M. Thevenin, and P. Valduriez. Efficient main memory data management using the DBGraph storage model. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *VLDB 1990*, pages 683–695. Morgan Kaufmann, 1990.

[17] Albrecht Schmidt, Martin Kersten, Menzo Windhouwer, and Florian Waas. Efficient relational storage and retrieval of XML documents. In Suciu and Vossen [23], pages 137–150.

[18] Albrecht Schmidt, Florian Waas, et al. Why and how to benchmark XML databases. *SIGMOD Record*, 30(3):27–32, 2001.

[19] Harald Schöning. Tamino – a DBMS designed for XML. In *ICDE 2001*, pages 149–154, Heidelberg, Germany, 2001. IEEE Computer Society.

[20] Takeyuki Shimura, Masatoshi Yoshikawa, and Shunsuke Uemura. Storage and retrieval of XML documents using object-relational databases. In Trevor J. M. Bench-Capon, Giovanni Soda, and A. Min Tjoa, editors, *DEXA' 99*, volume 1677 of *LNCS*, pages 206–217. Springer, 1999.

[21] Kimbro Staken. Introduction to dbXML. *XML.com*, Nov 2001. http://www.xml.com/pub/a/2001/11/28/dbxml.html.

[22] Kimbro Staken. Introduction to native XML databases. *XML.com*, Oct 2001. http://www.xml.com/pub/a/2001/10/31/nativexml.html.

[23] Dan Suciu and Gottfried Vossen, editors. *The World Wide Web and Databases, Third International Workshop WebDB 2000, Dallas, Texas, USA, May 18-19, 2000, Selected Papers*, volume 1997 of *LNCS*. Springer, 2001.

[24] D. Tsichritzis and F. H. Lochovsky. Hierarchical data-base management: A survey. *ACM Computing Surveys*, 8(1):105–123, 1976.

[25] WAP Forum, Ltd. *Binary XML Content Format Specification*, version 1.3 edition, Jul 2001. http://www.wapforum.org.

[26] T. A. Welch. A technique for high performance data compression. *IEEE Computer*, 17(6):8–20, Jun 1984.

[27] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, May 1999.