Compact In-Memory Representation of XML

Design and Implementation of a compressed DOM for data-centric documents

Mathias Neumüller and John N. Wilson

Department of Computer and Information Science, University of Strathclyde,
Glasgow G1 1XH, Scotland, U.K.
{mathias,jnw}@cis.strath.ac.uk

Abstract. Over recent years XML has evolved from a document exchange format to a multi-purpose data storage and retrieval solution. To make use of the full potential of XML in the domain of large, data-centric documents it is necessary to have easy and fast access to individual data elements. We describe an implementation of the Document Object Model (DOM) that is designed with these objectives in mind. It uses compression to allow large documents to be stored in the computer's main memory. Query-relevant DOM methods are optimised to work on top of the created data structure. Measurements indicate that compression up to a factor of 5 is possible without losing the ability to directly address individual elements. No prior decompression is needed to query and locate nodes.

1 Introduction

XML has been established in the domain of document management systems and is now emerging as an alternative to conventional database approaches. It already provides support for some of the features that have hitherto been provided by database systems. Further development of this capability requires that one must be able to query XML data sources with acceptable performance [25]. Although the syntax of XQuery [23], the upcoming query language for XML designed by the W3C, is on its way to completion, so far there is limited support for it in existing products. It is our belief that potential difficulties originate at the access level. While large volumes of XML data stored in textual form are hard to access, mappings to a relational database typically result in a large number of performance limiting joins [16]. Common implementations of the Document Object Model (DOM) [19] [21], defined to give easy access to individual elements of an XML document, struggle with memory limitations. We therefore decided to design a DOM implementation with low memory requirements that could serve as base for query engines.

2 Related Work

Significant research has already been carried out in the area of integrating XML data into relational [6] [5] and object orientated databases [26] [14]. Several

mapping schemata exist that allow storage of hierarchical XML into relational tables or object stores. The easiest approach is to store the entire document as one BLOB; all major database suppliers have incorporated specialised data types for such attributes into their products. More sophisticated methods usually break documents down into their elements to allow more efficient querying. On the downside these methods require a large number of joins to generate a view of the reconstructed document.

It is an accepted fact that data processing can be speeded up by avoiding access to secondary memory, usually magnetic media, and the use of memory-resident data instead. This technique can significantly improve querying processes that require random access to the stored data. Primary memory database systems were derived in the relational world in order to reduce access time [7]. Due to the price of solid-state memory, compact representation of the data is essential for the success of such systems. Because operations in main memory are typically several orders of magnitude faster than those accessing external storage, even penalties caused by compression/decompression algorithms can be tolerated if all data can be held in main memory [3].

Little research has been done in the area of compact representation of XML data. Its semi-structured, hierarchical nature makes it harder to treat than two-dimensional, strongly typed relational data. Existing research [12] [8] concentrates on compressing entire documents. Documents compressed in this way are unsuitable for querying. Access to the individual data elements requires decompression and subsequent parsing of the document.

Most implementations of the DOM so far have concentrated on extensive functionality rather than on compact representations. Thus those implementations can only handle relatively small documents. PDOM [9], which is a persistent DOM implementation that supports queries using XQL, allows to access elements of larger documents. The data is held on disk and copied into a buffer memory for processing. Compression is used for disk storage but not in memory. dbXML[1] [2], a native XML database, supports compression of individual elements, both in main memory and in secondary storage. The compression is limited to the tokenisation of element and attribute names and primarily used to increase the speed of querying. This approach is somewhat related to the approach suggested by the WAP forum [18], where element names are replaced by binary symbols to reduce transmission bandwidth. However, redundancy present in the data itself is not utilised, thus the compression achievable is moderate. Compression for document-centric XML is provided by some systems such as Tamino [15]. All of the designs mentioned are aimed at relatively small documents, though dbXML and Tamino allow documents to be grouped into collections and support queries across these collections.

[1] dbXML will become an Apache project named "Xindice" in the near future

3 Design

The basic design combines two approaches to achieve a compact in-memory representation: dictionary substitution for the occurring strings and minimising of the number of objects required to represent the DOM tree. Although the created structure is intended to be fully compliant with any well-formed XML document, a number of assumptions about typical documents are made.

3.1 Assumptions

The chosen compression method will only work well if a large number of identical data or metadata entries exist. A particular tag needs to occur sufficiently frequently to allow for multiple occurrences of the same values within its scope. Because the metadata, i.e. the XML tags, are compressed in the same way as the actual data, ideally only a few different tags appear many times in the document. In fact this assumes a non-random distribution of both data and metadata, which is true for practically every given data source. It is especially true for automatically created collections of data, like log entries, or data exported from relational databases, where the source dictionary is typically limited by explicit constraints. In general these kinds of XML documents are referred to as *data-centric* XML.

3.2 Dictionary Substitution

In a relational database, redundancy can be avoided by normalisation. Whether this is done in practice will often depend on the individual requirements of the application. However, in XML such an approach is not only impractical but will also result in a document that is more difficult to read, conflicting with the original design goals of XML. Links between different entities of an XML document are possible through the use of *ID* and *IDREF* attributes or *KEY* and *KEYREF* types using XML Schema [22] and can be used to achieve similar functionality as a primary/foreign key pair in a relational database. However, this will conflict with the idea of localising related information within one document. Furthermore it would introduce the need to normalise data prior to storing it. Since this kind of conventional logical design approach can not be applied to XML documents, other ways of reducing this redundancy must be found.

Dictionary substitution is a well understood, relatively simple compression mechanism [3]. It is is based on the replacement of textual representation of information by a short binary token, just big enough to represent the occurring values for the attribute in question. These tokens are fixed-length minimal bit-patterns. Every occurring data word is stored in a dictionary. Every occurrence of this word in the source document is replaced by its corresponding token. The document will be compressed if the same word occurs more than once. This compression technique is especially suitable for XML as its verbosity requires every piece of information to be expressed in textual form. In the extreme case of Boolean values the most common representation are the terms “TRUE” and

"FALSE". In the worst case 80 bits (5 x 16 bit Unicode characters) are used to express information content of just one bit.

In terms of our DOM, a data word is the name or the value of an attribute, an element name or any character data section. This means that the metadata contained in the element tags is compressed using the same compression technique as used for the data. Because data-centric documents have a large number of identical mark-up tags and may have a lot of identical attribute values and character data sections, considerable savings can be achieved. Figure 1 shows a simple example, a list of two server names together with their Internet domain names. It is a fairly regular structure, all tag names with the exception of *DN4* appear in both list entries. Some of the values are identical for the two different entries as well.

```
<?xml version="1.0" ?>
<DNS>
  <SERVER ID="www-0">
    <IP>130.159.23.21</IP>
    <DN0>uk</DN0>
    <DN1>ac</DN1>
    <DN2>strath</DN2>
    <DN3>www</DN3>
  </SERVER>
  <SERVER ID="www-cis">
    <IP>130.159.40.0</IP>
    <DN0>uk</DN0>
    <DN1>ac</DN1>
    <DN2>strath</DN2>
    <DN3>cis</DN3>
    <DN4>www</DN4>
  </SERVER>
</DNS>
```

Fig. 1. A simple example in XML

Our system builds a separate dictionary for every domain. The type of the node serves as a primary domain, e.g. all element names exist in one common domain *ELEMENT*. The domains of character data, attribute names and values are additionally subdivided by the name of their containing or direct parent element, e.g. all IP addresses in our example are stored in the domain *PCDATA:IP* as shown on the right side of figure 2.

The nodes of the created DOM tree will only store references to this dictionary. If a value occurs repeatedly, only a second reference to the same value will be stored. In the example the values for *DN0* to *DN2* are all identical for

Type	#
START-DOCUMENT	-
START-ELEMENT	1
START-ELEMENT	2
START-ATTR	1
PCDATA	1
END-ATTR	1
START-ELEMENT	3
PCDATA	1
END-ELEMENT	3
START-ELEMENT	4
PCDATA	1
END-ELEMENT	4
START-ELEMENT	5
PCDATA	1
END-ELEMENT	5
START-ELEMENT	6
PCDATA	1
END-ELEMENT	6
START-ELEMENT	7
PCDATA	1
END-ELEMENT	7
END-ELEMENT	2
START-ELEMENT	2
START-ATTR	1
PCDATA	1
END-ATTR	1
START-ELEMENT	3
PCDATA	1
END-ELEMENT	3
START-ELEMENT	4
PCDATA	1
END-ELEMENT	4
START-ELEMENT	5
PCDATA	1
END-ELEMENT	5
START-ELEMENT	6
PCDATA	1
END-ELEMENT	6
START-ELEMENT	7
PCDATA	1
END-ELEMENT	7
END-ELEMENT	2
END-ELEMENT	1
END-DOCUMENT	-

#	ELEMENT
1	DNS
2	SERVER
3	IP
4	DN0
5	DN1
6	DN2
7	DN3
8	DN4

#	ATTR:SERVER
1	ID

#	PCDATA:SERVER
1	www-0
2	www-cs

#	PCDATA:IP
1	130.159.23.21
2	130.159.40.0

#	PCDATA:DN0
1	uk

#	PCDATA:DN1
1	ac

#	PCDATA:DN2
1	strath

#	PCDATA:DN3
1	www
2	cis

#	PCDATA:DN4
1	www

Fig. 2. The structure arrays and corresponding dictionaries for the example file

the two entries. Only one entry per domain will be stored, reducing the memory requirements of the created data structure. Note though that the reoccuring string "www" will be stored twice, because it exists in the two distinct domains *PCDATA:DN3* and *PCDATA:DN4*. In the case of completely random data, as used for some benchmarks, this method can not result in any savings. Because all entries will have to be saved in the dictionary, the associated lexemes will actually increase the memory requirements. We rejected this limitation as irrelevant as no practical data source is completely random.

3.3 Object Minimisation

The most direct approach to implementing the DOM recommendations of the W3C is to parse every XML entity into a separate object. Each entity is represented by a corresponding node in the DOM tree. Depending on the kind of node and implementation system, more internal objects may be needed to store the contained information. In the case of Java, the implementation platform chosen, this will typically result in more than $2n$ objects for a DOM tree with n nodes (n objects for the nodes, n String objects to store the node names or values and some additional objects for node types that have additional properties such as attributes). Compared to elementary types such as integers, the use of many objects will result in an increased memory footprint, as references must be maintained [17].

Most DOM methods have sub-classes of Node objects as their result type. As a consequence, object representations of the different entities have to exist in order to communicate with DOM compliant applications. However, this does not imply that one object per node has to exist at any given time. In a typical usage situation, where a document is parsed once and then queried for a limited set of its individual elements, the requirements will be quite restricted. During the construction phase, one typically has one unconnected node, which is currently filled with data, and a reference to its future parent node. Tree traversal algorithms usually need a reference to one node and one of its children. Most query operations will also require only a reference to the node being queried and a limited set of nodes representing the query result. In all these cases the program will typically maintain a reference to the document root.

The idea behind the designed system was to reduce the number of objects held in memory to a minimum. This can be achieved by replacing object representation with primary types. This approach does not only allow for further memory savings but can also help to speed up searching processes [17]. Only node objects to which external references exist will persist, while other objects will be garbage collected and dynamically re-created on demand. Of course their data needs to be maintained somewhere: actual values of each node will be stored in the dictionaries, the structure of the tree and the references to the dictionaries will be stored in arrays containing only elementary types handled by the document node.

3.4 Structure Arrays

Conceptually XML data is represented as a tree although, as in the case with all trees, they can also be represented as a linear data structure. In the case of XML a possible flat representation is already given: XML text files are flat character files. The structure is given by corresponding start and end tags. It is reasonably easy to translate this into a series of tokens as shown in [18], which can be stored in a simple array of integers. This is shown in the left half of figure 2. Note that the "type" shown in the first column is also stored as a short integer type and just presented in textual form for the purpose of readability.

4 Implementation

The designed system, the Dictionary substitution based DOM implementation (DDOM), was implemented in Java. Classes for nodes as defined by the W3C name-binding schema for the DOM level 1 [20] with the exception of entities, entity references and notations, which need to be resolved by the underlying parser, were implemented. All classes were based on a common abstract class DNode, which implements most of the functionality. Nodes can either be connected or unconnected to a parent node. As long as they are unconnected, they hold their own data locally, together with a reference to the owning document. When they are appended to another node, which has to be a descendant of the abstract class DGroupNode, the parent node becomes responsible for maintaining the contained data. If no further external references to the child node exist, it can cease to exist. Once the entire document is built, either manually using the DOM interface or automatically by the parser using an existing XML source file, all structural information is maintained by the document node. All literal data is held in a collection of dictionaries, which are in turn maintained by the document object.

New nodes can only be appended to the end of an existing structure, i.e. as last child of their parent node. That is, it is effectively a write once, read multiple (WORM) data structure. Although this was not essential to the concept it simplified the implementation and seemed to be reasonable for the targeted application area, where a usual life cycle would consists of a single parsing process at the beginning followed by many read-only queries. The implemented parser appends nodes in the required order and is based on an underlying SAX parser.

Object minimisation preservers memory but may also result in excessive processing power requirements for frequent object creation and destruction. To minimise this, internal methods do not use the methods provided by the DOM interfaces, which generally return node objects and thus would require frequent object creation. Instead, they work directly on the arrays containing the document structure. Tree traversals are translated into linear search algorithms over the structure array. The method *Element.getElementsByTagNames* is especially useful for querying purposes and utilises the compressed data structure as well. Rather than generating and traversing a complete object tree, whose node values would then be compared to a given query expression, the query value is looked

up in the corresponding dictionary. If no entry exists, the query will return an empty result. Otherwise the corresponding token will be looked up in the structure array. Only those nodes that actually fulfil the query conditions will be instantiated.

5 Measurements

5.1 Test Data

We have evaluated the approach using two sets of data. The first set is a file containing DNS entries. Each row relates a server name to the 4 components of its IP address and up to 6 parts of its domain name, i.e. it is a slightly more complicated version of the example given in figure 1. The data was taken from the root domain server for the "ac.uk" domain, thus all rows have the constant entries "uk" and "ac" in the two most significant domain name fields. The data was loaded into a relational database and different sized subsets were exported into XML files. Each row resulted in an element with one child element per attribute. Attributes with null values were suppressed in the resulting XML. This is an example of a data-centric document. A suitable DTD was generated from the data files.

The second set of data was a collection of Shakespearean plays encoded in XML [1]. Although these documents were limited in size, they provided a less regular and redundant data source, falling into the category of document-centric documents. Measurements on these data sets were included to examine the performance of the developed solution under less optimal conditions. Finally some smaller documents with XML specific features such as entity references were tested to verify the standard conformance. Because classes for entities and associated references were not implemented, these needed to be resolved by the parser. This behaviour is in conformance with the W3C recommendation [19].

5.2 Results

The memory consumption of the DDOM was compared with those of the two most dominant implementations, Xerces [24] and Crimson [4]. All three implementations support JAXP [11] integration, hence only the property controlling the implementation to be used needed to be changed for the different measurements. However, all measurements were performed on a "clean" virtual machine, i.e. in separate runs. This was done to remove artifacts caused by incomplete garbage collection processes. The results for the DNS table can be found in tables 1 and 2. In addition figure 3 shows the size of the XML text file. Its compressed size using maximum gzip compression is shown as practical measure of its entropy.

The DDOM implementation showed the lowest memory consumption in all measurements performed. However, parsing was quite slow for large documents. This was caused by the actual parser implementation, which used a linear search

Table 1. Absolute and relative memory usage of different DOM implementations for the domain name server database

Entries	DDOM	Xerces (rel.)	Crimson (rel.)
100	92.0 KB	312.8 KB (3.4)	195.0 KB (2.1)
1,000	461.1 KB	1602.4 KB (3.5)	1916.3 KB (4.2)
10,000	3.2 MB	14.9 MB (4.6)	19.3 MB (6.0)
100,000	25.2 MB	141.7 MB (5.6)	191.0 MB (7.6)

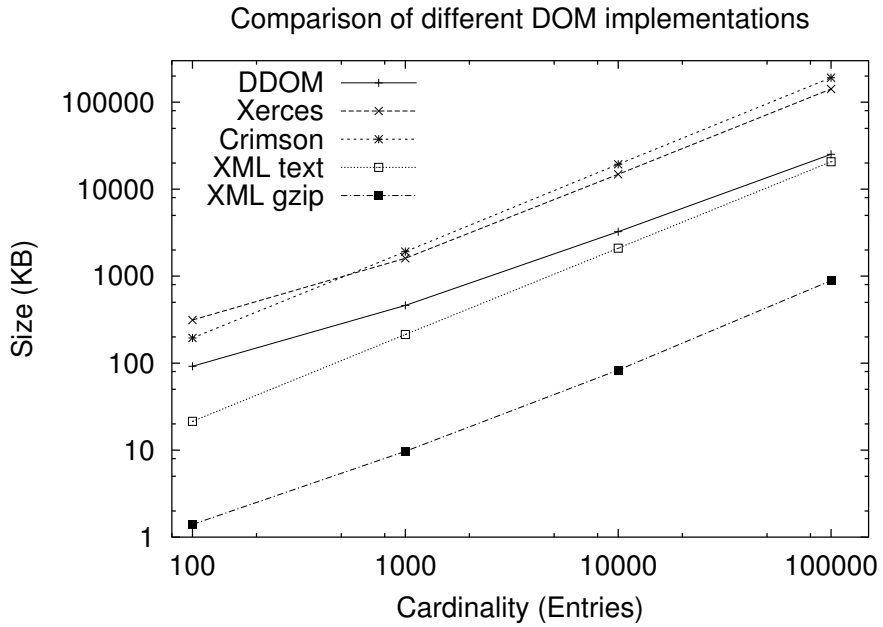

Fig. 3. Memory consumption of the different DOM implementations for the domain name server database. Note that the textual representation uses 8 bit character encoding whereas all DOM implementations store characters using 16 bits according to the W3C recommendation.

algorithm for every new entry in the dictionary, resulting in a $\mathcal{O}(n^2)$ run-time performance. This could be overcome by a better parsing process using an $\mathcal{O}(n \log n)$ algorithm without increasing the memory footprint as described in [13]. Both Crimson and Xerces showed linear growth in memory consumption for files above a certain threshold. The memory consumption of the DDOM grew less than linear, but depended on the redundancy present in the data.

In addition to the size of the actual input its format can also influence the memory requirements. In the absence of a DTD, additional white spaces between elements can not be detected, because they are indistinguishable from character data. Measurements for the DNS database using an XML file containing 10,000 entries were repeated using three different formats. By default, measurements were performed in the absence of a DTD, using XML text files formatted with white space to facilitate legibility. For the second run a suitable DTD was provided, allowing to detect the unnecessary white space in the otherwise unchanged XML file. The final run was performed in absence of a DTD but with no white spaces present in the source document. Different formatting of the XML source files and the presence or absence of a DTD had only a minimal effect on the DDOM as the additional white spaces are highly redundant and can be stored with minimal effort. Xerces and Crimson memory consumption suffered significantly if white spaces are to be conserved as shown in table 2.

Table 2. Absolute and relative memory usage of different DOM implementations for the domain name server database formatted using different styles

Format	DDOM	Xerces (rel.)	Crimson (rel.)
standard	3.24 MB	14.85 MB (4.6)	19.31 MB (6.0)
with DTD	3.32 MB	9.58 MB (2.9)	— (—)[a]
compact	3.19 MB	9.07 MB (2.8)	12.85 MB (4.0)

[a] caused runtime exception

On document-centric XML with low redundancy, the chosen approach had only minimal impact. Savings of about 20 % to 30 % were achievable, mostly due to the compression of the actual mark-up tags. Table 3 shows the results for the Shakespearean plays. The DDOM is capable of handling this kind of document-centric files without modification, although they do not conform to the assumptions stated earlier.

6 Conclusions and future work

The prototype implementation showed good compression ratios compared with standard DOM implementations. Real-world examples showed saving of about 30 % to 80 % in terms of memory requirements. Applications that could be expected to profit from this involve data-centric documents that need to be accessed randomly and repeatedly for processing. However, like most compression

Table 3. Absolute and relative memory usage of DDOM and Xerces DOM implementation for some of Shakespeare's plays

Play	XML file	DDOM	Xerces (rel.)
Macbeth	159.3 KB	628.2 KB	870.1 KB (1.39)
Julius Caesar	179.2 KB	739.6 KB	981.6 KB (1.33)
Henry VIII	212.7 KB	825.6 KB	1092.8 KB (1.32)

techniques, the designed structure suffers in the presence of key entries. As these are unique by definition, the application of compression actually results in higher memory consumption and some performance problems. Such entries can be detected automatically and then be excluded from the compression process.

More work will be required to analyse the query performance of this and other solution. The lack of a suitable query engine did not allow the resolution of queries nor the measurement of their performance. Although it is our belief, that this will be comparable or better to those of other implementations and may be significantly better for certain types of queries, this needs to be verified.

The success or failure of this and other implementations will depend heavily on the way future applications and middleware, especially XQuery engines, will store and access data. So far the rather complicated DOM standard seems to be more a hindrance than a unifying model to access XML data. Large, data-centric applications typically push the workload into a RDBMS whilst supplying only a lightweight front-end, translating XQuery expressions into equivalent SQL queries. Native XML databases tackle the problems inherent to this approach but are just entering the arena. They will depend on compact and easy to access representations of the contained data.

Currently we are using 32 bit integer numbers as tokens within the structure arrays. This limits every domain to a total of 2^{32} entries, although this limit would be hard to reach using current 32 bit implementations of the JVM. On the other end of the scale, in domains with only very few items, this represents a significant waste. Ways to improve the number of bits used per token need to be found. Within the dictionaries every word is stored as a separate *String* object. Using only one or a few *StringBuffer* instead and just noting the starting points of individual entries should allow for further savings. For long character data sections, conventional compression methods could push compression even further.

Given the compact, tokenised nature of the data representation we anticipate that query performance will exceed that of uncompressed DOM representations. String matching will be required only once per query and can be performed within the dictionary structure. In contrast conventional implementations have to compare strings once per node visited. Optimisation of the dictionaries is possible and should allow for better parsing and querying performance. Further improvements in compression may be also achieved by packing the tokens more tightly. Other performance improving techniques from the relational database world, especially indexing, could be applied to the generated structure.

We are currently planing to adapt a suitable query system to run against our DDOM representation to evaluate the performance. This will enable us to identify the circumstances under which dictionary based compression of XML will provide a basis for optimal query processing.

References

1. J. Bosak. Shakespeare's plays encoded in XML, 1996-1998. `http://www.ibiblio.org/xml/examples/shakespeare/`.
2. Tom Bradford. dbXML XML database application server version 0.4. dbXML core technical specification, dbXML Group, L.L.C., Dec 2000. "Xindice" since Dec 2001.
3. W. Paul Cockshot, Douglas McGregor, and John Wilson. High-performance operations using a compressed database architecture. *The Computer Journal*, 41(5):283–296, 1998.
4. The Apache XML project: Crimson XML parser. `http://xml.apache.org/crimson`.
5. Alin Deutsch, Mary Fernandez, et al. Querying XML data. *Data Engineering Bulletin*, 22(3):10–18, Sep 1999.
6. Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *Data Engineering Bulletin*, 22(3):27–34, Sep 1999.
7. H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. Knowledge Data Eng.*, 4:509–516, 1992.
8. M. Girardot and N. Sundaresun. Millau: An encoding format for efficient representation and exchange of XML over the web. In *Proceedings of the 9th WWW Conference*, pages 747–765, Amsterdam, Netherlands, May 2000.
9. Gerald Huck, Ingo Macherius, and Peter Frankhauser. PDOM: Lightweight persistency support for the document object model. In *Proceedings of the 1999 OOPSLA Workshop "Java and Databases: Persistence Options"*, Denver, CO, USA, Nov 1999.
10. IEEE Computer Society. *Proceedings of the 17th International Conference on Data Engineering, April 2–6, 2001, Heidelberg, Germany*, Heidelberg, Germany, 2001.
11. Java Community Process^{SM}: Java APIs for XML Parsing (JAXP). Project homepage. `http://java.sun.com/xml/xml_jaxp.html`.
12. H. Liefke and D. 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. ACM, 2000.
13. Mathias Neumüller. Compression of XML data. MSc thesis, University of Strathclyde, Glasgow, Scotland, UK, Sep 2001.
14. A. Renner. XML data and object databases: The perfect couple? In *ICDE 2001* [10], pages 143–148.
15. Harald Schöning. Tamino – a DBMS designed for XML. In *ICDE 2001* [10], pages 149–154.
16. Jayavel Shanmugasundaram, Kristin Tufte, et al. Relational databases for querying XML documents: Limitations and opportunities. In Malcolm Atkinson, Maria E. Orlowska, et al., editors, *VLDB 1999*, pages 302–314. Morgan Kaufmann, 1999.
17. Jack Shirazi. *Java Performance Tuning.* Java series. O'Reilly, Sebastopol, CA, USA, 2000.

18. WAP Forum members IBM, Motorola and Phone.com. WAP binary XML content format. W3C Note, Jun 1999. `http://www.w3.org/TR/wbxml/`.
19. World Wide Web Consortium. *Document Object Model (DOM) Level 1 Specification Version 1.0*, W3C recommendation 1 october, 1998 edition, 1998. `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001`.
20. World Wide Web Consortium. *Appendix C: Java Language Binding*, 2000. http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/java-language-binding.html.
21. World Wide Web Consortium. *Document Object Model (DOM) Level 2 Specification Version 1.0*, W3C recommendation 13 november, 2000 edition, 2000. `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113`.
22. World Wide Web Consortium. *XML Schema Part 2: Datatypes*, W3C recommendation 02 may 2001 edition, 2001. `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502`.
23. World Wide Web Consortium. *XQuery 1.0: An XML Query Language*, W3C working draft 7 june, 2001 edition, 2001. `http://www.w3.org/TR/2001/WD-xquery-20010607`.
24. The Apache XML project: Xerces XML parser for java. `http://xml.apache.org/xerces-j`.
25. Yasuo Yamane, Nobuyuki Igata, and Isao Namba. High-performance XML storage/retrieval. *Fujitsu Sci. Tech. J.*, 36(2):185–192, Dec 2000.
26. Ching-Long Yeh. A logic programming approach to supporting the entries of XML documents in an object database. In E. Pontelli and V.S. Costa, editors, *Practical Aspects of Declarative Languages*, volume 1753 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 2000.