PERSISTENT STORAGE AND QUERYING OF COMPRESSED XML DOCUMENTS ON THE WEB

Brian Demmings, Tomasz Müldner, Gregory Leighton, Obidul Islam, Andrew Young
Jodrey School of Computer Science, Acadia University,
Wolfville, NS, Canada B4P 2R6

ABSTRACT

We describe the design and implementation of a Web-based distributed system called TREESTORE, intended for storing compressed XML documents in a relational database. The use of a database is fully portable, requiring minimal changes to application code to substitute one database management system for another. In TREESTORE, compressed XML documents are shredded into a fixed number of relational tables. Besides the storage capability, TREESTORE provides online querying of compressed XML documents without requiring full decompression. Therefore, it is a useful component of any Web application as it combines space-efficient, database-portable access to persistent storage with online querying.

KEYWORDS

XML, compression, WWW, relational databases, XML-enabled databases

1. INTRODUCTION

The Extensible Markup Language (XML) [14] is a popular meta-language that facilitates the interchange and access of data. There are two classes of database management systems (DBMSs) that can store XML documents: *native* XML DBMSs [15], which are designed specifically for management of XML documents; and *XML-enabled* DBMSs [30], which provide an interface on top of relational database engines to handle conversions between XML and relational schemas. Both types of databases have their problems. Unlike *relational* databases, which have matured and provide efficient storage techniques, native XML databases represent a relatively new research area, and hence are not yet fully optimized. XML-enabled databases are non-standard, in the sense that each database vendor has defined different and incompatible interfaces for storing and querying XML, and changes to application code are likely to be required if the DBMS is changed. *Semi-structured* data formats such as XML are not easily mapped to relations for traditional relational databases, due partially to the hierarchical nature of semi-structured data versus the "flat" format of relational data. Several approaches for mapping between XML and relational formats have been proposed by the academic community. These strategies typically provide a facility for converting queries expressed using XPath or XQuery into equivalent SQL queries over the stored data. We survey such approaches in Section 2. To address many of these concerns, some vendors of relational DBMS products are beginning to move to a *hybrid* model, in which XML is treated as a native data type alongside the various data types defined by the SQL standard; see [8].

In this paper, we present the design and implementation of a data management framework called TREESTORE, that can be used to store *schema-less* XML documents in an arbitrary *relational* database using a *compressed format*. Our primary focus is on supporting efficient querying operations which do not require a compressed XML document to be fully decompressed beforehand. We envision that TREESTORE will prove especially useful for mobile applications where client-side storage is limited. Since the user will often wish only to extract a very small portion from the overall data set, the ability to extract the desired subset directly from the compressed domain tends to result in a substantial performance benefit, especially in cases where the data is highly-structured and therefore subject to higher compression rates.

This paper is organized as follows. Section 2 presents related work and our contributions, and Section 3 briefly describes the functionality and implementation of TREESTORE. Section 4 describes experimental results, and Section 5 concludes the paper and outlines future work.

2. RELATED WORK AND CONTRIBUTIONS

In this section, we describe previous work related to storing XML data in databases and compressing XML data. We close the section by presenting our contributions.

2.1 Storing XML in Databases

Native XML databases (hereafter referred to as NXDs) [11, 16, 22] are databases specifically designed to provide storage and manipulation of XML documents. Typically, XPath [34] and/or XQuery [36] are used for accessing documents; due to the current absence of a standardized XML update language, an implementation-specific language (e.g. XUpdate [38]) is used for updating documents in the database. Most NXDs support validation of documents against an associated *document type definition (DTD)* or schema, but some support storage of schema-less XML documents as well.

Storage of XML documents in relational databases requires them to be *shredded* (decomposed) and then mapped to tables. (Another less popular technique, storing an XML document as a single character large object value, frequently requires less storage space, yet is inefficient with respect to querying). There has been significant research on these techniques, which can be classified into several categories. S*chema-based approaches* assume that XML documents are valid according to a DTD or XML Schema [35]. For example, methods using this technique have been described by Shanmugasundaram et al [27]. This is an early paper that presents a prototype system which converts XML documents to relational tuples, translates semi-structured queries over XML documents to SQL queries over tables, and converts the results to XML. Tian et al [31] compare various strategies for storing DTD-based XML data to facilitate efficient query processing, including text files, RDBMS, and Object Manager. Amer-Yahia et al [2] describe mapping strategies to shred and load the XML Schema-based data into relational tables, and to translate XML queries over the original data into equivalent SQL queries over the mapped tables. LegoDB [4] is, an XML storage mapping system that automatically finds an efficient relational configuration for a target XML application.

The second category of approaches is *schema-oblivious*, meaning there is no requirement for XML documents to be valid according to a schema; examples of this strategy are outlined in [13] and [25]. There are also *hybrid approaches,* which may or may not employ a schema (e.g. STORED [10]). Recent versions of all commercial databases provide XML-to-relational data interfaces, which typically implement one of the techniques described above; (e.g. DB2 XML Extender [7] and Oracle XML DB [23]). However, these approaches have serious limitations in terms of capability and performance. Such XML-to-relational interfaces are not standardized, and therefore switching from one relational database to another may require major modifications of the application code, introducing an undesirable "tight coupling" between the application and data layers. In addition, the number of tables required by the database to store XML documents may grow as various XML documents are inserted.

2.2 XML Compression

In this section, we briefly review research on XML compression; for more details, see [17,21]. XML compressors can be categorized as follows:

- Is the compressor *XML-aware* (i.e. can it take advantage of understanding the underlying structure of XML documents during the compression process)?
- Is the compressor *single pass* (i.e. does it require only a single pass for compression and decompression)?
- Is the compressor *online* or *offline*? (An online compressor does not require the full compressed document to begin the decompression process)
- Is the compressor *query-able* (i.e. is it possible to query the document without full decompression)?

XML-aware compressors are superior for compressing XML documents over standard compressors such as gzip [14]. Here, we describe only a few XML-aware compressors that satisfy some of the above conditions. XMill [18] represents the pioneering work in the area of XML-conscious compression, but it is an *offline* non-query-able compressor. XMLPPM [5] is an online *non*-query-able compressor, but it achieves the highest degree of compression. XGRIND [32] and XPRESS [19] were the first query-able compressors, but neither is a single-pass compressor. XQueC [37] provides a compressed storage for XML data that can be tailored to a specified query workload, offering querying support for a subset of the XQuery [36] standard

over compressed data. However, the scheme used in XQueC relies on custom structures for storage and indexing, an approach which is not readily applicable to efficient streaming of XML documents over a network. Finally, TREECHOP [17], the compressor used in our proposal, is an online, single-pass, query-able compressor (described in further detail in Section 3.1.)

2.3 Contributions

To our knowledge, there have been no previous attempts to provide the solution proposed in this paper: a Web-based framework enabling the storage and querying of *compressed* XML data. Specifically, this paper describes the design and implementation of a distributed XML data management framework called TREESTORE, which can be used to store schema-less XML documents in a relational database over the WWW. TREESTORE is fully portable, meaning it can be interfaced with an arbitrary relational database supporting JDBC, and XML documents are stored in a relational database in a compressed form, using only four tables. TREESTORE is a *distributed* application that supports desktop clients storing and retrieving XML documents, and querying these documents. In addition, less powerful, thin mobile clients can query XML documents (in this case, the processing is done on the *server side*).

The TREESTORE client compression process and the TREESTORE server operate in an *online* fashion; the server does not have to wait until the entire compressed document has been received before it begins processing. XML data stored in the database can be subsequently queried *without having to fully decompress it*. Querying can span over multiple documents. Therefore, TREESTORE combines space-efficient persistent storage with efficient querying. Finally, TREESTORE additionally supports efficient conversion of TREESTORE managed documents into TREECHOP-compressed documents as the document is being retrieved from the database (without fully reconstructing the XML), so that the output can be piped into another application, or the compressed document can be stored by the client.

Comparing TREESTORE and DB2 with XML Extender [7] has shown that TREESTORE's performance, while worse than DB2 on insert cases, is comparable to DB2, or even faster in retrieval applications. This makes it a good candidate for use in Web applications or in a networked environment requiring storage of XML documents.

3. TREESTORE

We begin this section by briefly summarizing the operation of TREECHOP, the compression scheme used in TREESTORE. This is followed by a description of the functionality and a brief description of the Java-based implementation.

3.1 TREECHOP

TREECHOP is a lossless XML-aware compressor which supports querying over compressed data. The compression process begins by conducting a SAX-based [28] parsing of the XML document. To avoid building an in-memory representation of the entire document tree, tokens returned by the parser are re-interpreted as tree nodes and then written out to the compression stream in depth-first order. As node information is added to the compression stream, it is compressed using gzip. Each non-leaf node v is assigned a binary codeword. This codeword succinctly identifies a node's location within the document tree along the parent-child and sibling axes, and additionally indicates the node type (e.g. element, attribute, or #PCDATA). To allow the decoder to set about decoding the node without waiting for the entire codeword, information about the node is written to the encoding stream *immediately* after being assigned a codeword. Each non-leaf node is transmitted as a 3-tuple (L, C, D), where L is a byte indicating the bit length of the codeword; C is the codeword assigned to the node, consisting of a sequence of $\lceil L / 8 \rceil$ bytes; and D is textual data stored in the node. A reserved byte value is used to indicate to the decoder that raw character data is forthcoming in the encoding stream; once D has been transmitted, a second reserved byte value is used to signal the end of the character data sequence. For the second and subsequent occurrences of a particular codeword, only the 2-tuple *(L, C)* is transmitted, as the decompressor is able to infer the value of D at that point. Leaf nodes are transmitted in the same manner as D, described above.

Since tree node encodings are written to the compression stream in depth-first order, it is possible for the decompressor to regenerate the original XML document incrementally. A code table stores $(L, C) \rightarrow D$ mappings for previously-encountered tree nodes. In addition, a stack is employed to maintain proper nesting of elements during the decompression process. A more detailed description of the TREECHOP compression and decompression routines is provided in [17].

3.2 TREESTORE Functionality

TREESTORE provides *desktop clients* with storage, retrieval, and querying operations on XML documents, and it also provides thin, *mobile clients* with querying operations. Fig. 1 shows the overview of TREESTORE components. In this figure, TS stands for a TREESTORE stream and TC stands for a TREECHOP-compressed stream. The main difference between these streams is that while the TS stream contains gzip'ed data and other non-compressed items (such as codewords), the TC stream is a single, gzip'ed stream. Details of each of these operations are discussed below.

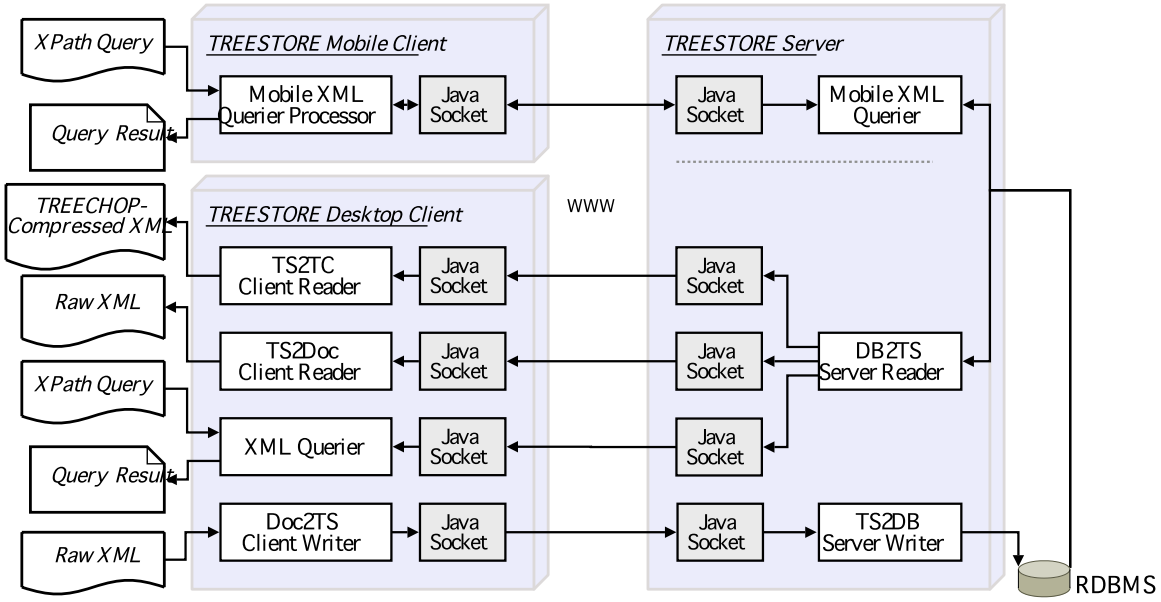


Fig. 1. High-Level Overview of TREESTORE.

3.2.1 Storage and Transmission of XML Documents

To store an XML document, the client component reads this document, shreds it using the technique developed for TREECHOP, then compresses its data (using gzip) and sends the resulting stream to the server component, which stores the shredded document in a relational database. This process offloads the compression process from the server to the client, thereby improving the scalability of the TREESTORE server.

On the receiving end, the TREESTORE server reads chunks (without decompressing), forms appropriate INSERT SQL statements and invokes these INSERT statements to store the received data in a relational database. Using JDBC, TREESTORE can connect to any relational database that supports SQL and has a JDBC driver, and so the underlying database can be changed without requiring the modification of application code outside the TREESTORE layer.

Finally, note that since mobile device clients have limited memory and processing capabilities, they are not able to store XML in TREESTORE.

3.2.2 Retrieval of XML Documents

To retrieve an XML document, the client component sends a request to the server component, which composes the TS stream (i.e. the shredded and compressed document) from the relational database and sends

it back to the client, which applies the decompression routine to restore the original XML document. As an alternative to the last step, the client component may instead convert the TS stream to a TREECHOP-compressed stream and "pipe" the resulting stream into another application, which can read this stream, or the stream may be decompressed and then piped.

3.2.3 Querying of Compressed XML Documents

By using the XML Query Processor component (see Fig. 1), a remote client can query an XML document stored in a database, without first completely downloading and decompressing the document. The current implementation of TREESTORE supports querying using absolute and relative XPath expressions XPath expressions, which may contain wildcards and predicates. These queries can be divided into *simple* and *conditional* query classes; *simple* queries may or may not contain wildcards, but must not contain predicate conditions; *conditional* queries support the notation supplied by simple queries, and additionally include predicates; see Table 1.

Table 1. Simple and Conditional Queries.

#	Simple	Conditional
1	`/employees/employee/fname`	`/employees/employee[fname='Bob']/fname`
2	`//employee`	`//employee[@id='1234']`
3	`//employee/lname`	`//employee[fname='Charles']/lname`
4	`/employees/employee/*`	`/employees/employee[fname='Randy']/*`

When a query match is made, the textual-content of the matched node is returned. For instance, if the query 3 from Table 1 is made, all last-names for employee's with the first name 'Charles' are returned.

3.2.4 TREESTORE-to-TREECHOP and Back Again

The close relationship between TREESTORE and TREECHOP allows for efficient conversion of TREESTORE-stored XML into streaming, query-able TREECHOP-compressed XML. The XML document decomposition of TREESTORE relies on the compression and decompression strategies developed for the TREECHOP compressor and decompressor [21].

When reading a TREECHOP-compressed stream, TREESTORE reuses the code-words and data from the TREECHOP stream. Important context information for each node, including first-child, following-sibling, data, code-word, and path are maintained by using a context-stack. The resulting information is stored in TREESTORE format in the relational database.

Using the first-child, following-sibling relationships, and codewords and data of stored nodes, TREESTORE can efficiently emit a TREECHOP-compressed XML document.

3.3 TREESTORE Architecture

In this section we discuss the architecture of TREESTORE, shown in Fig 1. The main components of TREESTORE are:

- TS2TC Client Reader, which inputs the TS stream, and outputs a TC stream;
- TS2Doc Client Reader, which inputs the TS stream and outputs the raw XML;
- Doc2TS Client Writer, which inputs a raw XML document and outputs the TS stream;
- Desktop XML Query Processor, which inputs the XPath Query and sends the request to the server; upon receiving the data, it forms the query result using a registered listener object, which carries out a defined sequence of operations on each node encoding as it is received from the server;
- Mobile XML Query Processor, which preprocesses the XPath Query, sends it for complete processing to the server's Mobile XML Query Processor, and receives the decompressed query results from the server;
- DB2TS Server Reader, which reads the compressed document from the database and outputs the TS stream; and
- TS2DB Server Writer, which reads the TS stream and writes the compressed document to the database.

The database schema used by TREESTORE is shown in Fig. 3.

Node_table(*ID*, *docID, fChild*, *fSibling, codeword, data*)
Metadata_table(*xmlIdentifier, docID, encoding, prologue*, *epilogue*)
Code_table(*path*, *codeword*, *codeLength*, *docID*)

Freenodes_table(*startNodeID, finishNodeID*)

Fig. 2. Relational Schema used in TREESTORE.

TREESTORE stores documents by associating a unique name (provided by the client) with each stored document. The unique name becomes the xmlIdentifier to which the server assigns an internal unique integer document identifier (docID) to each document, to permit timely lookup of nodes associated with a particular document. The server does not permit the client to overwrite existing documents with the same xmlIdentifer. The *node table* stores the tree structure of all documents, using a first-child, following-sibling format [1] along with the data and codeword for each node. Each XML node encountered during the depth-first traversal of the XML document is assigned a unique node ID, which is stored, along with the identifiers for the first-child and following-sibling nodes. The *metadata table* associates the xmlIdentifier with the ID of the document, the encoding format used for the original document (e.g. UTF-8), and the document's prologue and epilogue. The *code table* stores the path-to-codeword associations formed by the TREECHOP encoding of the document. The last table, called the *freenodes table*, is not used to store any document-specific information; instead it is used to keep a listing of unused node ID's so that TREESTORE can efficiently assign unique node identifiers for all nodes in the database.

For example, consider an XML document that contains a list of employees. The resulting decomposition into relational tables using TREESTORE is shown in Fig. 4. Note that for the sake of simplicity, the codewords in this example are artificial, only the code table and node table are shown, the data is uncompressed and the docID is missing and can be assumed to be 1 for this example.

Example XML

```
<employees>
  <employee id="1">
    <fname>Bob</fname>
    <lname>Jay</lname>
    <job>Cook</job>
  </employee>
  <employee id="2">
    ...
  </employee>
  ...
</employees>
```

Code Table

PATH	CODE
/employees	a
/employees/employee	b
/employees/employee/@id	c
/employees/employee/fname	d
/employees/employee/lname	e
/employees/employee/job	f

Node Table

ID	CODE	DATA	FCHILD	FSIB.
1	a	-	2	-
2	b	-	3	7
3	c	"1"	-	4
4	d	"Bob"	-	5
5	e	"Jay"	-	6
6	f	"Cook"	-	-
7	a	-	8	-
8	b	"2"	-	9
9	…	…	…	…

Fig. 3. Example XML Document Decomposition

As illustrated in Fig. 3, a constituent node within an XML document can easily be accessed without traversing the entire document as would be necessary if working with the raw XML. By using the unique docID all nodes for a document can be retrieved from the server, preserving the proper parent-child relationship between nodes and the document.

4. EXPERIMENTAL RESULTS

This section presents the results of experiments that were carried out to evaluate the effectiveness of using TREESTORE by comparing time efficiency for inserting and retrieving operations between the use of DB2 XML Extender [7] and TREESTORE interfaced with DB2 version 8.1.2. These experiments were performed

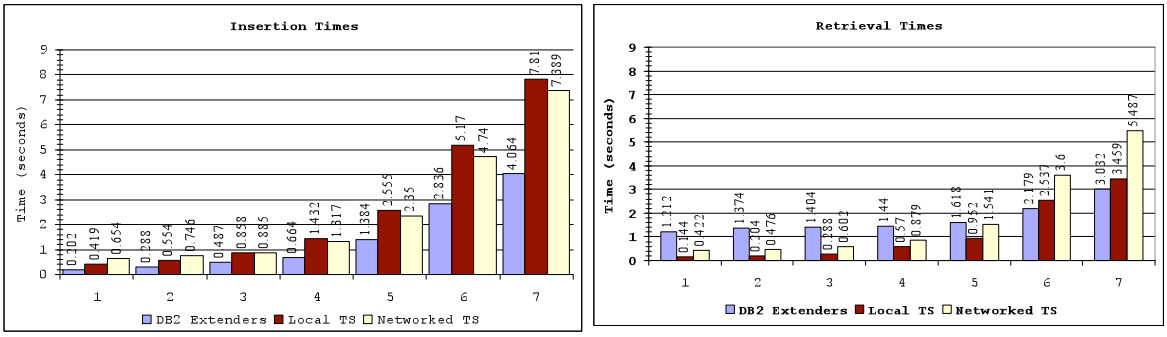
on Dell D600 laptops with 1400 MHz Pentium processors and 512 MB of RAM, running Windows XP. In all cases, the database ran on the server-side, meaning that the client was freed from the responsibility of managing the database. TREESTORE is implemented in Java, thus, SQL access to the DB2 server was provided by JDBC, and the DB2 XML Extender approach relied on the XML Collection method. All network communications with TREESTORE and DB2 Extenders were performed over a wireless 54MB network to better simulate real-world scenarios.

For all tests the same set of documents were used. Each document was a collection of employee records, with each record having an employee id number as an attribute, and nested elements to store first name, last name, age, job title, and phone number. Table 2 describes characteristics of the test documents.

Table 2. Test Documents

	XML Documents						
Document:	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Employees:	50	100	150	250	500	1000	1500
Size (Bytes):	8722	17292	25920	43208	86370	171491	258659
Elements:	301	601	901	1501	3001	6001	9001
Attributes:	50	100	150	250	500	1000	1500

The set of documents illustrated in Table 2 was inserted and retrieved from each of the databases 50 times and the average insertion and retrieval times were calculated. The results of these tests are shown in, Fig. 4 a. and b., respectively.



a. b.

Fig. 4. Insertion and Retrieval Test Data

Note that the DB2 XML Extender does not have to deal with the significant computation associated with applying compression. In the "Local TS" entries above, all operations including running DB2, decomposing the XML document, and inserting into the database occurred on the same computer. Thus, it is not surprising that the networked version was more efficient, because the workload of decomposing and compressing the document are handled on the client side, which is an incentive to distributing applications that rely on TREESTORE.

During retrieval, the results were much more favorable. As shown in Fig. 4.b, a computer running a local version of TREESTORE was able to be much better than DB2 at handling and storing small files. TREESTORE's performance is comparable with DB2, even on the larger documents (such as the 1500 employee document).

These results indicate that while TREESTORE is not necessarily better from an insertion stand-point as compared with DB2, it is time efficient for retrieval as compared to the XML Extender approach, which makes it beneficial for networked multi-client applications. Note that the additional time required for networked TREESTORE may be the result of using a wireless infrastructure. Furthermore, a significant advantage to using TREESTORE is that it distributes the load of document composition/decomposition and storage, which will scale more effectively, and that it is cross-platform and does not rely on a proprietary database interface.

5. CONCLUSIONS AND FUTURE WORK

This paper described TREESTORE, a system for storing schema-less XML documents in arbitrary relational databases. TREESTORE operates online and supports querying of compressed XML documents without requiring full decompression. TREESTORE combines a space-efficient, database-portable access to persistent storage with querying capability; therefore, it is a useful component of any Web application requiring this functionality.

Our experiments showed that, in addition to DB2 [8] TREESTORE is *compatible* with a multitude of popular databases, including Oracle [23], Derby [9], MySQL [20] and PostgreSQL [24].

We have compared the time efficiency of TREESTORE using DB2 version 8.1, versus the use of this version of DB2 with XML Extender. Experiments showed that TREESTORE's performance, while worse than DB2 on insertions, is comparable to DB2 (or faster) for retrievals. This makes it a good candidate for use in Web applications or in a networked environment requiring storage of XML documents. There are many additional advantages of using TREESTORE. First, it can be used for arbitrary relational databases as compared to XML-enabled databases. TREESTORE does not mandate the user to specify a mapping, a cumbersome process required by the DB2 XML Extender. Second, TREESTORE uses only four tables, while XML-enabled databases may create an arbitrarily large number of tables. Third, TREESTORE supports storing schema-oblivious XML documents. In conclusion, there is only a negligible cost of providing a fully portable connection for storing XML documents in a relational database.

We note that in typical database-backed Web applications, there is a high ratio of retrievals to insertions/updates. This makes TREESTORE an ideal candidate for use in Web applications or in a networked environment requiring storage of XML documents.

There are two main areas for future work on TREESTORE: to find ways to improve performance, and to remove existing restrictions. Currently with each data insert, a separate gzip compressor is used. This has the downside that if data is small enough, the overhead from compression may negate (or worse) the benefits of compression. To optimize this, the current idea is to use an "inlined" compressor per-document permitting TREESTORE to benefit from less overhead and improve overall compression. Future versions of TREESTORE could make use of a DTD or Schema to improve the efficiency of code-word generation. For instance, if it is known that a node has one possible child, a very simple code is required for that child.
Finally, with the current implementation, simultaneous client interactions with a remote TREESTORE server may result in various concurrency issues.

REFERENCES

[1] Aho, A. V., Hopcroft, J.E., and Ullman, J. D., "Data Structures and Algorithms". Addison-Wesley, Mass, 1983.

[2] Amer-Yahia, S., and F. Du, J. Freire: A comprehensive solution to the XML-to-relational mapping problem. WIDM 2004: 31-38

[3] Arion, A., Bonifati, A., Costa, G. et al. "Efficient query evaluation over compressed XML data," in 2004 Proc. Of EDBT, pp. 200-218.

[4] Bohannon, P., Freire, J., Roy, P., and Simeon, J. 2002. From XML Schema to Relations: A Cost-based Approach to XML Storage. In Proceedings of the 18th International Conference on Data Engineering. IEEE Computer Society, San Jose, CA, USA, 64–75.

[5] Cheney, J. "Compressing XML with multiplexed hierarchical PPM models," in 2001 Proc. IEEE Data Compression Conference, pp. 163-172.

[6] Cleary J.G. and Witten I.H., "Data compression using adaptive coding and partial string matching," IEEE Trans. Comm., vol. 32, no. 4, pp. 396-402, Apr. 1984.

[7] DB2 UDB Extenders for iSeries, Retrieved March 2006, http://www.common.be/pdffiles/16042002DB2XMLExtender.pdf.

[8] DB2 9 - pureXML™ and storage compression, Retrieved July 2006. http://www-306.ibm.com/software/data/db2/v9/

[9] Derby. Retrieved July 2006. http://db.apache.org/derby/

[10] Deutsch, A., Fernandez, M., and Suciu, D. 1999. Storing Semistructured Data with STORED. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data. ACM, Philadelphia, Pennsylvania, USA, 431–442.

[11] eXist, Retrieved March 2006. http://exist.sourceforge.net/.

[12] Extensible Markup Language (XML) 1.0 (3rd ed.), Retrieved March 2006. http://www.w3.org/TR/REC-xml.

[13] Florescu, D. and Kossmann, D. 1999. Storing and Querying XML Data Using an RDBMS. IEEE Data Engineering

Bulletin 22, 3 (September), 27–34.
[14] The gzip home page, Retrieved March 2006. http://www.gzip.org.
[15] Introduction to Native XML Databases, Retrieved March 2006. http://www.xml.com/pub/a/2001/10/31/nativexmldb.html.
[16] Jagadish H. V et al. « TIMBER: A Native XML Database". Retrieved July 2006. http://www.eecs.umich.edu/db/timber/files/timber.pdf
[17] Leighton G., Müldner T., Diamond J. "TREECHOP: A Tree-based Query-able Compressor for XML," Jodrey School of Computer Science Technical Report 2005-005, http://cs.acadiau.ca/technicalReports/files/tr-2005-005.pdf.
[18] Liefke H. and Suciu D., "XMill: an efficient compressor for XML data," in 2000 Proc. ACM SIGMOD Int'l Conf. on Management of Data, pp. 153-64.
[19] Min J-K., Park M-J. and Chung C-W., "XPRESS: a queriable compression for XML data," in Proc. 2003 ACM SIGMOD Int'l Conf. on Management of Data, pp. 122-33.
[20] MySQL. Retrieved July 2006. http://www.mysql.com/
[21] Müldner, T., Leighton, G. and J. Diamond 2005. Using XML Compression for WWW Communication, IADIS International Conference WWW/Internet
[22] Natix C++ XML Base System. Retrieved July 2006. http://pi3.informatik.uni-mannheim.de/natix.html.en
[23] Oracle Database. Retrieved July 2006. http://www.oracle.com/index.html
[24] PostGreSQL, Retrieved March 2006. xpsql. http://gborg.postgresql.org/project/xpsql/projdisplay.php
[25] Schmidt, A., Kersten, M. L., Windhouwer, M., and Waas, F. 2000. "Efficient Relational Storage and Retrieval of XML Documents. In Proceedings of the 3rd International Workshop on the Web and Databases. ACM, Dallas, Texas, USA, 137–150.
[26] Shanmugasundaram, J. E. J. Shekita, J. Kiernan, R. Krishnamurthy, S.Viglas, J. F. Naughton, I. Tatarinov. "A General Techniques for Querying XML Documents using a Relational Database System." SIGMOD Record 30(3): 20-26 (2001)
[27] Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D. J., and Naughton, J. F. 1999. Relational Databases for Querying XML Documents: Limitations and Opportunities. In Proceedings of the 25th International Conference on Very Large Data Bases. Morgan Kauffman, Edinburgh, Scotland, UK, 302–314.
[28] Simple API for XML (SAX), Retrieved March 2006. http://www.saxproject.org.
[29] SOAP, Retrieved March 2006. http://www.w3.org/TR/soap/.
[30] Storing XML in Relational Databases, Retrieved March 2006. http://www.xml.com/pub/a/2001/06/20/databases.html?page=2.
[31] Tian, F., and D. J. DeWitt, J. Chen, C. Zhang "The Design and Performance Evaluation of Alternative XML Storage Strategies," ACM SIGMOD Record Volume 31 , Issue 1 (March 2002)
[32] Tolani P.M. and Haritsa J.R., "XGRIND: a query-friendly XML compressor," in Proc. 2002 Int'l Conf. on Database Eng., pp. 225-34.
[33] Young, A. "Storing Large XML Documents in Relational Databases," Honours thesis, Acadia University, 2006.
[34] XML Path Language (XPath), Retrieved March 2006. http://www.w3.org/TR/xpath.
[35] XML Schema Language, Retrieved July 2006. http://www.w3.org/XML/Schema
[36] XQuery 1.0: an XML query language, Retrieved March 2006. http://www.w3.org/TR/xquery.
[37] XQueC. Retrieved March 2006. http://www.icar.cnr.it/angela/xquec/XML and Databases, Retrieved March 2006.
[38] XUpdate. Retrieved March 2006. http://xmldb-org.sourceforge.net/xupdate/