Updates of Compressed Dynamic XML Documents

Tomasz Müldner[1], Christopher Fry[1], Jan Krzysztof Miziołek[2], Tyler Corbin[1]

[1] Jodrey School of Computer Science, Acadia University, {tomasz.muldner, 062181f, 094568c}@acadiau.ca

[2] IBI AL, University of Warsaw, jkm@ibi.uw.edu.pl

Abstract: Because of the ever-growing number of applications that send numerous and potentially large XML files over networks there has been a recent interest in efficient updates of XML documents. However all known approaches deal with uncompressed documents. In this paper, we describe a novel XML compressor, XSAQCT designed to improve the efficiency of querying and updating XML documents with minimal decompression in a network environment.

1. Introduction

Although XML is now the de facto data standard for Web services, as well as for encoding semi-structured data, its verbose nature and the resulting large sizes of the underlying XML documents adversely affects various network-based XML services. One such service is remote XML storage and transmission of XML files to other nodes in the network (in particular, for devices with limited memory such as mobile devices and wireless sensors, which are becoming ubiquitous in today's society). Furthermore, while querying large XML documents is a commonly-executed operation, the high execution time and memory requirements of query tools (e.g. XQuery [XQ08]) severely limits their usefulness for very large documents; similar issues exist for XML update operations. Therefore, improving the efficiency of XML storage and processing is a key research challenge. However, practically all existing approaches to update operations are limited to updates of *uncompressed* documents, which suffer from scalability issues for large XML documents.

In this paper, we describe XSAQCT (pronounced *exact)*, which supports querying and updating XML documents using minimal decompression. We envision that XSAQCT will prove especially useful for mobile applications where client-side storage and CPU speed are limited. Less powerful, thin mobile clients can query or update potentially very large XML documents stored on a server without first completely downloading and decompressing the document (in this case, the processing is done on the server side).

The results described in this paper build directly on our recent work, based on intermediate representation of the XML structure in the form of an annotated tree, where each tree node is labeled by an *annotation* representing partial information on document structure via an integer sequence (the entire annotated tree represents the structure of the complete XML document). Specifically, our approach entails (1) encoding the document structure in an annotated tree; (2) storing the annotated tree and the document contents in separate containers; and (3) applying back-end compressors to the containers. We designed, implemented and tested a queryable XML compressor, called XSAQCT [M09], which relies on this representation and supports querying with lazy decompression. XSAQCT uses a single SAX pass of the input document and does not require building an in-memory representation of the document (such as a DOM tree). Consequently, our technique is applicable to processing very large XML documents and to streaming. We compared XSAQCT and TREECHOP [L05], the only other queryable XML compressor available for testing, on a standard XML corpus [W10]. Our findings demonstrated that XSAQCT achieves 50% to 80% higher compression ratio and, on average, 50% faster query time than TREECHOP. This improvement over TREECHOP did not sacrifice time efficiency as both compressors have a similar compression/decompression time. The annotated tree is designed to support extensions of XSAQCT to include updates. In this paper, we describe such an extension, in which the basic scenario for updating a compressed XML document considers multiple insert and deletes operations, interleaved with querying. To avoid potentially costly (in terms of the CPU and memory) updates of the corresponding annotations, each annotated tree node has a list of pending update operations (referred to as *pending list*).

Updates in the pending list will only be applied when a threshold is reached or when flushed by the user. A query operation involving a single node can be initiated without forcing a flush of the pending list. In related approaches, maintaining the pending list is sometimes referred to as "in-memory modifications", while flushing the list and updating the actual document is referred to as "in-place modifications").

Contributions. Our main contribution is to describe updates of compressed documents, with lazy decompression. We compared XSAQCT with Exist [Ea], BaseX [B10], QuizX [QX], Sedna [Se] and Oracle [Oa] and determined that out of the 32 different trials, XSAQCT achieved the best results, as it was placed first fourteen times, and it placed second fourteen times. QuizX achieved the second best results, as it was placed first fourteen times and it placed second 10 times. Since on average there is a high ratio of retrievals to insertions/updates, and XSAQCT does not force a complete re-writing of the underlying document nor does it force complete decompression, it is an ideal candidate for networked environment requiring storage of XML documents.

This paper is organized as follows. Section 2 describes related work, and Section 3 introduces XSAQCT, and its applications to updating, querying, compressing, and decompression. Section 4 provides results of testing of our compressor, and compares these results with other existing XML compressors. Conclusions and future work are described in Section 5.

2. Related Work

Given XML-related performance issues, there has been interest in devising various XML compression schemes. In many instances it is not practical to decompress an entire XML file to execute an operation such as a query or an update and provide *lazy decompression*, i.e. decompress "as little as possible". As far as query operations are concerned, recently there has been interest in queryable XML compressors that have the potential to improve response time by operating on (partially) compressed data (e.g. XQueC, [A03]). Because of space limitations, here we do not review this work. As far as update of XML documents is concerned, there are various XML updaters (mostly in the area of general databases or database engines), briefly reviewed below. Our review is focused on native XML databases rather than databases, which store XML documents as CLOBS. Since the current version of XSAQCT does not support optimization techniques, such as indexing or caching, here we do not review these techniques.

IBM DB2 pureXML [p10] treats XML as a first-class data type, and it stores XML documents intact in its native hierarchical format as type-annotated trees [DB2a]. The designer of a table may specify XML type for any column of the table, and then an XML file is represented by a row in this table. Large XML files are split into subtrees in an attempt to map them into various disk pages. XML documents with their type-annotated trees can be compressed by a dictionary type compression technique, which replaces tag names with unique integer values [ML05, DB2b]. In case of a query or update operation, these trees are fully decompressed. However, pureXML does not compress trees to a more concise format, similar to annotated trees in XSAQCT; nor does it compress XML data values. In conclusion, pureXML does not attempt to perform XML-conscious lazy decompression for query and update operations.

Oracle Berkeley XML DB [Oa, Ob] stores various kinds of items in separate containers, such as documents, indices and index statistics, data dictionary and other system metadata. By default, all XML documents stored in a container are compressed (metadata and indexes are not compressed) and they are fully decompressed when they are retrieved from those containers. Internally, XML nodes are stored in a B-tree, where nodes are allocated in document order, which also is an iteration order on the B-tree. Therefore, this database is not XML- conscious.

eXist [Ea, Eb] is probably the most widely deployed native open source XML database. eXist stores the XML tree as a modified, number scheme based, k-ary tree combined with structural, range and spatial indexing based on B+-trees, and a cache used for database page buffers, but it does not compress the documents.

In BaseX, [B10, N10] the XML tree is encoded and mapped in a simple table storing all of the node information. Processing time can then be improved by minimizing the table structure coupled with text, attribute, full-text (not default) and path indexing.

Sedna [Se] is a full-featured native XML database, in which nodes of an XML document are clustered together according to their positions in the descriptive schema of a document where direct pointers are used to represent relations between nodes of an XML document based on B-trees. It uses the numbering schema [A06], in which the nodes of the documents are labeled with certain unique identifiers. Comparing these identifiers, one can restore the sequence order of the nodes and to establish the hierarchical relationships.

Finally, Qizx [QX] is a native XML database engine, designed to perform high-speed querying, retrieval and processing of indexed XML contents. Updates are not applied immediately as the updating expressions are accumulated to a pending update list and the database is updated atomically, which is to help the re-indexing of the data model. Documents and indexes are compressed, and the compression mechanism is completely transparent to users or applications. As a result, partial updates of documents are not fast, because Qizx needs to entirely rebuild an updated document (but only once per transaction).

3. XSAQCT

For the sake of completeness, in Section 3.1 we briefly recall the description of the previous version of XSAQCT that supported querying with lazy decompression (for more details see [M09]), and then in Section 3.2 we describe updates. Note that the annotated tree representation is the internal representation used by our implementation and it is not visible to the user, who operates on XML documents as if they were uncompressed. In particular, the user will use standard XPath expressions to query and specify parts of the document, which are to be updated.

3.1. Basic Architecture of XSAQCT

Given a document D, we perform a single SAX traversal of D to encode it, thereby creating an annotated tree $T_{A,D}$, in which all similar paths are merged into a single path and each node is annotated with a sequence of integers; see Figure 1. Two absolute paths are called *similar* if they are identical, possibly with the exception of the last component, which is the data value. For example, the paths /a/b/t1 and /a/b/t2 are similar while the paths /a/b/t1 and /a/c/t1 are not. Note that $T_{A,D}$ provides a faithful but succinct representation of the structure of the input document D. Indeed, our tests performed on the files from the commonly-used Wratislavia corpus confirmed the succinctness of this representation.

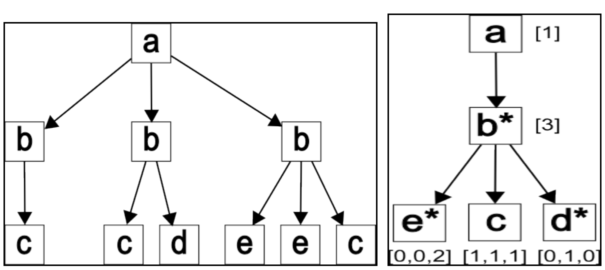


Figure 1 (a) XML document D; (b) the annotated tree $T_{A,D}$ representing D.

At the same time, data values are written to the appropriate data containers. Next, $T_{A,D}$ is compressed by first writing its annotations to one container and the skeleton tree T_D (with annotations stripped) to one or more containers. Finally, all containers are compressed, using back-end compressors, and written to create the compressor's output C_D. At the same time, data values are written to the appropriate data containers. The main back-end compressors used include GZIP [gzip], BZIP2 [bzip] and PAQ8 [paq] but the user can add more compressors. The main reason behind using an annotated tree representation is that it can be used to answer various queries and (as explained in the next section) to efficiently implement updates.

Note that an XML document D may have a *cycle* if there exists a node n in D such that there are two children x and y of n, which satisfy this condition: x < y and y < x (here, "<" denotes the document order); see Fig 2 (a). If there are cycles, then we add a "dummy tag name" to the annotated tree $T_{A,D}$, here denoted by \$, which will be used to avoid cycles; see Fig. 2 (b). Therefore, the annotated tree $T_{A,D}$ may have dummy nodes, and if so they will be removed by the decompressor to recreate the original document.

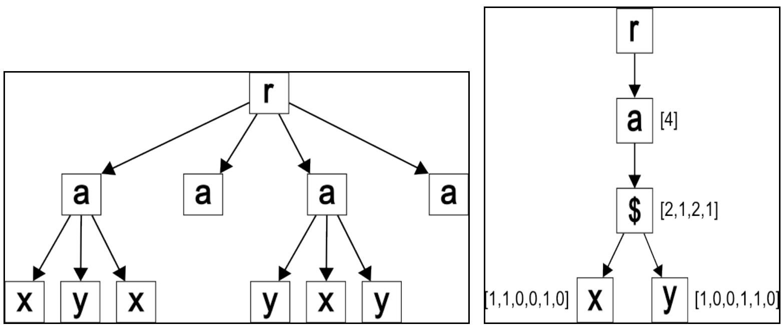


Figure 2 (a) XML document D with a cycle; (b) the annotated tree $T_{A,D}$ representing D, with dummy nodes.

3.2. Updates in XSAQCT

In Section 3.2.1, we describe update operations as seen by the user, and in Section 3.2.2 we describe the implementation of these operations.

3.2.1 Updates in XSAQCT: User perspective

At the present time, we support the following *basic update operations* on the document D:

1. insert a document D1 as a child of the node *n* of D at position *i* (here, the node *n* is specified by providing a path leading to it, using XPath syntax; the value of *i* equal to 0 represents inserting as the leftmost child, while the value of *i* equal to the number of children of *n* plus one represents inserting as the rightmost child)

2. delete a sub-document D1 of D rooted at the node *n* (here, the node *n* is specified as above)

3. insert a new text node as a child of the node *n* of D at position *i* (here, we use the same meaning of n and i as in operation 1 above) – possibly merging with sibling text nodes

4. delete the existing text node specified by the path

5. insert a set of new attributes of the node *n* of D (here, we use the same meaning of n as in operation 1 above)

6. flush the pending list and update the actual XML document

The final version of our system will support all update operators (such as move a sub-document), following the proposed W3C Update Facility [XQ08]; most of these operations can be easily implemented using the above basic operations.

The above basic operations are supported via the following three *basic* functions:

1. Boolean insertNode(path p, string node_name, int pos), which inserts an element or text node with the name node_name as a child of the node represented by p at the position pos. If the node cannot be inserted (e.g. the path p or the position i is incorrect), insertNode returns false; otherwise true is returned and the node will be inserted in the Annotated Tree. Inserting a document D1 is supported by traversing D1 top-down, and applying the function insertNode().

2. Boolean removeNode(path p), which removes the node n at path p and all of the child nodes of the node at p. If the node n cannot be deleted (the path p is incorrect, then removeNode returns false and removes no nodes; otherwise true is returned and the entire sub-document rooted at n will be removed from the Annotated Tree. If the path p points to a text node, then this node is removed.

3. void flush(), which flushes all pending lists

The above three basic update operations will be referred to as high-level XML update operations; these operations as well as the query operations are implemented via low-level update operations referred to as AU-operations, described in Section 3.2.2. XSAQCT supports two modes of operation: (1) with undoing of operations (like in XQuery Updates); and (2) without undoing, in which two operations appearing in the pending list ***may*** "cancel each other", for example the inserting some node followed by removing the same node.

In addition to the update operations, there are various *query* operations. Currently, only simple queries have been implemented (specifically, we have implemented absolute paths; similarly to TREECHOP, but additionally including some predicates such as "position()=2"). A query, which ends in an element, can be immediately answered using the annotated tree, and therefore it only requires a decompression of this tree. Now, consider a query, which calls for text values for a given path. As mentioned earlier, the compressor creates a separate container storing all values for similar paths. Therefore, to answer this type of a query it is enough to decompress a single data container, and then return the text values stored in this container.

In the following section we describe the implementation of XML updates. In this section, as well as in Section 4, where we present results of our tests, we use the following abbreviations: I stands for insert, R for removing a node, Q for query (here, "*" means all nodes, and text() to find the text values), T for adding a new text, and X for removing text.

3.2.2 Updates in XSAQCT: Implementation perspective

Update operations result in modifications of annotations, which may be costly (e.g. inserting or removing a single item from an annotation sequence). Therefore, our approach supports *lazy updates*, implemented by attaching to each annotated tree node a list of pending update operations (referred to as *pending list*). Updates in the pending list will only be applied when a threshold is reached (the value of threshold will be determined experimentally) or when the user explicitly requests such operation; at that time, the pending list is flushed. The (working) **state** of the pending list is defined as a sequence op_1, op_2,…,op_n, where op_i, i=1,2,…,n is an AU-operation. The list is called *clean* if n=0; otherwise it's called *dirty*. If a list is clean, then a modifier makes it dirty; and if the list is dirty then the flush operation makes it clean. Adding and removing text also makes changes to a pending list. This allows text to inserted and deleted as a child of some node N, without needing to decompress N's text containers. The changes to text are not written until the user initiates a flush.

Note that a query operation can be initiated without a flush of any pending lists. As mentioned previously there are two modes of update operations, one without an option of undoing (i.e. cancelling) an operation and another that gives the user an option of undoing any update operations. To support undo operation, we do not attempt to simplify operations in the pending list. After a flush, undoing previous operations becomes impossible.

Available AU-operations that may be stored in the pending list are listed below:

1. Boolean insertValue(int value, int position), which inserts a value at position. If the value cannot be inserted (the position is incorrect), insertValue returns false; otherwise true is returned and the insertValue operation will be appended to the pending list.

2. Boolean deleteValue(int position), which deletes a value at position. If the value cannot be deleted (the position is incorrect), deleteValue returns false; otherwise true is returned and the deleteValue operation will be appended to the pending list.

3. Boolean modifyValue(int position, int amount) which modifies value at position by adding the value of amount (note that this value may be positive or negative). If the value cannot be modified (the position is incorrect), modifyValue returns false; otherwise true is returned and the modifyValue operation will be appended to the pending list.

In addition, to support queries, there is an operation Boolean getValue(int position). This operation will not be stored in the pending list; rather it is implemented by traversing the pending list backwards and then using the appropriate annotation sequence. Finally, to support lazy insertion of nodes, which are not present in the annotation tree, a node n may have a distinguished annotation denoted by {i} that stands for the annotation consisting of i zeros.

Example.

Consider an XML document D of the form

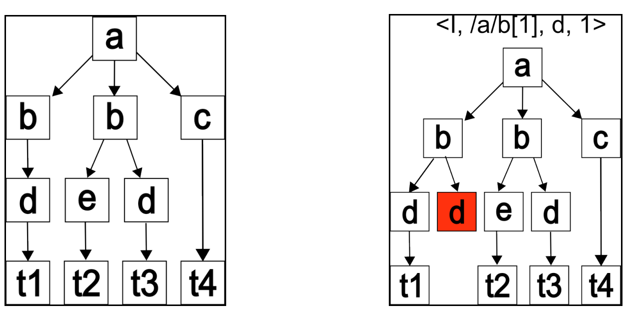
```
<a> <b> <d> t1 </d> </b>
    <b> <e> t2 </e>
        <d> t3 </d>
    </b>
    <c> t4 </c>
</a>
```

for which the corresponding document tree is shown at the top left corner of Fig. 3. The three remaining trees shown in the same figure show the *user perspective* of executing high level update operations; specifically:

1. The top right corner shows the result of executing insertNode(/a/b[1], d, 1)

2. The bottom left corner shows the result of executing insertNode(/a, b, 2)

3. The bottom left right shows the result of executing insertTextNode(/a/b[1]/d[2], t5).



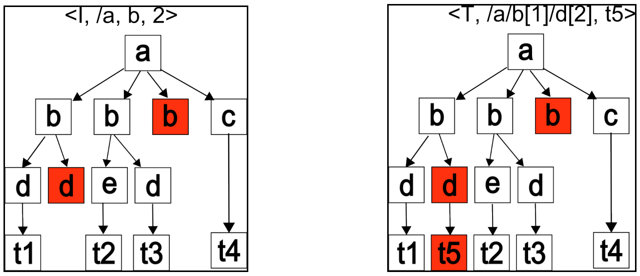
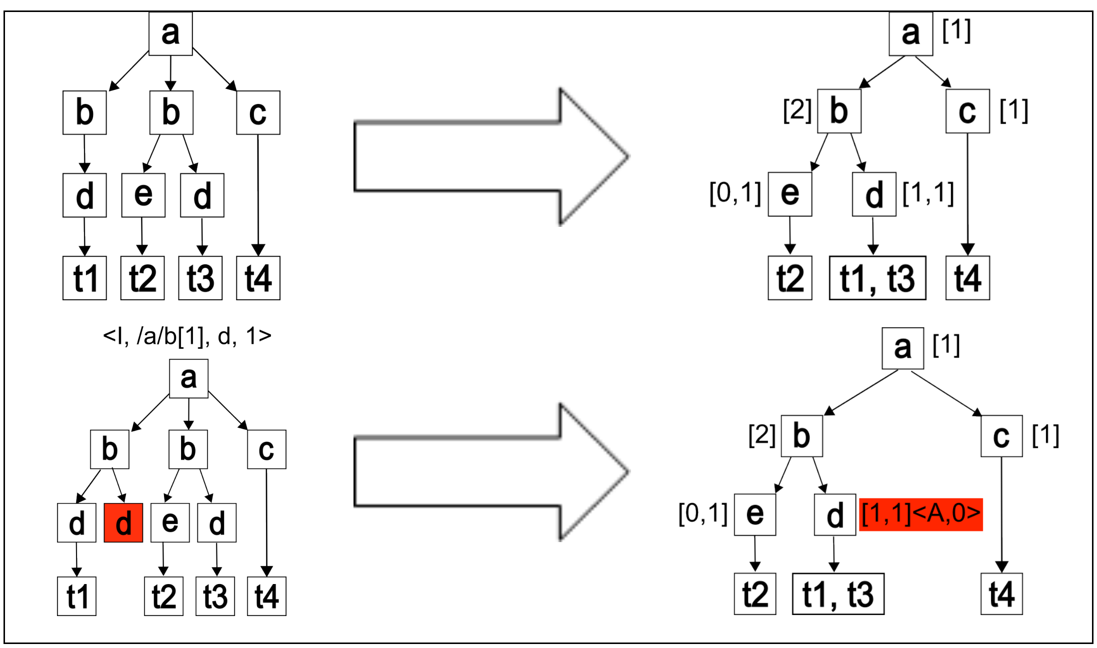


Figure 3. Updates of an XML document D: user perspective

Fig. 4 shows the changes in the annotated tree $T_{A,D}$ representing updates of D; specifically:

1. The top left corner of part (a) shows the original tree and the corresponding annotated tree (top right corner)
2. The bottom left corner of part (a) shows the user's perspective of executing insertNode(/a/b[1], d, 1) and the corresponding annotated tree with the pending list (bottom right corner)
3. The left corner of part (b) shows the user's perspective of executing insertNode(/a, b, 2) and the corresponding annotated tree with the pending lists (right corner)
4. The left corner of part (c) shows the user's perspective of executing insertTextNode(/a/b[1]/d[2], t5) and the corresponding annotated tree with the pending lists (right corner)



(a)

(b)

(c)

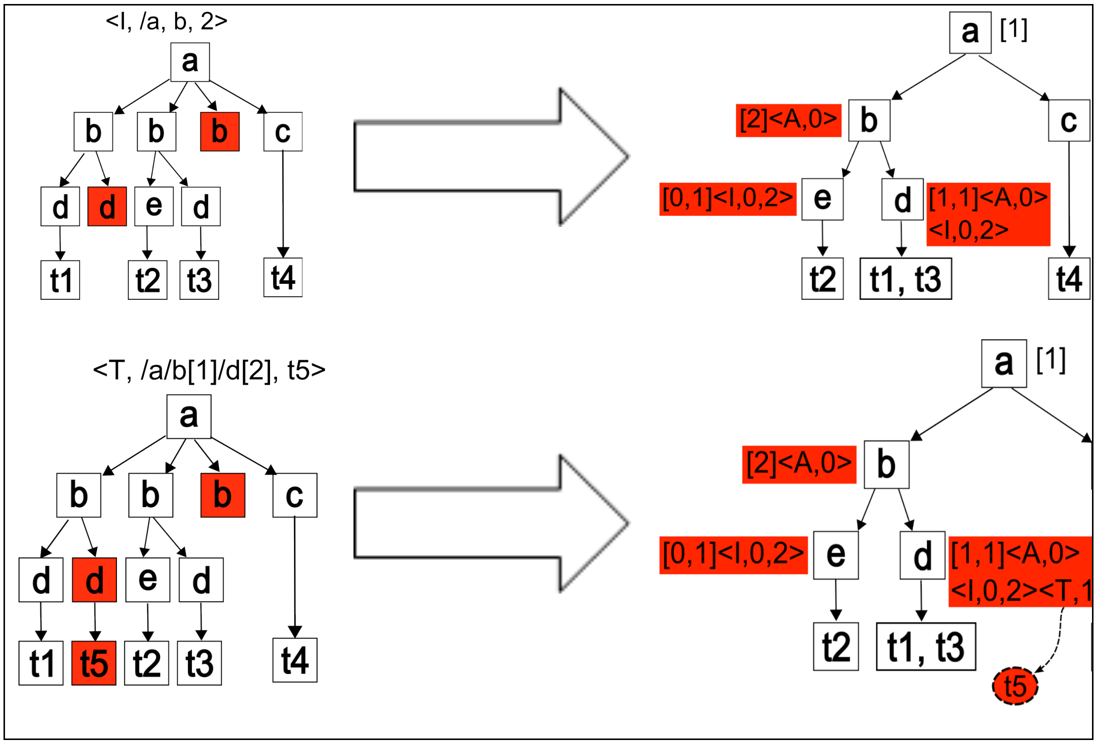


Figure 4. Updates of an XML document D: implementation perspective

Note that after a few insertions or deletions it may not be beneficial to write to a file, but in general it is a “space versus time” trade-off: writes are costly in terms of time but preserve space). It is best to think about the design as if it had two layers, the top layer is not concerned with issues such as “removed compressed annotations create a *hole* in a file”, while the bottom layer is concerned with usage of the compressed file (which is maintained in a way similar to heap management). This description does not provide details of the lower layer.

In the next section, we provide the results of tests performed on XSAQCT and some related XML updaters.

4. Results of Tests

Because of the on-going nature of our project and the fact that currently it does not support indexing or any kind of caching, fair comparisons of query and update times is very difficult. Instead, we conducted various preliminary tests on several open source applications (including Exist [Ea], BaseX [B10], QuizX [QX], Sedna [Se], Oracle [Ob] and our XSAQCT) to assess some sort of benchmark. When dealing with any kind of performance comparison for XML compressors, one compares their application with others, using a specific set of input documents. In this paper, for our experiment we use the file uwm.xml (of the size 2.2M) from the Wratislavia XML corpus [W10]. In total, we performed 32 operations, shown in Table 1, where in each row the best updater’s time is shown in bold face, and the second best result is shown in italics. To remove cache interference, tests were performed five times, with the application being restarted after every trial, and we computed the average result. Out of the 32 different trials, XSAQCT achieved the best results, as it placed first fourteen times, and it placed second fourteen times. QuizX achieved the second best results, as it placed first fourteen times and it placed second 10 times. Oracle was placed first four times, and second 10 times.

Table 1. Results of the tests

Operation/Updater	**Exist**	**BaseX**	**QuizX**	**Sedna**	**Oracle**	**XSAQCT**
<I, /root, course_listing, 0>	184.8	115.99	1629.8	*77.2*	186.80	**12.76**
<I, /root/course_listing[1], course[1], 0>	158	19.58	1071.6	73.2	*7.87*	**1.37**
<Q, /root/course_listing[1]/course[1]>	90.8	5.62	**1.85**	23.2	6.23	*4.55*
<T,/root/course_listing[1]/course[1], "alpha">	93.2	20.73	989.4	74.8	*8.37*	**0.94**
<Q, /root/course_listing[1]/course[1]/text()>	55.6	7	*2*	13.4	5.73	**0.87**
<Q, /root/course_listing[2]/course[1]/text()>	54.6	8.22	**0.85**	20.8	5.64	*1.21*
<R, /root/course_listing[1]/course>	53.4	16.7	984	95.8	*6.51*	**1.07**
<R, /root/course_listing[1]>	62.2	17.34	986	83.2	**7.34**	*11.83*
<Q, /root/course_listing[1]/*>	47.6	3.93	*1.85*	34	6.56	**0.69**
<Q, /root/course_listing[1]/course[1]/*>	42.8	4.06	**0.85**	13.4	6.57	*1.09*
<Q, /root/course_listing[1]/course[1]/text()>	46.8	6.42	**0.85**	15.2	7.54	*1.29*
<R, /root/course_listing[1]/note>	191	122.24	1405.6	85.4	**3.54**	*8.52*
<Q, /root/course_listing[1]/course[1]/text()>	158.8	*4.48*	**1.4**	12.8	6.41	4.91
<Q, /root/course_listing[2]/course[1]/text()>	87	5	**1**	19	6.84	*1.06*
<Q, /root/course_listing[1]/title/text()>	49.4	5.18	**1**	13.4	5.96	*1.19*
<I, /root/course_listing[1], course, 0>	167.8	114.33	1346.4	82.8	**3.99**	*11.49*
<Q, /root/course_listing[1]/course[1]/text()>	161.2	5.07	**1.8**	14.8	5.49	*4.88*
<T, /root/course_listing[1]/course, "216XXX">	122	23.12	915.6	76	*6.93*	**0.93**
<Q, /root/course_listing[1]/course[1]/text()>	41.2	4.67	*1.2*	19.2	4.59	**1.05**
<X, /root/course_listing[1]/course>	57.2	15.39	879.2	89.8	*8.63*	**0.83**
<T, /root/course_listing[1]/course, "216TM")	41.2	21.46	900.8	85.8	*8.51*	**0.85**
<Q, /root/course_listing[1]/course[1]/text()>	37.4	6.06	*1.8*	16.4	7.34	**1.09**
<Q, /root/course_listing[2]/course[1]/text()>	44.8	4.55	**1**	10.6	6.18	*4.62*
<Q, /root/course_listing/course[1]/text()>	755	93.27	**70.2**	989.8	*82.3*	315.02
<R, /root/course_listing[1]>	400	117.44	1621.2	76.8	*53.09*	**25.52**
<Q, /root/course_listing[1]/course[1]/text()>	197.8	5.22	**2.6**	11.6	6.39	*4.67*
<Q, /root/course_listing[4]/section_listing[2]/*>	76.8	5.78	**0.55**	30.6	6.60	*1.35*
<Q, /root/course_listing/course[1]/text()>	1103	*90.58*	110.8	626.2	**79.17**	269.42
<Q, /root/course_listing[1]/section_listing>	69.8	4.42	**0.85**	24	7.08	*1.06*
<I, /root, course_listing, 0>	54.6	*22.83*	1043.4	88.4	181.41	**2.58**
<I, /root/course_listing[1], course, 0>	57.2	13.92	1032.6	72.6	*7.29*	**1.26**
<Q, /root/course_listing/course[1]/text()>	690.8	83.19	**75.6**	535.8	*79.96*	184.61

5. Conclusions and Future Work

For the proposed framework to be generally applicable, it is paramount that storage considerations are addressed. Storing the compressed document as a single file in a standard file system has several weaknesses; for example querying may require storing offsets within a file to retrieve the required compressed container; and flushing the pending list may require re-writing the entire file. Therefore, we will investigate storing the compressed document through a specially-designed layer, implemented using either a file system or a database (as appropriate, based on experimentation). We will also investigate compressing pending lists.

In addition, we will investigate adding *versioning* of compressed XML documents, where switching between different versions will not require full data decompression. When a user decides that a series of updates form a new version, the pending list will be assigned this version's number. Any subsequent updates will then be included in the successive version.

References

[A03] Arin, A.; Bonifati, A.; Manolescu, I. and A. Pugliese. XQueC: A Query-Conscious Pushing Queries to Compressed XML Data. Proceedings of the Twenty-Ninth International Conference on Very Large Data Bases, pp. 1065-1068, 2003.

[A06] Aznauryan, N. A.; Kuznetsov, S. D.; Novak, L. G. and M. N. Grinev. SLS: A numbering scheme for large XML documents, Volume 32(1), 2006, pp. 8 – 18.

[A06] Aznauryan, N. A.; Kuznetsov, S. D.; Novak, L. G. and M. N. Grinev. SLS: A Numbering Scheme for Large XML Documents. pp. 8-18. Programming and Computer Software. Springer Science+Business Media LLC. Volume 32(1), January, 2006.

[B10] BaseX: http://basex.org/, retrieved on January 20, 2010.

[bzip] bzip compression, http://www.gzip.org/, retrieved on January 20, 2010

[DB] Getting Started with Berkeley DB XML: http://www.oracle.com/technology/documentation/berkeley-db/xml/gsg_xml/cxx/index.html, retrieved on January 20, 2010

[DB2a] DB2 pureXML: http://domino.watson.ibm.com/comm/research.nsf/pages/r.datamgmt.innovation.purexml.html, retrieved on January 20, 2010

[DB2b] Compression in DB2: http://www.ibm.com/developerworks/forums/thread.jspa?threadID=315868&tstart=0, retrieved on January 20, 2010

[Ea] Exist: http://exist.sourceforge.net/, retrieved on January 20, 2010

[Eb] Exist: http://exist-db.org/webdb.pdf, retrieved on January 20, 2010

[gzip] bzip compression, http://www.bzip.org/, retrieved on January 20, 2010

[L05] Leighton, G.; Müldner, T. and J. Diamond. TREECHOP: A Tree-based Query-able Compressor for XML, The Ninth Canadian Workshop on Information Theory, June 2005.

[L08] Li, C.; Ling, T-W. and M. Hu. Efficient updates in dynamic XML data: from binary string to quaternary string. The International Journal on Very Large Data Bases, 17(3), pp. 573–601, 2008.

[M09] Müldner, T.; Fry, C.; Miziołek, J. K. and S. Durno. XSAQCT: An XML queryable compressor. Balisage: The Markup Conference, August 10 -14, 2009.

[ML05] Matthias N.; van der Linden, B. Native XML Support in DB2 Universal Database. Proc. of the 31st VLDB Cnference, 2005.

[N10] Native XML Processing: http://www.inf.uni-konstanz.de/dbis/teaching/ws0708/xml/24-Native.pdf, retrieved on January 20, 2010

[Oa] Oracle XML DB: http://www.oracle.com/database/berkeley-db/xml/index.html, retrieved on January 20, 2010

[Ob] Oracle DB Reference Guide: http://www.oracle.com/technology/documentation/berkeley-db/xml/ref_xml/xml/arch.html, retrieved on January 20, 2010

[p10] pureXML: http://www-01.ibm.com/software/data/db2/xml/, retrieved on January 20, 2010

[PAQ] PAQ Compression, http://cs.fit.edu/~mmahoney/compression/paq.html#paq8, retrieved on January 20, 2010

[QX] QuizX: http://www.xmlmind.com/qizx/, retrieved on January 20, 2010

[Se] Sedna: http://modis.ispras.ru/sedna/, retrieved on January 20, 2010

[T01] Tatarinov, I.; Ives, Z. G.; Halevy, A.Y. and D. S. Weld. Updating XML. SIGMOD Conference 2001, pp. 413-424, 2001.

[W10] Wratislavia corpus: http://www.ii.uni.wroc.pl/~inikep/research/Wratislavia/, retrieved on January 20, 2010.

[XQ08] W3C XQuery Update Facility 1.0, Candidate Recommendation 1 August 2008, http://www.w3.org/TR/xquery-update-10/, retrieved on January 20, 2010.