Design and Implementation of an Online XML Compressor for Large XML Files

Tomasz Müldner
Jodrey School of Computer Science
Acadia University
Wolfville, B4P 2A9 NS, Canada
e-mail: Tomasz.muldner@acadiau.ca

Jan Krzysztof Miziołek
IBI AL
University of Warsaw
Warsaw, Poland
e-mail: jkm@ibi.uw.edu.pl

Tyler Corbin
Jodrey School of Computer Science
Acadia University
Wolfville, B4P 2A9 NS, Canada
e-mail: 094658c@acadiau.ca

Christopher Fry
Jodrey School of Computer Science
Acadia University
Wolfville, B4P 2A9 NS, Canada
e-mail: chrisfry99@gmail.com

***Abstract*—Network-based applications using XML experience a performance penalty resulting from the verbose nature of this data format. This paper presents a novel XML-conscious compressor designed to alleviate these problems, using it for online compression and decompression. Two versions of the compressor were designed and implemented to find the most optimal solution and they were compared with offline compression/decompression. The tests show that for existing files online compression is less efficient than offline compression, however, online compression is superior for streaming or when compared to offline compression combined with sending the file through the network and subsequent decompression.**

Keywords-XML; compression; network performance.

I. Introduction

The eXtensible Markup Language (XML) [16] is the most popular meta-language for the interchange and access of data. In particular, XML has been adopted as one of the main formats for online communications and Web applications. However, XML's markup and resulting verbose nature may increase the size of a dataset as much as ten-fold. For XML-based network applications, network bandwidth tends to become the bottleneck in the interchange of information; therefore these applications will experience a performance benefit from compressing XML data.

There has been considerable research on XML-conscious compressors, which unlike general data compressors can take advantage of the XML structure; see [2][3][4]. Most recently, there has been research on queryable XML compressors for which queries can be answered using lazy decompression, i.e., decompressing as little as possible when executing a specific query; see [5][6]. Also, there has been research on updateable XML compressors, for which updates can be saved without full decompression; see [7][8]. Online XML compressors are typically defined as compressors, which decompress chunks of compressed data whenever possible rather than processing it offline when the entire compressed file is available; see [9][10]. Clearly, for a compressor to be online implies that *only one pass* through the document is required to compress it. This class of compressors is particularly useful for networked applications, specifically on networks with limited bandwidth. Numerous applications of XML use *streams*, abstract representations of sources/sinks, where the sources of data are dynamic and their contents are not known beforehand, e.g., measurements or logging. The contents are processed at run-time, either by XML Streaming Parsers, such as SAX [17] or StAX [24] or by ordinary text compressors such as GZIP [18]. Another approach is taken by Efficient XML Interchange (EXI) format [25], a compact representation of XML designed to reduce bandwidth requirements while maintaining efficient use of various resources such as memory and processing power (implemented using EXIficient [27]). While Snyder [26] determined that using EXI can double a bandwidth potential, it should be noted that EXI is not a queryable compressor.

This paper presents an online compression algorithm based on XSAQCT, an XML compressor developed by our group, see [11]. There are other online compressors, e.g., TREECHOP [12], but XSAQCT has a number of distinctive features, in particular it is queryable using lazy decompression, updateable [7], supports the streaming of data in a more compact representation than ordinary text compressors, and finally the structure of the XSAQCTs compression scheme allows a large reduction in processing time through parallelization on multi-core machines [13]. Possible educational applications of XSAQCT are described in [14]. Similar to TREECHOP, XSAQCT supports compression where the decompressor's output is the same as the original input (i.e., the document is semantically equivalent to the original document) or the output generates a *canonicalized* [15] XML document. Design of an early version of the compressor described in [1] did not support XML documents with mixed contents, attributes, or cycles, e.g., nodes with the consecutive children b, c and b. For example, if in Figure 1 (a) the node t2 were actually a tag node "c", then there will be a cycle (for more on cycles, see [11]). This paper presents a design and implementation of the new version, which removes all these limitations and supports arbitrary XML files. In addition, this paper presents the implementation and results of tests on 11 sample XML documents aimed to evaluate the design and implementation.

Contributions. Design, implementation and test results of two versions of the novel online XML compressor, XSAQCT are presented. These two versions are tested and compared with: (1) Send-and-Compress, i.e., sending a single XML file D over the network from node N1 to node N2 and then compressing *offline* in N2; and (2) Compress-and-Send, i.e., compressing D *offline* on N1 and sending to N2. Recall from [1] that online XSAQCT **not only** *decompresses* the data whenever enough data is available, but it also *compresses* online, which is essential for the case of a network node N1 receiving *streamed* XML data from one or more sources, which are to be stored in a compressed form. The tests show that for existing files online compression is less efficient than offline compression. However, online compression in its natural environment (e.g., streaming) is a more space efficient and faster technique.

This paper is organized as follows. Section II gives a short introduction to the design and functionality of the previous offline version of XSAQCT, and Section III describes its current extension, i.e., online XSAQCT. Section IV is on characteristics of the test suite used in this paper, and Section V provides the description of the implementation and testing results. Section VI describes applications of XSAQCT for online communication, and finally, Section VII provides conclusions and describes future work.

II. OUTLINE OF OFFLINE XSAQCT

For the sake of completeness, we briefly recall here a description of offline XSAQCT; for more details, see [11][7]. Given an XML document D, we perform a single SAX (specifically using Xerces, [17]) traversal of D to encode it, thereby creating an annotated tree $T_{A,D}$, in which all *similar* paths (i.e., paths that are identical, possibly with the exception of the last component, which is the data value) are merged into a single path and each node is annotated with a sequence of integers; see Fig. 1. When the annotated tree is being created, data values are output to the appropriate data containers. Next, $T_{A,D}$ is compressed by writing its annotations to one container and finally all containers are compressed using selected *back-end compressors,* e.g., GZIP [18]. While GZIP was chosen (because HTTP standard uses it), another suitable data compressor can be used as a back-end compressor.

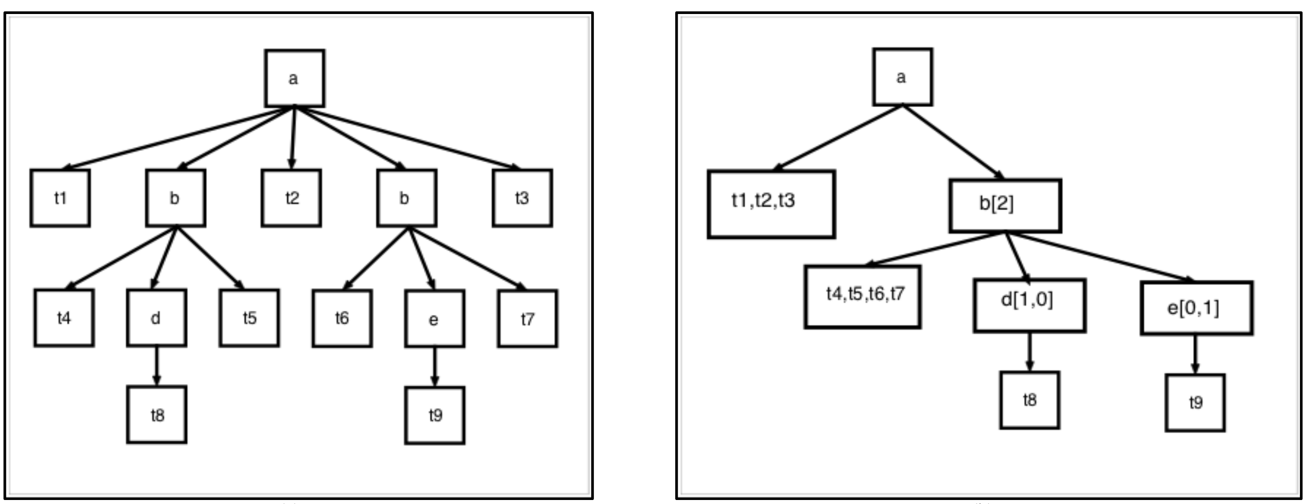
Note that if there was another node labeled “c” in Fig. 2 c) then the document D2 would have a cycle.

III. ONLINE XSAQCT

In this section, we present our online algorithms.

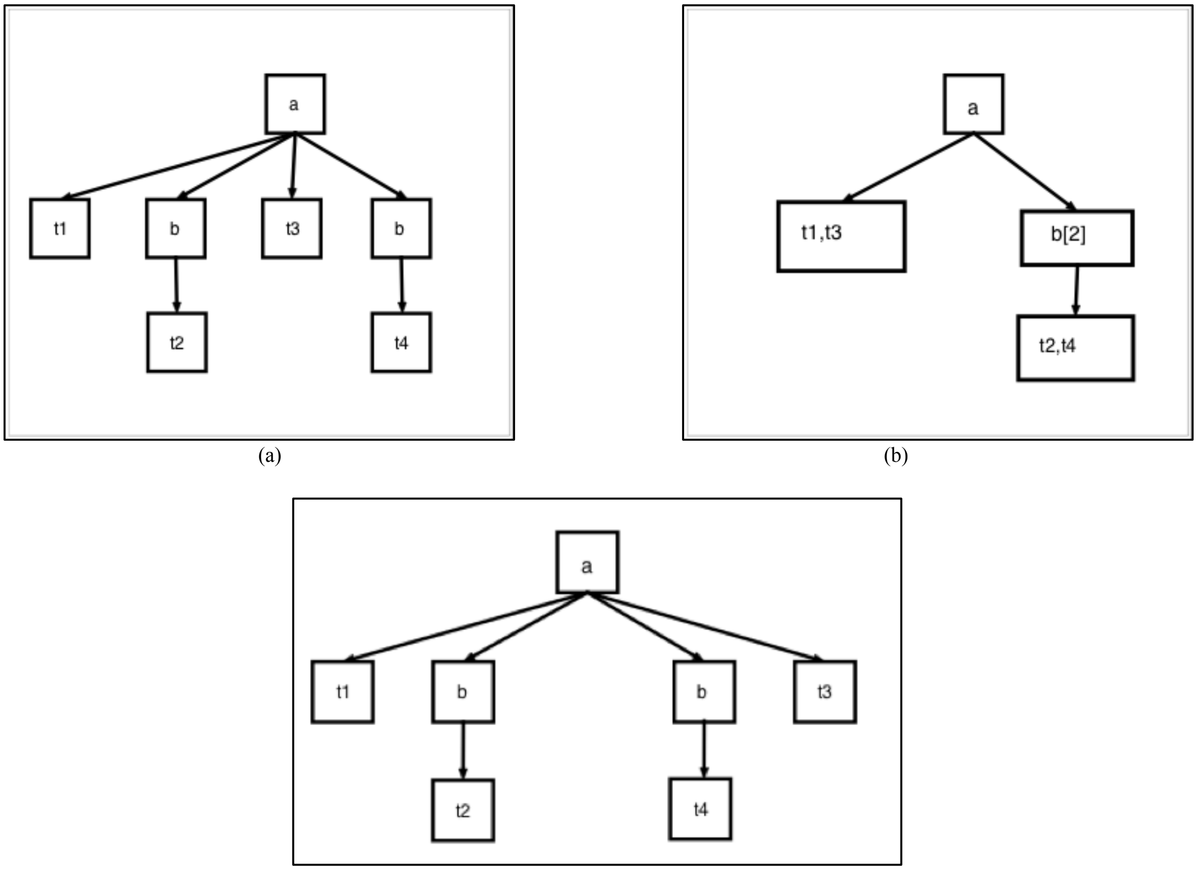
A. Notations and Terminology

In this paper, XML documents may have *mixed* contents, assuming “full mixed content”, i.e., there exists a text child separating any two siblings, and there are text children respectively before the first child and after the last child.



(a) (b)

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



(a) (b)

(c)

Figure 2. XML document D1 (a), the annotated tree for D1 (b), another XML document D2 (c)

Example of full mixed contents is shown in Fig. 1 (a) and its annotated tree is shown in Fig. 1 (b). The use of full mixed contents is required; otherwise an annotated tree would not uniquely represent every XML document. For example, for the XML document D1 from part (a) and D2 from part (c), the annotated tree shown in part (b) of Fig. 2 is the same. Common occurrences of nodes that do not exhibit the full mixed content property are elements that use font-style tags, e.g.,

Bold TextOther Text. Note that to achieve full mixed content, the sending node may have to insert empty text (consisting only of ASCII zero) whenever the text is missing; the receiving end outputting the decompressed file will neglect such empty texts.

In Fig. 2 (c), there would be an empty text between the two occurrences of “b”.

The skeleton tree T_D denotes the tree labeled by tag names (with no annotations) and ANN denotes the sequence of all annotations. Annotations for a node of $T_{A,D}$ may be stored with this node, or the node may store a (logical) pointer to ANN (e.g., the offset within ANN). In the annotated tree, for each node n the text for all similar paths ending with n is stored as the leftmost child of n (strictly speaking a *container* for all texts, separated by ASCII zero); see Fig. 1 (b).

We assume that an annotated tree $T_{A,D}$ is implemented so that following functions are available:

- Node add_RC(Node n, Tag p, annotation a) creates and returns a new rightmost child of n with the tag p and the annotation a;
- void add_Text(Node n, Text t) adds text t to the leftmost child of n (creating it if necessary)
- Node create_Root(Tag p) creates a new root with tag p;
- Node get_LC(Node n) returns the leftmost child of n;
- Node get_RS(Node n) returns the right sibling of n;
- bool function is_Text(Node n) returns true iff n is a special tree node to store text;
- Node get_Parent(n) returns the parent of n;
- Node get_Tag(n) returns the tag of n;
- Text get_Text(n) returns the text child of leaf node n.

In addition, we assume that a data structure Path stores tags or text value, with the operations append_Node(Path p, Node n) which appends n to the path p, append_Text(Path p, Text t) which appends text T to the path p, clear_Path(Path p) which sets the path p to empty, and set_Path(Path p, int k) which stores k as the first element of p. Finally, we use the following notations:

a(n)	annotation of the node n
a(n)+=j	increase the last annotation of n by j

a(n)+=",0"	add ", 0" to the annotation of n, e.g., if a(n)=[1] then it becomes [1,0]
$[0^{a(m)},1]$	if a(m) is [1], then $[0^{a(m)},1]$ is [0,1] otherwise $[0^{a(m)},1]$ is [0,…,0, 1] where $0^{a(m)}$ is the sum of all annotations in a(m), minus 1; e.g., if a(m) = [2,1], then $[0^{a(m)},1]$ is [0, 0, 1].

B. *Online Compression*

This section describes two algorithms used for online compression, starting with a general description.

SN denotes a sending node and RN denotes a receiving node. SN and RN communicate using message passing; here SN is a producer using *send(packet)*, RN is a consumer using *receive(packet)*, where a packet is defined as a collection of data used for one processing branch (a series of data of the form:[annotation operation, text operation]); finally, synchronization is taken care of by these procedures. SN parses XML and sends packets to RN, which first creates an annotated tree (as described below) and then follows the compression process from XSAQCT [11]. To reduce the overload of sending tag names, the parser creates a dictionary of tags, which is built incrementally by SN and RN. Specifically, for a new tag T, which has not been encountered yet, SN adds T to the dictionary and sends to RN the packet containing the tag and its key in the dictionary. Then, RN uses this packet to update its dictionary, while for an existing packet only the key is sent. As a result, RN can create an annotated tree labeled by indices rather than tags. For the sake of readability the description provided in this paper shows sending and receiving tags rather than indices but our implementation operates on indices.

1. *Basic Algorithm:* The online compression is performed by two procedures, respectively executed by SN and by RN.

```
int k = -1; Path p;
// initially stores only the tag of the root
// of the XML tree
void SN_send_compress(Node n) {
    c = LC(n); // must be text, possibly empty
    append_Path(p, get_Text(c));
    c = RS(c);
    while(c != 0) {
        if(is_Text(c)) {
            append_Path(get_Text(c));
            c = RS(c);
            if(c==0) break;
        }
        append_Node(p, get_Tag(c));
        SN_send_compress(c);
        k++;
        c=RS(c);
    }
    set_Path(p, k);
    send(p);
    clear_Path(p);
    k=0;
} // SN_send_compress()
```

The pseudo-code for procedure SN_send_compress() is shown as if it was a recursive procedure running on the XML tree, but in the actual implementation the tree is not created in memory, instead an event-based SAX [17] parser implements the actions of SN_send_compress().When SN_send_compress() is called, it sends a packet of the form (-1, the path of the leftmost path rooted at the root of the tree), and at this time the value of the "*current node*" c is set to n_k; then this procedure is called recursively.

```
void RN_receive_compress() {
  bool flag; Node m; Text t;
  Node c; // current node
  receive(k, p1,…,pN , t);
  if(k==-1) { // initialization, the path received starts
    // with a node (the root), create the tree
      c = create_Root(p1);
      add_Text(c, p2);
      for(i=3; i<N; i+=2) {
          c = Add_RC(c, pi, [1]);
          add_Text(c, p(i+1));
      }
  }
  while (true) { // until the final packet
      receive(k, p1,…,pN , t);
      // the path received starts with a text
      if(k == -2)
          return; // done
      //move current based on the value of c
      for(i=1; i<=k; ++i) // set the current
          c = get_Parent(c);
      add_Text(c, p1);
      //check every tag in the received path
      for(i=2; i<=N; i+=2) {
          flag = false;
          for (m= RS(LC(c)); m <> 0; m= RS(m))
              if(get_Tag(m) == pi) {
                  a(m)+=1;
                  c = m;
                  flag = true;
                  for (every non-text child m of c)
                    a(m) += ",0";
                  add_Text(c, pi+1);
                  break;
          } // end of if and of inner for
          if(!flag) {
              c = add_RC(c, pi, [0a(c),1]);
              add_Text(c, pi+1);
          }
      } // for i=1…
    } // while(true)
} // RN_receive_compress()
```

Example 1: Compression.

(a) SN_send_compress()
For the XML file from Fig. 1 (a), we show the *trace* of the execution and packets (numbered p1, p2,…) sent by SN_send_compress(). Packets sent are shown in bold.

```
SN(a):   // SN denotes: SN_send_compress
p={a}
         c=t1
         p={a, t1}
         c=b
         p={a, t1,b}
         SN(b):
                  c=t4
                  p={a, t1, b, t4}
                  c=d
                  p={a, t1, b, t4, d}
                  SN(d):
                       c=t8
                       p={a, t1, b, t4, d, t8}
                       c=0
                       p1:{-1, a, t1, b, t4, d, t8}
                  p={}
                  c=t5
                  p={t5}
                  c=0
                  p2: {1, t5}
         p={}
         c=t2
         p={t2}
         c=b
         p={t2, b}
         SN(b):
                  c=t6
                  p={t2, b, t6}
                  c=e
                  p={t2, b, t6, e}
                  SN(e):
                           c=t9
                           p={t2, b, t6, e, t9}
                           c=0
                           p3={1, t2, b, t6, e, t9}
         p={}
         c=t7
         p={t7}
         c=0
         p4={1, t7}
         p={}
         c=t3
         p={t3}
         c=0
         p5={1, t3}
// end of trace
```

(b) RN_receive_compress()
Fig. 3 shows the state of the annotated tree after each packet has been processed by RN_receive_compress(), (un-annotated nodes have annotation [1]). Note that the last state shows the same annotated tree as in Fig. 1 (b).

2. Improved Algorithm: This algorithm is similar to algorithm 1), but it removes some overhead of sending some packets. According to the *Basic Algorithm*, for Example 2 the packets sent would start with the following packets (\0 is required to denote end of packet):

1. {-1, a, *t1*, b, t2, c, t3, \0}
2. {1, *t4*, d, t5, \0}
3. {1, *t6*, e, t7, \0}
4. {1, *t8*, f, t9, \0}

and the occurrence of consecutive leaf nodes cause at minimum six bytes of overhead with the "1" and "\0" bytes. The *Improved Algorithm* removes that overhead by encoding the packets to be:

1. {-1, a, *t1*, b, t2 c, t3, \0}
2. {-2, 3, *t4*, d, t5 e, t7, t8, f, t9}

where the value of -2 is a special action indicator (similar to what -1 represents in "root node"). One issue not mentioned before is that the packets are encoded in a preorder fashion, implying that the online algorithms have a secondary functionality and through the use of a stack, they can be used to rebuild the original XML file D as opposed to an annotated tree T_D. This is beneficial because it allows a streaming node to pipe XML data directly into a WWW application. Note that there are some boundary cases that need to be considered; for example, consider the following XML fragments:

```
<a>
  <b> text </b>
     ...
  <b> text </b>
<a>
```

which are mostly long sequences of leaf nodes. If one parent has say 10,000 such leaf nodes, each with their own text data, then a substantial buffering would be required.

C. Online Decompression

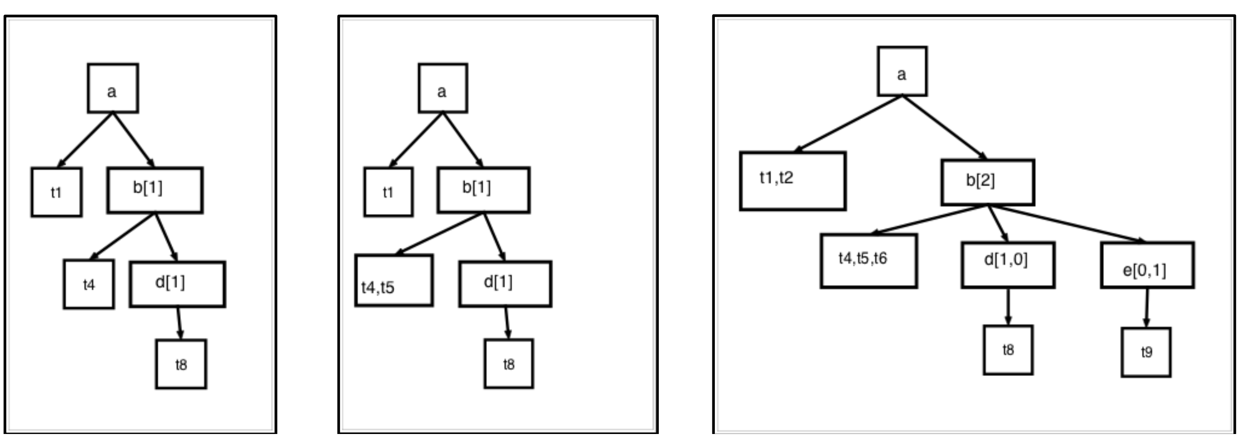
The sending node SN is assumed to be able to decompress all annotations, restore the skeleton tree and send it to RN, then re-annotate it as well as run a procedure SN_send_decompress(AnnotationTreeNode) shown below. As far as the receiving node RN is concerned, it runs a procedure RN_restore_decompress(SkeletonTreeNode) shown below. RN implements the "AA", an abstract data type, which stores sequences of annotations with the following operations (initially, the annotations for every node are un-initialized):

- void AA_delete(Node n) removes the first element of the annotations for n;
- void AA_store(Node n, sequence of integers seq) stores seq as the annotations for n;
- void AA_init(Node n) initializes the annotations for n;
- bool AA_isInit(Node n) returns true iff the annotation for n has been initialized;
- int AA_getFirst(Node n) returns the first element from the annotations for n;
- AA_get_Text(Node n, binary b) where b contains a compressed text, performs the following actions: b is decompressed, stored into a container, and then the iteration AA_nextTextIter(Node n) is started, this iteration returns the next text in the container;
- bool AA_hasReceivedText(Node n) returns true iff the text for n has been received.

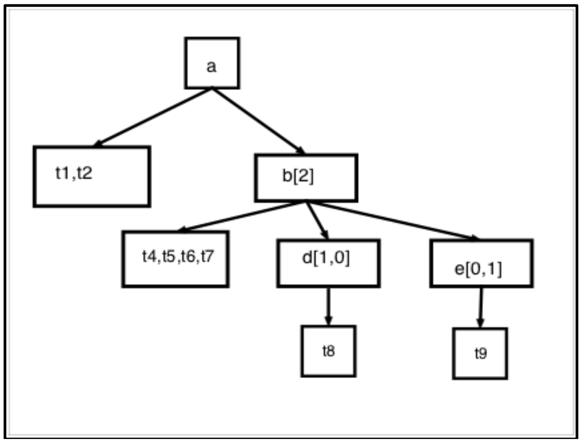
Initialization

SN restores the skeleton tree T_D and then the annotated tree $T_{A,D}$ (but it does not decompress text containers), finally it sends the *skeleton* tree to RN, which receives it.

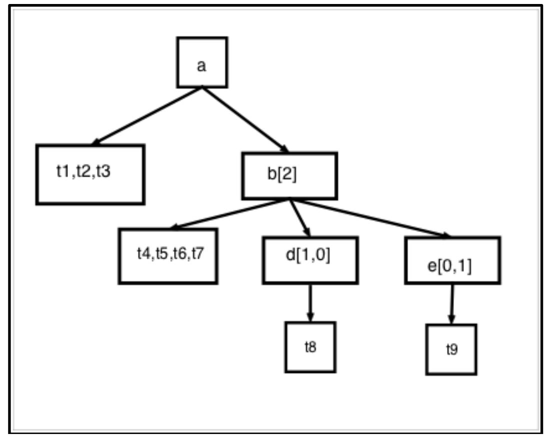
After the initialization, SN runs the procedure SN_send_decompress(AnnotationTreeNode).



(a) (b) (c)



(d)



(e)

Figure 3. Packets sent by SN_send_compress: (a) p1:{-1, a,t1,b,t4,d,t8}, (b) p2: {1,t5}, (c) p3={1,t2,b,t6,e,t9}, (d) p4={1,t7}. (e) p5={1,t3

```
SN_send_decompress(AnnotationTreeNode f) {
    for (every child c of f)
        if(isText(c)) send(c, text of c);
        // sends the entire text container,
        else {
          send(ANN(c));
          SN_send_decompress(c);
        }
} // SN_send_decompress()
```

For RN, the following code is executed:

```
output(Document Headings)
output("<" + tag(root of T_D) + ">")
RN_restore_decompress(root of T_D)
output("</" + tag(root of T_D) + ">")
output(Document Trailings)
```

where RN_restore_decompress() is shown below.

Example 2: Decompression.

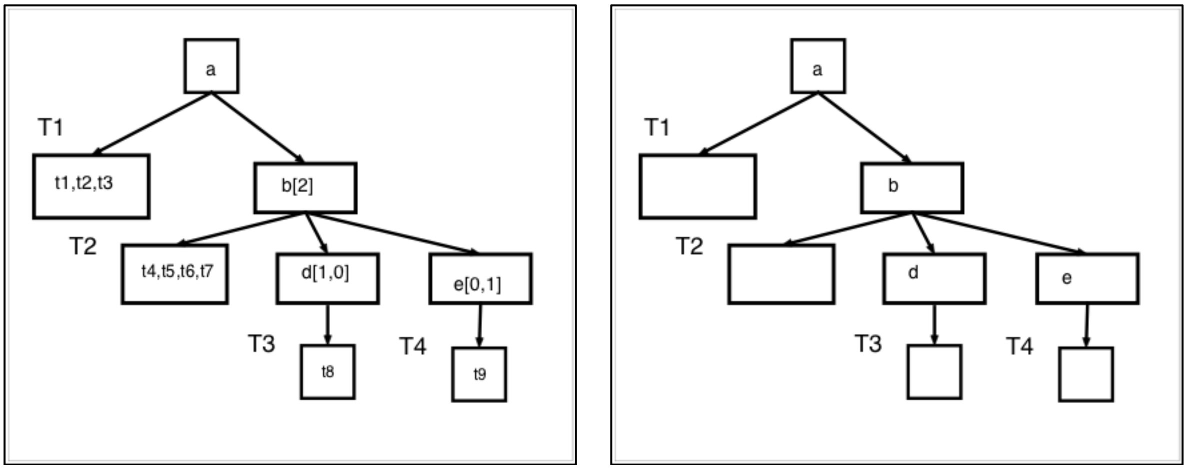
For the XML file from Fig. 1 (a), Fig. 4 (a) shows its annotated tree and Fig. 4 (b) shows the initial state of the skeleton tree. Table I shows the trace the execution of SN_send_decompress() denoted below as SN() and RN_restore_decompress() denoted below as RN(). T1,..,T4 denote text containers.

IV. CHARACTERISTICS OF THE TEST SUITE

Our experiments used the following 11 files listed here in the order of their sizes (from 5,685.77 GB to 159 KB). Specifically, we use enwiki-latest-stub-articles.xml (from [19]), 1gig.xml (a randomly generated XML file, using xmlgen [20]), enwikibooks-20061201-pages-articles.xml, dblp.xml, SwissProt.xml, enwikinews-20061201-pages-articles.xml, lineitem.xml, shakespeare.xml, uwm.xml (all from the Wratislavia corpus [21]), baseball.xml (from [22]), and macbeth.xml (from [23]).

```
RN_restore_decompress(SkeletonTreeNode f) {
  c = LC(f);  //must be text
  if (!AA_hasReceivedText(c)) AA_getText(c);
  Text t = AA_nextTextIter(c); // it shouldn't happen
   // that we reached the end of iteration before this call
  if(!empty_text(t))  output(t);
  c=RS(c);
  while(c<>0) {
    if (!AA_isInit(c)) {
      receive(ann);
      AA_init(c); AA_store(c,ann);
    }
    while (AA_getFirst(c) > 0) {
      output("<" + tag(c) + ">");
      RN_restore_decompress(c);
      a(c)+=-1;
      output("</" + tag(c) + ">");
      output(AA_nextTextIter(LC(f)));
    } // inner while
    AA_delete(c);
    c=RS(c);
  } //outer while
} // RN_restore_decompress()
```

Performance of various algorithms tested in this paper depend on the characteristics of an XML file, such as the size, the number of tags and attributes, the number of unique paths, the distribution of data among the paths and their respective sizes (in Kbytes). Table II provides an overview of these characteristics, where reserved characters are defined as all the static characters defined in the XML grammar (e.g., <, >, /). As it can be seen from Table II, files used for testing greatly vary in various characteristics and in general provide an appropriate test suite. In addition, this suite is designed to simulate streaming, as Send-and-Compress would not be an optimal because it would require *buffering all of the data internally before sending*.



(a) (b)

Figure 4. (a) Annotated tree $T_{A,D}$, (b) skeleton tree T_D

Note that two Wikipedia files (enwiki-books and enwiki-news) have their own schema specifications for rendering to a webpage. However, for the largest Wikipedia XML file (enwiki-latest-stub-articles.xml), for the "text" tag, there is a reference to 112KB of text data, whereas in enwiki-books and enwiki-news that tag would contain all of that data rather than a reference. One common characteristic, not shown in the Table II, is that the height of the XML document, i.e., the length of the longest path from the root of the tree to the leaf, never exceeds six. In other words, XML files used here are often wide but never high, and our design is suited for such files. It should be noted that this is typical of most XML documents used for everyday life and for the Wratislavia corpus [21], used by most researchers for testing their compressors; however, one can construct atypical XML documents with a large height. Another important characteristic is the number of unique paths in each XML file, which determines how many text containers will be created in the annotated tree. For the test suite used, this number varies from 19 to 548.

Fig. 5 provides a visualization of these characteristics using fractions, e.g., “node tags” represents the percentage of these tags when compared with the entire document (in this figure, some values are too small to be shown). The total sizes of element names and attribute names, calculated as a percentage of the total size, vary respectively from 64.5% to 3.71% and from 8.5% to 0%, and determine how much can be saved using the dictionary for the sender and the receiver. The total sizes of element values and attributes values, i.e., all text values, vary respectively from 93.4% to 10.9% and from 12% to 0%, and determine which data sizes can be reduced, and which cannot. Finally, the total size of reserved characters varies from 20.5% to 3.2%. Fig. 6 shows a comparison of the amount of reducible data, i.e., the amount of overhead through node tags, attribute tags, reserved characters, structure data, etc., in comparison to the amount of text data (denoted by ELB, our estimated lower bound). The accumulation of text data is defined as the *estimated lower bound* because regardless of the compression scheme applied to the XML structure, this data must be sent to the recipient node. It defines the amount of overhead we are dealing with in comparison to actual data. In general, from this figure and more accurate calculations, one can find out that the ratio of ELB over other reducible data varies from 10% to 89%.

Table III provides sizes of the test files compressed respectively with *offline* XSAQCT and GZIP, compression ratios are calculated as the size of the compressed file over the size of the original file, and finally a comparison of XSAQCT with GZIP is performed by dividing XSAQCT’s compression ratio by the GZIP compression ratio (therefore, values less than one indicate that XSAQCT’s compression is better). From Table III, it can be seen that in all cases the XSAQCT’s compression ratios are better than those for GZIP.The "text" tag in enwiki-latest-stub-articles.xml looks as follows:

```
<page>
  <title>Agriculture</title>
  <ns>0</ns>
  <id>627</id>
  <revision>
    <id>493785573</id>
    <timestamp>2012-0522T06:48:15Z</timestamp>
    <contributor>
      <username>FrescoBot</username>
      <id>9021902</id>
    </contributor>
    <minor/>
    <comment>Bot:[[User:FrescoBot/Section
      wikilinks|fixing section wikilinks]]</comment>
    <text id="496854391" bytes="112070" />
    <sha1>ozdbwwwn9r6if5sz0gcu1558jkrs6</sha1>
  </revision>
</page>
```

TABLE I. TRACE OF THE EXECUTION OF SN_SEND_DECOMPRESS()

SN(a)	**RN(a)**	**Output**
		//initially <a>
c= T1; send(T1)	c=T1; receive(t1,t2,t3), t= t1	t1
c=b[2]; send([2])	c=b, receive[2]; c=b[2]	
SN(b)	RN(b)	
c=T2, send(T2)	f=b; c=T2, receive(t4,t5,t6,t7), t=t4	t4
c=d[1,0], send([1,0]	c=d, receive([1,0]), c=d[1,0]	<d>
SN(d)	RN(d)	
c=T3, send(T3)	c=T3, receive(t8), t=t8, return	t8
	c=d[0,0], c=d[0]	</d>
c=0, return		t5
c=e[0,1], send([0,1])	c=e, receive([0,1]), c=e[0,1]	
SN(e)	c=e[1], c=0	
	c=b[2], c=b[1], c=0, return	
c=T4	c=b[1]	t2
	RN(b)	
	c=T2, t=t6, c=d[0], c=d[], c=e[1]	t6
	RN(e)	<e>
send(T4)	c=T4, receive(T4), t=t9	t9
	c=0, return	
c=0	c=e[0], c=e[]	</e>
Return	c=0, return	t7
Return	c=b[0], c=b[],return	
	return	t3
		//at the end

TABLE II. SOME CHARACTERISTICS OF FILES FROM THE TEST SUITE (SIZES ARE IN BYTES)

File	**Node Tags**	**Attribute Tags**	**Reserved**	**Attribute Text**	**Text Values**	**Total**
enwiki-latest-stub-articles	1,549,965,749	114,862,558	925,621,022	263,920,333	3,011,045,032	5,865,414,694
1gig	185,893,521	25,554,558	92,699,718	46,661,400	815,377,949	1,166,187,146
enwikibooks-20061201-pages-articles	5,791,956	392,912	2,845,193	441,905	146,789,119	156,261,085
Dblp	38,958,602	1,361,043	18,278,604	7,682,331	67,571,145	133,851,725
SwissProt	30,361,262	9,824,703	23,644,591	13,877,139	37,112,515	114,820,210
enwikinews-20061201-pages-articles	3,186,100	221,345	1,485,980	196,858	41,316,971	46,407,254
Lineitem	20,820,560	2	5,114,884	8	6,299,843	32,235,297
Shakespeare	1,808,406	0	898,463	0	4,941,139	7,648,008
Uwm	963,400	24	333,676	72	1,040,357	2,337,529
Baseball	454,720	0	141,530	0	73,032	669,282
Macbeth	40,052	0	19,888	0	103,149	163,089

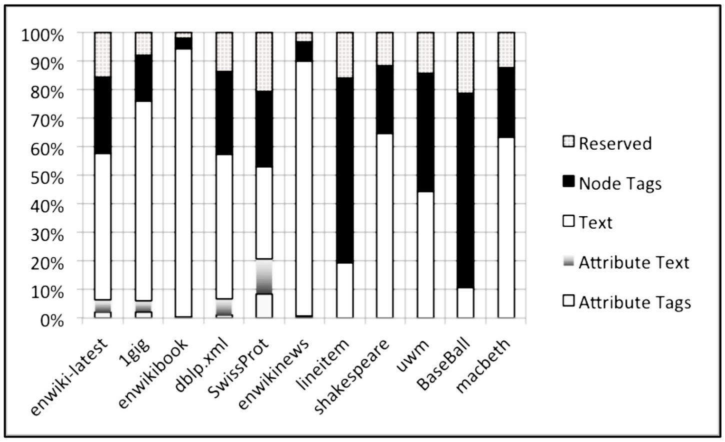


Figure 5. Characteristics of the XML suite

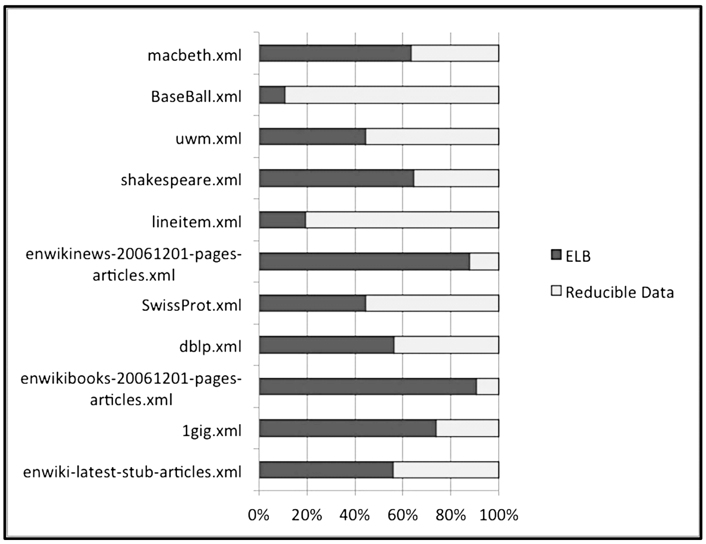


Figure 6. The comparison of reducible and non-reducible data

TABLE III. SIZES (IN MB) AND COMPRESSION RATIOS USING OFFLINE XSAQCT

File	XSAQCT	Compression ratio	GZIP	Compression ratio	XSAQCT compared to GZIP
enwiki-latest-stub-articles	678,268.57	0.1164	931,249.52	0.1599	0.7283
1gig	321,525.55	0.2808	375,695.46	0.3282	0.9743
enwikibooks-20061201-pages-articles	43,475.50	0.2848	44,621.68	0.2923	0.7921
Dblp	18,941.69	0.1449	23,912.73	0.1829	0.5394
SwissProt	7,448.27	0.0664	13,808.91	0.1232	0.9676
enwikinews-20061201-pages-articles	12,322.58	0.2718	12,735.6	0.2809	0.4928
Lineitem	1,401.13	0.0445	2,843.06	0.0903	0.8836
Shakespeare	1,846.92	0.2473	2,090.27	0.2799	0.6298
Uwm	99.44	0.0436	157.90	0.0692	0.6988
Baseball	45.57	0.0694	65.20	0.0994	0.9299
Macbeth	42.37	0.2661	45.56	0.2861	0.8558

V. IMPLEMENTATION AND EXPERIMENTAL RESULTS

This section starts with a brief description of the implementation and testing environment, followed by the implementation details. Then, it provides data transfers and timing results of experiments carried out in this environment to evaluate the effectiveness of online compression and decompression.

A. *Implementation and Testing Environment*

For the implementation language, Java version 1.7.0_05 was used. GZIP [18] was used as the back-end compressor for XASQCT (e.g., compressing the annotation lists and text containers) and in some experiments, for wrapping the sockets I/O stream (to be described later in this paper).

The following three computers were used for testing: (1) an Apple Mac box, here referred to as "SmallMac", with 2.66 GHZ, i7 processor, 8GB 1067 MHz DDR3 RAM, SATA2 SSD; (2) another Apple box, here referred to as "BigMac", an eight-core with 2.8GHz Quad-Core Intel Xeon chips (Harpertown/Penryn) processors and 12MB of L2 cache per processor; and (3) a Linux box, here referred to as "XPS", with Intel Duo Core processor, 2.40 GHZ, 4GB 1067 MHz DDR3 RAM, and 7200 RPM HDD. The experiments were carried out on LAN using nodes N1 (XPS) and N2 (SmallMac) and N1 and N2 located one hop away. For the sake of completeness note that a 100 Mbit switch connects XPS and SmallMac. Tests were also carried out for sending data from XPS to BigMac and vice versa. The XPS's upload rate is 150 KB/s and BigMac's upload rate is 2.5 MB/s. The XPS has a 2.5 MB/s download rate. The routing times between the XPS and BIGMAC, over the Internet (using traceroute) were:

```
1   dd-wrt  : 2949.275 ms  0.290 ms  0.193 ms;
2   modem : 55.319 ms  28.205 ms  19.802 ms;
3   hop 1 : 9.809 ms  9.440 ms  26.398 ms;
4   hop 2 : 9.684 ms  11.549 ms  9.547 ms;
5   firewall : 11.608 ms  11.091 ms  11.307 ms;
6   destination :11.655 ms  10.395 ms  9.621 ms.
```

All tag names are encoded as variable sized integers depending on the number of unique elements in our synchronous dictionary. For example, the following approach could be used in determining the encoding:

```
if (elementDictionary.size() < 127)
        //code is a byte
else if (elementDictionary.size() < 32767)
        //code is a short
else  // resort to integer
```

Values 127 and 32767 (or $2^{(8-1)}$-1 and $2^{(2*8-1))}-1$) are used because the most significant bit in each encoding is used for handling attribute elements, e.g., if the bit is set, a specific tag includes an attribute added encoding.

For sending annotations, which are possibly very long sequences of non-negative integer values, one modification can be made to the algorithm to improve performance. There are several possibilities as to how annotation data can be sent from the sending node to the receiving node: (1) sending annotations ANN(n) for each tag node n as this node is encountered during the online decompression; (2) sending the entire sequence ANN of all annotations after decompressing has been completed; and (3) sending ANN compressed (compressing ANN(n) would be useless as these sequences may be short and so the compression may actually be detrimental). For the case of sending all annotations, let us recall from [11] that based on the parents annotation summation, one can figure out the number of integers required for each child, and this is how XSAQCT stores the annotations. It appears that sending compressed annotations should be advantageous and to decide on which option should be chosen, and to test this claim a series of experiments to find out the size of data was carried out. The results are provided in Table IV, in which "uncompressedI" and "uncompressedV" denote respectively sending all data (including annotations) encoded as Integers or Variable Length Integers, and "compressed" means sending all data, including compressed annotations. Based on results from Table IV, compressed annotations encoded as Variable Length Integers (determining the variable length can be stored during the parsing/compressing procedure) are sent on a per-node basis. Finally, note that "Per Node Uncompressed" is not the same as "All Annotations Uncompressed" because of the concept of clean nodes (all annotations are equal to'1') dirty nodes (all remaining nodes). Thus, annotations for clean nodes do not have to be stored; rather nodes are qualified as clean or dirty.

B. *Data Transfers and Timing Results*

The implementation was tested for offline and online XSAQCT. Four algorithms were compared: (1) Send-and-Compress, denoted below by SC, sending a single XML file D over the network from node N1 to node N2 and then compressing offline in N2; (2) Compress-and-Send, denoted below by CS, compressing D offline on N1 and sending to N2; (3) compressing D using online XSAQCT with the basic algorithm; and (4) compressing D using online XSAQCT with the improved algorithms (both online algorithms were described in Section III B2, in all tables these algorithms are denoted respectively by Online (1) and Online (2)). The sizes of data transferred for each algorithm were computed using both the RAW mode (data sent uncompressed) and the COMPRESS mode (data sent compressed with GZIP).

Each timing test was repeated *three times* and all tables show *the average times* (in seconds) for compression and for decompression, respectively.

There are two possible transmission scenarios: saturated and unsaturated. If the transmission is unsaturated, i.e., the maximum transfer rate is greater than the maximum receiving and then the processing rate, the receiver will never have to block, i.e., wait for data.

TABLE IV. OVERHEAD OF SENDING ANNOTATIONS

File	Per node uncompressedI	Per node compressedI	Per node uncompressedV	Per node compressedV	All annotations uncompressed	All annotations compressed
enwiki-latest-stub-articles	1,126,519,254	691,503,851	794,899,751	691,164,783	1,126,519,147	691,503,851
1gig	373,697,822	328,214,230	337,804,153	327,762,877	373,696,652	328,214,230
enwikibooks-20061201-pages-articles	45,654,608	44,324,089	44,771,053	44,505,494	45,654,524	44,518,913
Dblp	62,291,449	19,433,781	30,420,794	19,218,025	62,291,170	19,396,295
SwissProt	30,485,489	7,663,524	13,175,233	7,469,093	30,484,796	7,627,024
enwikinews-20061201-pages-articles	13,191,594	12,614,494	12,749,145	12,613,409	13,191,510	12,618,319
Lineitem	1,434,856	1,432,365	1,434,860	1,434,880	1,434,791	1,434,759
Shakespeare	2,995,612	1,896,990	2,140,288	1,881,201	2,995,456	1,891,251
Uwm	135,414	101,556	109,031	101,497	298,563	101,826
Baseball	298,620	49,554	100,157	44,296	135,322	46,660
Macbeth	64,452	43,652	47,993	43,575	64,354	43,389

If the transmission is saturated, the receiving node sometimes has to wait for data to process, and so it will sometimes block. Timing is more important for the unsaturated transmission, whereas data transfer is more important for the saturated one. However, the results for latter type of transmission fall in line with what was described in section I.

To test various kinds of environments, we created the three experiments: XPS -> (1) BigMac was heavily saturated: (2) BigMac -> XPS was semi-saturated, and (3) LAN was unsaturated.

In our future work, we will try to develop a saturation metric, e.g., Saturation estimate = amount of time on IO wait queue / total amount of processing time (the higher the number, the more network-dependent the processing is).

Tables V and VI provide RAW and GZIP data transfer results, respectively. These two tables show that the offline compression CS is always the most space-efficient algorithm, i.e., it transfers the least amount of data. Note, however, that for the GZIP mode the differences between the online algorithms and the offline algorithms are less profound. To explain the reason for these results, note that in a RAW mode, using CS, text and annotations are always compressed, while in online compression the packets (specifically text data) are not compressed.

TABLE V. RAW DATA TRANSFER RESULTS (IN BYTES)

File	File Size	CS	SC	Online (1)	Online (2)
enwiki-latest-stub-articles	5,961,966,106	694,547,020	5,961,966,106	4,124,439,288	4,055,267,064
1gig	1,172,322,551	329,242,185	1,172,322,551	947,901,973	937,671,984
enwikibooks-20061201-pages-articles	156,300,597	44,518,962	156,300,597	143,338,449	143,316,025
dblp	133,862,399	19,396,313	133,862,399	92,448,731	87,775,673
SwissProt	114,820,211	7,627,047	114,820,211	70,294,919	66,572,952
enwikinews-20061201-pages-articles	46,418,850	12,618,367	46,418,850	41,775,533	41,746,933
lineitem	32,235,298	1,434,781	32,235,298	11,294,640	9,609,740
shakespeare	7,647,996	1,891,276	7,647,996	5,774,201	5,617,525
uwm	2,337,523	101,843	2,337,523	1,337,690	1,311,656
baseball	671,924	46,682	671,924	212,536	163,496
macbeth	163,077	43,410	163,077	121,776	118,196

TABLE VI. GZIP DATA TRANSFER RESULTS (IN BYTES)

File	File Size	CS	SC	Online (1)	Online (2)
enwiki-latest-stub-articles	953,599,509	694,547,020	953,599,509	886,753,284	886,449,292
1gig	384,712,148	329,242,185	384,712,148	371,180,220	371,174,799
enwikibooks-20061201-pages-articles	45,692,602	44,518,962	45,692,602	45,070,832	45,064,449
dblp	24,486,638	19,396,313	24,486,638	22,718,443	22,661,594
SwissProt	14,140,327	7,627,047	14,140,327	12,328,793	12,308,808
enwikinews-20061201-pages-articles	13,041,266	12,618,367	13,041,266	12,817,088	12,815,965
lineitem	2,911,297	1,434,781	2,911,297	2,331,926	2,197,236
shakespeare	2,140,436	1,891,276	2,140,436	2,020,343	2,033,234
uwm	161,692	101,843	161,692	142,912	141,733
baseball	66,769	46,682	66,769	54,109	48,225
macbeth	46,658	43,410	46,658	44,450	44,809

Therefore, comparing these ways of compressing data is not quite fair (the difference in amount of data that has to be transferred shows this.) Our future work will consider a way to deal with this issue by not compressing the annotation and text containers in RAW mode.

For all algorithms in GZIP mode, all data for Online (1), Online(2) and SC are compressed. These results are not surprising because offline and online algorithms have several distinctively different features. Specifically, in terms of document scope, online XSAQCT has a scope local to a path and it is forced to interleave more data thereby increasing the amount of information entropy and reducing the compression ratio. At the same time, offline XSAQCT has a scope of an entire file (and similar data can be compartmentalized by using the container methodology and compressed at a lower rate).

The remaining part of this section discusses timing results. Table VII provides the LAN-based (unsaturated) compression timing results using the RAW mode. For each file, the most efficient timing of the online algorithm is shown in bold face, the most efficient timing of the offline algorithm is shown in italics.

TABLE VII. LAN-BASED *RAW* COMPRESSION TIMING RESULTS

File	CS	SC	Online (1)	Online (2)
enwiki-latest-stub-articles	*431.713*	583.283	1514.077	**1490.39**
1gig	*124.715*	159.527	335.796	**296.67**
enwikibooks-20061201-pages-articles	*15.774*	18.418	32.749	**31.478**
dblp	*12.533*	14.136	33.282	**32.544**
SwissProt	*9.415*	10.57	31.077	**29.705**
enwikinews-20061201-pages-articles	*4.945*	5.084	9.688	**9.477**
lineitem	*2.711*	2.923	7.899	**7.804**
shakespeare	1.708	*1.464*	2.886	**2.539**
uwm	0.889	*0.367*	1.405	**1.025**
baseball	0.991	*0.414*	1.276	**0.959**
macbeth	0.416	*0.231*	0.428	**0.402**

TABLE VIII. COMPARISON OF LAN-BASED *RAW* COMPRESSION TIMING RESULTS

File	SC vs. Online (1)	SC vs. Online (2)	CS vs. Online (1)	CS vs. Online (2)
enwiki-latest-stub-articles	3.5071	3.4523	2.5958	2.5552
1gig	2.6925	2.3788	2.1049	1.8597
enwikibooks-20061201-pages-articles	2.0761	1.9956	1.7781	1.7091
dblp	2.6555	2.5967	2.3544	2.3022
SwissProt	3.3008	3.1551	2.9401	2.8103
enwikinews-20061201-pages-articles	1.9592	1.9165	1.9056	1.8641
lineitem	2.9137	2.8786	2.7024	2.6699
shakespeare	1.6897	1.4865	1.9713	1.7343
uwm	1.5804	1.1530	3.8283	2.7929
baseball	1.2876	0.9677	3.0821	2.3164
macbeth	1.0288	0.9663	1.8528	1.7403

TABLE IX. LAN-BASED *GZIP* COMPRESSION TIMING RESULTS

File	SC	CS	Online (1)	Online (2)
enwiki-latest-stub-articles	*442.281*	578.697	1333.298	**1322.663**
1gig	*135.782*	159.901	287.905	**286.292**
enwikibooks-20061201-pages-articles	*17.186*	18.006	31.93	**30.5**
dblp	*13.159*	14.098	31.649	**31.092**
SwissProt	*9.993*	10.631	28.565	**28**
enwikinews-20061201-pages-articles	5.294	*5.271*	9.938	**9.159**
lineitem	2.942	*2.706*	7.714	**7.306**
shakespeare	1.736	*1.38*	2.91	**2.618**
uwm	0.864	*0.372*	1.383	**1.039**
baseball	0.998	*0.454*	1.169	**1.07**
macbeth	0.42	*0.302*	0.476	**0.388**

TABLE X. COMPARISON OF LAN-BASED GZIP COMPRESSION TIMING RESULTS

File	SC vs. Online (1)	SC vs. Online (2)	CS vs. Online (1)	CS vs. Online (2)
enwiki-latest-stub-articles	3.0146	2.9905	2.3040	2.2856
1gig	2.1203	2.1085	1.8005	1.7904
enwikibooks-20061201-pages-articles	1.8579	1.7747	1.7733	1.6939
dblp	2.4051	2.3628	2.2449	2.2054
SwissProt	2.8585	2.7864	2.6870	2.6191
enwikinews-20061201-pages-articles	1.8772	1.7301	1.8854	1.7376
lineitem	2.6220	2.4833	2.8507	2.6999
shakespeare	1.6763	1.5081	2.1087	1.8971
uwm	1.6007	1.2025	3.7177	2.7930
baseball	1.1713	1.0721	2.5749	2.3568
macbeth	1.1333	0.9238	1.5762	1.2848

TABLE XI. COMPARISON OF XPS-BIGMAC RAW COMPRESSION TIMING RESULTS

File	CS vs. Online (1)	CS vs. Online (2)	SC vs. Online (1)	SC vs. Online (2)
1gig	2.7629	2.7256	0.8510	0.8395
enwikibooks-20061201-pages-articles	2.9429	3.0194	0.9093	0.9329
dblp	3.8989	3.9489	0.6381	0.6463
SwissProt	7.5003	7.4432	0.6228	0.6180
enwikinews-20061201-pages-articles	3.2949	3.2647	0.9502	0.9415
lineitem	6.0347	5.2047	0.3320	0.2864
shakespeare	2.8017	2.4821	0.8275	0.7331
uwm	5.9046	5.8614	0.4975	0.4939
baseball	1.2254	1.2269	0.4229	0.4234
macbeth	1.5505	1.1284	1.1118	0.8092

TABLE XII. COMPARISON OF XPS-BIGMAC GZIP COMPRESSION TIMING RESULTS

File	CS vs. Online (1)	CS vs. Online (2)	SC vs. Online (1)	SC vs. Online (2)
enwiki-latest-stub-articles	1.4705	1.3260	1.1789	1.0630
1gig	1.0699	1.0682	0.9738	0.9723
enwikibooks-20061201-pages-articles	0.9470	0.8740	0.9209	0.8499
dblp	1.0905	1.0019	0.9606	0.8825
SwissProt	1.2103	1.1432	0.7632	0.7209
enwikinews-20061201-pages-articles	0.9099	0.8622	0.8745	0.8286
lineitem	1.3191	1.1031	0.7305	0.6109
shakespeare	0.8870	0.8963	0.8230	0.8316
uwm	0.8489	0.8435	1.1849	1.1773
baseball	0.7132	0.7421	1.5320	1.5940
macbeth	0.8433	0.8762	1.5504	1.6110

TABLE XIII. COMPARISON OF BIGMAC-XPS RAW COMPRESSION TIMING RESULTS

File	CS vs. Online (1)	CS vs. Online (2)	SC vs. Online (1)	SC vs. Online (2)
enwiki-latest-stub-articles	1.0844	0.9798	1.1119	1.0046
1gig	1.6375	1.5859	0.8379	0.8115
enwikibooks-20061201-pages-articles	1.8667	1.8947	0.9188	0.9326
dblp	1.8849	1.7962	0.6987	0.6658
SwissProt	2.3250	2.3710	0.6353	0.6479
enwikinews-20061201-pages-articles	1.4902	0.8523	1.5875	0.9080
lineitem	2.1719	2.2143	0.5404	0.5509
shakespeare	1.2205	1.2135	0.7730	0.7685
uwm	0.9653	0.9784	0.9734	0.9867
baseball	1.8118	0.9524	1.8691	0.9826
macbeth	0.7094	0.7117	1.4904	1.4952

TABLE XIV. COMPARISON OF BIGMAC-XPS GZIP COMPRESSION TIMING RESULTS

File	CS vs. Online (1)	CS vs. Online (2)	SC vs. Online (1)	SC vs. Online (2)
enwiki-latest-stub-articles	2.1716	2.0828	2.6257	2.5184
1gig	1.0640	1.0607	0.9467	0.9437
enwikibooks-20061201-pages-articles	0.6199	0.6194	0.9985	0.9976
dblp	0.5847	0.5442	1.1176	1.0403
SwissProt	0.7437	0.7366	1.2010	1.1894
enwikinews-20061201-pages-articles	0.6035	0.5981	0.9672	0.9585
lineitem	1.0138	0.9686	1.3551	1.2947
shakespeare	0.6804	0.6356	0.9356	0.8740
uwm	0.8685	0.8200	0.8685	0.8200
baseball	0.7255	0.7098	0.7255	0.7098
macbeth	0.7083	0.7083	0.7083	0.7083

These results show that for the first seven largest files CS is more efficient than SC, while Online(2) is always more efficient than Online (1). A comparison of timing results for the offline and online algorithms is provided in Table VIII, where "X vs. Y" gives the ratio of the timing result of Y divided by the timing result of X; therefore the value greater than one indicates that X is more efficient than Y. These results indicate that offline algorithms are more efficient than online algorithms.

Table IX is similar to Table VII, but it provides the LAN-based compression timing results using the GZIP mode. These results confirm that for the first seven largest files CS is the most efficient algorithm, while for the remaining four smaller files, SC is the most efficient. A comparison of the four algorithms is provided in Table X, using the same technique as in Table VIII. Results from this table confirm that both offline algorithms are less time-efficient than the online algorithms.

Besides LAN-based tests, two other sets of tests (using the RAW and the GZIP mode) were performed respectively sending data from XPS to BigMac (with 150 KB/s download rate) and sending data from BigMac to XPS (with 2.5 MB/s upload rate). For the former case, the results for the largest (over 5G in size) enwiki-latest-stub-articles.xml file in the RAW mode are not provided because it takes too much time to transfer data. Results provided in Tables XI and XII indicate that for sending data from XPS to BigMac, Online(2) is faster than SC, but the offline algorithm CS is the fastest of the four algorithms. Tables XIII and XIV are similar to Tables XI and XII, but they provide comparison of the timing results for sending data from BigMac to XPS, respectively using the RAW and GZIP mode. Results from these tables are similar to previous results and show that offline algorithms are faster than online algorithms. Now, we describe *decompression.* Here, the **client** rebuilds the XML file and the **server** sends the compressed representation. Therefore, timing results may be disproportionate, because the client has to decompress every text container and then rebuild the document. However, in a client-server paradigm, where a server may be answering many clients' requests, in our future work this disproportionality may prove to be more beneficial.

Table XV provides LAN-based decompression timing results using the RAW and GZIP mode. While our algorithms are not very fast comparing to decompression of GZIP-ed file, they always send less data than just using GZIP (see Tables III, V and VI), which is another argument for using XML-based compression techniques. Tables XVI and XVII provide similar results to the results from Table XV, but for sending data between BigMac and XPS (missing row in Table XVII indicates that decompression of the corresponding files was taking too much time). Results from these tables are consistent with other results and show that in non-low-bandwidth situations, online decompression is slow in comparison to just decompressing an ordinary GZIP-ed file. In low bandwidth situations, the added compression proves to overcome the disparity in processing times.

The remaining part of this section describes results of tests aimed to compare both versions of online algorithms with the algorithm referred to as SCU, which performs the offline compression, then it sends the compressed file and finally the receiver performs the offline decompression.

Tables XVIII to XXIII provide respectively RAW and GZIP results of tests. Table XVIII shows that online algorithms are best (with Online(2) being marginally a winner). Note, however, that Online(1) and Online (2) are very similar and "marginally better" falls into the margin of error.

Table XIX shows that base (sending GZIP as is) is the best because data is being sent so fast that there is no point in running an extra algorithm on it (recall from the first paragraph in this paper "low bandwidth networks").

Table XX shows that SCU is the best, because decompression time is less than time to send online (in raw mode, the amount of data one has to send is very large in comparison to SCU). To understand results from the remaining tables, it is useful to recall the Tables V and VI showing the amount of data to transfer.

TABLE XV. LAN-BASED DECOMPRESSION TIMING RESULTS (IN S)

File	RAW	GZIP	Server	Client
enwiki-latest-stub-articles	594.018	217.179	1542.516	347.794
1gig	114.79	84.579	201.277	185.177
enwikibooks-20061201-pages-articles	14.643	10.741	19.951	11.186
dblp	12.704	6.197	41.191	18.847
SwissProt	9.978	3.453	32.294	7.009
enwikinews-20061201-pages-articles	3.996	2.731	7.284	3.724
lineitem	2.797	1.039	5.947	0.425
shakespeare	0.663	0.51	0.217	0.814
uwm	0.224	0.075	0.548	0.111
baseball	0.07	0.027	0.43	0.184
macbeth	0.048	0.124	0.216	0.07

TABLE XVI. BIGMAC TO XPS -BASED DECOMPRESSION TIMING RESULTS

File	RAW	GZIP	Server	Client
enwiki-latest-stub-articles	2543.355	406.214	1588.146	444.263
1gig	499.474	179.658	260.695	244.751
enwikibooks-20061201-pages-articles	64.419	20.938	26.542	18.478
dblp	55.493	12.37	39.658	18.096
SwissProt	48.8778	9.03	31.338	7.429
enwikinews-20061201-pages-articles	18.875	5.148	8.657	5.37
lineitem	13.044	1.356	6.141	0.768
shakespeare	2.987	1.295	2.333	0.932
uwm	0.702	0.114	0.586	0.141
baseball	0.252	0.084	0.417	0.149
macbeth	0.316	0.069	0.226	0.065

TABLE XVII. XPS TO BIGMAC -BASED DECOMPRESSION TIMING RESULTS

File	RAW	GZIP	Server	Client
1gig	6349.561	2413.013	2076.258	2040.6
enwikibooks-20061201-pages-articles	1027.346	281.652	246.217	232.465
dblp	840.802	147.557	158.461	103.082
SwissProt	730.773	84.544	102.709	40.591
enwikinews-20061201-pages-articles	288.547	78.493	71.056	64.388
lineitem	201.886	15.837	22.677	6.226
shakespeare	47.243	10.156	12.508	8.629
uwm	15.82	0.058	1.785	0.301
baseball	4.672	0.028	0.776	0.187
macbeth	1.131	0.015	0.355	0.073

Tables XXI and XXII show that online algorithms are best, because decompression time is greater than time to send the extra data. Table XXIII shows that for enwiki-latest-stub-articles.xml file, the base algorithm is the best (otherwise, online algorithms are the best). The reason for this result is that this file is very “text heavy” and for creating packets, text data has to be encoded and buffered. The internal libraries used for this encoding is quite memory/computationally expensive resulting in slow-downs. Our future work will deal with these shortcomings.

TABLE XVIII. LAN-BASED COMPARISON USING SCU OF RAW TIMING RESULTS

File	Online(1)	Online(2)	SCU	Base (sending as is)
enwiki-latest-stub-articles	403.7920	402.9060	2645.9100	594.0180
1gig	115.7440	115.1730	281.7290	114.7900
enwikibooks-20061201-pages-articles	14.1610	14.4710	26.0260	14.6430
dblp	9.7080	8.5060	37.6960	12.7040
SwissProt	8.0270	7.1300	35.2200	9.9780
enwikinews-20061201-pages-articles	4.2470	4.5280	8.7070	3.9960
lineitem	1.9380	1.7380	7.7970	2.7970
shakespeare	1.1030	1.2610	2.8070	0.6630
uwm	0.7790	0.7860	1.1120	0.2240
baseball	0.6660	0.6000	1.4000	0.0700
macbeth	0.3090	0.4060	0.4020	0.0480

TABLE XIX. LAN-BASED COMPARISON USING SCU OF GZIP TIMING RESULTS

File	Online(1)	Online(2)	SCU	BASE (sending GZIP-ed)
enwiki-latest-stub-articles	468.5560	459.3120	2645.9100	217.1790
1gig	118.8480	117.0300	281.7290	84.5790
enwikibooks-20061201-pages-articles	15.2410	15.1560	26.0260	10.7410
dblp	12.3800	12.8280	37.6960	6.1970
SwissProt	10.6750	8.7150	35.2200	3.4530
enwikinews-20061201-pages-articles	4.3790	4.6250	8.7070	2.7310
lineitem	2.7780	2.3230	7.7970	1.0390
shakespeare	1.4140	1.4540	2.8070	0.5100
uwm	0.7640	0.7800	1.1120	0.0750
baseball	0.5810	0.5040	1.4000	0.0270
macbeth	0.3360	0.3240	0.4020	0.1240

TABLE XX. XPS-BIGMAC-BASED COMPARISON USING SCU OF RAW TIMING RESULTS

File	Online(1)	Online(2)	SCU	Base (sending as is)
1gig	5337.4770	5224.8040	2578.4710	6349.5610
dblp	801.7320	789.7230	349.1410	1027.3460
SwissProt	517.5450	480.0680	214.6720	840.8020
enwikinews-20061201-pages-articles	382.7000	362.6890	134.6920	730.7730
lineitem	228.2950	234.8440	99.0010	288.5470
shakespeare	59.9060	50.8520	30.3240	201.8860
uwm	6.7080	5.8030	3.2170	15.8200
baseball	0.6810	0.6200	1.6600	4.6720
macbeth	0.3350	0.2770	0.8030	1.1310

TABLE XXI. XPS-BIGMAC-BASED COMPARISON USING SCU OF GZIP TIMING RESULTS

File	Online(1)	Online(2)	SCU	BASE (sending GZIP-ed)
enwiki-latest-stub-articles	5811.5030	5729.5110	7747.8340	5951.0740
1gig	2186.6430	2169.7950	2578.4710	2413.0130
enwikibooks-20061201-pages-articles	271.0320	246.7336	349.1410	281.6520
dblp	133.3010	125.6450	214.6720	147.5570
SwissProt	69.4780	68.8730	134.6920	84.5440
enwikinews-20061201-pages-articles	70.5520	68.0460	99.0010	78.4930
lineitem	13.1660	10.7800	30.3240	15.8370
shakespeare	10.6080	10.5420	22.3760	10.1560
uwm	1.0240	1.0130	3.2170	0.0580
baseball	0.7710	0.7320	1.6600	0.0280
macbeth	0.3100	0.2950	0.8030	0.0150

TABLE XXII. BIGMAC-XPS-BASED USING SCU COMPARISON OF RAW TIMING RESULTS

File	Online(1)	Online(2)	SCU	Base (sending as is)
enwiki-latest-stub-articles	1703.8838	1691.5070	2710.9930	2543.3550
1gig	381.8510	377.2150	391.7770	499.4740
enwikibooks-20061201-pages-articles	59.0470	57.0220	41.7830	64.4190
dblp	37.9620	36.1100	47.7650	55.4930
SwissProt	28.9790	27.5730	38.4140	48.8778
enwikinews-20061201-pages-articles	17.1140	17.0990	12.6370	18.8750
lineitem	4.6600	3.9560	8.1360	13.0440
shakespeare	2.6050	2.2130	2.9730	2.9870
uwm	0.6210	0.6190	0.9630	0.7020
baseball	0.4240	0.4280	0.8420	0.2520
macbeth	0.2150	0.2140	0.3400	0.3160

TABLE XXIII. BIGMAC-XPS -BASED USING SCU COMPARISON OF GZIP TIMING RESULTS

File	Online(1)	Online(2)	SCU	Base (sending GZIP-ed)
enwiki-latest-stub-articles	692.1700	637.3570	2710.9930	406.2140
1gig	173.8370	167.4110	391.7770	179.6580
enwikibooks-20061201-pages-articles	22.2400	20.8570	41.7830	20.9380
dblp	17.1090	16.2570	47.7650	12.3700
SwissProt	12.1040	11.5370	38.4140	9.0300
enwikinews-20061201-pages-articles	6.3650	5.9300	12.6370	5.1480
lineitem	3.1150	2.7520	8.1360	1.3560
shakespeare	1.8560	1.4790	2.9730	1.2950
uwm	0.6000	0.6030	0.9630	0.1140
baseball	0.4770	0.4440	0.8420	0.0840
macbeth	0.2390	0.2520	0.3400	0.0690

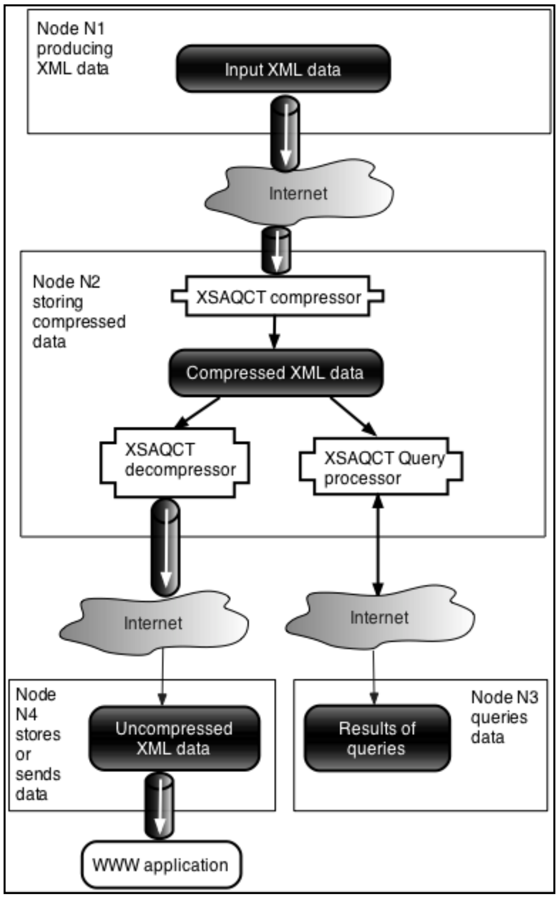


Figure 7. Applications

VI. EXAMPLES OF APPLICATIONS

For the sake of completeness, here we recall from [1] an example of an application. Consider Fig. 7, in which the network node N1 produces XML data to be sent to the network node N2, where they are compressed online by XSAQCT and then they can be queried by N3. This data can also be decompressed online by XSAQCT and sent to a new network node N4, which can either store this uncompressed data, or pipe it into any WWW application.

VII. CONCLUSION AND FUTURE WORK

This paper presented XSAQCT, an online XML compressor/decompressor. The original hypothesis was that the online compression will be more efficient than the offline compression because for the online some actions may be performed "in parallel", i.e., when N1 sends to N2 online, N2 will start decompressing as soon as it gets a chunk of data and at the same time N1 will be sending the next chunk.

The problem with this claim was the dependence on network bandwidth. In low bandwidth situations, several issues might invalidate it because in case of producing data faster than transferring it, all modern operating systems will intentionally block the process because internal network buffers are full, or cannot accommodate the required data.

This paper provided a brief outline of the implementation and results of tests to evaluate the effectiveness of online XSAQCT; specifically amounts of data transfers and compression and decompression times (in s).

The tests show that for high bandwidth network, and for *existing* files the online compression is less efficient than the offline compression. However, the online compression is superior when compared to offline compression combined with sending the file through the network and subsequent decompression. In addition, the online compression is useful for streaming, i.e., when (potentially generated) XML data is streamed from another network node.

Note that timing results are less important than actual compression ratios because characteristics of the different hardware and operating system may affect timing results as packets are sent through the networking stack.

In our future work, we will attempt a development of a formalization of conditions (which do not factor in processing loads) under which one type of compression would perform better than the other: Let X be the Offline Compression Time, Y be Offline Compressed Size, Z be the Online Compressed Size, and U be the Upload Rate. Assuming that the Online Compression Time is 0 (because there is no waiting period to send data, let T(Offline) = (X + (Y/U)), T(Online) = (Z/U), and R = T(Offline) / T(Online). Based on the value of R, one can define (with a pretty high accuracy) the conditions required for online compression to be better than offline compression and vice-versa.

We will also design, implement and test other versions of the online compression by mimicking the SAX parser on the receiving end, rather than sending full information about the nodes (here by mimicking, we mean sending bit-encoded SAX events). Therefore, instead of using a byte, or a variable-length byte encoding, we will investigate working on the bit level.

We will also test different ways of compressing data, and annotations specifically, instead of using ordinary GZIP, we will use BZIP, LZMA, Golomb, and Delta Encoding combined with GZIP. The latter type of compression may be beneficial as typically annotation lists are not a list of random numbers and there is some inherent pattern to them. Instead of using a static dictionary, we will use a more adaptive approach (e.g., a frequency based dictionary) to achieve higher compression rates. Also the future version will add more querying and updating facilities. The complexity of our algorithms will be analyzed, including their footprint.

Finally, we will add parallelization to the online compressor, based on our earlier work reported in [13].

ACKNOWLEDGMENT

The authors would like to thank anonymous reviewers for their helpful comments.

REFERENCES

[1] T. Müldner, J. K. Miziołek, and C. Fry, "Online Internet Communication Using an XML Compressor," The Seventh International Conference on Internet and Web Applications and Services, ICIW 2012, Stuttgart, Germany, 2012, pp. 131-136.

[2] H. Liefke and D. Suciu, "XMill: an efficient compressor for XML data," Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, United States, 2000, pp. 153–164.

[3] P. Tolani and J. Haritsa, "XGRIND: a query-friendly XML compressor," Proc. of 18th IEEE Intl. Conf. on Data Engineering (ICDE), San Jose, USA, February 2002, pp. 225-234.

[4] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese, "XQueC: pushing queries to compressed XML data," Proceedings of the 29th international conference on Very large data bases - Volume 29, Berlin, Germany, 2003, pp. 1065–1068.

[5] P. Skibiński and J. Swacha, "Combining efficient XML compression with query processing," Advances in Databases and Information Systems, 2007, pp. 330–342.

[6] Y. Lin, Y. Zhang, Q. Li, and J. Yang, "Supporting Efficient Query Processing on Compressed XML Files," SAC '05 Proceedings of the 2005 ACM symposium on Applied computing, Santa Fe, New Mexico, pp. 660-665.

[7] T. Müldner, C. Fry, J. K. Miziołek, and T. Corbin, "Updates of Compressed Dynamic XML Documents," The Eighth International Network Conference (INC2010), Heidelberg, Germany, July 2010, pp. 315–324.

[8] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld, "Updating XML," Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, California, United States, 2001, pp. 413–424.

[9] S. Sakr, "An Experimental Investigation of XML Compression Tools," CoRR, vol. abs/0806.0075, 2008.

[10] T. Müldner, G. Leighton, and J. Diamond, "Using XML Compression for WWW Communication," IADIS International Conference WWW/Internet 2005, Lisbon, Portugal, October 2005, pp. 459–466.

[11] T. Müldner, C. Fry, J. K. Miziołek, and S. Durno, "XSAQCT: XML Queryable Compressor," Balisage: The Markup Conference, Montréal, Canada (August 2009, DOI 10.4242/BalisageVol3.Muldner01 Montréal, Canada, 2009.

[12] G. Leighton, T. Müldner, and J. Diamond, "TREECHOP: A Tree-based Query-able Compressor for XML," The Ninth Canadian Workshop on Information Theory, Montréal, Canada, June 2005, pp. 115-118.

[13] T. Müldner, C. Fry, T. Corbin, and J. K. Miziołek, "Parallelization of an XML Data Compressor on Multi-cores," Torun, Poland, 2011, PPAM 2, volume 7204 of Lecture Notes in Computer Science, Springer, (2011), pp. 101-110.

[14] T. Müldner, J. K. Miziołek, and C. Fry, "Updateable Educational Applications based on Compressed XML Documents," Proceedings of the 3rd International Conference on Computer Supported Education, Volume 1, Noordwijkerhout, Netherlands, 6-8 May, 2011, pp. 369–371.

[15] W3C, Canonical XML. http://www.w3.org/TR/xml-c14n, retrieved on July 20, 2012.

[16] W3C Extensible Markup Language (XML) 1.0 (Fifth Edition), http://www.w3.org/TR/REC-xml/, retrieved on July 20, 2012. 2012.

[17] "Xerces," http://xerces.apache.org/xerces-j/, retrieved on July 20, 2012.

[18] The GZIP home page. http://www.gzip.org/, retrieved on July 20, 2012.

[19] enwiki-latest-stub-articles.xml. http://dumps.wikimedia.org/enwiki/latest/, retrieved on July 20, 2012.

[20] "xmlgen - The Benchmark Data Generator". http://www.xml-benchmark.org/generator.html.

[21] Wratislavia XML Corpus. http: //www.ii.uni.wroc.pl/ \extasciitildeinikep/research/Wratislavia, retrieved on July 20, 2012.

[22] Baseball.xml. http: //rassyndrome.webs.com/ CC/ Baseball.xml. retrieved on July 20, 2012.

[23] Macbeth.xml. http: //www.ibiblio.org/xml/examples/ shakespeare/macbeth.xml, retrieved on July 20, 2012

[24] "STAX Parsing. Streaming XML API". http: //java.sun.com/webservices/reference/tutorials/jaxp/html/stax.html#bnbdx, retrieved on July 20, 2012

[25] Efficient {XML} Interchange {(EXI)} Format 1.0, http://www.w3.org/TR/exi/, retrieved on October 2012

[26] S. Snyder, "Efficient XML Interchange (EXI) Compression and performance benefits: Development, Implementation and Evaluation Naval Postgraduate School, Monterey, California, Masters Thesis 2010. http://www.dtic.mil/cgi-bin/GetTRDoc?AD= ADA518679, retrieved on October 2012

[27] exificient, http://exificient.sourceforge.net, retrieved on October 2012.