Pre-order Compression Schemes for XML in the Real Time Environment

Tyler Corbin[1], Tomasz Müldner[1] and Jan Krzysztof Miziołek[2]
[1]*Jodrey School of Computer Science, Acadia University, Wolfville, B4P 2A9 NS, Canada*
[2]*IBI AL, University of Warsaw, Warsaw, Poland*

Keywords: XML, Online Compression, Network Performance.

Abstract: The advantages of using XML come at the cost, especially when used on networks and small mobile devices. This paper presents a design and implementation of four online XML compression algorithms, which exploit local structural redundancies of pre-order traversals of an XML tree, and focus on reducing the overhead of sending packets and maintaining load balancing between the sender and receiver. For testing, we designed a suite consisting of 11 XML files with various characteristics. Ten encoding techniques were compared, compressed respectively using GZIP, EXI, Treechop, XSAQCT and its improvement, and our algorithms. Experiments indicate that our new algorithms have similar or better performance than other online algorithms, and have only worse performance than EXI for files larger than 1 GB.

1 INTRODUCTION

The eXtensible Markup Language, XML (XML, 2012) is a World Wide Web Consortium (W3C) endorsed standard for semi-structured data. XML is simple, open-source, and platform independent, and has become the most popular markup language for the interchange and access of data between heterogeneous systems. Because of its textual format, XML's data sets may increase as much as ten-fold. A naive solution to reduce the XML format overhead uses a general-purpose data compressor, e.g., (GZIP, 2012). However, such compressors do not take advantage of the XML structure, thereby increasing first order entropy of the data. Therefore, there has been considerable research on *XML-conscious compressors*, e.g., XMill (Hartmut and Suciu, 2000), XGrind (Tolani and Haritsa, 2002) and XQueC (Arion et al., 2007).

XML-conscious compressors can be *homomorphic*, in which each node is processed during a pre-order traversal thereby preserving the original tree structure in the compressed representation, or *permutation-based*, in which the structure is separated from content to which a partitioning strategy is applied to group content nodes into a series of data containers compressed using general-purpose compressor (a *back-end compressor)*. A specific class of XML compressors consist of *queryable* XML compressors, which answer queries using *lazy decompression*, i.e., decompressing as little as possible; e.g., (Lin et al., 2005), and (Ng et al., 2006). Here, querying is based on path expressions, see (XPath, 2012) and (XQuery, 2012). Finally, another subset of XML compressors, which we focus on in this paper, is centred on real-time network activities. *Online* XML compressors are defined as compressors that work in real time, therefore the encoder or decoder processes chunks of data whenever possible rather than doing it *offline* when the entire document is available. Online XML compressors are particularly useful for streaming data, especially in conditions where the entire document may never be available without substantial buffering or on devices with limited hardware resources. For online compression, the efficiency of an algorithm involves several factors, including compression ratio, encoding/decoding times and network bandwidth. Therefore a collection of algorithms that can be tuned is often desired.

Contributions. This paper presents a design and implementation of four online XML compression algorithms, which exploit local structural redundancies of pre-order traversals of an XML tree. Our algorithms focus on reducing the overhead of sending packets and maintaining load balancing between the sender and receiver with respect to encoding and decoding. Specifically, we present (1) a SAX-Event based encoding scheme; (2) its improvement using bit packing; (3) path-centric compression, and its improvement with variable size buffers. For testing, we designed a suite consisting of 11 XML files with various characteristics. In order to take into account network issues, such as its bottleneck, we took two measures of the encoding process. Ten encoding techniques were compared, using GZIP, EXI, Treechop, XSAQCT and its improvement, and our new algorithms. Experiments show that our algorithms have similar or better performance than existing online algorithms, such as XSAQCT and Treechop, and have only worse performance than EXI for files larger than 1 GB.

This paper is organized as follows. Section 2 provides a description of related work and Section 3 describes our algorithms. Section 4 gives a description of the implementation and results of testing aimed at evaluating the efficiency of our algorithms, and finally Section 5 provides conclusions and future work.

2 RELATED WORK

Treechop (Leighton et al., 2005) is a queryable, online XML compressor, which allows streaming XML files. XQueC (Arion et al., 2007) is queryable, online XML compressor that given a query workload (a set of queries) can be tailored to decide the best method of compressing. However, a complexity analysis in (Leighton and Barbosa, 2009) showed that the problem of selecting an optimal compression configuration is NP-hard. XSAQCT(Müldner et al., 2009) is a queryable and updateable compressor. Although the original version of XSAQCT was offline, there is a more recent online version of XSAQCT (Müldner et al., 2012b) and (Müldner et al., 2012a). The Efficient XML Interchange, EXI (EXI, 2012) provides a very compact XML representation thereby optimizing performance and efficiently utilizing computational resources. While (Snyder, 2010) determined that using EXI can double a bandwidth potential and it is well-suited for the real-time network.

For a detailed comparison and evaluation of some of above-mentioned XML compressors, and other offline-variants, see (Sakr, 2008). Now, we will describe previous research on characteristics of XML files. (Qureshi and Samadzadeh, 2005) described the complexity of XML documents using the number of elements, the number of distinct element, the size of the document, and the depth of the document. (Measurements, 2012) adds two more characteristics, namely (1) content density defined as the ratio of the sum (in characters) of all texts and attribute values, over the size of the entire document in characters; (2) the structure regularity defined as one minus the ratio of the total number of distinct elements over the total number of elements in the XML document (documents with a few distinct elements have high structure regularity, while documents using many different elements have low structure regularity). Furthermore, Structure Regularity provides a meaningful value only when the distribution of element-occurrences is uniform. Otherwise the distribution of occurrences would provide more insights to the regularity of occurrence. In addition, (Ruellan, 2012) considers the first order entropy, and finds that while the entropy of XML documents is lower than the document's size, the compressed size (using GZIP) is sometimes lower than the entropy, showing that the entropy value does not represent the quantity of information contained in a XML document (the symbols used to compute the first-order entropy are not independent).

3 ONLINE ALGORITHMS

Our work on online compression is partially based on the XSAQCT compression process, therefore we start by recalling the basic terminology, based on (Müldner et al., 2012b) and (Müldner et al., 2012a).

3.1 Introduction to XSAQCT Compression

This section recalls the first phase of offline compression, in which the compressor transforms the original XML document into a more compressed form, called an *annotated tree*. Then it strips all annotations to a separate *container* and finally compresses all containers using a back-end compressor. These actions are performed using a SAX compliant parser (SAX, 2012).

Definition 1. *An **annotated tree** of a XML document is a tree in which all similar paths (i.e., paths that are identical, possibly with the exception of the last component, which is the data value) of a XML document are merged into a single path labeled by its tag name and each node is annotated with a sequence of integers (referred to as an **annotation list**).*

Definition 2. ***Character data** is data that must be associated with the correct parent node no matter what transformation of the input XML document is used. Conversely, **Markup Data** is data used to provide self documenting structure of the character data.*

For character data (and not markup data), compression must be strictly lossless; e.g., the text defined by XPath query /a[1]/b[101]/text() never changes.

Definition 3. *A* ***text container*** *of a XML path P is an ASCII zero delimited (and possibly indexed) list of all character data for all paths equal to P.*

Given an XML document as shown in Figure 1, its associated annotated tree is shown in Figure 3. Because of space limitations, we refer the reader to (Müldner et al., 2008) for the algorithms to create an annotated tree and to (Müldner et al., 2009) for the description of how cycles (consecutive children X, Y, X) are dealt with. For a XML document to be uniquely represented by a single annotation tree, it has to satisfy the full mixed content property, i.e., all tags in an XML document have to be separated by character data (see (Müldner et al., 2012a)).

An example of full mixed content is shown in Figure 1 and one that does not exhibit full mixed content would be when the node t3 from the same document was missing (i.e. the document ended *</b></a>*). To achieve full mixed content the encoder inserts empty text, consisting only of ASCII zero, whenever a text-node is missing; the decoder will neglect such empty texts.

3.2 Online Compression

Although the annotated tree represents a permutation-based representation of an XML document, it is a lossless representation of the XML document. For network transfers, assuming the annotated tree is already constructed, the encoder would first encode the tree using a left-child/right-sibling encoding scheme followed by the annotation list for each path in depth first order, and finally the actual compressed text for each text container (again in depth first order), thereby creating the compressed output O. Since the mapping which converts the input XML document to an annotated tree is one-to-one, after rebuilding the annotated tree, the receiver can decompress O to produce the original XML document D, query O using *lazy decompression* or write O to persistent storage for later use. Although this technique can satisfy soft-real time requirements, it is not scalable (i.e., the entire tree needs to be created) and does not translate well to hard-time requirements.

```
<?xml version="1.0" encoding="UTF-8"?>
<a>(t1)
        <b>(t4)
                <d>(t8)</d>(t5)
        </b>(t2)
        <b>(t6)
                <e>(t9)</e>(t7)
        </b>(t3)
</a>
```

Figure 1: A sample XML document D. (t#) represents (possibly whitespace) character data.

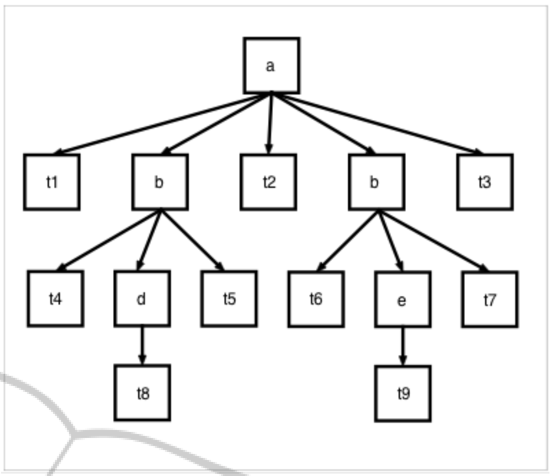

Figure 2: Tree representation of Figure 1.

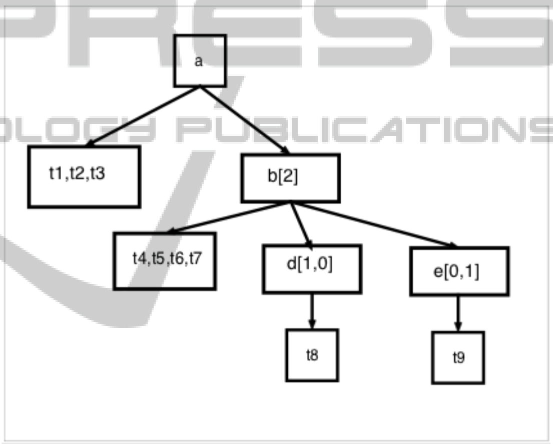

Figure 3: The annotated tree for the document D.

Definition 4. ***Online Compression*** *or real time compression is defined as an encoding or decoding scheme where the encoder operates on data as it is being received and the decoder is not forced to wait until the entire encoded document has been received; instead decoding of the document can be started as soon as the beginning of the encoded data stream is received.*

The majority of online compressors are homomorphic. While existing work on real time compression has focused on maximizing compression ratios; processing-time and (least frequently considered) bottleneck effects caused by network bandwidth have not been considered.

Definition 5. *There are two basic applications of the decoder: (1)* ***restoration****, the decoder restores the original XML document, and (2)* ***querying****, the decoder saves or transforms the data into a represen-*

tation suitable for subsequent queries.

In case of offline algorithms, for *restoration* the encoder builds a model/compressed representation of the original XML document; the decoder decompresses the compressed data and restores the original XML document. If this approach were used on a network, we would have singular-processing, i.e., the encoder/decoder would be inactive for the time it takes for decoding/encoding, and the network would be inactive during encoding and decoding. For *querying*, the encoder performs the same actions as for restoration, while the decoder stores the compressed data (or queries the compressed data). Therefore, depending on processing powers and bottleneck effects, in some cases offline compression in the networked environment would be beneficial.

In the case of online algorithms, for *restoration*, the encoder node produces encodings while the decoder node receives this encoding and outputs (to a network application or storage device) the original XML document, or in most of our applicative cases the annotated representation of the original XML document. From a network perspective, three situations can occur: (1) Perfect network utilization, in which the encoder and decoder are never bottlenecked by a network, or by one another and never have to wait or the process has to sleep for data to process; (2) Decoder starvation, in which the decoder (or its associated process) is often located on its associated operating system I/O wait queue, because the encoding process or the network bandwidth is limited; (3) Encoder starvation, in which the encoder (or its associated process) is often located on its associated operating system I/O wait queue, which can arise when the decoding process (or the network bandwidth) is lagging and the encoding socket is waiting for a RR or ACK request for specific TCP/UDP packet/frame. Therefore, cases (2) and (3) often occur in parallel on low-bandwidth networks. However, in terms of processing load, online algorithms have *intersection processing loads*. For *querying*, the encoding node produces encodings while the decoding node translates these encodings into a format that can be used to answer specific queries. In any case, an encoding can be translated back into its original XML representation and be queried using tools such as SOAP (soap, 2012), the encoding can directly interface with the *Query Streams* design pattern as described in (Leighton et al., 2005), or the encoding could be directly converted into an annotated representation and be queried as well. Regardless of situation, the processing load negatively favours the decoder.

The offline XSAQCT compression algorithm described in (Müldner et al., 2008) will be used as a baseline for comparison for two reasons. First, under most circumstances, the client receiving this data will be building the annotated tree for future querying (and use). Therefore if an online algorithm can transfer less encoded data than what an annotation algorithm produces, we will consider the online algorithm to be *efficient*. Second, while the scope of compression is greater for the annotated algorithm (i.e., a model is developed for the entire XML document, while for online compression a model is developed for *specific branches* of tree), compression may not necessarily be optimal for entire document modelling. For example, the zeros in Figure 3 tell the decoder that this specific subtree has no tag element. Therefore the goal of *our approach to online compression* will be to produce an encoding for XML that is more space efficient than the offline XSAQCT counterpart. This comparison will mostly revolve around the number of bits in representing the structure of an XML file, as character data (defined in Section 3.1) is often not manipulated. This is because learning grammars, caching substrings can incur severe space/time overheads, where-as using a real-time text compressor could be just as beneficial.

i. {-1, a, t1, b, t4, d, t8}
ii. {1, t5}
iii. {1, t2, b, t6, e, t9}
iv. {1, t7}
v. {1, t3}
vi. {EOF} // End of File

Figure 4: Example of XSAQCT online encoding.

Encoding. Figure 4 shows the encoding generated by the algorithm discussed in (Müldner et al., 2012b) for the document in Figure 1. The encoding is created by a pre-order traversal on the non-text nodes and whenever there is traversal up the tree, a new encoding packet is created. For the sake of brevity, the first element of each block (packet) of data is used to represent the number of elements required to be popped off from a stack (number of traversals going up the tree). In general, within a XML document, tag names may be repeated many times. The example shown in Figure 4 has only one tag repetition ('b'), but in general tag names are long and also dependant on the character-set of the data. Using a dictionary that can be synchronously built on the fly by both the encoder and the decoder, for tag names that have been seen before the encoder will only have to send indices within the dictionary, which will decrease the overhead of sending tag names.

Figure 5 depicts the encoding using a synchronous

i. {-1, 1, a, t1, 2, b, t4, 3, d, t8}	D={a,b,d}
ii. {1, t5}	D={a,b,d}
iii. {1, t2, 2, t6, 4, e, t9}	D={a,b,d,e}
iv. {1, t7}	D={a,b,d,e}
v. {1, t3}	D={a,b,d,e}
vi. {EOF} // End of File	D={a,b,d,e}

Figure 5: Node Name Dictionary.

dictionaries generated by the algorithm discussed in (Müldner et al., 2012b) for the document in Figure 1. The reasoning behind using a dictionary is that tag names (and attribute names) are often repeated many times. The example shown in Figure 5 has only one tag repetition ('b'), but in general tag names are long and also dependant on the character-set of the data. Therefore, a dictionary that can be synchronously built on the fly by both the encoder and the decoder, for tag names, is an easy technique for reduction of markup.

To handle XML attributes in a pre-order encoding, all attribute data can be encoded along with a dictionary index. For example, if the "d" node in Figure 1 had an attribute "href", a standard encoding would be {...3, nameAndAttrEncoding(d)...}. The next sections describe our main contributions.

3.2.1 Lightweight SAX Parsing

For compressing data we are interested in the four major components of an XML Document: (1) XML Declaration; (2) Start Element Tags; (3) End Element Tags; and (4) Character Data, which can be subcategorized as parsed character data, unparsed character data (processing instructions, comments), and intermittent namespace definitions. An XML declaration only occurs once, start and end tags define the pre-order traversal of a tree, and character data (recall Definition 2) encompasses everything else. Since we assume full-mixed content, all Start Element and (non-child) End Element events are followed with a text string which is, at minimum, an ASCII zero byte.

Encoding. Define the following two elements, both encoded by single bytes: (1) Start Element, encoded by 0x0; and (2) End Element, encoded by 0x1. The description of Lightweight SAX is provided in Algorithm 1. Figure 6 shows the encoding generated by this algorithm for the document in Figure 1. Note that the synchronous dictionary techniques can be used to reduce the overhead in sending node names.

Bit Packing Improvement. The encoding bytes 0x0

i. {0x0, a, t1, 0x0, b, t4, 0x0, d, t8}
ii. {0x1, t5, 0x1, t2}
iii. {0x0, b, t6, 0x0, e, t9, 0x1, t7}
iv. {0x1, t3, 0x1=EOF} // End of File

Figure 6: Encoding with Lightweight SAX.

Algorithm 1: Lightweight SAX.

```
Require: n is the Root on initial function call
Require: XML Declaration Information has all ready been
    transferred.
function ENCODE(Node n)
    // Start Element
    send(0x0);
    // send name+attribute
    send(handleAttributes(n));
    // send text, possibly empty
    send(getText(n));
    // Pre-order Traversal
    for Each Child Node s Of n do
        Encode(s);
        // Full mixed content states that
        // text exists between all children.
        send(getText(n,s))
    end for
    // End Element
    send(0x1);
end function
```

and 0x1 can be represented using a single bit. Introducing bit-level operations can reduce each start/end declaration by at minimum seven bits. Using this technique, the encoding from Figure 6 changes to the one shown in Figure 7. However, a low-level design choice must be made for this technique: either (1) the entire encoding will be done on the bit level, i.e., one bit will be written for start/end declaration followed by multiple bytes that include tag index, character data, etc., or (2) an algorithm is forced to *buffer* eight bits, each representing a SAX event. While both techniques may induce extra processing and memory overheads, for example, using the second technique, the corresponding tag-dictionary keys and associated character data must be buffered until a byte is packed and the amount of data to buffer is undefined. The second method is preferred for the following reason. By packing bits, the entropy of the entire encoding will be smaller because no character data (the most common form of data) is being bit-shifted before the encoder outputs it and by extension, allows more separation between structure and data.

3.2.2 Path-centric Compression

Recall that for offline XSAQCT a text container for path P contains a list of text data for all paths similar to P. This technique provides a good compression

i. {00011001_2, a, t1, b,t4, d, t8,t5,t2, b,t6, e, t9,t7}

ii. {11_2, t3, EOF }

Figure 7: Encoding using two bits.

ratio because the types of data for each path are typically related (e.g. a ID tag will only have characters defined by the class $[0-9]^+$) and thus will have a smaller entropy in comparison to all of the text in total. Consider the two algorithms provided before. For both algorithms, every packet of compression focuses on a single pre-ordered branch of the XML document, which makes it more difficult to deal with text data. If *Z* text nodes of each similar *path* are buffered and that buffer is compressed before transfer, we will decrease N-order entropy, or less formally, the entropy in specific locales of the document by exploiting character locality properties used by permutation-based XML compressors. However, if the receiving node is piping XML content directly into an application, text data should be transferred as rapidly and frequently as possible into the client. This conundrum necessitates the reason for introducing context as a way to increase compression ratios and potentially reduce processing and transfer times. For storing XML content, which would be queried upon, the same text containers used for Offline Compression could be losslessly constructed on the sending node.

Deferred Text Compression. Similar to offline XSAQCT, the encoder constructs text containers, one for each unique path. However, each container is a statically defined buffer (with a statically defined buffer size). When the buffer becomes full, the encoder writes 0x2 (or 10_2) and the contents of the buffer are written to the output stream. One slight modification to the algorithm would allow variable sized buffers that expand or contract depend on the usage of the text container. In the following example let square brackets [text] denote a buffer, and for the following example assume it has a maximum capacity of two elements (in our implementation it is actually based on a specified number of bytes).

i. {0x0, a, 0x0, b, 0x0 d, 0x1}

ii. {0x2, [t4,t5], 0x1, 0x2, [t1,t2]}

iii. {0x0, b, 0x0, e, 0x1, 0x2, [t6,t7]}

iv. {0x1, 0x1, [t3], [\0], [t8], [t9]}

Figure 8: Buffering text.

Figure 8 gives the output of encoding for Figure 1 using the Algorithm 2. The last encoding line in this figure includes all leftover text data, sent in a commonly agreed upon ordering (for this example, depth-first fashion, the implementation uses the synchronous dictionaries natural ordering). One additional note too make is that since three codes are reserved, only four commands can be packed per byte.

Algorithm 2: Deferred Pre-order Encoding.

```
Require: n is the Root on initial function call
function ENCODE(Node n)
    preorder(n);
    for Each Unique Path : q do
        // Any trailer data in the buffer.
        buffer(q).finalize().close();
        send(buffer(q));
    end for
end function
function BUFFER(Path path, Text t)
    buffer(path, getText(n));
    // The buffer would do the following
    // as data is being buffered:
    if buffer(path).isLimitExceeded() then
        send(0x2);
        // Offload the full buffer.
        send(buffer(path));
        // Reset the buffer
        buffer(path).clear();
    end if
end function
Require: n is the Root on initial function call
function PREORDER(Node n)
    // Start element
    send(0x0);
    // Send name+attribute information.
    send(handleAttributes(n));
    // Buffer (possibly empty) left-most text node.
    buffer(p, getText(n));
    // Pre-order traversal
    if !isChild(n) then
        c = LC(c);
        while $c \neq$ null do
            preorder(c);
            // Buffer right-most text child
            // or separation text.
            buffer(p, getText(n));
            c = RS(c);
        end while
    end if
    send(0x1);
end function
```

4 IMPLEMENTATION AND RESULTS

4.1 Testing Environment

All our algorithms were implemented using Java version 1.7.0. Based on the previous research on characteristics of XML files, to create a test suite described in the following section, we chose to consider the following characteristics: file size, general characteristics (such as number of elements and attributes), first order entropy, content density and a variant of structure regularity (total number of elements over total number of distinct elements). For encoding process, two measures were taken: (1) an ordinary encoding of the entire document, which represents the number of raw bytes generated by the encoder; (2) a compression of the encoding from (1) using a general purposed back-end compressor that is used often in real time environments. GZIP (GZIP, 2012) was chosen as the back-end (and text) compressor because many bandwidth aware transfer protocols (including HTTP, see (HTTP, 2012)) use GZIP or a DEFLATE variant for speedup. While the first measure may provide insights to how efficient each encoding is in comparison to the original data, the second measure describes compressibility of each encoding. Therefore our proposed encoding strategies focus on high compression ratio (reducing entropy) at the expense total number of bytes sent. This philosophy is also quite different from the EXI standard that adopts the philosophy of pre-compressing specific pieces of data (common string subsequences), at the cost of reducing the overall (how it appears contiguously) compressibility. In total, ten total encoding techniques are compared. First, we measured a naive solution of using GZIP (or no compression depending on the measure), followed by an EXI Implementation (Peintner, 2012) (using all fidelity options, default encoding options and a compression-oriented coding mode). Next, we measured performance of Treechop (Leighton et al., 2005), the original algorithm as discussed in (Müldner et al., 2012b) and a leaf improvement algorithm (grouping together long-sequences of child-siblings) is borrowed from (Müldner et al., 2012b), followed by our new algorithms, Lightweight SAX and its bit-packed improvement. Finally, we measured path-centric compression with its bit packing improvement.

4.2 Characteristics of 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): enwiki-latest-stub-articles.xml (in the tables and pictures referred to as e.w.-stub) from (enwiki dumps, 2012), 1gig.xml (a randomly generated XML file, using xmlgen (xmlgen, 2012)), enwikibooks-20061201-pages-articles.xml, (in the tables and pictures referred to as e.w.-books),dblp.xml, SwissProt.xml, enwikinews-20061201-pages-articles.xml (in the tables and pictures referred to as e.w.-news), lineitem.xml, shakespeare.xml, uwm.xml (all from the Wratislavia corpus (Wratislavia, 2012)), baseball.xml (from (Baseball.xml, 2012)), and macbeth.xml (from (macbeth, 2012)). Table 1 provides an overview of each XML file, using characteristics described in Section 4.1. The measurements show that the (first order) entropy for each XML document is greater than four. In comparison, Shannon estimated the entropy of written English to be between 0.6 and 1.3. Secondly, while the number of unique paths in 1gig.xml file is greater than 500, the number of unique element names is roughly 80. This phenomenon is common in XML files where many subtrees differ by the name of a parent node. Fig-

Table 1: Overview of XML Test Suite. (Sizes in Bytes).

XML File	Size	Entropy	Size (Gzip)	Unq. Paths
e.w.-stub	5.96E+09	4.905	9.54E+08	37
1gig	1.17E+09	4.768	3.85E+08	548
e.w.books	1.56E+08	5.152	4.57E+07	29
dblp	1.34E+08	5.198	2.45E+07	145
SwissProt	1.15E+08	5.54	1.41E+07	264
e.w.news	4.64E+07	5.202	1.30E+07	29
lineitem	3.22E+07	5.042	2.91E+06	19
shakespeare	7.65E+06	5.189	2.14E+06	58
uwm	2.34E+06	4.752	1.62E+05	22
BaseBall	6.72E+05	4.867	6.68E+04	46
macbeth	1.63E+05	5.164	4.67E+04	22
XML File	Elements	Attributes	Density	Regularity
e.w.-stub	1.70E+08	3.04E+07	0.549	4.60E+06
1gig	1.67E+07	3.83E+06	0.735	3.05E+04
e.w.books	5.34E+05	4.91E+04	0.942	1.84E+04
dblp	3.33E+06	4.04E+05	0.562	2.30E+04
SwissProt	2.98E+06	2.19E+06	0.444	1.13E+04
e.w.news	2.79E+05	2.46E+04	0.894	9.61E+03
lineitem	1.02E+06	1.00E+00	0.195	5.38E+04
shakespeare	1.80E+05	0.00E+00	0.646	3.10E+03
uwm	6.67E+04	6.00E+00	0.445	3.03E+03
BaseBall	2.83E+04	0.00E+00	0.109	6.15E+02
macbeth	3.98E+03	0.00E+00	0.633	1.81E+02

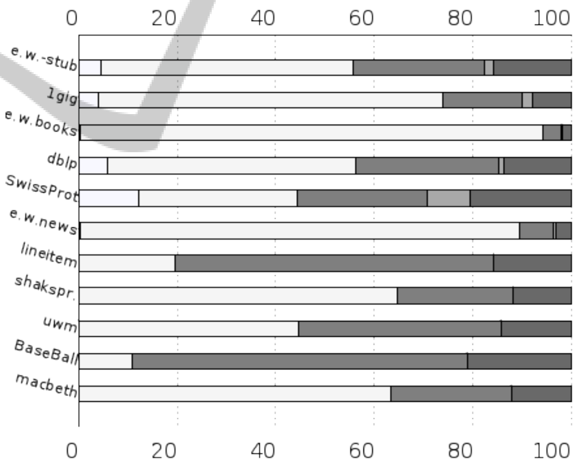

Figure 9: Character Density (Attribute Text + Element Text) vs. Markup Density (Node Tags + Attribute Tags + Reserved Chars).

ure 9 provides a comparison of Content Density and Markup Density. This comparison shows the major issue with using XML as a representation language for semi-structured data. Specifically, it shows that in the worst case, for BaseBall.xml, ninety percent of the document is considered markup overhead. In the best case, for enwikibooks.xml, over ninety percent of the document consists of text. Otherwise, the rest of the suite lies within the two extremes. Table 2 provides a breakdown of XML Test Suite in percentage points. These two tables show that our XML suite includes XML files with low/high number of nodes and attributes, and various amounts of text, such as attribute text or element text.

4.3 Results of Testing

Reducing Markup Data. Table 3 compares the efficiency of each algorithm described in Section 3 in terms of encoding *only* the markup of a XML document. The compression ratio is calculated as the size of the encoding divided by the size of the markup data (column two). Note that for online compression the size of the markup includes all occurrences of ASCII Zero to satisfy the full-mixed content property (for algorithms that require it) and all additional data required to describe text data (for example, block lengths, text terminators, etc.).

Figure 10 provides two measures: (a) A grey bar with a white bar stacked on top. This represents the best compression of markup an online algorithm could achieve, while the white bar stacked-upon it represents the size of its GZIP'd encoding. (b) The new grey bar with a black bar stacked on top. This represents the number of bytes required to encode an annotated tree, while the white bar stacked-upon it represents the size of its GZIP'd encoding. For the online algorithms we achieve over a 25% compression ratio (or a space savings greater than 75%) for all files. For the offline algorithm, some files produce encodings that contain many zero annotations and thus contain large portions of overhead. There is no decisive winner when one compares both encodings compressed as these encodings produce compression ratios of under 2% (space savings of 98%) of the original markup. However, comparing the two encodings after compression is also quite misleading, but still provides interesting information none the less (how compressible/similar the data is). For online algorithms, these encodings aren't continuous as large portions of intermittent character data separate portions of this markup, thus not allowing us to encode markup data as one contiguous sequence.

Switching focus to only the online algorithms, the

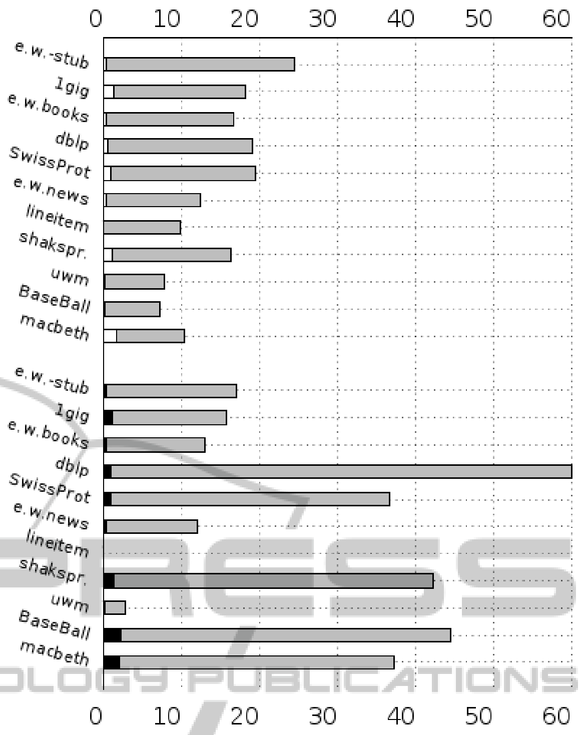

Figure 10: Compression of Markup Data. Best Online Compressor (Top), Offline XSAQCT (Bottom) vs. GZIP'ed encodings.

first general trend is that as the file gets bigger (bottom to top in Figure 10), the worse the compression ratio of markup data gets. From Table 1, we see that the four documents with the highest number of tag elements are also the four least-compressed documents (enwiki-latest, 1gig, dblp, SwissProt). This phenomenon can be attributed to the full-mixed contents assumption. For each algorithm an ASCII zero is required to delimit text so the decoder knows when the character portion ends. In our future work, more compressible text delimitation and full mixed content techniques will be studied, as well as introducing context models, similar to the ideas EXI proposes, to increase path-centric compressibility.

Reducing XML Data. Tables 4 and 5 encompass an entire XML encoding and transfer process. Each element in the table represents the number of bytes physically transferred over a socket. Let us first compare the our basic online algorithms, with their improved variants. The improved variant of the original XSAQCT algorithm performs slightly better. However, the improved variant of the Lightweight SAX implementation that uses bit-packing (unsurprisingly) does not, because it may represent a fewer number of total bytes, and the self-information (entropy) of the encoding tends to be much higher. This increased self-information is also present in the Path-centric Al-

Table 2: Breakdown of XML Test Suite in Percentage Points. For character set, Number of Bits in Character Encoding / Number of Unique Characters.

File	Node Tags	Attribute Tags	Reserved	Attribute Text	Element Text	Character Set
e.w.-stub	26.426	1.958	15.781	4.5	51.336	16/12460
1gig	15.94	2.191	7.949	4.001	69.918	8/80
e.w.-books	3.707	0.251	1.821	0.283	93.938	16/23795
dblp	29.106	1.017	13.656	5.739	50.482	8/126
SwissProt	26.442	8.557	20.593	12.086	32.322	8/86
e.w.-news	6.866	0.477	3.202	0.424	89.031	16/1777
lineitem	64.589	0	15.867	0	19.543	8/69
shakespeare	23.645	0	11.748	0	64.607	8/80
uwm	41.214	0.001	14.275	0.003	44.507	8/68
BaseBall	67.941	0	21.147	0	10.912	8/68
macbeth	24.558	0	12.195	0	63.247	8/70

Table 3: Compression Ratio of Structure Data: Number of Bytes to Encode Markup / Original Markup Size.

File	XML-Markup	XSAQCT	Leaf Imprv.	Lw SAX	Bit Packing	Path Cent.	Defer. Text	Min
e.w.-stub	2.59E+09	0.331	0.304	0.359	0.244	0.376	0.26	0.244
1gig	3.04E+08	0.282	0.249	0.312	0.216	0.325	0.181	0.181
e.w.-books	9.03E+06	0.279	0.277	0.312	0.208	0.317	0.167	0.167
dblp	5.86E+07	0.294	0.214	0.305	0.206	0.312	0.191	0.191
SwissProt	6.38E+07	0.302	0.244	0.316	0.235	0.37	0.194	0.194
e.w.-news	4.89E+06	0.27	0.264	0.3	0.2	0.305	0.123	0.123
lineitem	2.59E+07	0.193	0.128	0.197	0.128	0.197	0.099	0.099
shakespeare	2.71E+06	0.308	0.25	0.332	0.216	0.332	0.163	0.163
uwm	1.30E+06	0.233	0.213	0.257	0.167	0.257	0.077	0.077
BaseBall	5.96E+05	0.234	0.152	0.238	0.155	0.238	0.072	0.072
macbeth	5.99E+04	0.312	0.252	0.335	0.219	0.336	0.104	0.104

Table 4: Encoding the Entire XML Document. (Sizes in Bytes).

File	Offline	EXI	Treechop	XSAQCT	Leaf	Lw. SAX	Bit Pck.	Path Cnt.	Defer.
e.w.-stub	4.09E+09	*	4.66E+09	4.12E+09	4.06E+09	4.21E+09	3.91E+09	4.25E+09	3.94E+09
1gig	9.47E+08	4.79E+08	1.24E+09	9.48E+08	9.38E+08	9.57E+08	9.28E+08	9.61E+08	9.17E+08
e.w.-books	1.49E+08	1.39E+08	1.45E+08	1.43E+08	1.43E+08	1.50E+08	1.49E+08	1.50E+08	1.41E+08
dblp	1.26E+08	8.69E+07	1.07E+08	9.24E+07	8.78E+07	9.31E+07	8.73E+07	9.35E+07	8.63E+07
SwissProt	8.26E+07	4.37E+07	8.46E+07	7.03E+07	6.66E+07	7.12E+07	6.60E+07	7.46E+07	6.25E+07
e.w.-news	4.27E+07	3.95E+07	4.27E+07	4.18E+07	4.17E+07	4.30E+07	4.25E+07	4.30E+07	4.10E+07
lineitem	8.35E+06	7.31E+06	1.19E+07	1.13E+07	9.61E+06	1.14E+07	9.62E+06	1.14E+07	9.40E+06
shakespeare	6.44E+06	5.94E+06	6.38E+06	5.77E+06	5.62E+06	5.84E+06	5.53E+06	5.84E+06	5.19E+06
uwm	1.21E+06	7.68E+05	1.52E+06	1.34E+06	1.31E+06	1.37E+06	1.26E+06	1.37E+06	8.33E+05
BaseBall	3.95E+05	1.77E+05	1.52E+06	2.13E+05	1.63E+05	2.15E+05	1.66E+05	2.15E+05	4.31E+04
macbeth	1.33E+05	1.21E+05	1.35E+05	1.22E+05	1.18E+05	1.23E+05	1.16E+05	1.23E+05	6.25E+03

* – Requires extraordinary amount of RAM. Crashes with 16GB of allocated memory.

Table 5: Using a Backend Compressor on the Encodings. (Sizes in Bytes).

File	Offline	EXI	Treechop	XSAQCT	Leaf	Lw. SAX	Bit Pck.	Path Cnt.	Defer.
e.w.-stub	6.95E+08	*	9.14E+08	8.87E+08	8.86E+08	8.95E+08	9.39E+08	6.99E+08	6.95E+08
1gig	3.29E+08	1.57E+08	3.90E+08	3.71E+08	3.71E+08	3.70E+08	3.74E+08	3.32E+08	3.31E+08
e.w.-books	4.45E+07	4.39E+07	4.52E+07	4.51E+07	4.51E+07	4.54E+07	4.56E+07	4.44E+07	4.38E+07
dblp	1.94E+07	1.86E+07	2.36E+07	2.27E+07	2.27E+07	2.29E+07	2.34E+07	1.94E+07	1.92E+07
SwissProt	7.63E+06	7.71E+06	1.33E+07	1.23E+07	1.23E+07	1.24E+07	1.36E+07	7.74E+06	7.77E+06
e.w.-news	1.26E+07	1.26E+07	1.29E+07	1.28E+07	1.28E+07	1.29E+07	1.30E+07	1.26E+07	1. 25E+07
lineitem	1.43E+06	1.44E+06	2.34E+06	2.33E+06	2.20E+06	2.34E+06	2.44E+06	1.45E+06	1.44E+06
shakespeare	1.89E+06	1.99E+06	2.07E+06	2.02E+06	2.03E+06	2.03E+06	2.05E+06	1.91E+06	1.90E+06
uwm	1.02E+05	1.10E+05	1.50E+05	1.43E+05	1.42E+05	1.47E+05	1.66E+05	1.02E+05	1.01E+05
BaseBalll	4.67E+04	3.87E+04	5.29E+04	5.41E+04	4.82E+04	5.45E+04	5.41E+04	3.49E+04	3.46E+04
macbeth	4.34E+04	4.50E+04	4.56E+04	4.45E+04	4.48E+04	4.46E+04	4.53E+04	4.34E+04	4.32E+04

* – Requires extraordinary amount of RAM. Crashes with 16GB of allocated memory.

gorithm, however not to the same extent.

We did not report the compression ratio of enwiki-latest-stub.xml with EXI due to hardware limitations. In general, we made the hard restriction that the amount of internal memory allocated to a process can never exceed twice the file size. Not only did EXI exceed this, but it exceeded three times the original file size. Section 5 provides a brief discussion on why we think this occurs, and what can be done to improve it. However, in terms of our online compression algorithms (which don't produce substantial memory footprint), we can see that they scale well to data of all sizes.

Relating each individual algorithm to the baseline transfers (Offline & GZIP) we see that in each case all online algorithms perform better than GZIP. However, in each case as we relax the "online constraint" (the more data we are allowed to buffer), the better and more significant results we receive. Comparing Offline XSAQCT, which requires processing the entire document before transfer to Lightweight Sax, which processes on the fly, the Offline XSAQCT performs on average, 15% better. However, comparing Offline to Path-centric algorithm, Offline XSAQCT performs on average, 2% better, which is significant because Path-centric algorithm does not require a full document scope, which would be optimal in the domain of large-scale XML streaming. In comparison to Treechop, the original and lightweight SAX variants also prove to be quite competitive and once again the Path-centric algorithms beat Treechop. Finally, in comparison to EXI, the Path-centric algorithms tend to be similar for documents whose size was larger than 1GB. For example, for 1gig.xml, a document which contains a lot of character data, EXI beats "all" algorithms by a substantial margin.

5 CONCLUSIONS AND FUTURE WORK

This paper provided a design and implementation of four new versions of online XML compression, which exploit local structural redundancies of pre-order traversals of an XML tree and focus on reducing the overhead of sending packets and maintaining load balancing between the sender and receiver with respect to encoding and decoding. For testing various algorithms, a suite consisting of 11 XML files with various characteristics was designed and analyzed. In order to take into account network issues, such as its bottleneck, two measures of the encoding process were considered to compare ten encoding techniques, using GZIP, EXI, Treechop, XSAQCT, and its improvement, and our new algorithms. Our experiments indicated that our new algorithms have similar or better performance than existing online algorithms, such XSAQCT and Treechop, and have only worse performance than (a non-queryable XML compressor) EXI for files larger than 1 GB.

For future work we will consider the amount of overhead of performing each encoding as in the amount of resources required for encoding. Recall the missing data in Table 4. The reason for the extraordinary resource requirement can be attributed to the fact that EXI also performs manipulation on the text to increase compression ratios, which appears to be resource heavy. In addition, more techniques will be introduced that alleviate the vast overhead of sending ASCII zero characters to delimit text and to satisfy the full mixed content property. Next, context-modeling techniques (such as those used in the PPM & PAQ variants of data compression) will be introduced as a pre-processor on character data to allow higher compression ratios. Finally, a more concise and meaningful definition of Structure Regularity that takes into account the skewness of occurrences, by using the annotated tree representation, will be defined and studied.

REFERENCES

Arion, A., Bonifati, A., Manolescu, I., and Pugliese, A. (2007). XQueC: a query-conscious compressed XML database. *ACM Transactions on Internet Technology*, 7(2).

Baseball.xml (2012). baseball.xml, retrieved October 2012 from http://rassyndrome.webs.com/cc/baseball.xml.

enwiki dumps (2012). enwiki-latest.xml, retrieved October 2012 from http://dumps.wikimedia.org/enwiki/latest/.

EXI (2012). Efficient XML Interchange (EXI) Format 1.0, Retrieved October 2012 from http://www.w3.org/TR/exi/.

GZIP (2012). The gzip home page, retrieved October 2012 from http://www.gzip.org.

Hartmut, L. and Suciu, D. (2000). XMill: an efficient compressor for XML data. *ACM Special Interest Group on Management of Data (SIGMOD) Record*, 29(2):153–164.

HTTP (2012). HTTP RFC 2616, retrieved October 2012 from http://www.w3.org/protocols/rfc2616/rfc2616.html.

Leighton, G. and Barbosa, D. (2009). Optimizing XML compression. XML Database Symposium (XSym) '09, pages 91–105, Berlin, Heidelberg. Springer-Verlag.

Leighton, G., Müldner, T., and Diamond, J. (2005). TREECHOP: A Tree-based Query-able Compressor for XML. *The Ninth Canadian Workshop on Information Theory*, pages 115–118.

Lin, Y., Zhang, Y., Li, Q., and Yang, J. (2005). Supporting efficient query processing on compressed XML files. Proceedings of the Symposium on Applied Computing (SAC) '05, pages 660–665, New York, NY, USA. ACM.

macbeth (2012). macbeth.xml, retrieved October 2012 from http://www.ibiblio.org/xml/examples/.

Measurements (2012). Efficient XML Interchange Measurements Note, retrieved October 2012 from http://www.w3.org/tr/exi-measurements/.

Müldner, T., Corbin, T., Miziołek, J., and Fry, C. (2012a). Design and Implementation of an Online XML Compressor for Large XML Files. *International Journal On Advances in Internet Technology*, 5(3):115–118.

Müldner, T., Fry, C., and Miziołek, J. (2012b). Online Internet Communication using an XML Compressor. In *The Seventh International Conference on Internet and Web Applications and Services*, pages 131–136, Stuttgart, Germany. International Academy, Research, and Industry Association. (IARIA).

Müldner, T., Fry, C., Miziołek, J., and Durno, S. (2008). SXSAQCT and XSAQCT: XML Queryable Compressors. In S. Böttcher, M. Lohrey, S. M. and Rytter, W., editors, *Structure-Based Compression of Complex Massive Data*, number 08261 in Dagstuhl Seminar Proceedings.

Müldner, T., Fry, C., Miziołek, J., and Durno, S. (2009). XSAQCT: XML queryable compressor. In *Balisage: The Markup Conference 2009*, Montreal, Canada.

Ng, W., Lam, W.-Y., Wood, P., and Levene, N. (2006). XCQ: a queriable XML compression system. *Knowledge and Information Systems*, 10(4):421–452.

Peintner, D. (2012). EXI: EXIficient retrieved October 2012, from http://exificient.sourceforge.net.

Qureshi, M. H. and Samadzadeh, M. H. (2005). Determining the complexity of XML documents. International Conference on Information Technology: Coding and Computing(ITCC) '05, pages 416–421, Washington, DC, USA. IEEE Computer Society.

Ruellan, H. (2012). XML entropy study. In *Balisage: The Markup Conference 2012*, Montreal, Canada.

Sakr, S. (2008). An experimental investigation of XML compression tools. *The Computing Research Repository (CoRR)*, abs/0806.0075.

SAX (2012). Simple API for XML (SAX), retrieved October 2012 from http://www.saxproject.org.

Snyder, S. (2010). Efficient XML Iinterchange (EXI) compression and performance benefits: Development, implementation and evaluation, retrieved October 2012 from http://www.dtic.mil/cgi-bin/gettrdoc?ad=ada518679. Master's thesis, Naval Postgraduate School, Monterey, California.

soap (2012). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), retrieved October 2012 from http://www.w3.org/tr/soap12-part1/.

Tolani, P. and Haritsa, J. (2002). XGRIND: a query-friendly XML compressor. *International Conference on Data Engineering (ICDE)' 02*, pages 225–234.

Wratislavia (2012). Wratislavia XML corpus, retrieved October 2012 from http://www.ii.uni.wroc.pl/∼inikep/research/wratislavia/.

XML (2012). Extensible markup language (XML) 1.0 (Fifth edition), retrieved October 2012 from http://www.w3.org/tr/rec-xml/.

xmlgen (2012). The benchmark data generator, retrieved October 2012 from http://www.xml-benchmark.org/generator.html.

XPath (2012). XML Path Language (XPath), Retrieved October 2012 from http://www.w3.org/TR/xpath/.

XQuery (2012). XQuery 1.0: An XML Query Language (Second Edition), Retrieved October 2012 from http://www.w3.org/TR/xquery/.