TR-2005-005 August, 2005

TREECHOP: A Tree-based Query-able Compressor for XML

Gregory Leighton, Tomasz Müldner, James Diamond
Jodrey School of Computer Science, Acadia University,
Wolfville, NS, Canada B4P 2R6
{005985l;tomasz.muldner;james.diamond}@acadiau.ca

Abstract

XML is a popular meta-language that facilitates the interchange and access of data. However, XML's verbose nature may increase the size of a data set as much as tenfold. In this report, we present a novel technique for lossless XML compression, called TREECHOP, which supports querying of compressed XML data without requiring full decompression. Unlike other query-capable XML compression schemes, TREECHOP requires only a single pass over the input document during the compression process, resulting in an efficient, online operation that is well-suited for transmission of compressed XML documents over a network.

1 Introduction

The Extensible Markup Language (XML) [6] is a World Wide Web Consortium (W3C) endorsed standard for semi-structured data. It allows data to be surrounded by textual markup (*elements* and *attributes*) that serves to describe its semantics. The inclusion of structural information with the data grants XML great flexibility, at the cost of increased verbosity. It is not uncommon for the XML representation of a set of data to be as much as ten times as large as alternative representations (e.g. data in comma-separated value format).

In recent years, messaging has been one of the most common applications of XML. One example is the Web Services initiative, in which network services can be discovered, described, and invoked in a platform- and implementation-independent way via the exchange of XML messages. Such applications would benefit from a compression scheme which operates online and allows queries to be carried out directly on compressed data. In this report, we present TREECHOP, an XML-conscious compression scheme which achieves both of these objectives.

TREECHOP supports querying of compressed XML data without requiring full decompression. In addition, it is an online operation that is well-suited for transmission of compressed XML documents over a network. Unlike other query-capable XML compression schemes, TREECHOP requires only a single pass over the input document during the compression process. The current implementation of TREECHOP supports queries based on XPath expressions (using exact matches).

1.1 Related Work

In [9, 10] we describe AXECHOP, an XML-conscious compression scheme that combines a grammar-based compression of document structure with a Burrows-Wheeler Transform-based [2] compression of the data portions of a document. While this approach results in significant compression rates, it involves a re-ordering of the document during compression in order to localize redundancies. This precludes online operation, as the decoder cannot begin rebuilding the original document until the entire compressed data stream has been processed.

XMill [11] represents the pioneering work in the area of XML-conscious compression. Its compression strategy separates the structural information of an XML document from

the contained data. Data values are then grouped in containers according to the identity of the enclosing element or attribute, and GZIP [7] is subsequently applied to each individual container. A container expression language allows the user to substitute alternative compression strategies for GZIP. Although XMill often outperforms GZIP on XML data, the original structure of the document is disrupted during the compression process, which precludes online processing. This serves to limit the usefulness of XMill for XML messaging applications. In addition, the XMill encoding format does not allow querying of compressed data.

XMLPPM [3] achieves a higher degree of compression via the use of multiplexed hierarchical models and the PPM [4] text compression method. As with XMill, compressed data cannot be queried.

XGRIND [14] was the first XML-conscious compression scheme to support querying without full decompression. Element and attribute names are encoded using a byte-based scheme, and character data is compressed using non-adaptive Huffman coding [8]. Use of the latter technique significantly slows down the compression process, since two passes over the original document are required (first to gather probability data, and a second time to perform the encoding).

XPRESS [12] also supports querying of compressed data and claims to achieve better compression than XGRIND. However, it uses a semi-adaptive form of arithmetic coding which also necessitates two passes over the original XML document.

1.2 Contributions

This report presents linear time strategies for compressing, decompressing, and querying XML data. Unlike the query-able XML compression schemes described in [14] and [12], compression requires only a single pass through the input XML document.

1.3 Organization

Section 2 of this report describes the compression, decompression, and querying strategies used in TREECHOP. Experimental results comparing the compression and speed of TREECHOP versus alternative compression routines are presented in Section 3. Section 4 concludes the report.

2 TREECHOP

In this section, we define some notational conventions used in the subsequent discussions on the compression, decompression, and querying strategies employed in TREECHOP. We begin by describing the XML document tree structure used to visualize an input XML document.

Definition 1: Each node in the XML document tree possesses a textual *label*. In the case of XML elements, the label is the name of the element; for XML attributes, the label is formed by concatenating '@' with the name of the attribute. In the case of comments,

processing instructions, and CDATA sections, the label consists of all text between the delimiting section markers.

As an example, the root element of the document tree in Figure 2 has the label “PurchaseOrder”, while the “no” attribute associated with the “PurchaseOrder” element is assigned the label “@no”.

The root node of the document tree corresponds to the root element in the XML document. Any information appearing before the root element (such as the XML declaration, DOCTYPE declaration, processing instructions, or comments) is stored in a data container called the *prologue*. Similarly, any comments or processing instructions occurring after the end tag of the root element are stored in a data container called the *epilogue*.

Character data (such as attribute values and text occurring between an XML element's beginning and ending tags) are leaf nodes in the tree. Instances of CDATA sections, comments, and processing instructions are also modelled as leaf nodes. Each occurrence of an attribute or element in the XML document is represented in the tree as a non-leaf node, as follows:

- *attribute node*: this subtype corresponds to the occurrence of an attribute within the source XML document; the node label records the name of the attribute. Each attribute has a single child, a leaf node containing the attribute's data value.
- *element node*: this subtype represents an occurrence of an XML element within the document. The node label stores the name of the element, and additionally each element node may have multiple children, including nodes representing nested elements, attributes, comments, or processing instructions. Non-empty element nodes have a leaf node child containing the character data enclosed by the element's start and end tag. The root node in the document tree is always an instance of this type.

Figure 1 depicts an XML document and Figure 2 shows the equivalent document tree representation. A particular node within the document tree can be identified and extracted by specifying its location according to the following scheme.

Definition 2: The *path* of a node v_n in the XML document tree is a sequence $/L_1/L_2/\ \ldots/L_n$ of one or more ‘/’-separated labels that traces a route from the root node v_1 to v_n, where L_i is the label of node v_i.

In the document tree depicted in Figure 2, each of the nodes labeled “Quantity” is assigned the same path value “/PurchaseOrder/Order/Item/Quantity”, while the node labeled “@no” has the path value “/PurchaseOrder/@no”.

2.1 Compression Strategy

The compression process in TREECHOP begins by conducting a SAX-based [13] parsing of the XML document. As tokens are returned by the parser, new tree nodes are created and then written out to the compression stream in depth-first order. This approach avoids the necessity of building an in-memory representation of the entire document tree.

```
<?xml version=”1.0” encoding=”UTF-8”?>
  <!-- start of PO -->
  <PurchaseOrder no=”1456”>
    <Date>06/05/05</Date>
    <CustomerID>765345</CustomerID>
    <Order>
      <Item>
        <ProductNo>P-4534</ProductNo>
        <Quantity>2</Quantity>
      </Item>
      <Item>
        <ProductNo>P-9182</ProductNo>
        <Quantity>1</Quantity>
      </Item>
    </Order>
  </PurchaseOrder>
  <!-- end of PO -->
```

Figure 1: Example XML Document

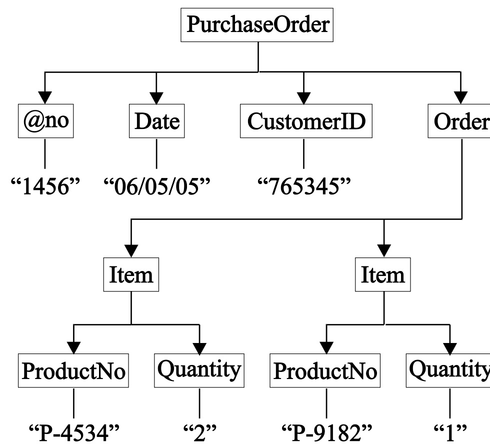

Figure 2: Tree Representation of XML Document in Figure 1

Each CDATA section, comment, processing instruction, and non-leaf node is assigned a binary codeword. This codeword is uniquely assigned based on the path of the tree node. If there are multiple nodes with the same absolute path, each occurrence will receive the same codeword. For example, in the tree shown in Figure 2, each of the two instances of "/PurchaseOrder/Order/Item" will be assigned the same codeword.

The codeword $C(v)$ assigned to a node v with parent node p is formed from the concatenation of three codes $C(p)$, $G(v)$, and $T(v)$, where

- $C(p)$ represents the codeword assigned to p;
- $G(v)$ is a Golomb code assigned to v based on its ordering relative to p. If v is the n-th distinct child node of p in left-to-right order (where two nodes are said to be *distinct* if they have different paths), then $G(v)$ is formed by concatenating the unary code for $q+1$ with the binary code for r, where q and r are calculated as

$$q = \lfloor (n-1)/3 \rfloor \tag{1}$$

$$r = n - 3q - 1 \tag{2}$$

  After experimenting on several XML documents of varying structural characteristics, it was observed that $m = 3$ tended to produce the best average compression results. (In general, smaller values for m are best in cases where the range of n-values is also small; this is true in the current case, since typically any element in an XML document has a relatively small number of distinct child elements and attributes.)
- $T(v)$ is a sequence of 3 bits used to indicate the node type. Table 1 lists the $T(v)$ value for each node type.

Node Type	$\boldsymbol{T(v)}$
Element	000
Attribute	001
Comment	010
CDATA	011
Processing Instruction	100

Table 1: Values for $T(v)$ by Node Type

The codeword of the root node is always 00000. An example of the codeword assignment scheme is provided in Table 2, pertaining to the document tree depicted in Figure 2.

This encoding scheme has three important properties:

1. the codeword for each node is prefixed by its parent's codeword;
2. two nodes in the XML document tree share the same codeword if and only if they have the same path; and

Node Path	$C(v)$
/PurchaseOrder	00000
/PurchaseOrder/@no	0000000001
/PurchaseOrder/Date	00000010000
/PurchaseOrder/CustomerID	00000011000
/PurchaseOrder/Order	00000100000
/PurchaseOrder/Order/Item	0000010000000000
/PurchaseOrder/Order/Item/ProductNo	000001000000000000000
/PurchaseOrder/Order/Item/Quantity	0000010000000000010000

Table 2: Assigned Codewords for the Document Tree in Figure 2

3. the structure of the original XML document is maintained by the encoding scheme.

Data is added to the compression stream in the following order. First, the contents of the prologue are written after applying GZIP, followed by a depth-first ordering of the encoded document tree. Finally, the GZIP-compressed contents of the epilogue are written to the compression stream.

The encoding information for each tree node is written to the encoding stream in an adaptive fashion. Each non-leaf node is encoded as a 3-tuple (L, C, D), where L is a byte indicating the bit length of the codeword; C is the codeword assigned to the node, consisting of a sequence of $\lceil L/8 \rceil$ bytes; and D is the textual data stored in the node. A reserved byte value is used to indicate to the decoder that raw character data is forthcoming in the encoding stream; once D has been transmitted, a second reserved byte value is used to signal the end of the character data sequence.

Leaf nodes are transmitted in the same manner as D, described above. For the second and subsequent occurrences of a particular codeword, only the 2-tuple (L, C) is transmitted since the decompressor is able to infer the value of D at that point.

Information about a node N is written to the encoding stream immediately after N is assigned a codeword. This allows the decoder to set about decoding N before the encoding of other nodes has been received. As node information is added to the compression stream, it is compressed using GZIP.

2.2 Decompression Strategy

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

As each tree node is encountered in the compression stream, the decompressor determines the node type by examining the final three bits in the codeword. The type information is then used to surround the D value for this node with the appropriate XML syntax before emitting it to the decompression stream. In the case of non-leaf

nodes, the compression stream is examined to determine whether the next occurring node is a data value node belonging to the current element or attribute node.

2.3 Querying Strategy

Exact-match queries can be carried out via a single scan through the compression stream. The query processor employs a stack to keep track of the current path; when the query predicate path is first encountered, the associated codeword C is recorded and the next occurring D value is extracted from the compression stream as a query match. Subsequently, the remainder of the stream is scanned for future occurrences of C. With each match, the associated D value is extracted from the stream.

Range queries are handled in a similar manner, except that each query match additionally requires that the corresponding D value be converted into a numeric value and tested to see if it falls within the query range before it is returned as a search result.

2.4 Implementation

We have completed a Java-based implementation of the TREECHOP compression, decompression, and querying algorithms presented in Sections 2.1, 2.2 and 2.3 respectively. In this section, we focus on the design of this implementation. Appendix A lists several illustrative example programs which employ the TREECHOP implementation to perform compression, decompression, and querying of XML documents.

2.4.1 Encoder

The encoder component is responsible for compressing an input XML document by applying the compression strategy described in Section 2.1. The client program normally constructs an instance of `XMLDocumentOutputStream`, with a `java.io.OutputStream` object passed as an argument to the constructor. This parameter refers to the output stream on which compressed XML data will be written (e.g. a `java.io.FileOutputStream` instance can be specified to write the compressed data to a local file).

The implementation of `XMLDocumentOutputStream` extends `OutputStream`. A key departure is that the `write()` methods of `OutputStream` are replaced with several `writeXMLDocument()` methods, each of which takes a single input parameter identifying the XML document to be compressed. Figure 3 illustrates the public methods for this class.

A delegate class called `XMLDocumentTreeEncoder` is used to perform the actual compression. As depicted in Figure 4, this class has a `process`*`X`*`()` method for each SAX event type X; as tokens are returned by the SAX parser, the corresponding process method is called in `XMLDocumentTreeEncoder` to create a new tree node and write it to the compression stream.

To facilitate sending compressed XML documents over a TCP/IP network, client programs can utilize the `XMLSocket` class. This class provides many of the same methods as the `java.net.Socket` class, and also handles compression of the source XML document prior to transmission. Outgoing communications for the `XMLSocket` take

treechop::io::XMLDocumentOutputStream
+ XMLDocumentOutputStream(out: OutputStream)
+ close()
+ write(val: int)
+ writeXMLDocument(source: InputStream)
+ writeXMLDocument(source: Reader)
+ writeXMLDocument(sourceFile: File)
+ writeXMLDocument(filename: String)

Figure 3: Class Diagram for XMLDocumentOutputStream

treechop::XMLDocumentTreeEncoder
+ XMLDocumentTreeEncoder(out: DataOutputStream)
+ processCDATASection(contents: String)
+ processComment(commentData: String)
+ processDataValue(data: String)
+ processDocDecl(data: Object)
+ processEndDocument()
+ processEndElement()
+ processProcessingInstruction(procInstData: String)
+ processStartElement(name: String, attrs: Map)
+ processXMLDecl(data: Object)

Figure 4: Class Diagram for XMLDocumentTreeEncoder

place over an XMLDocumentOutputStream, while incoming communications travel over an XMLDocumentInputStream. The getOutputStream() and getInputStream() methods are used to obtain references to the streams associated with a given XMLSocket. Figure 5 shows the public methods for XMLSocket.

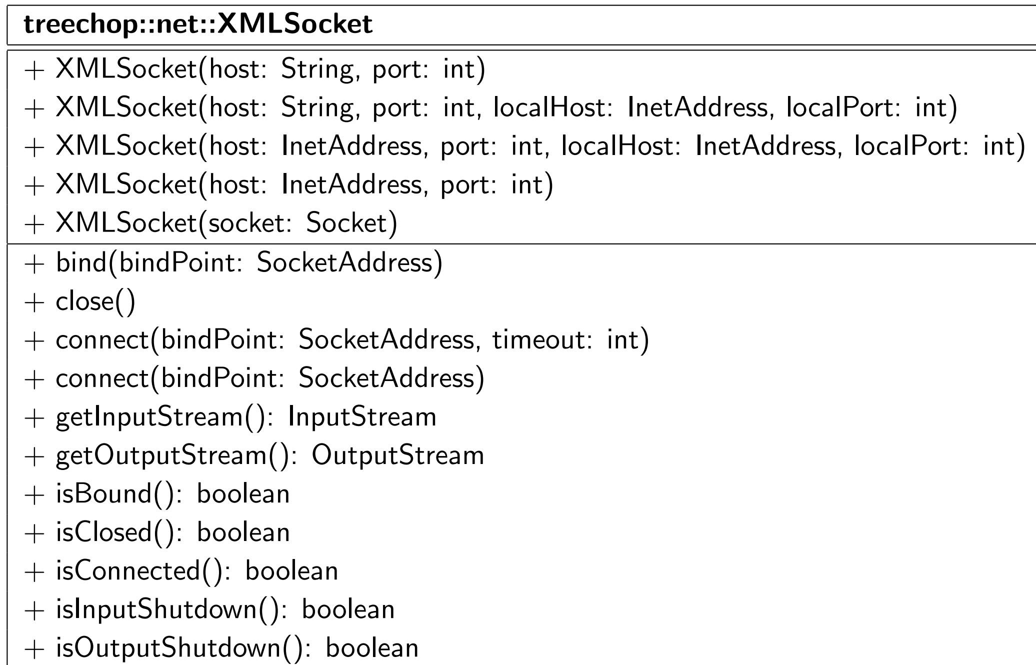

Figure 5: Class Diagram for XMLSocket

2.4.2 Decoder

Client programs wishing to decode compressed TREECHOP data begin by creating an instance of the XMLDocumentInputStream class. Figure 6 lists the public methods for this class.

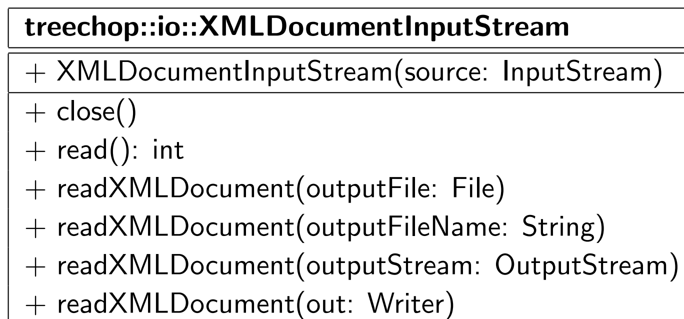

Figure 6: Class Diagram for XMLDocumentInputStream

This class extends java.io.InputStream to provide methods for reading compressed XML data. In the constructor, the client program specifies the underlying InputStream from which the compressed data originates (e.g. in the case where a local file holds the

compressed data, this parameter would be an instance of `java.io.FileInputStream`, whereas a call to the method `java.net.Socket.getInputStream()` would be used to obtain the `InputStream` reference used to read incoming compressed data over a TCP socket connection).

Once the `XMLDocumentInputStream` instance has been constructed, the client program starts the decompression process by calling one of the `readXMLDocument()` methods. Each of these overloaded methods takes a single input parameter used to identify the `File`, `OutputStream`, or `Writer` instance to which the decompressed XML document is written.

A delegate instance of the `XMLDocumentTreeDecoder` class performs the actual decompression. Figure 7 lists the public methods for this class. When the client program calls one of the `readXMLDocument()` methods of the `XMLDocumentInputStream` instance, `XMLDocumentTreeDecoder.decode()` is invoked to perform a reading of the compressed data stream. As each tree node is encountered, its type is determined and the appropriate static method of the `XMLDocumentWriter` class is invoked to output the decompressed node data. Figure 8 depicts the publicly accessible methods for this class.

treechop::XMLDocumentTreeDecoder
+ XMLDocumentTreeDecoder(inStream: DataInputStream, out: Writer)
+ decode()

Figure 7: Class Diagram for XMLDocumentTreeDecoder

treechop::XMLDocumentWriter
+ writeAttribute(attName: String, attValue: String, out: Writer)
+ writeCDATASection(cdata: String, out: Writer)
+ writeClosingElement(elName: String, out: Writer)
+ writeComment(comment: String, out: Writer)
+ writeContainer(containerContents: String, out: Writer)
+ writeDataValue(data: String, out: Writer)
+ writeProcessingInstruction(pi: String, out: Writer)
+ writeStartElement(elName: String, out: Writer)
+ writeStartElementClosing(out: Writer)

Figure 8: Class Diagram for XMLDocumentWriter

For networked applications, the `XMLServerSocket` class can be used to simplify the process of decoding compressed XML data sent using an instance of `XMLSocket`. The relationship between the `XMLServerSocket` and `XMLSocket` classes mirrors that of the `java.net.ServerSocket` and `java.net.Socket` classes respectively. The client program on the receiving end constructs an `XMLServerSocket`, and calls the `accept()` method to obtain an `XMLSocket` object bound to the incoming request. Figure 9 illustrates the key public methods for `XMLServerSocket`.

treechop::net::XMLServerSocket
+ XMLServerSocket(port: int, backlog: int)
+ XMLServerSocket(port: int, backlog: int, bindAddr: InetAddress)
+ XMLServerSocket(port: int)
+ accept(): XMLSocket
+ bind(endpoint: SocketAddress, backlog: int)
+ bind(endpoint: SocketAddress)
+ close()
+ isBound(): boolean
+ isClosed(): boolean

Figure 9: Class Diagram for XMLServerSocket

2.4.3 Query Processor

To enable direct querying of a TREECHOP-compressed data stream, the client program on the receiving end of the data stream constructs an instance of the `XMLDocument-QueryableInputStream` class. This class is a subclass of `XMLDocumentInputStream`, containing additional logic for mapping query predicates to corresponding code words, and for reporting query matches as they are encountered within the data stream. Figure 10 displays the various public methods for `XMLDocumentQueryableInputStream`.

treechop::io::XMLDocumentQueryableInputStream
+ XMLDocumentQueryableInputStream(source: InputStream)
+ addQuery(queryPath: String, handler: QueryHandler)
+ close()
+ read(): int
+ readXMLDocument()
+ removeQuery(queryPath: String, handler: QueryHandler)

Figure 10: Class Diagram for XMLDocumentQueryableInputStream

The `XMLDocumentQueryableInputStream` instance contains a delegate instance of `XMLDocumentQueryableTreeDecoder` (the publically-accessible methods for this class are shown in Figure 11). This latter class maintains a hash table whose entries consist of a code word (the key value) mapped to a `java.util.List` containing one or more instances of the `QueryHandler` interface. Each time that a key value from this table is encountered in the compressed data stream, the associated D value for the current tree node is extracted from the data stream and passed to each of the corresponding `QueryHandler` instances.

The `QueryHandler` interface (displayed in Figure 12) defines a single method `handleQuery()`. For a particular application, the desired query processing logic is incorporated by creating one or more implementations of this interface and associating each implementation with the appropriate tree path predicate through an invocation of the `XMLDocumentQueryableInputStream.addQuery()` method.

treechop::XMLDocumentQueryableTreeDecoder
+ XMLDocumentQueryableTreeDecoder(source: DataInputStream)
+ decode() + registerQueryHandler(xpath: String, handler: QueryHandler) + removeQueryHandler(xpath: String, handler: QueryHandler)

Figure 11: Class Diagram for XMLDocumentQueryableTreeDecoder

For networked applications, `XMLQueryableSocket` and `XMLQueryableServerSocket` provide query-capable analogues for the `XMLSocket` and `XMLServerSocket` classes presented earlier.

treechop::query::QueryHandler
+ QueryHandler()
+ handleQuery(match: String)

Figure 12: Class Diagram for QueryHandler

3 Experimental Results

This section presents the results of three experiments that were carried out to evaluate the effectiveness of TREECHOP. The first experiment compares the achieved compression rates for TREECHOP on a corpus of XML documents with those of other compressors. The second experiment serves to evaluate the transmission speed of XML data over a TCP/IP network for TREECHOP and alternative compression schemes. The final experiment rates the efficiency of TREECHOP compressed queries.

In all three experiments, the local system was a Dell D600 laptop computer with a 1400 MHz Pentium 4 M processor and 512 MB of RAM, running Windows XP Professional. For the second and third experiments, an ADSL connection was used to connect to a remote server running FreeBSD 4.10.

3.1 Compression Rate

This experiment was designed to measure the compression rates achieved by TREECHOP, GZIP, and XGRIND on a corpus of four XML files. Table 3 describes the structural characteristics of each test file, including the original document size, the number of elements and attributes, and the total number of characters in the data sections. These test files include:

- "baseball.xml" contains player statistics from the 1998 Major League Baseball season, expressed as XML;
- "macbeth.xml" is an XML representation of Shakespeare's play *Macbeth*, which serves as an example of a document with large sections of free-form English text;

File	Size (KB)	Elements	Attributes	Data Characters
baseball	788	27080	0	230970
macbeth	175	3975	0	97625
150emp	26	901	150	8277
100000emp	16831	600001	100000	5534311

Table 3: Characteristics of XML Files in Compression Corpus

Compressor	Baseball	Macbeth	150emp	100000emp
TREECHOP	0.60	2.08	1.07	0.81
GZIP	0.63	2.10	1.10	0.85
XGRIND	2.07	3.81	3.79	2.51

Table 4: Compression Performance of TREECHOP, GZIP, and XGRIND on Corpus (measured in bits-per-character)

- "150emp.xml" is a highly-structured document containing 150 employee records, where each record consists of an attribute referring to the employee id number and nested elements for first name, last name, age, job title, and phone number; and
- "100000emp.xml" is similar to the previous document, with 100,000 employee records.

Table 4 lists the compression ratios achieved by each compressor on the four test files. Each entry is expressed as a bits-per-character value. The testing results indicate that XGRIND performs significantly worse than either GZIP or TREECHOP on each file, while TREECHOP slightly outcompresses GZIP in all cases.

3.2 Compression/Decompression Speed

To evaluate the speed of TREECHOP's compression and decompression strategies, an experiment was carried out in which a set of documents ranging in size from 2 KB to 1 MB were compressed, transmitted over a TCP socket connection to a remote server located 20 km (12 miles) from the client, and decompressed on the server side to reproduce the original document. The time measurements were taken on the remote system and reflect the total time elapsed from initiation of the socket connection to completion of the decompression process. Each test document consisted of a set of employee records, sharing structural similarity with "150emp" and "100000emp" from the corpus used in Section 3.1. The performance of TREECHOP was compared with GZIP and with uncompressed transmission of each document. The results of this experiment are depicted in Figure 13. Results for XGRIND were not included since the current implementation does not support online transmission of compressed data between networked systems.

Not surprisingly, both GZIP and TREECHOP experience a widening performance advantage over raw XML data transmission as the document size increases. In addition, GZIP performs slightly faster than TREECHOP on the larger documents in the test set

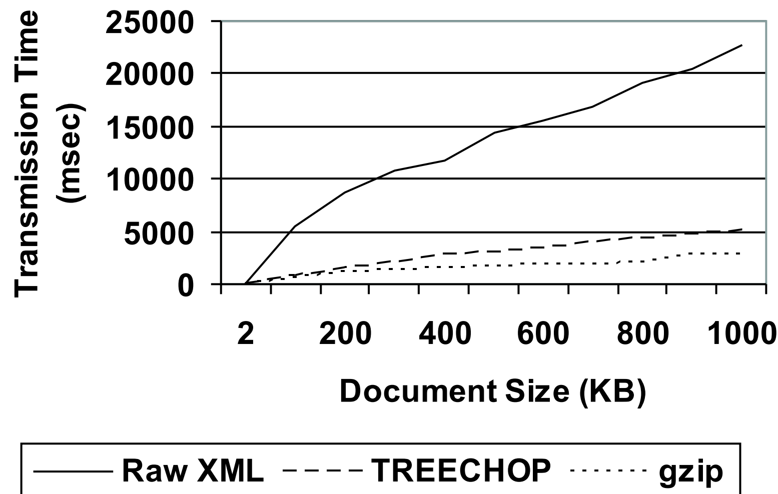

Figure 13: Transmission Speed of TREECHOP, GZIP, and Raw XML Data Over a TCP/IP Socket Connection

(presumably due to the additional computational expense of calculating and decoding the codeword for each tree node).

When interpreting the results, it is worth noting that as the physical distance between the client and server systems is increased, the slight speed advantage of GZIP during the compression and decompression phases may eventually be eclipsed by the sheer expense of sending data across the network (if the network approaches its saturation point). When this is the case, TREECHOP's ability to compress data at a better rate than GZIP will allow it to achieve superior transmission rates. Additionally, in cases where the receiving server program is only interested in a subset of the document content (e.g. the name of each employee), it would be more efficient to perform a search on TREECHOP-compressed data in lieu of carrying out a full decompression of the document.

3.3 Querying

As a means of evaluating the query processing performance of TREECHOP, a set of experiments was conducted in which the same set of XML documents from the experiment in Section 3.2 (ranging in size from 2 KB to 1 MB) was transmitted across a distance of 20 km (12 miles) from a client system to a remote server.

A query handler was registered on the decoder to extract the last name value within each employee record and print it to the console. For comparison purposes, we also evaluated the result of applying an XSLT stylesheet that carried out similar functionality to incoming streams of GZIPped and uncompressed XML data. The Xalan-Java XSLT processor [15] was used for the latter two sets of tests.

The time measurements were taken on the remote system and reflect the total time elapsed from initiation of the socket connection through to completion of the process of

writing query matches to the console.

(Since the current implementation of XGRIND does not support streaming transmission over a network, we did not include it in this experiment.)

As shown in Figure 14, both TREECHOP and the combination of GZIP and XSLT perform much faster on even small XML documents than XSLT applied to raw XML data. As document size increases, this difference becomes more pronounced. On files less than roughly 500 KB in size, GZIP/XSLT slightly outperforms TREECHOP querying; on larger files, TREECHOP queries run faster.

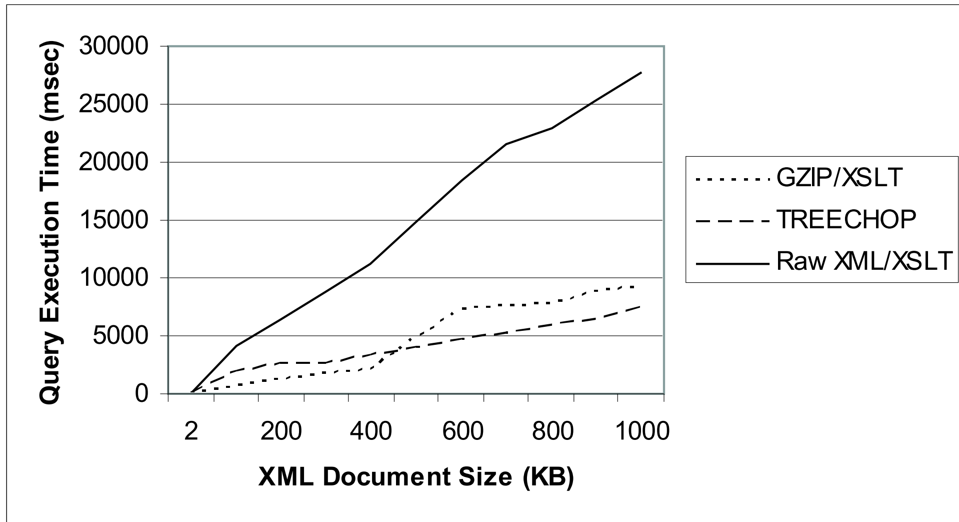

Figure 14: Query Times for TREECHOP, GZIP/XSLT, and Raw XML/XSLT

4 Conclusions

In this report, we have described an online compression and querying strategy for XML data which betters the compression rate of GZIP while providing query-friendly annotations to the compressed data stream.

As part of our future work, we intend to examine ways in which the existing compression rates and execution speed of TREECHOP compression can be improved. One possibility is to take advantage of an existing DTD or schema document to predefine codeword mappings and avoid the necessity of transmitting the D value for element and attribute nodes in the compression stream. In addition, query processing could be sped up via the inclusion of indexing information at the beginning of the compression stream (at the probable expense of the achievable compression rate and an increase in the time required to perform the compression). This would be a desirable feature in data retrieval applications, where a set of XML data is compressed infrequently but queried frequently.

An active area of research involves the use of XML for Selective Dissemination of Information (SDI) [1, 5], where massive amounts of data are continuously collected,

filtered against a set of user profiles, and delivered to interested users. The ability of TREECHOP to efficiently filter specific portions of the XML document tree from a compressed data stream would likely translate effectively to this application domain.

References

[1] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the Twenty-Sixth International Conference on Very Large Data Bases*, pages 53–64, 2000.

[2] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Systems Research Center, 1994. Research Report 124.

[3] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proceedings of IEEE Data Compression Conference*, pages 163–172, 2001.

[4] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, April 1984.

[5] Y. Diao, S. Rizvi, and M. Franklin. Towards an Internet-scale XML dissemination service. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 612–623, 2004.

[6] Extensible Markup Language (XML) 1.0 (3rd ed.), 2005. Retrieved March 2005 from http://www.w3.org/TR/REC-xml.

[7] The gzip home page, 2005. Retrieved March 2005 from http://www.gzip.org.

[8] D. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[9] G. Leighton, J. Diamond, and T. Müldner. AXECHOP: A grammar-based compressor for XML. In *Proceedings of the 2005 IEEE Data Compression Conference*, page 467, 2005.

[10] G. Leighton, J. Diamond, and T. Müldner. A grammar-based approach for compressing XML. Technical Report TR-2005-004, Jodrey School of Computer Science, Acadia University, June 2005.

[11] H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, 2000.

[12] J. Min, M. Park, and C. Chung. XPRESS: a queriable compression for XML data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 122–33, 2003.

[13] Simple API for XML (SAX), 2005. Retrieved March 2005 from http://www.saxproject.org.

[14] P. Tolani and J. Haritsa. XGRIND: a query-friendly XML compressor. In *Proceedings of the 2002 International Conference. on Database Engineering*, pages 225–34, 2002.

[15] Xalan-Java version 2.6.0, 2005. Retrieved March 2005 from http://xml.apache.org/xalan-j/.

Appendix A: Example Programs Using the TREECHOP Library to Compress, Decompress, and Query XML data

Example 1: Compressing an XML document ("10emp.xml") residing on the local file system

```
package treechop.examples;

import java.io.File;
import java.io.FileOutputStream;

import treechop.exceptions.CompressorException;
import treechop.io.XMLDocumentOutputStream;

/**
* Program that reads in an XML file from the local file system,
* compresses it, and writes the compressed version to a second file.
*
*/
public class XMLFileCompressor
{
   public static void main(String[] args)
   {
      File inFile = null;
      File outFile = null;
      XMLDocumentOutputStream out = null;

      try
      {
         inFile = new File("10emp.xml");   // input XML document
         outFile = new File("10emp.cmp");  // output file
      }

      catch (Throwable t)
      {
         t.printStackTrace();
         System.exit(-1);
      }

      try
      {
         // instantiate output stream for writing compressed XML
         out = new XMLDocumentOutputStream(
            new FileOutputStream(outFile));

         // perform the compression
         out.writeXMLDocument(inFile);
```

```
      }

      catch (CompressorException e)
      {
         e.printStackTrace();
         System.exit(-2);
      }

      catch (RuntimeException e)
      {
         e.printStackTrace();
         System.exit(-3);
      }

      catch (Throwable t)
      {
         t.printStackTrace();
         System.exit(-4);
      }

      finally
      {
         if (out != null)
         {
            try
            {
               out.close();
               out = null;
               inFile = null;
               outFile = null;
            }

            catch (CompressorException e)
            {
               e.printStackTrace();
               System.exit(-5);
            }
         }
      }
   }
}
```

Example 2: Reading in compressed XML data from a local file and decompressing to a separate local file

```
package treechop.examples;

import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileWriter;

import treechop.io.XMLDocumentInputStream;
import treechop.exceptions.DecompressorException;

/**
* Program that reads in compressed XML data from the local file
* system, decompresses it, and writes the decompressed version to a
* second file.
*
*/
public class XMLFileDecompressor
{
   public static void main(String[] args)
   {
      File outFile = null;
      File inFile = null;
      XMLDocumentInputStream inStream = null;

      try
      {
         outFile = new File("10emp_dcp.xml");
         inFile = new File("10emp.cmp");
      }

      catch (Throwable t)
      {
         t.printStackTrace();
         System.exit(-1);
      }

      try
      {
         // instantiate input stream for reading in compressed
         // XML data
         inStream = new XMLDocumentInputStream(
            new BufferedInputStream(
               new FileInputStream(inFile)));
```

```
         // perform decompression
         inStream.readXMLDocument(new FileWriter(outFile));
      }

      catch (Throwable t)
      {
         t.printStackTrace();
         System.exit(-2);
      }

      finally
      {
         if (inStream != null)
         {
            try
            {
               inStream.close();
               inStream = null;
            }

            catch (DecompressorException e)
            {
               e.printStackTrace();
               System.exit(-3);
            }
         }
      }
   }
}
```

Example 3: Querying a compressed XML document

```
package treechop.examples;

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.PrintStream;

import treechop.exceptions.DecompressorException;
import treechop.io.XMLDocumentQueryableInputStream;
import treechop.query.SimpleQueryHandler;

/**
 * Program that queries a compressed XML document and outputs
 * query matches to console.
 *
 */
```

```
public class XMLFileQuerier
{
   public static void main(String[] args)
   {
      XMLDocumentQueryableInputStream inStream = null;

      try
      {
         // instantiate queryable input stream for reading in
         // compressed XML data
         inStream = new XMLDocumentQueryableInputStream(
            new BufferedInputStream(
               new FileInputStream("10emp.cmp")));

         // register query handler which searches for employee last
         // name values within the compressed data stream, and
         // outputs each query match to System.out
         inStream.addQuery("/employees/employee/lname",
            new SimpleQueryHandler(System.out));

         // perform the processing
         inStream.readXMLDocument();
      }

      catch (Throwable t)
      {
         t.printStackTrace();
         System.exit(-2);
      }

      finally
      {
         if (inStream != null)
         {
            try
            {
               inStream.close();
               inStream = null;
            }

            catch (DecompressorException e)
            {
               e.printStackTrace();
               System.exit(-3);
            }
         }
      }
```

```
   }
}


package treechop.query;

import treechop.exceptions.DecompressorException;
import java.io.PrintStream;

/** Writes each matching occurrence of the query predicate
 *  to the specified PrintStream.
 *
 */
public class SimpleQueryHandler extends QueryHandler
{
   private final PrintStream out;

   /** Constructs a new SimpleQueryHandler.
    *
    * @param out a <code>PrintStream</code> object on which query
    * matches will be written
    *
    */
   public SimpleQueryHandler(PrintStream out)
   {
      this.out = out;
   }

   /** Writes query match to standard output.
    *
    * @param match the contents of the query match
    */
   public void handleQuery(String match) throws DecompressorException
   {
      out.println(match);
   }
}
```

Example 4: Compressing and transmitting an XML document over a TCP/IP socket connection

```
package treechop.examples;

import treechop.io.XMLDocumentOutputStream;
import treechop.net.XMLSocket;
import treechop.net.XMLSocketException;
```

```
/**
 * Program that transmits a compressed XML document over a TCP/IP socket
 * connection.
 *
 */
public class XMLSocketSender
{
   // name of remote system
   private final static String REMOTE_HOST_NAME = "localhost";

   // remote system port number
   private final static int REMOTE_PORT_NO = 6000;

   // document to be sent
   private final static String DOCUMENT_NAME = "10emp.xml";

   public static void main(String[] args)
   {
      XMLSocket theSocket = null;

      try
      {
         // create a new socket connection to the remote host
         theSocket = new XMLSocket(REMOTE_HOST_NAME,
            REMOTE_PORT_NO);
         theSocket.setKeepAlive(true);

         // get a reference to the OutputStream associated
         // with socket connection
         XMLDocumentOutputStream out =
            (XMLDocumentOutputStream)
               theSocket.getOutputStream();

         // write compressed document to the socket
         out.writeXMLDocument(DOCUMENT_NAME);

      }

      catch (XMLSocketException e)
      {
         e.printStackTrace();
      }

      finally
      {
         if (theSocket != null)
```

```
         {
            try
            {
               theSocket.close();
               theSocket = null;
            }

            catch (XMLSocketException e)
            {
               e.printStackTrace();
               System.exit(-1);
            }
         }
      }
   }
}
```

Example 5: Receiving compressed XML data over a TCP/IP socket connection, decompressing and displaying contents on console

```
package treechop.examples;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

import treechop.io.XMLDocumentInputStream;
import treechop.net.XMLSocket;
import treechop.net.XMLServerSocket;
import treechop.net.XMLSocketException;

/**
 * Program that receives a compressed XML document over a TCP/IP socket
 * connection and writes decompressed document to a local file.
 *
 */
public class XMLSocketReceiver
{
   // port number for incoming connections
   private final static int PORT_NO = 6000;

   public XMLSocketReceiver() throws XMLSocketException
   {
      XMLServerSocket theServerSocket = new
         XMLServerSocket(PORT_NO);

      while (true)
```

```
   {
      // spawn a separate thread to handle each incoming
      // connection
      XMLSocket conn = theServerSocket.accept();
      new ConnectionHandler(conn).start();
   }
}

public static void main(String[] args)
{
   XMLSocketReceiver receiver;

   try
   {
      receiver = new XMLSocketReceiver();
   }

   catch (XMLSocketException e)
   {
      e.printStackTrace();
      System.exit(-1);
   }

   catch (Throwable t)
   {
      t.printStackTrace();
      System.exit(-2);
   }
}

class ConnectionHandler extends Thread
{
   XMLSocket theSocket = null;

   ConnectionHandler(XMLSocket s)
   {
      this.theSocket = s;
   }

   public void run()
   {
      try
      {
         // get a reference to the InputStream for the
         // socket
         XMLDocumentInputStream in =
            (XMLDocumentInputStream)
```

```
                  theSocket.getInputStream();

            // read in the next document from the input
            // stream, and write decompressed contents to
            // local file named "10emp_dcp.xml"
            in.readXMLDocument(new BufferedWriter(new
               FileWriter("10emp_dcp.xml")));
         }

         catch (IOException e)
         {
            e.printStackTrace();
         }

         catch (XMLSocketException e)
         {
            e.printStackTrace();
         }

         finally
         {
            if (theSocket != null)
            {
               try
               {
                  theSocket.close();
                  theSocket = null;
               }

               catch (XMLSocketException e)
               {
                  e.printStackTrace();
                  System.exit(-3);
               }
            }
         }
      }
   }
}
```