Schema Based XML Compression

Naphtali Rishe[1], Ouri Wolfson[2], Ben Wongsaroj[3], Damian Small[1], Mauricio Alarcon[1], Naidel Lorenzo[1], Ricardo Koller[1], Sajib Kundu[1], Scott Graham[1], Kevin Alexander[3], Malek Adjouadi[1]

Florida International University [1] | *University of Illinois at Chicago[2]* | *Florida Memorial University[3]*

Abstract

XML has grown into a widely used and highly developed technology, due in part to the subcomponents built around the technology (advanced parsers, frameworks, libraries, etc). The use of XML reduces development time and increases the robustness of distributed applications. Due to these advantages, a large and growing range of distributed applications, such as Web-services, use XML as the basic unit of communication.

This paper surveys the topic of XML compression and proposes a new method that uses schema information for the compression algorithm. The schema provides valuable information to the compressor by specifying the data type and format of each element in the XML document. For example, if the compressor knows that a portion of data is numeric, it can intelligently save it using a binary representation instead of trying to compress the string representation.

1. Introduction

XML [4] is a markup language and is used as a means of structuring data. It is a standard that defines a set of rules needed by any text file in order to define itself as a valid XML document. XML documents are structures which are organized in a hierarchical "tree-like" manner. The documents have two basic sections: the structure (skeleton) or metadata of the document and the data itself. The metadata is a tree of nodes where each node is a container for the data. XML is widely used for two reasons: the simple set of rules specified in the XML document and the highly evolved parsers and set of technologies built around the document. These are referred to as XML technologies and comprised of the following: XPath, XQuery, XSLT, etc. Software designers are considerably moving towards XML technologies for data storage, data communication and data transformation [16]. Applications reading information from a file with an arbitrary structure would need to know the structure of the document and then build a parser that understands the document. With XML this can be easily achieved, if the file is saved as an XML document where you can use an XML parser to query the document for the needed data. Thus, software companies have used them as basic blocks of communication. A very common example of this is Web Services [5].

When a file is adhering to the XML standard rules, we say that the XML is correct. Specific information is needed in order to read or write an XML file for some specific application. This information is a set of rules that define the structure and related data that the XML documents can implement. These rules are wrapped in another document called schemas which has two specific forms: DTD or XML [6, 7]. If an XML follows the rules defined in its related schema or DTD file, we say that the XML document is valid. These rules [7] define the structure by specifying all the nodes required, order and relations in an XML document. Schemas also specify the data that the container nodes can have. The data specification is achieved by the definition and use of the following data types: integers, floats, dates, enumerations and user defined data types using regular expressions.

Nevertheless, there is a problem with XML when used as a communication medium due to the high amounts of metadata produced which consequently causes too much overhead to be used directly in communication or storage. A common solution has been the use of regular compression to reduce the metadata size before sending it through the network. The deficiency in regular compressions is in the time spent deciphering and translating the structure of the data itself to determine and apply the appropriate dictionary transformations. No distinctions are made between files; all that is known is the data stream of the data itself. The XML compressor will spend valuable time performing this operation.

Knowing that a file is an XML document can be very useful for the compressor as it can determine which sections of the stream are data and which sections are metadata. Current XML compression algorithms remove the metadata and treat the metadata like any normal compression algorithm would.

A naive approach that is widely used is to compress the XML using a regular compressor. However, there are faster and more efficient XML compression techniques and methodologies [1][2][3]

XML also contains redundant information. The first part of the information is already specified in the schema, for example the order of the children may not be important for the XML to be valid. In fact, the compressor may be able to reconstruct some of the XML by using the schema. That is if we assume the XML is valid, then the validity of the XML document is essential in order for data transmission to occur. This may imply high levels of compression because the order information of the example can be repeated throughout the XML document. Also in a transmission the schemas does not need to be compressed and transmitted since it already exists in the receiver. XML schemas and DTD's already specify the data it has hence there is no need to infer the data type by adding an extra algorithm. Prior approaches automatically infer the data types by obtaining statistics about the data or by querying the user for that information [1, 2, 3]. These reasons make XML a very convenient file for compression.

This paper is organized as follows: Sec.2 depicts previous work performed on compression algorithms. Sec.3 describes our proposal to the paper and Sec. 4 outlines our conclusion.

2. Previous Work

Compression algorithms can be divided in two basic groups: lossy and lossless [8]. A lossless compressed document can be decompressed and reverted back to the original document. A lossy compressed document cannot be reverted to the same original state, but it can be reverted to a very similar document where the similarity is measured by a concept called distortion. Distortion is typically defined as the accepted distance that any two characters in the dictionary can have. It provides a means to measure the distance that any two blocks of characters can have. A typical example of lossless compressor is gzip. An example of lossy compression is jpeg.

Lossless compression algorithms maps blocks of data into sequences of bits. The compressor's objective is to minimize the number of bits used to represent each block. Unfortunately there is a theoretical limit on this mechanism: entropy (H) [8]. Some algorithms use a fixed set of mappings in order to convert blocks into bits. This mapping can be viewed as a large table that relates all known blocks into binary representations, or sequence of bits (Table 1: Mapping Table). This type of compression is called dictionary based compression. Other compressors dynamically build the mapping table based on the text characteristics. These compressors are usually referred to as adaptive and semi-adaptive. Some of them map blocks of fixed size into variable number of bits, while others map blocks of variable size into a fixed number of bits. Huffman [15] encoding and Lempel Ziv [14] encoding are examples of semi-adaptive and adaptive encoding respectively. Arithmetic encoding is another example of adaptive encoding.

Table 1 : Mapping Table

Block	BitsSequence
acb	11
aa	10101011
yhnb	11111
hjkb	1011
asdfb	10101011
oopb	10101011
pb	11011
yb	1111011
eb	1010
fbcc	10101111
gabg	1010101010101011

Huffman compressors use variable length sequence bits in such a way that most frequent blocks are saved using shorter bits sequences. But in order to know the distribution of the blocks it needs to first read the whole data just to gather the proper statistics. This type of compression is called semi-adaptive, while compressors such as dictionary based (which does not need to build this type of mapping) are referred to as static compressors. Lempel Ziv compression constructs the mapping table while compressing the document. It starts by mapping blocks of size one and for each new match in the map table it adds a new entry made of the matched entry plus the next character of the stream. These types of algorithms that construct the map table dynamically are called adaptive compressors.

We mentioned that there is a limit on the compression rate we can get with any compression algorithm. But there are algorithms that take into account the characteristics of the data similar to the methods mentioned above. This has been applied to many specific sources including XML. XML compressors can be divided into two groups: queriable and non-queriable. Queriable compression may be needed in cases where there is too much overhead in decompressing the XML file, or when the space requirements are so high that the XML document cannot be saved into a plain text format; therefore it becomes necessary to answer queries directly on the compressed

document. Experiments have shown that querying compressed documents in this manner can take less time than on uncompressed documents [2]. In order to query compressed documents we need to physically maintain the structure of the XML in a compressed manner; this method is called homomorphism.

2.1. XML Structure Compression

As the name suggests, this paper [9] proposes a method to compress the structure of an XML document by proposing a novel algorithm given the DTD of a document, the algorithm separates the structure of the document from its data. For simplicity, the authors only work with DTDs that define elements. For actual encoding to take place, the parse tree (d, D) is constructed from the existing document (d) using the schema information in the DTD (D). The nodes in the parse tree correspond to the elements of the XML document d and the operators from the regular expressions used in content models in D.

The compression occurs in two steps. First, all the leaf nodes containing texts are pruned from the parse tree. This new tree is called asstruct (d). The encoding algorithm takes PARSE(d, D) as input and produces a minimum length encoding ENCODING(d, D) as output. The compression process applies further pruning to the parse tree maintaining a tree representation only of the structures that needs to be encoded in order for the decoding to reconstruct the document given DTD D. This tree is called PRUNE(d, D). The tree is then encoded PRUNE(d, D) using a breadth-first traversal in such a way that each repetitive node is encoded by a number of bits, say B, encoding the number of children of the repetitive node and where each decision node is encoded by a single bit (which may be 0 or 1) according to its child.

The compression of the algorithm thus contains three elements :(1) DTD, (2) encoded structure(in terms of bits) conforming the DTD, and (3) the actual data contained. The outputs can then be further compressed by piping them through standard text compression tools. The decoding algorithm takes ENCODING(d, D) and D as input and reconstructs STRUCT(d). The data can then be added to STRUCT(d) in a simple way to obtain the original document d. This encoding algorithm is in the spirit of the Minimum Description Length (MDL) of the information theory, which is based on the idea of choosing the model that minimizes the lengths of the encoding. It assumes that document elements are independent of each other. The overall length of the encoding is O(n) bits, where n is the number of nodes in the parse tree.

Unlike XMill, which will be discussed in 2.2, the algorithm takes the advantage of using the DTD to separate structure from text. Instead of relying upon existing compressing software to encode the structure, the technique also states an algorithm to compress the structure in an efficient way which ensures a good compression ratio of the XML document. The algorithm does not provide a way to construct the pruned tree from the parse tree. Moreover, a lot of computing overhead is incurred in computing the pruned tree of a large XML file.

2.2. XMILL

XMill [2] implements a clever and simple idea on compressing an XML document. It applies a 3 step process to compress the XML document, by applying specific compressors to specific data types to obtain the best compression ratio. This idea is very effective; furthermore, it is even better to transform a non-xml data and then compress it, than compress without transforming it; since you will end with a small size compressed data.

The main reason for this is that XML groups data into logical related elements. After the separation and the compression stage (via the XMill operation) the result is transformed into a highly compressed set of data that has a better compression ratio than its counterpart (same data in a non-xml format).

The first step that XMill does is separate structure from data, what essentially happens is the tags are dictionary-encoded while data is assigned to specific containers through *"container numbers"* [2]. Subsequently all the related data items are grouped by their *"container numbers"* and finally, through the use of *"semantic compressors"* [2], the data is compressed.

Since data comes in a variety of specialized data types such as integer, dates, zip codes, etc., this kind of data is better compressed using specialized semantic compressors. As a result integers receive a completely different treatment to strings or dates ending with a better compression ratio than a mix of compressed data types.

The approach taken by XMill has the following two limitations: first, the result of the compressed data is not usable by query engines. Second, this process is useful only when data is greater than 20KB. Therefore, XMill seems more oriented to storage and transport than on the fly processing of the compressed data.

2.3. XGRIND: A Query-friendly XML Compressor

XGrind [3] capitalizes on a weakness of XMill regarding the direct querying of compressed data. XMill is designed to minimize the size of the compressed XML document, which reduces the network bandwidth required for transmission, and the disk space required for storage of the original document. However, this compression approach is not intended for directly querying or updating the compressed document which XGrind provides.

The following are advantages of direct queries on compressed data:

(1) Disk seek times are reduced since the compressed data fits into smaller physical disk area.

(2) Disk bandwidth is effectively increased due to the increased information density of the transferred data.

(3) Memory buffer hit ratio increases since a larger fraction of the document now fits in the buffer pool.

XGrind compresses individual element/attribute values using a context-free compression scheme.

Using this scheme, the exact-match and prefix-match user queries can be executed directly on the compressed document with decompression restricted to only the final results provided to the user. This means that the compressed document can be parsed using exactly the same techniques as those used for parsing the original XML document.

XGrind uses different techniques for compressing meta-data, enumerated-type attribute values and element/attribute values. These techniques are as follows:

Meta-Data Compression

XGrind follows the XMill compression approach of separating structure from content. The method to encode metadata is similar to that in XMill. Each start-tag of an element is encoded by a 'T' followed by a uniquely assigned element-ID. All end-tags are encoded by '/'s. Attribute names are similarly encoded by the character 'A' followed by a uniquely assigned attribute-ID.

Enumerated-type Attribute Value Compression

XGrind identifies such enumerated-type attributes by examining the DTD of the document and encodes the values using a simple $\log_{2K}$ encoding scheme to represent an enumerated domain of K values.

General Element/Attribute Value Compression

This is where the bulk of the processing is done. For Element/Attribute compression, two passes have to be made over the XML document: the first is to collect the statistics and the second is to perform the actual encoding. This methodology can be made more efficient by simply making one "sweep" over the data document to gather the necessary statistics and to perform the data encoding in one pass of the document.

In principle we could use a single character-frequency distribution for the entire document. However, in XGrind the compressed XML document can be viewed as the original XML document with its tags and element/attribute values replaced by their corresponding encodings. The advantage of doing so is that the variety of efficient techniques available for parsing/querying XML documents can also be used to process the compressed document.

XGrind Architecture

```
<?xml:namespace prefix = v
  ns = "urn:schemas-microsoft-com:vml"
/>
<?xml:namespace prefix = o
  ns = "urn:schemas-microsoft-
  com:office:office" />
```

Document compression is as follows:

(1) Invocation of the DTD Parser which parses the DTD of the XML Document and initializes frequency tables for each element/non-enumerated attributes; populating a symbol table for attributes having enumerated-type values.

(2) The Kernel then invokes the XML Parser which scans the XML document and populates the set of frequency tables containing statistics (in the form of frequencies of character occurrences) for each element and non-enumerated attribute.

(3) The XML Parser is invoked a second time by the kernel to construct a tokenized form – tag attribute or data value – of the XML document. These tokens are supplied to the kernel which calls for each token based on its type to either the Huffman-Compressor or Enum-Encoder.

Enum-Encoder is used for meta-data and enumerated type data items. Each start-tag of an element is encoded by a 'T' followed by a unique element-ID. All end-tags are encoded by '/'s. Attribute names are encoded by the character 'A' followed by a unique attribute-ID.

Huffman-Compressor is used for non-enumerated data items. It encodes each element/attribute value with the help of its associated Huffman tree which is constructed from its corresponding frequency table.

(4) The compressed output of the above encoders along with the various frequency and symbol tables is called the Compressed Internal Representation (CIR) of the compressor and is fed to XML-Gen which converts the CIR into a semi-structured compressed XML document.

2.4. Millau

Millau [10] is defined as an encoding format extension of the WAP binary XML format, aka WBXML [11]. It defines a compact binary representation of XML and is designed to reduce the transmission size of XML documents (with no loss of functionality or semantic information).

In addition Millau addresses the main drawbacks found in the WBXML format. WBXML does not compress data nor attributes that were not defined in the DTD and it does

not suggest strategies to build the code space in an efficient way.

Millau also presents an interesting idea that makes use of proxies in an efficient way to un/compress data for data exchange. This makes it useful for transmission of small chunks of data over the network, which makes the model easily adaptable to any pair of client/server architecture typical found in web services or any other xml data interchange model [5][13].

3. Our Proposal

Our fundamental contribution is the idea of using XML schemas for compression. This is done in two ways: (1) adding the concept of semi-lossy metadata compression based on the schema definition and (2) specific data type compression expanded to the use of regular expression definitions. Previous data type based compression [1, 2, 3] were restricted from any knowledge of the DTD or what the user was allowed to amend to this knowledge. The concept behind this model is to infer the data characteristics from the schema data type definition itself without actually reading the data. This also includes the ability to infer data characteristics and take advantage of the data types based on the definition of regular expressions. For example, we shall assume that there is an element called choice that only accepts one letter chosen from x, y and z.

```
<xs:element name="choice">
<xs:simpleType>
 <xs:restriction base="xs:string">
  <xs:pattern value="[xyz]"/>
 </xs:restriction>
</xs:simpleType>
</xs:element>
```

This type of data can be easily compressed by acquiring and utilizing the knowledge that appears in the Schema. Other examples include:

Table 2: Summary of regular expression definitions.

Type	RegExp
html tag	</?[a-z][a-z0-9]*[\^<>]*\>
currency	[0-9]\{1,3\}(?:,?[0-9]\{3\})*
US postal code	[0-9]\{5\}(?:-[0-9]\{4\})?

Although, there are other cases in which the specification is too broad to take advantage of during the compression process. We set the limit that it is better to use regular compressors for the specific data as a set of values for the 4 complexity indicators developed in [12].

Semi-lossy compression is a new term and new type of compression that is only applicable to certain files. As previously discussed; there are two types of compressions: the ones that cannot revert the compression to the original state, and the ones that can. However, there are certain files that are exempt from needing to revert to exactly the original state as there is simply no need for performing the conversion. The perfect case of this is XML. To further elaborate, an XML schema specifies that a specific node can only have a set of specific children in any order. Order related information is something we do not need to specify in our compressed document [7], as there is no need to compress nodes that are not specified in the Schema of the XML.

We can also make use of regular expressions by adding a small set of auto discovery rules based on the use of regular expressions. Our approach is to have an expandable repository with a common set of rules that define typical data.

Thus, while the XML document is parsed, the set of regular expressions present in the repository is used to match and group the related data even if the data has different paths. e.g. *Home phone number, Office phone number, Home postal code, Office postal code.* This approach not only has the advantage of using the schema information as a faster match for data, but also provides a means for discovering knowledge that can be easily expandable depending on the needs of the user and business logic.

4. Conclusions

Over the years a large amount of work has already been done in this area. We have observed that the size of the XML document is the main obstacle towards the wide use of XML in preserving web documents and sending web scripts across the network. The basic difference in compressing a normal document over XML data is that we can compress a lot of it if we can cleverly use the structured information associated with XML metadata. We have studied several existing compression technologies, e.g. XMill, XGrind, etc. All of these technologies can be broadly divided into two parts: 1) one which achieves higher compression ratio, but does not support queries, and 2) one which supports query over compressed data, but sacrifices compression ratio. We made relative comparisons of these existing compression techniques and concluded by suggesting an idea of XML compression by efficiently using the XML schema information: the use of regular expressions to define typical data set and to have a repository of rules based on regular expressions which can be very efficient in compressing the XML data.

5. Acknowledgements

This research was supported in part by NSF grants HRD-0317692, CNS-0220562, CNS-0320956, CNS-0426125, OII-0611017, DGE-0549489, IIS-0513736, and IIS-0326284, and NATO grant SST.NR.CLG:G980822.

6. References

[1] I. M. Author, “XPRESS: A queriable Compression for XML Data,” *the journal*, Vol. 17, pp. 1-100, 1987.

[2] H Liefke, D Suciu, “XMill: an efficient compressor for XML data,” *Proceedings of the 2000 ACM SIGMOD international conference*, Volume 29, Issue 2, pp. 153 164, 2000

[3] I. M. Author, “XGrind,” *the journal*, Vol. 17, pp. 1-100, 1987.

[4] I. M. Author, “XML specification www.w3.com/xml,” *the journal*, Vol. 17, pp. 1-100, 1987.

[5] I. M. Author, “Web Services,” *the journal*, Vol. 17, pp. 1-100, 1987.

[6] w3, “XML DTD www.w3.com,” *the journal*, Vol. 17, pp. 1-100, 1987.

[7] w3, “XML Schema www.w3.com,” *the journal*, Vol. 17, pp. 1-100, 1987.

[8] C. E. Shannon, “A Mathematical Theory of Communication,” *the journal*, Vol. 17, pp. 1-100, 1947.

[9] Mark Levene and Peter Wood, “XML Structure Compression,” Birkbeck College, University of London

[10] M Girardot, N Sundaresan, “Millau: an encoding format for efficient representation and exchange of XML over the Web” *Computer Networks*, Vol 33, pp.747-765, 2000

[11] Wap Binary XML Content Format, W3C Note 24 June 1999, http: //www.w3.org/TR/wbxml

[12] A. Ehrenfeucht, P. Zeiger Complexity Measures For Regular Expressions Department of Computer Science, University of Colorado, Boulder, Colorado

[13] W3C Web Services Activity home page, *http://www.w3.org/2002/ws/*

[14] J.Ziv, A.Lempel, A universal algorithm for sequential data compression, IEEE Transactions on information theory, 1977

[15] Huffman's original article: D.A. Huffman, "A method for the construction of minimum-redundancy codes" (PDF), Proceedings of the I.R.E., sept 1952, pp 1098-1102

[16] Maruyama, H. 2002. New trends in e-Business: from B2B to web services. New Gen. Comput. 20, 1 (Jan. 2002), 125-139.