XCOMP: AN XML COMPRESSION TOOL

By

Weimin Li

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2003

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other instituions or individuals for the purpose of scholarly research.

Signature of Author

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions of individuals for the purpose of scholarly research.

Signature of Author

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Table of Contents

Abstract

XML, as a data exchange and storage format, is extremely verbose, requiring compression for efficient transmission. Although this can be addressed by a general purpose compression tool, the effects of these tools are not satisfactory. In this thesis, an XML-specific compression tool, XComp, is proposed that exploits XML's self-describing property to re-group XML data. It follows the principle of separating structure from data, and grouping data based on semantics. The structure is encoded as a sequence of integers, while the data grouping is based on XML tags/attributes and their levels in the document tree. The re-organized data are compressed by existing compression algorithms. Performance evaluation shows that XComp outperforms both general purpose compression tools and previously proposed XML-specific compression approaches.

Acknowledgements

I recognize and appreciate all of the help from many people, without which my work could not be done.

First, I would like to thank Professor M. Tamer Özsu, my supervisor, for his many suggestions and constant support and encouragement during this research. Even in the recovery after a heart surgery, he kept reviewing my thesis and stimulated me to finish my work. His hardworking spirit and persistent pursue of best solutions in academic problems inspire me all the time.

I have been very fortune to have Dr. Grant Weddell and Dr. Frank W. Tompa read my thesis. I thank them for their precious time and valuable feedback.

I also want to thank for the help from Benjamin Bin Yao, who helped me set up the data set for the experiments, and Ning Zhang, who helped me figure out some Latex problems in thesis writing.

Finally, I acknowledge the continued love and support of my wife. Her support and encouragement always accompanied me throughout these years.

Chapter 1

Introduction

1.1 Overview

XML, the *eXtensible Markup Language* [24], is becoming a standard format for data storage and exchange, in particular over the Internet. It is based on and is a subset of the first truly viable commercial markup language *Standard Generalized Markup Language (SGML)* [9]. Another famous markup language is *HyperText Markup Language (HTML)* [21], which is commonly used in presenting Web Pages. HTML is also a subset of SGML. The markup refers to the method of conveying the meta-data. All these SGML-based markup languages use *tags* to delimit the major components of the meta-data, called *elements*. Example 1.1 shows tags and elements in a segment of an XML document.

```
<Person ID="IS9095">
    <FirstName>Wally</FirstName>
    <LastName>Robertson</LastName>
    <Address>200 University Ave.</Address>
    ...
</Person>
```

Example 1.1: An XML Segment

`<FirstName>`, `<Person>` are start-tags and `</FirstName>` and `</Person>` are end-tags. Start-tags and end-tags with the same *tag names*, such as `Person`, appear together as a pair. Such pair delimits an element value, i.e. `<FirstName> Wally </FirstName>`. Data content or nested elements are contained between them. We call a piece of continual data content that does not include any markups a *data item*. "*Wally*" and "*200 University Ave.*" are all examples of data items. In XML documents, data items are all presented as strings.

A marked up document can contain both data and information that tells what kind of roles these data play in that document. Such information will help applications in processing or presenting these data. Unfortunately, SGML and HTML have a number of disadvantages. SGML is too large and too complicated to handle easily. Although HTML was a success in Web page presentation, its extensibility is limited. Its pre-defined tags are used more in describing the appearance of the data rather than in defining the data structure. Moreover it doesn't provide a mechanism to create specific markup tags for the numerous types of entities in a Web page.

The variety of Internet calls for applications need an extensible, flexible and concise markup language to exchange data on the Internet. Driven by this need, *World Wide Web Consortium (W3C)* has created XML. The design goals [24] for XML are:

"1. XML shall be straightforwardly usable over the Internet.

2. XML shall support a wide variety of applications.

3. XML shall be compatible with SGML.

4. It shall be easy to write programs which process XML documents.

5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.

6. XML documents should be human-legible and reasonably clear.

7. The XML design should be prepared quickly.

8. The design of XML shall be formal and concise.

9.XML documents shall be easy to create.

10. Terseness in XML markup is of minimal importance."

The two most important virtues that make XML a success are *self-description* and *extensibility*. In XML, the tags that delimit XML components can also be used to describe the data contained in these components. By carefully choosing some semantic tag names, the document creators are able to tell the document consumers what roles these data play. For instance, a person's information, including name and address, can be easily extracted from the earlier XML segment because of the meaningful tags. In this case a consumer does not need a data schema to understand the documents; a shared XML vocabulary is enough.

Using vocabulary rather than schema gives rise to another benefit of XML — extensibility. XML provides a basic syntax but does not define the actual tags; the tag set can be extended by anyone for his own purpose, and a set of tags can be combined to define complicated data structures. Unlike HTML, which describes data and its presentation, XML separates semantics from its presentation. The self-describing documents tell how to interpret them and leave the presentation problem to document consumers. This mechanism makes XML documents more flexible and extensible, especially when they are exchanged between different kinds of applications.

However, such virtues are not free. One of the disadvantages is that XML documents are verbose, requiring compression before transmission over the Internet. This thesis deals with the problem of XML document compression. In the next section the

verbosity problem is explained and the necessity of compressing is discussed.

1.2 Why is XML Compression Important?

Meaningful tags make an XML document self-describing, but they also increase the document's size. In most cases, an element contains at most one data item. In other words a data item usually comes with a start-tag and an end-tag, as "*Wally*" does in `<FirstName>Wally</FirstName>` in Example 1.1. In some cases, the tags occupy even more space than data items. We use an example document to show how XML format increases the document's size.

Consider the invoice record in Example 1.2, which is a sales and shipping document including the customer's name and address, the ordered products, and payment information. We assume one invoice contains only one customer and several products.

```
<?xml version="1.0" encoding="utf-8" ?>

<Invoice ID="ST87302" date="2003-03-17" paid="true" >
  <Customer ID="2376">
    <Name>
      <FirstName>Wally</FirstName>
      <LastName>Robertson</LastName>
    </Name>
    <BillingAddress>
      <Street>200 University Ave.</Street>
      <City>Waterloo</City>
      <Province>ON</Province>
    </BillingAddress>
  </Customer>

  <ProductList>
    <Product ID="HI0941" >
      <name>MP3 Player</name>
```

```
      <Price>139.99</Price>
      <Units>2</Units>
    </Product>

    <Product ID="XP6015" >
      <name>CD Writer</name>
      <Price>79.99</Price>
      <Units>4</Units>
    </Product>
  </ProductList>
</Invoice>
```

Example 1.2: An Invoice Record In XML Format

This document contains 19 data items and its length is 688 bytes. The data itself, stored in plain text as in Example 1.3, has only 144 bytes. This value still can be further reduced if we use binary format. The repeating tag characteristic makes the XML version almost 5 times longer than the plain text version.

```
ST87302 2003-03-17 true 2376
Wally Robertson
200 University Ave. Waterloo ON
HI0941
MP3 Player
139.99 2
XP6015
CD Writer
79.99 4
```

Example 1.3: An Invoice Record In Plain Text

The verbosity of XML causes doubts about its efficiency as a data format in exchanging and archiving data. Although individual XML files transferred on the network may not be very large, the number of files is usually so large that the total

amount of data will consume a lot of bandwidth. Under the circumstances where bandwidth is limited and data traffic is very heavy, the transmission of XML documents may become a bottleneck for those applications. When XML is used to archive data, concerns still exist. The querying performance is also influenced by the size of the XML data. Because of these concerns, many applications (e.g. Web logs, biological data, etc) use other, specialized data formats to archive and exchange data, which are more economical than XML [13].

Compression of XML documents for efficient storage and transmission is thus needed. One alternative is to use general purpose compression tools, such as compress, zip, and gzip. Although these tools can reduce the size of an XML document, because they were developed for general use, they can not exploit the properties of XML to obtain the best compression results. Most of these general purpose compression tools use Lempel-Ziv (LZ) family of algorithms [12, 30, 31] as their compression engines. Here we only give a brief description of the Lempel-Ziv family algorithms. We will discuss them in detail in Chapter 2.

Many lossless compression algorithms *parse* the source symbol string into a sequence of *phrases*, then generate *codes* to represent these phrases. For example, LZ77 [30] parses the sequence 000101100000101001001000010011... into phrases as $0, 00, 1, 011,\ 0000, 01010, 0100, 10001, 0011, ...$ Then these phrases are adaptively encoded according to their frequencies in the sequence. The phrases with higher frequencies will get shorter codes, while the phrases with lower frequencies will get longer ones. By adaptive, we mean that an algorithm can adapt to the statistics of the source. For example, if the algorithm has read many symbol strings `00100`, it should be able to shorten the code for string `00100`. While the source statistics may change,

it can follow these changes by adjusting its codes and do it rapidly, so that it can be widely applicable to different kinds of source data.

While these algorithms are applied over XML data, they just treat the data as a piece of ordinary text data. The compressed XML document's size is still much larger than that of the compressed plain version. There are some places we can improve.

First of all, any well-formed XML document must have a hierarchical (tree) structure, which is constructed by tags. Each node of this tree represents an *element* or *attribute*. This tree structure can be coded with much fewer bytes if we extract it from the XML document. However, general purpose compression tools can't exploit this characteristic, instead compressing the entire document as a whole.

Secondly, in most XML documents, it is reasonable to expect that related data have similar data type and format. It is possible to group such data together according to the element type name. For example, in the invoice document of Example 1.2, the element `FirstName` contains a person's first name so that its data item usually only contains English letters. Similarly `Units` element contains a product's quantity so that its data item is always a positive integer. XML's self-description property provides a possibility to group similar data items according to element type name and attribute name. Compressing data items as groups will be better than compressing them together.

Thirdly, we know that any whitespace characters, i.e. space, tab, end-of-line character, except the ones within the content, have no semantic meaning. They may be removed by any XML parser. This feature allows us to remove any such whitespace characters while compressing without affecting any semantic information this document carries. Of course, such a compression policy can not reconstitute the

exact original files, but it does not influence information content of the XML files.

We can improve XML compression by these strategies, only when the compressor understands XML syntax. An XML-specific compressor is what is needed for this purpose.

1.3 Problem Definition

Our goal in this thesis is to develop an XML-specific compressor which has good performance in both compression ratio and transmission speed over the network. So, reducing XML documents' size is not the only target of the compressor. When XML documents are transferred on the network, as described in Figure 1.1, we expect the transmission speed to be improved if we add a compressor and a decompressor at the sender side and receiver side, respectively. To get better overall performance, compared with the case without compression, the total transmission time $T_{total} = T_{compress} + T_{transmit} + T_{decompress}$ must be less than the time used to send the original file directly. The cost factors that we can control are T_{total} and $T_{decompress}$. Therefore the compression speed is also a consideration. We expect a tradeoff between compression efficiency and speed to get the optimal transmission efficiency.

Apart from the above targets of compression ratio and transmission speed, we also set the following requirements for our compressor:

1. The compressor should be a universal XML compression tool. It should have a consistent performance over any well-formed XML document, no matter its size, its regularity (or irregularity), or its structure. In other words, it should not favor some kinds of documents, and work badly over others.

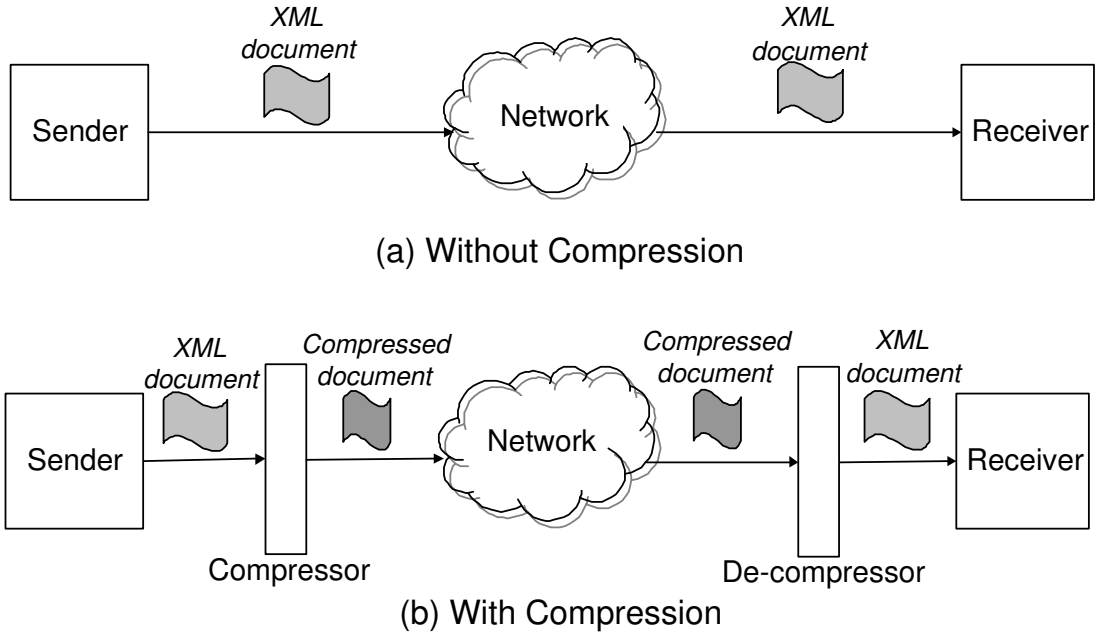

Figure 1.1: Transferring XML Document on the Network

2. It must work in a stream model. This means that it should work as a document *filter* that reads from an input stream and outputs the result as an output stream. The stream model allows it to consume input and produce output in parallel. This feature can significantly improve the transmission efficiency. Note that in this case the serial T_{total} formula is revised and needs to take into consideration the parallelism of compression and decompression activities at the sender and receiver sides.

3. It must compress XML data without the aid of any predefined data schema. *Document Type Definition (DTD)* [24] and *XML Schema* [25] provide mechanisms to describe the document structure so that an XML document can be validated, e.g., element parent-child relationships are respected, attributes have

valid values, all referenced entities have been properly defined. Therefore the information contained in DTD or XML Schema indicates data types of element content and attribute value that will improve compression efficiency. However, we can not assume that such data definition is ubiquitous. As mechanisms to perform structural or content verification, DTD and XML Schema are not likely to exist with every XML document that is exchanged between Internet applications. To make the compressor simple, we do not use any document definition to improve compression.

Another problem in XML compression is direct querying that supports querying a compressed document without decompressing it. XGrind [18] uses a two-pass Huffman code [7] to compress at the granularity of individual element/attribute values, so that *exact-match* and *prefix-match* queries can be executed directly on the compressed data. Because of the reduced document size, such a compressor can improve the query speed by saving the scanning time. In this thesis, we do not consider direct querying. Combining the decompressor with a query engine or an XML parser is left as future work.

1.4 Novelties

In this paper, we propose a new XML-specific compressor, called *XComp* that performs better than general-purpose compressors Zlib [3] and other XML compressors: XMill [13], XGrind and [18]. The basic idea behind XComp is separating structure from data and compressing them individually. The structure is encoded into a sequence of integers, and the data are grouped according to their element tags and attribute names and levels in the document tree. We do not develop a new compression

algorithm, but leverage the power of existing ones.

XComp can be applied to any well-formed XML document without any hints that imply the document structure or the data types. Written in C++, it can be used, as a file filter, by any application directly with its default settings. These default settings have been tuned to get an optimal effect on general XML files. Of course, users can modify these settings for specific classes of XML files to improve the results.

A novel feature of XComp is that it employs a parse-compress architecture in which an XML document is parsed first and then its components are re-organized and forwarded to the compression engine. During the parsing process, data are loaded into main memory so that the compression is a main memory operation. A memory window is applied when the document size exceeds a set value. XComp allows parsing and compressing to be done in parallel when the document can not fit in one memory window.

XComp improves compression by grouping related data items. Data items with the same element tag or attribute name and appearing at the same document tree level are compressed together. Such data usually have the same data type and similar data values. For instance, values of attribute `paid` may always be *true* or *false* in Example 1.2.

We also propose an algorithm to encode the document tree structure. There is the option of preserving white space or eliminating it. When white space is preserved, it becomes a lossless compression. In both cases, the XML structure is encoded into a sequence of integers.

1.5 Thesis Organization

The rest of this thesis is organized as follows. Background material on compression techniques and some related work on XML compression are introduced in Chapter 2. In Chapter 3, we present the architecture and implementation details of XComp compressor. The experimental results are discussed in Chapter 4. Finally, in Chapter 5, we draw our conclusions of this study and outline the future avenues to explore.

Chapter 2

Background of Data Compression and Related Work

In this chapter, we introduce the basics of information theory and two well-known and commonly employed lossless compression techniques, Huffman code and Lempel-Ziv compression. Some existing XML compressors are discussed in the second half of this chapter.

2.1 Information Theory

2.1.1 Lossless Compression

Lossless compression techniques provide exact recovery of the original data from their compressed version. Any information contained in an original file can not be lost during compression and reconstruction. These techniques are used widely in applications to save storage space and network bandwidth. For instance, most modems operating at 14,400 baud or higher use Lempel-Ziv in their default configuration. Since an XML compressor needs to preserve all data content, only lossless compression techniques can be used. Therefore, we restrict our interests to them in this thesis.

There are generally two kinds of compression techniques. One assumes a priori

knowledge of the statistics (i.e. the probability distribution $p(.)$ of single source letters) of the data source. Huffman code [7] is an example of this class. The second type works without such information; the encoder can *adapt* to the statistics of the source when it has read enough data from the source. We call them *universal compression techniques*.

2.1.2 Variable-Length Source Coding

In this and the next section, we assume that the source is a sequence of *independent, identically distributed* (i.i.d.) random symbols. The i.i.d characteristic indicates the fact that there is no statistical dependence among its successive source symbols. Given an alphabet $A = \{a_1, a_2, ..., a_m\}$, this means that $p(a_i|a_j)$, which is the probability that the symbol following an a_j is a_i, equals $p(a_i)$. This assumption precludes the possibility that any specific symbol strings intentionally appear repeatedly in the source, e.g. "the" has high appear frequency in an English document. We assume that the symbols are randomly distributed, although such data sources are not frequent in real world. However, we usually do not know the higher-order probability distributions that govern pairs, triples and longer n-tuples of source symbols. Unless otherwise noted, we shall assume i.i.d. model, not only for the above reason, but also because i.i.d. model is relatively easy to understand yet general enough to illustrate most of the fundamental concepts.

The essential principle of data coding is assigning short codes to frequently occurring symbols and long codes to ones that occur less often. Let us consider an example source with symbols and their probabilities as given in Table 2.1. These symbol probabilities satisfy the requirement $p_1 + ... + p_8 = 1$[1].

[1]We use p_i as a shorthand to represent $p(a_i)$

Source Symbol	Symbol Probability	Binary Codeword
a_1	0.35	000
a_2	0.20	001
a_3	0.15	010
a_4	0.15	011
a_5	0.05	100
a_6	0.05	101
a_7	0.03	110
a_8	0.02	111

Table 2.1: Code A: Fixed-Length Binary Code

In code A, each symbol is represented by a binary string, called *codeword*. Since the length of all codewords is the same, code A is a *fixed-length* code. Correspondingly, a *variable-length* code has codewords with different lengths. Codes B and C, shown in Tables 2.2 and 2.3, respectively, are both examples of variable-length code, where symbols with higher frequency have shorter codewords. Another type of code classification is based on the number of symbols used in codeword's alphabet. A *D-ary* code is one whose codeword's alphabet has D symbols. Codes A, B and C are all binary code.

Source Symbol	Symbol Probability	Binary Codeword
a_1	0.35	0
a_2	0.20	1
a_3	0.15	00
a_4	0.15	01
a_5	0.05	10
a_6	0.05	11
a_7	0.03	000
a_8	0.02	001

Table 2.2: Code B: Optimum Variable-Length Code

Source Symbol	Symbol Probability	Binary Codeword
a_1	0.35	0
a_2	0.20	10
a_3	0.15	1100
a_4	0.15	1101
a_5	0.05	1110
a_6	0.05	11110
a_7	0.03	111110
a_8	0.02	111111

Table 2.3: Code C: UD Variable-Length Code

We use the average codeword length to evaluate the efficiency of these codes. Assuming l_i denotes the codeword length of symbol a_i, the average codeword length is given by

$$\overline{l} = \sum_{i=1}^{m} p_i l_i$$

For code A, each symbol codeword always occupies 3 digits; regardless of the symbol probabilities, we always have $\overline{l}_A = 3$. For codes B and C we have

$\overline{l}_B = (0.35 + 0.20) \times 1 + (0.15 + 0.15 + 0.05 + 0.05) \times 2 + (0.03 + 0.02) \times 3 = 1.5$

$\overline{l}_C = 0.35 \times 1 + 0.20 \times 3 + (0.15 + 0.15 + 0.05) \times 4 + 0.05 \times 5 + (0.03 + 0.02) \times 6 = 2.9$

Code B has the shortest $\overline{l}$ among these codes. However, it has a fatal problem in that it is not uniquely decipherable. Consider the case where we have a segment of encoded binary sequence 11111. There is no way to decode it only according to these binary digits, for 11111 may stand for $a_2a_2a_2a_2a_2$ or $a_2a_6a_6$ or $a_6a_2a_6$ or other strings. This sequence is ambiguous unless "commas" are introduced to delimit the codewords. But comma is a new code symbol. Therefore, permitting the use of comma changes the binary code to a ternary one and also increases the length of

encoded sequence. Any attempt to use some delimiter between the codewords will increase $\overline{l}$.

One solution to this problem is to let the codewords themselves be able to delimit the encoded string. Such codes are called *uniquely decipherable* (UD) codes. A code is uniquely decipherable only when the code translates any finite-length encoded string to at most one possible source sequence. In our examples, code C is UD while code B is not. For lossless compression, we only consider the codes that are UD.

2.1.3 Entropy

Code C is not the only UD code for our example source. In Section 2.2.1, we will see that Huffman code is also UD and has smaller $\overline{l}$ than code C. One may ask whether it is possible to find a UD code with the shortest $\overline{l}$?

In late 40's and early 50's, Claude Shannon and others laid the foundations and gave birth to Information Theory. In his classic paper "Mathematical Theory of Communication" [16], Shannon introduced some notions including *entropy* as well as the phrase "Information Theory". An entropy formula was defined to qualify the amount of information contained in a signal. The entropy of an i.i.d. source with probability distribution $\{p_i\}$, is given by

$$H = -\sum_{i=1}^{m} p_i \log p_i$$

The base of the logarithm determines the units of H. Usually base 2 is used, in which case H is measured in bits. He proved that entropy sets the low bound of lossless compression. The average codeword length of any D-ary UD code is always at least as great as the source entropy calculated using base D. For example, for our

example source, H calculated with base 2 is $0.35\log_2 0.35{+}0.2\log_2 0.2{+}0.15\log_2 0.15{+}$ $0.15\log_2 0.15 + 0.05\log_2 0.05 + 0.05\log_2 0.05 + 0.03\log_2 0.03 + 0.02\log_2 0.02 \approx 2.512$. Code C is 15.4% bigger than the entropy. The gap between $\overline{l}$ and H is known as the *redundancy* of the coding scheme.

Consider another example where a source has a 256-symbol alphabet, in which each symbol has a uniform distribution ($p_1 = ... = p_{256} = 1/256$). Hence it needs $H = \log_2 256 = 8$ bits per symbol, which exactly equals $\overline{l}$ of any code that assigns one distinct 8-bit codeword to each symbol. In this case all such codes can reach the lowest bound of compression. In other words, no compression effect over these codes is available because the symbols are randomly and evenly distributed.

2.2 Compression Techniques

2.2.1 Huffman Code

Before Shannon's time, the variable-length codes were used in telecommunication systems, i.e. telegraph and telephone, to encode information symbols. Shannon's theories gave a theoretical bound that these codes could achieve. But he left open a major question in variable-length source coding: the algorithm that finds a variable-length code with minimum $\overline{l}$. This work was done by David Huffman [7]. He found such an algorithm that produces an optimal UD binary code for any given source symbol distribution $\{p_i\}$.

Given an alphabet and corresponding distribution, Huffman's algorithm generates a binary tree. Each leaf node in this tree is a symbol in the alphabet and each link is labelled with 0 or 1. When the tree is constructed, each symbol codeword can be obtained by traversing the path from the root to the leaf node. The tree is constructed

in a recursive way: each time two nodes with minimum probabilities are added to a new node as its children, the sum of their probabilities becomes the probability of the new node. This process is repeated until all symbol nodes are in the tree.

The following 5 steps show how Huffman's algorithm is applied to our example source given in Tables 2.1 to 2.3.

1. Label each of the eight nodes with its symbol probability as shown in Figure 2.1.

2. Merge the nodes labelled by the two smallest probabilities, in this case 0.02 and 0.03, by making them be the children of a new node.

3. Label two links as 0 and 1 arbitrarily, and set the new node's probability as $0.02 + 0.03 = 0.05$. Now we have seven nodes to merge, namely the six original nodes that are not yet used and the ancestor node just created.

4. Find the two smallest probabilities in the seven nodes, in this case 0.05 and 0.05. Merge them and label the new node and links as in Steps 2 and 3.

5. Continue iteratively, at each step merging two nodes labelled by the smallest probabilities among those that so far have no ancestor. Stop the process once there is only one node without ancestor; let it be the root of the tree.

In Figure 2.1 there are two Huffman codes (other possible Huffman codes exist). The difference results from different tie-breaking methods. Although their codeword length sets are different, the mean codeword lengths are identical:

$$\overline{l}_{H_a} = (0.02+0.03)\times 6+0.05\times 5+0.05\times 4+(0.15+0.15+0.20)\times 3+0.35\times 1 = 2.6$$

$$\overline{l}_{H_b} = (0.02+0.03+0.05+0.05)\times 4+(0.15+0.15)\times 3+(0.20+0.35)\times 2 = 2.6$$

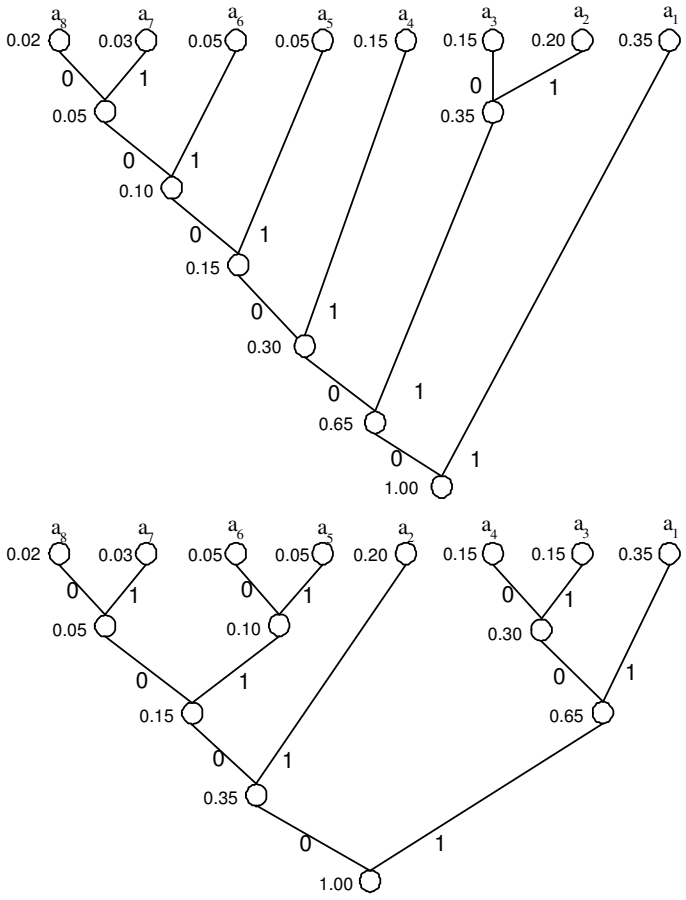

Figure 2.1: Two Huffman Trees for Source of Example

This, plainly, is a requirement for Huffman's algorithm to be optimum. The Huffman codes shorten $\overline{l}$ 10.3% over the previous UD code C and only exceed entropy by 3.5%.

Huffman algorithm can be extended to create non-binary codes. For instance, to create ternary codes, we change the binary Huffman tree to a ternary tree where every link is labelled with a ternary symbol. At each iteration, we merge three smallest probabilities instead of two. In this way, Huffman algorithm can be easily extended to

generate D-ary Huffman codes which also achieve maximum proximity to the entropy calculated by base D.

All the codes we presented up to now are *instantaneous* codes that can code a source symbol instantaneously upon its arrival, and each symbol is treated individually. This is not necessary. We often (not always) can achieve a lower redundancy by treating symbols as pairs and then constructing a Huffman code for these pairs. If the source alphabet size is 8, there are $8^2 = 64$ distinct pairs. Assuming i.i.d., probability of pair $a_i a_j$ over any possible pair combinations in this source always equals $p_i \times p_j$. Applying Huffman algorithm to these pairs, we get a Huffman code for pairs where a codeword represents two contiguous symbols instead of one. If we treat the source in units of n symbols, the code is called *n-tuple* code. In the limit as $n \to \infty$, the average codeword length of Huffman codes approaches entropy H [5]. However, we pay a price both in complexity and in delay for this approach. There are more and longer codewords to store and to look up while coding and decoding. Moreover we lose the instantaneousness property : for an n-tuple Huffman code, a symbol's codeword can not be determined until all symbols in the n-tuple have been received.

Huffman code is applied in various applications ranging from facsimile [8] to high-definition television [1]. We also use Huffman code as the compression engine in XComp. Its advantages are as follows:

1. It achieves the lowest possible redundancy among all UD codes by a very simple algorithm. The process of coding and decoding is also faster than universal compression techniques.

2. For instantaneous Huffman code, it does not need to store any temporary source data or coded data in memory.

3. It has a set of fixed codewords, which allows exact match search in the coded data. The search key can be coded by the same Huffman code, then be searched in the coded data without decoding.

However, Huffman code also has two major problems. First, it requires a priori knowledge of source statistics. To get this information, applications using Huffman code usually go through the source data once before coding. That is why the compression techniques requiring source statistics are called *two-pass compression algorithms*. Second, Huffman code assumes that the source data model is i.i.d., but the actual world is not so simple — most data are not memoryless. For instance, "the" in any English document or "XML" in this thesis have higher occurrence probabilities than any other combinations of these three letters. Furthermore statistics may vary in one source. Huffman code can not exploit this property to improve compression, because it uses an *average* statistics for the whole source.

2.2.2 Universal coding

The term "universal" was first used by A. Kolmogorov [11] to refer to the compression algorithms that do not depend on a priori knowledge of the source distribution. In universal coding, the encoder can exploit the fact that it reads the source output and thus can "learn" the statistics from it and adapt to it. The natural idea is to estimate the "future" symbol probabilities based the knowledge of source data that was read. Initially, we assume a distribution for the source, and code it by Huffman code. When the encoder reads enough data, it modifies the probabilities according to statistics of the data source consumed. As well, the codewords are also modified according to the change of these probabilities. By doing so the encoder can adapt to the statistics of

the source. Actually that is precisely what the adaptive Huffman code [2, 4, 10, 19] scheme does.

The most widely used universal coding method is the algorithm introduced by A. Lempel and J. Ziv in [12, 30, 31]. Unlike the algorithms we mentioned so far in this chapter, Lempel-Ziv (LZ) does not map fixed-length symbols to variable-length codewords. It uses the idea of *parsing* to segment the source into a sequence of symbol strings. The principle of LZ parsing is that the next phrase is the shortest phrase never seen before.

There are many variants of Lempel-Ziv algorithms, and they don't use the same parsing method. The primary versions are LZ77 [30] and LZ78 [31] and LZW [20]. The first two are named from the years when they were first proposed in literature, while the last one is a major variant of LZ78 developed by T. A. Welch. In this chapter, we only focus on the first two algorithms.

Suppose that we have a source sequence 000101100000101001001000010011. LZ77 parses it into a sequence of phrases 0, 00, 1, 011, 0000, 01010, 0100, 10001, 0011, ... In this sequence, each phrase is the longest phrase that has appeared in the data source before it concatenates one more symbol. For instance, when the first five phrases have been parsed, we have consumed 00010110000. Then in the immediate data sequence, 0101 is the longest substring of 00010110000. Thus the sixth phrase is 01010, which is 0101 plus its next symbol 0.

LZ78 parses the same sequence as 0, 00, 1, 01, 10, 000, 010, 100, 1001, 0001, 001, 1..., ... Unlike LZ77 that looks for the longest phrase in the window of all parsed data, LZ78 searches it in the domain of all parsed phrases. For example, when the first five phrases have been parsed, we match these phrases in the immediate data sequence.

Phrases 0 and 00 match the head of the rest and 00 is the longer one between them. Adding the next symbol 0 immediately to the end of this maximal match, we get the sixth phrase 000.

Next, we look at how to code these phrases. LZ77 uses three items to describe the kth phrase. We use W_{k-1} to donate the data window of previously parsed data sequence before the kth phrase. The three items are:

- I_k the index of the beginning of the kth phrase in W_{k-1},
- L_k the length of the phrase,
- S_k the terminal symbol of the phrase.

An encoder uses these items to tell the decoder how to reconstruct the original data source.

Compared with LZ77, LZ78 parses most data sequences into more phrases (in other words shorter phrases), e.g. 12 vs. 9 in our example.[2] However, it only requires the id of the appropriate phrase that has appeared earlier and the terminal symbol to describe an LZ78 phrase. So LZ78 uses less bits for a phrase than LZ77.

The phrases in LZ78 are usually stored in a tree structure as depicted in Figure 2.2. As in Huffman tree, each link is labelled either 0 or 1. Each node except the root in the tree represents a phrase; the path from root to a node comprises the binary value of the phrase. While parsing, the tree grows by one node at a time. Initially, the tree only has the root node. Since the first phrase is 0, node 1 is added as the left child of root. The second phrase is the concatenation of the first phrase and 0, so node 2 is added to the left branch of node 1. When a new phrase has been delimited,

[2]The twelfth phrase is not complete in LZ78 yet.

a new node will be added as a child of an existing node. The tree contains all the information needed to restore the original data. Each phrase can be described by specifying its immediate ancestor and whether it is the left or the right descendant of that.

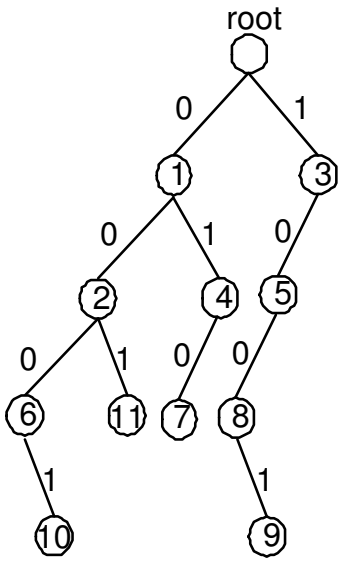

Figure 2.2: LZ78 Tree

2.3 XML Compression

In this section, we review some XML specific compressors, including XMill [13], XGrind [18] and Millau [6].

2.3.1 XMill

The XMill compressor [13] is designated to minimize the size of the equivalent XML documents. XMill incorporates some novel ideas of XML-specific compression, which

are also followed by other XML compressors. The most important ones are *separating structure from data* and *grouping items with related meaning*.

In the XMill model, each XML document is composed of three kinds of tokens: tags, attributes and data items. These tokens are organized as a tree structure, with nodes labelled with element type names or attribute names. Each data item is always associated with a node in this tree. XMill extracts the tree structure and data items from XML documents and compresses them separately.

The extracted data items are re-organized to group the ones with related meaning into one container. XMill uses *container expressions* to group data items on semantics. The container expression is a regular expression derived from XPath [23]. Users may use it to indicate absolute or relative path to a node. All data items of nodes with the same container expression are placed in one container and compressed together. The motivation for such semantic grouping is that data belonging to the same group usually have similar characteristics and therefore can be compressed better than data sorted only in syntax order.

Based on the above principle, XMill works in following manner. First, the XML document is sent to an XML parser. The parsed structure components, tags and attributes, are stored in a special container, and data items are forwarded to the container expression processor as strings. The container expression processor groups these data items according to the pre-defined containers expressions. A Container expression allows defining the data type of its data items. The data type can be atomic (integer and boolean), or complex as combination of atomic types (date or IP address). If a data type is designated, data items can be stored as in binary format rather than string in the container, e.g., using 4 bytes to represent an IP

address instead of a string like `206.102.2.1`. Such processing before compression helps to improve the compression ratio. After parsing and grouping, the contents of the containers are compressed individually by zlib [3] and the results are concatenated into a single document.

The performance study over a wide variety of XML documents demonstrated that XMill consistently provides improved compression ratios as compared to zlib, since zlib treats the entire document as a continuous stream of bytes and does not associate any semantics with the contents. While the compression ratio is improved by 40%, XMill only uses 20% more time than zlib. We compared XMill with XComp in our experimental evaluation. The results are given in Chapter 4.

XMill has a number of limitations. First, it is not designed to work with a query processor. It is designed to save storage space and network bandwidth, and therefore, only focuses on reducing document size rather than speeding up query processing by compressing XML data. Second, XMill's performance is not very good for small documents. When file size is smaller than 20KB, it loses its advantage over general purpose compressors. Thirdly, it does not comply with XML specification well. It only supports ASCII documents, rather than Unicode [17] that is the default character set of XML. It even does not accept apostrophe as an attribute value delimiter.

2.3.2 Millau

The Millau [6] coding format is an extension of WAP (Wireless Application Protocol) Binary XML (WBXML) [22] format. The WBXML defines a binary representation of XML data to reduce the transmission size of XML documents over wireless communication networks. In WBXML model, the XML structure is encoded as a sequence of

tokens, each of which occupies one byte. There are two categories of tokens: global tokens and application tokens. Global tokens are context free and assigned to indicate the position of data items and special markups. Application tokens have context-dependent meaning and are split into three overlapping *code spaces*: tag code space, attribute name code space, and attribute value code space. Tag tokens are used to indicate the beginning and end of elements, while attribute name tokens and attribute value tokens are used to indicate attributes. The sequence of these tokens represents the structure of the original XML document.

Millau also follows the approach of separating structure from data. It uses two streams to send XML data. One stream is the token sequence while the other stream is the data stream. Millau does not group hierarchical data items but sends them in the order they appear in the XML document. The token stream is not compressed so that a receiver can scan it directly. The data item stream can be compressed by general purpose compressor or left uncompressed.

Besides the compression algorithm, Millau also implements DOM [28] and SAX [14] parsers for Millau streams of XML documents. DOM and SAX are two API specifications for XML parsers. SAX parsers process the XML document sequentially while DOM parsers typically load the entire document into memory. The Millau SAX parser and DOM parser read input as encoded Millau structure stream and data stream.

The experimental evaluation shows that Millau outperforms zlib: the total size of structure stream and compressed data stream is smaller than the size of file compressed by zlib. However, because Millau does not re-organize the data content, which is just compressed plainly by zlib, the entire improvement of compression ratio is not

significant.

2.3.3 XGrind

Tolani and Haritsa proposed a query-friendly XML compressor, XGrind, that supports queries in the compressed domain [18]. XMill and Millau both use zlib, which is based on LZ77, as their compression engine. For the adaptive compression algorithms, i.e. LZ77, it is impossible to search keywords in the compressed data directly because the codes keep changing. Given the goal of efficiently querying compressed documents without decompression, only a *context-free* compression scheme is permitted here. Context-free compression code encodes data independent of their position in the document. This feature allows searching without decompression; the keywords to be searched can be coded by the same code and then looked up in the compressed data. The authors implemented (non-adaptive) Huffman code in XGrind. As discussed in Section 2.2.1, Huffman code is one of the context-free compression codes. To provide statistical information for Huffman code, a two-pass process is used in XGrind: the first to collect statistics and the second to do the actual coding.

XGrind has another novel feature to enable direct querying. Although it distinguishes structure from data, XGrind keeps the document structure. It uses different schemes to encode structure and content data. The structure is encoded in the similar way as in XMill. Each element tag and attribute name is assigned a unique element-ID and attribute-ID, and the data items are also grouped on the semantics as in XMill. Each data group has its own distribution table and is encoded individually. However, the data in one group is not encoded continually, but on the granularity of individual data item. Moreover, they are not stored together in the output, but still left in the their original positions. The output of the compressed XML data

remains the structure of original XML document. In fact, the compressed document can be viewed as the original one with tags and content values replaced by their corresponding encodings. That is why we say XGrind follows the principle of separating structure from data while maintaining document structure at the same time.

With the above scheme, queries can be executed over the compressed document without decompressing it. *Exact-match* (the search key is a specific data value) and *prefix-match* (the search key is a prefix of a data value) queries can be *completely* carried out directly on the compressed document, while *range* (the search key covers a range of data values) and *partial-match* (the search key is a substring of the data values) queries require *on-the-fly* decompression of only the element/attribute values that are part of the query predicates.

Unfortunately, the compression ratio of XGrind is not as good as XMill's. It only reaches two thirds of the latter's.

Chapter 3

XComp Compression

In this chapter, we introduce our XML compressor: XComp. First we present the principles, on which XComp is built, namely separating structure from data and grouping data based on semantics. Then we discuss the architecture of XComp. At last we describe how it work by giving an compression example in detail.

3.1 XComp Compression Model

In this section we present how XComp compresses an XML document. XComp follows two principles, similar to XMill and XGrind that are discussed in Chapter 2: (1) separating document structure from data, and (2) grouping content data based on semantics.

3.1.1 XML Document Structure

An XML document basically consists of elements in a nested structure, which can be easily represented as a tree or directed graph. There are other special markups, such as *DTD* and *CDATA Section*, also playing important roles in XML. The choice of the data model to explain and process XML documents depends on the needs

of different applications. There are standard XML data models developed by W3C that represent documents as trees: *XML Information Set* [26], *XPath 1.0 data model* [23], *DOM* [28], *XQuery 1.0 and XPath 2.0 data model* [27]. Underneath surface similarities in modeling documents as trees, they differ on the details to cater different needs [15]. Instead of using one of those, we have chosen to use a simplified model for the following reason. In XML compression, we need to preserve all the semantic information so that all markups and their content are important to the compressor. However, we do not care about the actual content or meaning of the special markups. Therefore XComp uses a simplified model to process special markups — most are not parsed but stored as a whole. The details of our model are explained in the remainder of this section.

In XComp, tags, attributes and data items are the three kinds of basic component. We have introduced tags and data items in Section 1.1. Another special tag is *empty-tag*. An element may not contain any nested elements and data items, e.g. `<number></number>`. Such an element has an equivalent expression as `<number />`, which we call the *empty-tag*.

Attributes always appear in name-value pairs connected by "`=`", i.e. `ID="IS9095"` in Example 1.1, where `ID` is the attribute name and "*IS9095*" is the attribute value. Attribute can only exist in start-tags or empty-tags which may have more that one attribute. In XComp the attribute value is also treated as a data item of an attribute.

In addition to elements and attributes, there are special markups: *XML Declaration*, *Document Type Declaration* (*DTD*), *Processing Instruction (PI)*, *Comment*, and *CDATA Section*. XML Declaration and DTD appear at most once and only at the beginning of an XML document. PI, Comment and CDATA may occur anywhere

data contents may occur. The formal definitions can be found in [24]. Since these markups usually occupy a small part of XML documents, we do not pay too much attention to their semantics. XComp directly extracts the contents of these markups and stores them in respective *containers*, each of which is memory space containing related data that will be compressed together. For example, we define PI as

```
PI ::= '<?' %string without ?% '?>'
```

So XComp treats the following PI as the string between the delimiters `<?` and `?>`.

```
<?xml-stylesheet href="mystyle.css" type="text/css" ?>
```

XComp does not parse the string "`xml-stylesheet href="mystyle.css" type= "text/css" `", but stores it as a whole data item in the PI container. Moreover, for some special markups that have similar structure, we use a unified definition. For example, XML Declaration begins with `<?xml` and ends with `?>`. We also treat it as a special case of PI; the string "*xml*" after `<?` is taken as part of the data item string.

The XML document definition used in XComp is given in Appendix A. It simplifies the definitions of these special markups, but keeps the accurate definitions of element and attribute as in [24].

An XML document has a hierarchical (tree) structure which is comprised of element and attribute nodes. One element may contain several attributes and/or several child elements. While an element may have several children, an attribute must be a leaf node. In Example 1.1, element `Person` has one attribute, `ID`, and three child elements: `FirstName`, `LastName` and `Address`. Every element except `Person` and every attribute have a data item associated with it. The tree structure of this file is depicted in Figure 3.1.

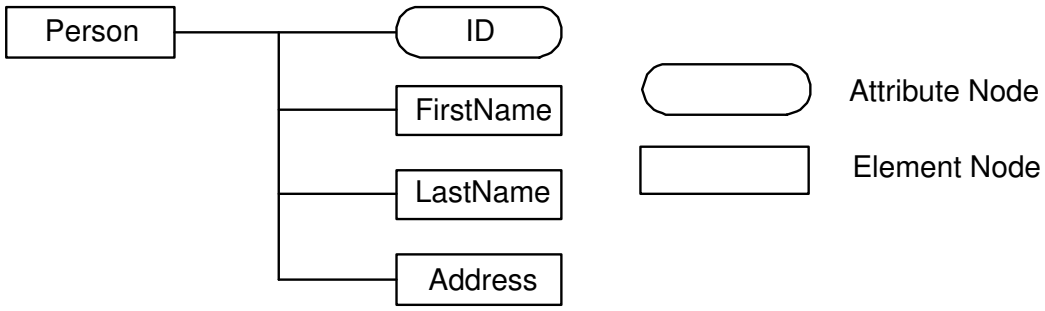

Figure 3.1: The XML Document Tree

3.1.2 Separating Structure From Data

The structure refers to the document structure that is composed of the markups, and data refers to the data items of these markups. In Example 1.1, "*Robertson*" is the data item of element `FirstName`, and "*IS9095*" is the data item of attribute `ID`. For special markups, the string values between their delimiters are their data items. Separating structure from data permits us to use different schemes to process them to improve compression. In XComp, we extract the document structure from the data, and use a sequence of integers to record the markups and data items' positions. The data are re-organized to compress related data together. This is discussed further in the next section. The structure is encoded as a concise representation of a hierarchy.

XComp encodes the structure by assigning non-negative integers to tags, attributes and data items (with 1, 2 or 4 bytes, see Section 3.4). Each distinct tag or attribute is assigned a positive sequence id. The sequence ids begin at 10, since the integers before 10 are reserved for special markups and other notations (see Table 3.1). In Example 1.1, we have $id_{Person} = 10$, $id_{ID} = 11$, $id_{FirstName} = 12$, $id_{LastName} = 13$ and $id_{Address} = 14$. In addition, code 0 is used to indicate the end-tags and 1 is used

to indicate the positions of data items. The encoded structure of the example XML file is 10, 11, 12, 1, 0, 13, 1, 0, 14, 1, 0, 0. Note that there is no 1 directly after 11, because 11 represents an attribute that can only have one data item. Thus the code 1 for attribute values can be saved.

Here we only record the positions of data items, whose values will be grouped and stored in other places. In order to segment them so that they can be laid continuously, we use a set of integers to indicate their lengths. In Example 1.1, the lengths are 6, 5, 9, 19 for data items "*IS9095*", "*Wally*", "*Robertson*", "*200 University Ave.*". In this case, we use $12 + 4 = 16$ bytes to represent the document structure and data lengths. There is another scheme where we replace 1's with the length values in 10, 11, 12, 1, 0, 13, 1, 0, 14, 1, 0, 0 so that we can save 3 bytes.[1] We do not use this scheme because the former one works with LZ algorithm better. In practice, most XML documents are regular; repeated structure usually appears in XML documents. Consider an XML document containing a large collection of personal information as the record in Example 1.1. If all `Person` elements have the same structure, the whole structure representation will consist of a large number of repeated sequences such as the following:

```
10, 11, 12, 1, 0, 13, 1, 0, 14, 1, 0, 0,
10, 11, 12, 1, 0, 13, 1, 0, 14, 1, 0, 0, ...
```

¿From Section 2.2.2, we know that LZ77, the core algorithm of zlib library, is good at compressing such repeated data. For example, an experimental document having 512 `Person` elements has a structure sequence with length $12 \times 512 = 6144$ bytes, which zlib compresses it to only 39 bytes! Good compression is still obtained even when the structure has small variations, e.g. some persons may have a middle

[1] Note we have three 1's and four length values.

name. However if the length values of data items are inserted into the structure representation, the repetition characteristic is corrupted. Therefore XComp uses two containers: the structure container and the data length container for the structure sequence and the length sequence, respectively.

Up to this point, we ignored all whitespace characters in our encoding, since all of these characters that are not in data items may be removed without influencing the information contained in the XML documents. Optionally, XComp provides another extended encoding scheme that preserves whitespace characters. This encoding scheme guarantees the recovery of the *exact* version of the original document. 4 is used to indicate the positions of whitespace characters and their lengths are stored in the length sequence as data item lengths as well. In order to express the whitespace within the tags, we also add 2 to indicate '=' and 3 to indicate '>' of a start-tag. (We do not assign any code for '<', because '<' always appears at the beginning of start-tags or end-tags and no whitespace is allowed between it and tag name.) All of the codes defined in XComp structure representation are showed in Table 3.1.

Code	Meaning
0	End tag
1	Data item position
2	'=' position (used only when preserve whitespace)
3	'>' position (used only when preserve whitespace)
4	Whitespace position (used only when preserve whitespace)
5	The position of any characters before XML Declaration
6	PI
7	DTD
8	Comment
9	CDATA section

Table 3.1: XComp Structure Representation Codes

To show how to encode whitespace characters, let us consider the case in Example 3.1:

```
<A  B = "IS9095"  >
    <C></C>
    abcd
    <C/>
    xyz
</A>
```

Example 3.1: An XML Segment

In this example, element `A` has two empty child elements, one of which has a start-tag and an end-tag while the other has an empty-tag. It also has two data items: *abcd* and *xyz*. Note that there are whitespaces within and outside of tags. Its structure representation is 10, 4, 11, 4, 2, 4, 1, 4, 3, 4, 12, 3, 0, 4, 1, 4, 12, 0, 4, 1, 4, 0. Each 4 represents a piece of whitespace. For example, the first 4 represents the whitespace between `<A` and `B` while the last 4 represents the one between `xyz` and `</A>` (the return character). We also need to record their length values. The length sequence is 2, 1, 1, 6, 2, 5, 5, 4, 5, 5, 3, 1, where each integer represents a whitespace length or data item length. To make it clear, we show these two sequences together: 10, $4_{(2)}$, 11, $4_{(1)}$, 2, $4_{(1)}$, $1_{(6)}$, $4_{(2)}$, 3, $4_{(5)}$, 12, 3, 0, $4_{(5)}$, $1_{(4)}$, $4_{(5)}$, 12, 0, $4_{(5)}$, $1_{(3)}$, $4_{(1)}$, 0, where every 1 and 4 has a subscript indicting its length. This representation is different from the one that does not preserve whitespace in four aspects:

1. It uses 4's to indicate the positions of whitespace.

2. It uses 2's to indicate the positions of '`=`', so that it is possible to partition the whitespace before and after '`=`'.

3. It uses 1's to indicate not only the positions of data items, but also the ones of attribute values, while in no whitespace representation attribute values' positions are implied by attribute ids. This allows partitioning the whitespace before and after the attribute value.

4. It uses 3's to indicate the positions of '>' of a start-tag, so that it is possible to partition the whitespace within and between tags, e.g. the fourth and the fifth whitespaces in Example 3.1. Moreover, now we can distinguish empty-tags from pairs of start-tag and end-tag that have no content between them. A pair of start-tag and end-tag always has this representation: *id*, ..., 3 (indicating the >), ..., 0 (indicating the end-tag) while an empty-tag's representation is *id*, ..., 0. For example, representation of <C></C> is 12, 3, 0, while the presentation of <C/> is 12, 0.

3.1.3 Grouping Data

Grouping data based on the semantics is the second principle we follow in XComp. We exploit self-descriptiveness of XML documents to group data content. It is observed in common XML documents that tag and attribute names usually indicate the type of data they contain. For example, <telephone> elements usually contain telephone number and <postcode> elements typically contain postal codes. The data items with the same tag or attribute name usually have a semantic relationship between them. We expect them to have similar characteristics and they can be compressed better if they are put together. In XMill, they share one container. For example, all <name> data items form one container, while all <year> data items form another container.

However, sometimes a tag or attribute name may have different meanings in different contexts. In Example 1.2, there are three kinds of attribute `ID`s: the invoice id, the customer id, and the product id, and two kinds of element `name`s: the customer name and the product name. The three kinds of ids may be generated by different rules so that they have different formats and domains; customer name and product name also have different semantics. So, it is better to compress them separately. In XComp, the data item grouping is based not only on the tag/attribute names, but also on their levels in the document tree and their types. Data items are put in the same container only when their tags/attributes have the same name, and are at the same level, and are of the same node type (tag or attribute). In Example 1.2, `ID` has three containers and `name` has two containers, while other tags and attributes have only one container. If one name appears as both tag and attribute name and at the same level, as `new` in the XML document of Example 3.2, we use different containers since they have different node types.

```
<p new="true">
  <new>It's a new one.</new>
</p>
```

Example 3.2: A Name Appears Both As Tag and Attribute

3.2 System Architecture

The architecture of XComp compressor is shown in Figure 3.2. It has four major modules: the *XML Parser*, the *Dictionary Manager*, the *Containers* and the *Compression Engine*. This architecture follows a parsing-compressing order: an XML document is parsed first, then its structure and data are compressed individually. The data in

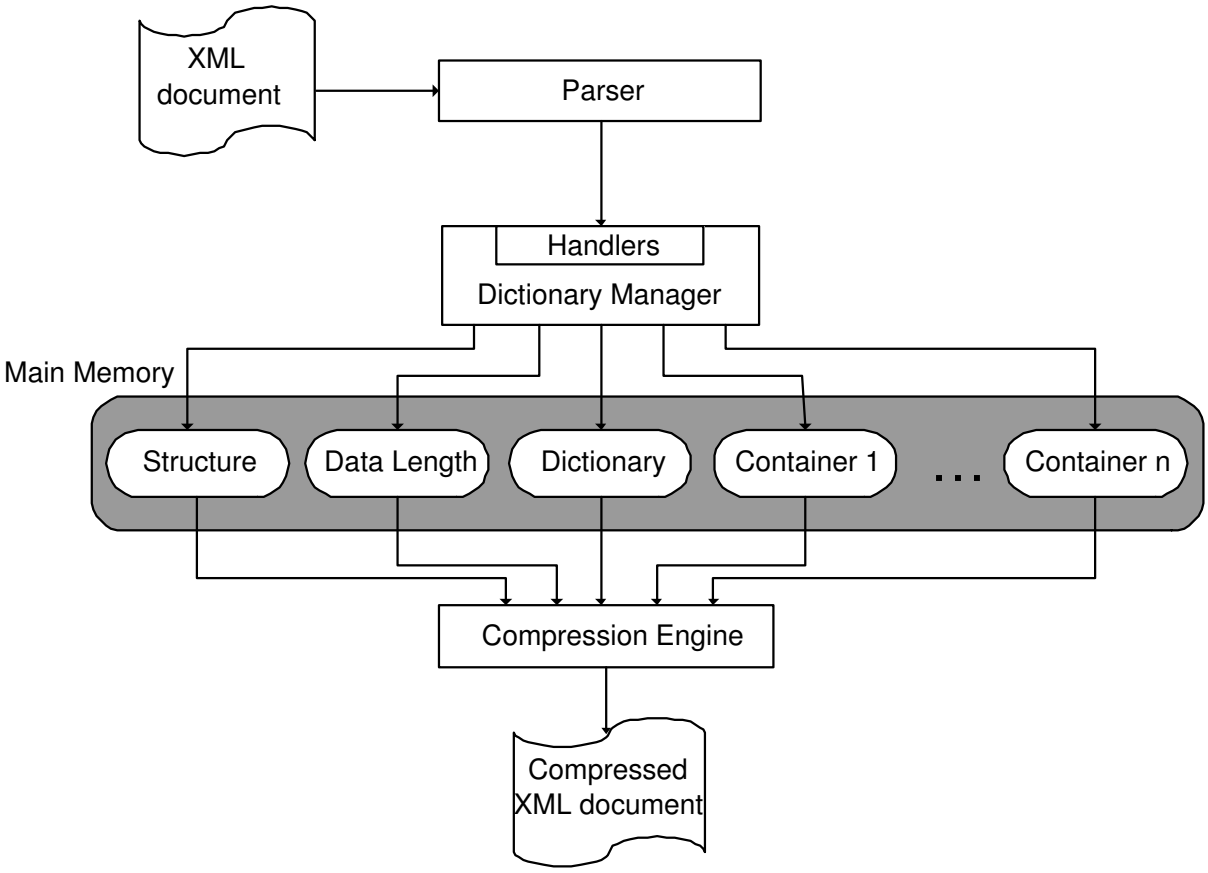

Figure 3.2: The Architecture of XComp Compressor

an XML document go through these four modules in sequence as shown in Figure 3.2. First they are sent to the parser. The parser decomposes the document into XML components like start-tags, end-tags, and data items. As soon as it parses one component, it calls a handler function of the dictionary manager to deal with it. The dictionary manager has a set of handlers, each of which is responsible for handling one kind of component. The handler interfaces are presented in Appendix B.

The structure encoding and data grouping are done by the dictionary manager. It writes structure sequence, data length sequence, and data items to their containers,

respectively. The dictionary manager also has a *dictionary table*, in which distinct tag/attribute names and the relationships between them and containers are maintained. We call these tag/attribute names *words*.

As indicated before, a container is an individual memory space that contains related data. Its size can be increased when its space is full. However, the total size of all the containers can not exceed the size of *memory window*. There are four kinds of containers:

1. **Structure Container**: The structure container holds the structure sequence. Integers are stored in binary format with 1, 2 or 4 bytes.

2. **Data Length Container**: The data length container contains the data length sequence. It stores the integers in the same way as structure container.

3. **Dictionary Container**: The dictionary container stores the tag and attribute name values. To delimit these string values, each string is preceded by an integer indicating its length. The integers are formatted as in structure and data length containers.

4. **Data Containers**: Data containers contain the related data items based on the tag/attribute name and its type and its level. Every distinct tag/attribute name has its own container, in which data items are stored contiguously as strings. Unlike dictionary container, we do not need length values of these strings since they are already stored in data length container. Besides the general data containers for element content and attribute values, XComp also has some special containers for whitespace, PI, DTD, comment, CDATA section

and PreText. The last one contains the document control characters before the text in an XML document.

When a document is all parsed or the memory window is full, all containers are sent to the compression engine. In the latter case, the parser and dictionary manager stop their work until all the data in the containers are compressed and sent to the output. There are two optional compression engines in XComp: zlib and Huffman code. Both compress containers one-by-one and stream the result as output. As discussed in Chapter 2, Huffman code compression needs a priori statistical information about the document. The collection of the information is done by the parser when the Huffman code engine option is chosen. Each container has its own alphabet frequency table since they are compressed individually.

The decompression process is the reverse of the compression process. The architecture of XComp decompressor is shown in Figure 3.3. Compared to the compressor, the decompressor has an *XML writer* instead of the XML parser and no dictionary manager. The compressed data are first decompressed and stored in the containers. The XML writer reads the structure sequence from the structure container and translates it into XML tokens. When it needs a data item value, the XML writer retrieves the string value from the corresponding data container. Furthermore the decompressed XML document is written to the output in a streaming fashion.

¿From next section, we present XComp compression by showing how each XComp module works over Example 1.2. For case of reading, we repeat the example here:

```
<?xml version="1.0" encoding="utf-8" ?>

<Invoice ID="ST87302" date="2003-03-17" paid="true" >
  <Customer ID="2376">
```

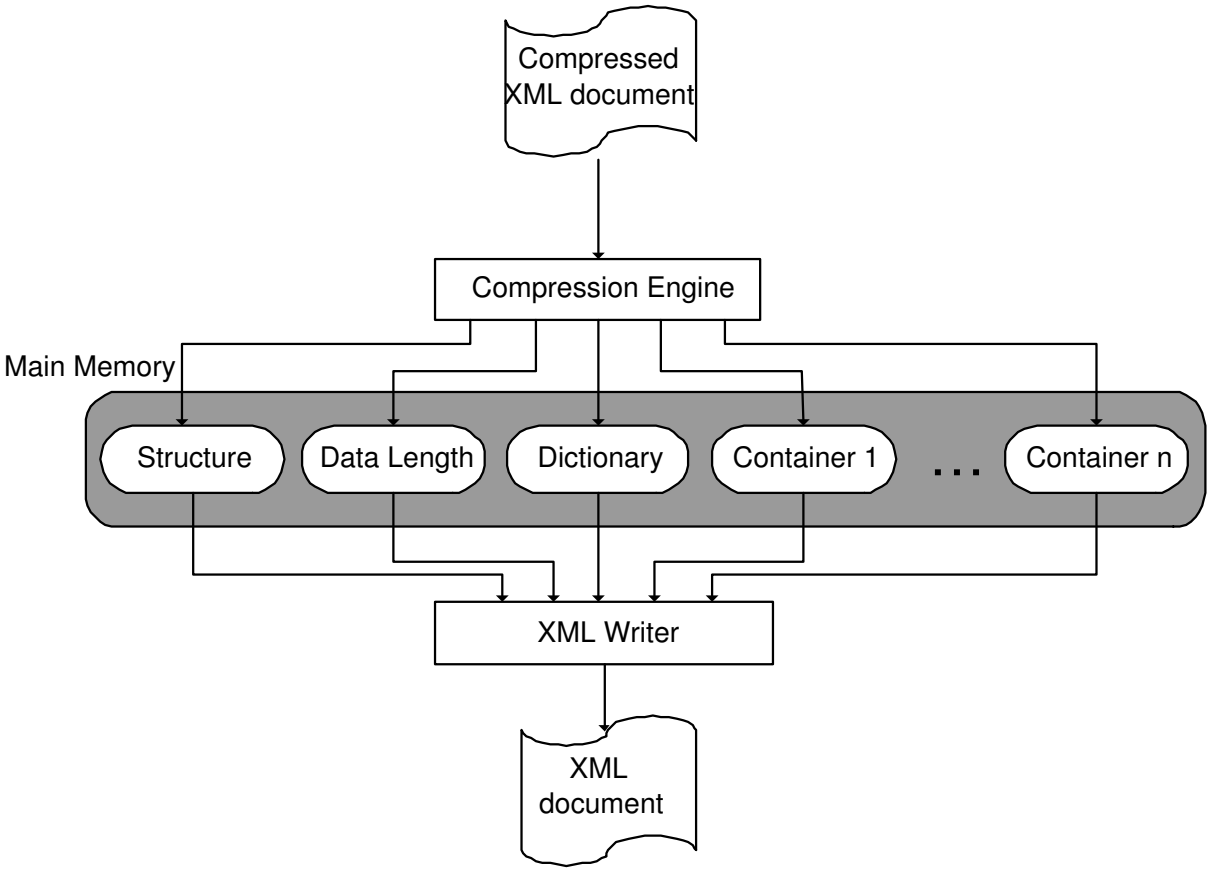

Figure 3.3: The Architecture of XComp Decompressor

```
  <Name>
    <FirstName>Wally</FirstName>
    <LastName>Robertson</LastName>
  </Name>
  <BillingAddress>
    <Street>200 University Ave.</Street>
    <City>Waterloo</City>
    <Province>ON</Province>
  </BillingAddress>
</Customer>

<ProductList>
```

```
    <Product ID="HI0941" >
      <name>MP3 Player</name>
      <Price>139.99</Price>
      <Units>2</Units>
    </Product>

    <Product ID="XP6015" >
      <name>CD Writer</name>
      <Price>79.99</Price>
      <Units>4</Units>
    </Product>
  </ProductList>
</Invoice>
```

3.3 Parser

The compression process begins by the XML parser. We use a SAX [14] parser for XComp. Unlike a DOM parser [28], a SAX parser does not load the whole document into main memory to process it. It reads through the XML document and reports some *events* depending on the tags encountered. The event-driven processing feature is what we need for XComp, which should compress the XML document in a streaming fashion. Meeting (the end of) any XML token incurs an event given by a SAX parser. The events defined in SAX include *startDocuemnt*, *endDocument*, *startTag*, *endTag*, *contentData* and so on. They occur in the order of their positions in the XML document.

A SAX parser will break down the example XML document into a series of linear events as

```
startDocument
XMLDeclaration
startTag: Invoice
```

```
Attribute:ID
AttValue: ST87302
Attribute:date
AttValue: 2003-03-17
...
endTag:   Invoice
endDocument
```

To process these events, we must implement the handlers that act on specific events. For XComp compressor, these actions include encoding XML structure and grouping data items. As SAX is aimed at general purpose XML parsers, it defines a set of comprehensive events that are not all required by XComp. For example, SAX parsers report some DTD events while parsing a DTD. XComp does not need to understand DTD and treats the content of a DTD as a string. Therefore some SAX events are not needed in XComp. To save processing time and simplify our implementation, we built a light-weight SAX parser for XComp. It only has a subset of SAX specification (see Appendix B). Furthermore, some interface definitions are sightly different from standard SAX for our specific use in XComp.

The parser works in the following manner. It reads an XML document as a stream of characters. The input data are first loaded into an internal buffer. The parser searches XML delimiters, such as "<", ">", "<?" and "?>", in this buffer to locate the positions of markups. For example, when "<!--" is read, it means the beginning of an XML comment according to the XML document definition (see Appendix A). Then the parser searches "-->", which is the end of comment, in the rest of the buffer. When "-->" is found, there is a comment event. Then the parser calls the handler function *handlerComment()* to deal with this event. It passes the information of the beginning position of the comment string and its length between "<!--" and "-->" to the handler by the function parameters (see Appendix B). In

our simplified definition of XML, most special markups are simply processed in this way. Elements are more complex because an element may have not only attribute name-value pairs and data items, but also nested elements and other special markups. The parser processes elements in a recursive way. The algorithm of parsing elements is given in Algorithm 3.1.

Algorithm 3.1: Element Parsing Algorithm

```
find the position of "<"
find the end of tag name
report a startTag event
while there is an attribute pair do
    find the end of attribute name
    issue an attribute event and call corresponding handler
    skip "="
    find a "'" or "\""
    find the end of attribute value by meeting the corresponding "'" or "\""
    issue an attVAlue event and call corresponding handler
end while
if the next non-whitespace char = "/" then
    issue an endOfEmtpyElement event and call corresponding handler
else if the next non-whitespace char = ">" then
    skip ">" {It's the end of a start tag.}
    while the next two non-whitespace chars ≠ "</" do
        {It's the beginning of a new markup.}
        if the next non-whitespace char = "<" then
            if the char after "<" = "?" then
                find "?>"
                issue a PI event and call corresponding handler
            else if the char after "<" = "!" then
                if the char after "!" = "-" then
                    find "-->"
                    issue a Comment event and call corresponding handler
                else if the char after "!" = "[" then
                    skip "CDATA["
                    find "]]>"
                    issue a CDSect event and call corresponding handler
                end if
            else
```

```
32:             recursively call this algorithm to parse a nested element
33:           end if
34:         else
35:           find the end of the data item
36:           issue contentData event and call corresponding handler
37:         end if
38:       end while
39:       skip "</"
40:       find the end of tag name
41:       issue an endElement event corresponding handler
42:   end if
```

The parser is also responsible for collecting data statistics for Huffman code compression. As mentioned in Section 3.1, each container has one frequency table, which is a 256-ary integer array. The ith integer indicates the number of characters with binary code i. Upon receiving startTag and attributeName events, the handlers must return the array associated with the container of the element or attribute to the parser. While the parser scans the data items of the element or attribute to find their ends, it counts the character occurrences at the same time. For example, in Example 1.2, "*Wally*" is the first data item of `FirstName`. Before the parser reads it, all entries of `FirstName`'s container's frequency table are 0. When "*Wally*" is scanned, value of entries 87 (ASCII code of "W"), 97 ("a"), 121 ("y") become 1 while entry 108 ("l") becomes 2. This design avoids the two-pass loading of Huffman code; the frequency counting is done while parsing.

3.4 Containers

The containers are the basic units for grouping XML data. All containers are kept in main memory. While compressing, we use a memory window to limit the memory space usage — the total size of containers can not exceed the memory window size.

When the limit is reached, the content of all containers will be compressed and sent to the output. After that, the parsing process will resume. As the number and sizes of containers needed for a document are not known beforehand, it is not possible to assign some memory blocks with given sizes to containers. In XComp, the container sizes increase with the parsed data volume. Because the cost of adjusting memory block size is high, we do not use a single memory block for one container. A container consists of a series of memory blocks whose sizes comply with memory size incremental function (Formula 3.4.1). The rules of the incremental function include the following:

1. The first memory block size is small for some containers that may have very few data. If we assign a large initial block to containers, there is likely to be some waste.

2. The block size should increase exponentially rather than in linearly. As the first block is small, we want the size to increase fast. The cost to allocate 256 1KB blocks is higher than the cost of allocating one 256KB block.

3. There must be a maximum limit for the block size, since as the total size becomes large, (i.e. 1 MB), the allocation cost again increases. This is due to the need for reorganizing its memory fragments to get a large one.

4. To avoid wasting memory space, the block size begins to decrease when the total size of containers approaches the memory window size.

The incremental function is as follows,

$$S_n = \begin{cases} \min(S_{max}, S_{n-1} * R_{increae}) & \sum\text{containers sizes}<90\% \text{ of memory window size,} \\ \max(S_{min}, S_{n-1}/R_{decrease}) & \text{otherwise.} \end{cases} \tag{3.4.1}$$

where S_i is the ith block size, S_{max} and S_{min} are the maximum and minimum block sizes, $R_{increase}$ is the block size increase ratio and $R_{decrease}$ is the block size decrease ratio.

When the memory window is full, the data in the containers are compressed. However XComp does not release the memory blocks back to the system, but collects them in a *memory store* that groups these blocks based on their sizes. In the next run of filling the containers, these blocks are reused to save memory allocation time. Therefore, the memory space is allocated only once.

The compressor and decompressor both have the containers module and manage their memory usage in the same way.

All data items are stored as strings in the containers, but for structure and data length and dictionary structure, we also need to store integers. All these integers are non-negative. We use 1, 2 or 4 bytes to encode integers in XComp. The first byte's value indicates the length of this integer. The codes between 0 and $F3$ (in hexadecimal) are reserved for one-byte integers, while the codes between $F4$ and FD are reserved for two-byte integers and FE and FF are for four-byte integers. The integer values and their code space are described in Table 3.2.

This coding can accommodate integers no larger than 33557235, which is the largest sequence id and data item length allowed in XComp. In practice, most ids and lengths use only one byte.

Value	Code Space in Hex
0 - 243	00-F3
244 -2803	F400 - FDFF
2804 - 33557235	FE000000- FF000000

Table 3.2: Integers Coding

3.5 Dictionary Manager

The dictionary manager is responsible for encoding structure and distributing data items. As indicated earlier, it contains the event handler routines and the management of the dictionary table.

3.5.1 Handler Algorithms

Event handling is given in Algorithm 3.2.

Algorithm 3.2: Handler Algorithm

```
1: initialize dictionary table
2: while not end of the input file do
3:    switch event type
4:       case: meet a whitespace event
5:          if preserve whitespace then
6:             write 4 to structure container
7:             write length of whitespace to data length container
8:             write string value of whitespace container
9:          end if
10:      case: meet a preText event
11:         write 5 to structure container
12:         write length of preText to data length container
13:         write string value of preText data container
14:      case: meet a PI event
15:         write 6 to structure container
16:         write length of PI to data length container
17:         write string value of PI data container
18:      case: meet a DTD event
```

19: write 7 to structure container
20: write length of DTD to data length container
21: write string value of DTD data container
22: **case**: meet a `Comment` event
23: write 8 to structure container
24: write length of Comment to data length container
25: write string value of Comment data container
26: **case**: meet a `CDADA Section` event
27: write 9 to structure container
28: write length of CDADA Section to data length container
29: write string value of CDADA Section container
30: **case**: meet a `startTag` event
31: call getWord() routine of dictionary manager to get the word and its data container of this tag {see Algorithm 3.3}
32: push the container into the container stack
33: write the word id to structure container
34: **if** use Huffman code **then**
35: return the frequency table of the data container of this tag
36: **end if**
37: **case**: meet an `endTag` event
38: pop the top container off the container stack
39: write 0 to structure container
40: **case**: meet an `endOfEmptyElement` event
41: pop the top container off the container stack
42: write 0 to structure container
43: **case**: meet an `attributeName` event
44: call getWord() routine of dictionary manager to get the word and its data container of this attribute
45: push the container into the container stack
46: write the word id to structure container
47: **if** use Huffman code **then**
48: return the frequency table of the data container of this tag
49: **end if**
50: **case**: meet an `attributeValue` event
51: **if** preserve whitespace **then**
52: write 1 to structure container
53: **end if**
54: write length of attribute value to the data length container
55: write the attribute value to the top container of the container stack
56: pop the top container off the container stack
57: **case**: meet a `contentData` event
58: write 1 to structure container
59: write length of the data item to the data length container

```
60:         write value of the data item to the top container of the container stack
61:       case: meet a '=' event
62:         if preserve whitespace then
63:           write 2 to structure container
64:         end if
65:       case: meet a '>' event
66:         if preserve whitespace then
67:           write 3 to structure container
68:         end if
69:     end switch
70:   end while
71:   call compress() routine to compress data in all containers
```

Let us consider how the algorithm is applied over Example 1.2. We assume that whitespace is preserved. The first event given by the parser is a PI event (recall that we treat XML Declaration as PI). Now XComp needs to record the PI in structure sequence and write its value to the PI container. Note that we already have special data containers for whitespace, PI, Comment and so on which are all initially empty. We do not have other containers since we do not know what tag/attribute we will meet later. The handler is called to write 6, the id of PI, to the structure container and write 35 to the data length container and write "`xml version="1.0" encoding="utf-8"` " to structure container (lines 14, 15, 16 of the algorithm).

Then the parser reads the whitespace between `?>` and `<Invoice`, and issues a whitespace event. The handler is called to write 4 to the structure container and 2 to the data length container and the whitespace characters to whitespace container as lines 5, 6, 7 of the algorithm.

When `<Invoice` is read, the parser issues the first `startTag` event. The handler calls getWord() to look up the tag name `Invoice` in the dictionary table. Since now there is no entry in this table, dictionary manager will construct a new one with id 10 and a new data container for `Invoice`. We will show the dictionary algorithm

later (Algorithm 3.3) and discuss it with dictionary table. The handler writes the id to structure container and push the data container to container stack. If Huffman code is used, the frequency table of this container is returned to the parser to collect statistical information of the data items of this container. The container stack keeps the data containers of all nodes on the path from root to the current node. The top element is the container of the current node. When the dictionary manager meets a start-tag, its container will be pushed onto this stack for future retrieval when the dictionary manager has its data items. Similarly, when an end-tag is met, the stack is popped since now the current node is the parent element node in the document tree. Then the compressor processes the next whitespace after `Invoice`.

By now, the containers have their contents as in Table 3.3, and there is one element in the stack as in Table 3.4. The whitespace is not shown since it holds all whitespace characters.

Container	Content
structure	6, 4, 10, 4
data length	35, 2, 1
PI	xml version=”1.0” encoding=”utf-8”
Invoice	

Table 3.3: Containers' Contents (1)

Invoice

Table 3.4: Container Stack (1)

The parser meets the attribute `ID` and issues an attribute event. It is assigned id 11, a word entry in the dictionary table, and a container. The container is pushed

onto the stack and 11 is written to the structure container. Furthermore, the frequency table is returned to the parser. '=' is read and 2 is written to the structure container. Then the parser issues an attValue event when it meets the attribute value "*ST87302*". The handler writes 1 to the structure container and 7 to the data length container and inserts the string value into the container at the stack top. Now we need to pop the stack since the current attribute pair is processed (line 55). The attribute pair contributed 11, 2, 1 to the structure sequence. Similarly, the compressor processes the rest of the whitespace and attribute pairs in this start tag until it meets '>', for which it writes 3 to the structure container. But the container stack is not popped, because we just meet the end of a start tag, not an end tag. Now we have four words, `Invoice`, `ID`, `date` and `paid`, each of which has a data container. The contents of these containers are shown in Table 3.5. The stack content is the same as in Table 3.4.

Container	Content	Container	Content
structure	6, 4, 10, 4, 11, 2, 1, 4, 12, 2, 4, 1, 13, 2, 1, 4, 3	ID	ST87302
data length	35, 2, 1, 7, 1, 10, 1, 4, 1	date	2003-03-17
PI	xml version="1.0" encoding="utf-8"	paid	true
Invoice			

Table 3.5: Containers' Contents (2)

The compressor continues to process the nested element `Customer` where we meet attribute `ID` again. The dictionary manager finds the word `ID` in its table, but does not return the existing container of `ID`. The existing container belongs to attribute `ID`

at level 2, while now we meet `ID` at level 3. The dictionary manager will also return a new container that is associated with `ID`. So we will have two containers for word `ID`, which have contents "*ST87302*" and "*2376*", respectively. Now the container stack is as shown in Table 3.6

ID (at document level 3)
Customer
Invoice

Table 3.6: Container Stack (2)

When the compressor reads the `ID` in `<Product ID="HI0941">` where the `ID` appears at level 4. Hence, a third container will be assigned to `ID`.

So far we described how the dictionary manager processes events to construct structure sequence and data containers. We do not describe the rest of the progress, since it reiterates some of the work we already described before. When the end of the XML document is reached or the memory window is full, the routine compress() is called to compress the containers one-by-one. This is done by the compression engine.

3.5.2 Dictionary Table

The dictionary manager module contains a collection of words, each of which is a distinct tag or attribute name. Therefore actually this collection is the vocabulary of the XML document. Each word is assigned a unique id. For Example 1.2, when the whole document has been processed, the structure of the dictionary looks like a 2-field table as shown in Table 3.7.

It is easy to implement this dictionary as an array. However, the compressor's working pattern requires that the dictionary data structure can easily grow and can be

10	Invoice
11	ID
12	date
13	paid
14	Customer
15	Name
16	FirstName
17	LastName
18	BillingAddress
19	Street
20	City
21	Province
22	ProductList
23	Product
24	Price
25	Units

Table 3.7: An Example of Dictionary Table

looked up very fast. While parsing, the compressor does not know the exact number of words. The words should be inserted to the dictionary one-by-one. Moreover, compared with insertion, search operation is more frequent; each time the parser meets a tag or attribute, its name should be searched in the dictionary. For example, parsing a 10MB XML document may need to insert tens of words and search for more than hundreds of thousands times. For these requirements, we use a hash table for the dictionary. The hash table has 256 entries, each of which points to a link of words. A hash function maps the word's string value to a one-byte integer. We use a hash table, because the cost of comparing string values is expensive. Grouping them by their hash values will narrow the search range. In a link, all words are sorted by their string values. We expect one entry in the hash table to contain no more than 10 words. Figure 3.4 displays a part dictionary data structure of Example 1.2.

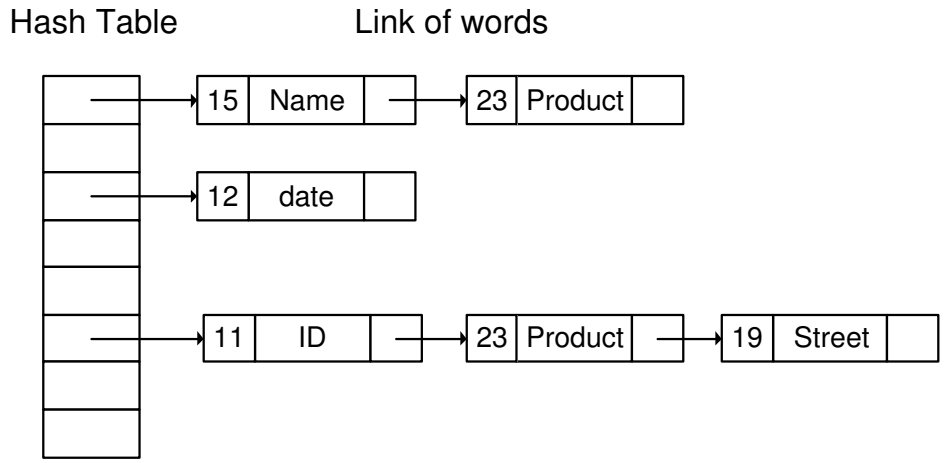

Figure 3.4: A Dictionary Data Structure

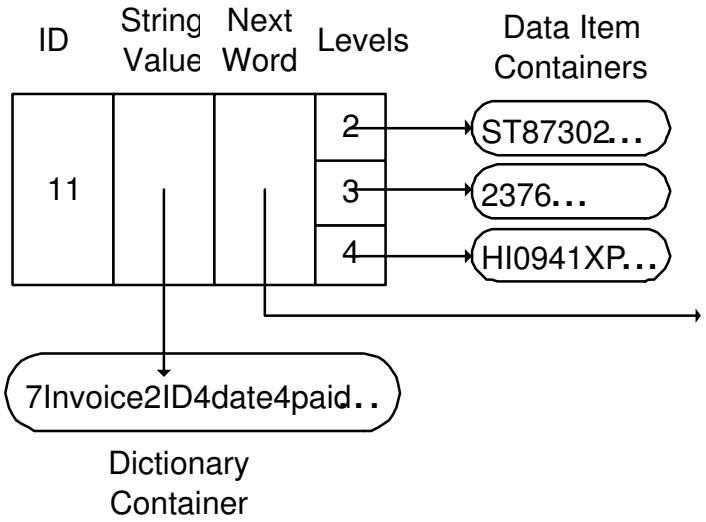

Figure 3.5: A Word Data Structure

Figure 3.5 shows the internal structure of the node `ID` in Figure 3.4. For simplicity, we directly place the word's value in the node. Actually, the string values of words are not stored in words themselves, but in the dictionary container. A word only contains a pointer that points to the string value in that container. The dictionary container is also shown in Figure 3.5. Each string is proceeded by an integer that indicates the length of this string. As indicated earlier, one word might be associated with several data containers, since it appeared in different levels or as different node types. The word structure includes two arrays: one is the array of level numbers and the other is the array of pointers pointing to the data item containers. The level number is also used to indicate the node type: the first bit is used for that purpose and the rest is the actual value of level.

Algorithm 3.3 shows the algorithm of getWord().

Algorithm 3.3: getWord() Algorithm

```
1: compute the hash value for the string
2: get the entry of the hash value in the hash table
3: while the current word string value is greater than this string value do
4:    move to the next element in the link
5: end while
6: if the current word string value equals this string value then
7:    {We found the word.}
8:    if this string's level and node type does not accord with anyone in the level array
      then
9:       construct a new container {Need a new container.}
10:      add one element to the level array
11:   end if
12: else
13:   {No such word.}
14:   construct a new word
15:   construct a new container
16:   add one element to the level array
17:   write the string length to word container
18:   write the string value to word container
```

19: **end if**
20: return the word and the corresponding container

3.6 Compression Engine

Two optional compression Engines, zlib and Huffman code, are used in XComp. The engines first compresses the structure and data length and dictionary container. Then the data containers are compressed in the order of the sequence ids of tags and attributes they belong to. The decompressor decompresses them in the same order. While the zlib is a ready-made tool, the Huffman code engine was written by ourselves. For each container, the engine constructs a Huffman tree for it. This Huffman tree is also written to the output to let the decompressor know the codes.

Chapter 4

Experimental Evaluation

In this chapter, we present the results of our experiments that compare XComp with XMill, XGrind and zlib. First we introduce the experimental framework, including the setup and the methodology. Then we discuss the experiments that we conducted to determine the optimal parameters for XComp. The comparison results with others are presented finally.

4.1 Experimental Framework

4.1.1 Methodology

As indicated in Chapter 1, we use two metrics in our evaluation of the efficiency of XML compression tools: the compression ratio and the compression time.

- **Compression Ratio**: We express the compression ratio as the ratio of the size of the compressed document to the original document. For example, if a 10MB file can be compressed to 2.5MB, the compression ratio is 25%. Lower compression ratios are, obviously, better.

- **Compression Time**: Compression time represents the elapsed time during compression, expressed as the time from the start of program execution on a

document until all the data are written to disk. We do not separate CPU and I/O times. For this reason, the compression ratio can influence compression time. A lower compression ratio always means fewer output, thus lower compression time.

Of these two metrics, the compression ratio is the easier one to measure. For the same XML document, an algorithm always produces the same compression ratio value if we don't change its parameters. It won't be influenced by the status of the computer on which the programs run. But the compression time is not stable. The major factor that causes variations in compression time is the effect of operating system buffer, since the amount of the document that is in the buffer is in reversely proportional to the time needed to load the document. The first 2-3 runs usually have high variance. So in our experiments each experiment was run 8 times. Ignoring the 2 slowest ones and the fastest one, we use the average time of the remaining 5 runs as the result value.

4.1.2 Classes of Experiments

There are two classes of experiments that we conducted to evaluate XComp. One is compressing documents locally, where the compression programs read the XML documents from hard disk, compress them and write the compressed document back to hard disk. In these experiments, compression ratio and compression time were both measured.

The second class of experiments is the Internet transmission experiments. We wrote a client and a server program to do the data transmission. The transmission

programs run between the compressors and the decompressors, which run on geographically distributed computers. The client establishes a TCP connection with the server, receives compressed data from the compressor and sends them to the server, while the server sends data to the decompressor. The aim of these experiments is to see how the compression tools can speed up XML data transmission.

4.1.3 Platform

The local compression experiments were all conducted on a P4, 1.5GHz machine running Windows 2000 Professional, with 256MB memory. While performing Internet transmission experiments, in addition to this machine at the University of Waterloo, another machine located at the University of Alberta running Unix was also used. We transferred data over TCP protocol for which we measured an average transfer rate of 408 KB/s.

4.1.4 Data Source

We evaluate XComp on XBench [29], which is a family of benchmarks that captures different XML application characteristics. The databases it generates have data-centric and text-centric models that cater for the needs of different applications. Data-centric (DC) model contains data that are not originally stored in XML format, i.e. e-commerce catalog data and transactional data. Text-centric (TC) model's data (e.g. news article archives) are text data which are more likely stored as XML. XBench also categorizes its database models as single document (SD) and multiple document (MD). SD model has only one complex XML document while MD model is composed of a set of XML documents. The two kinds of classifications are combined to generate four cases: TC/SD, DC/SD, TC/MD, DC/MD. As XComp works on a

single XML document, we only use TC/SD and DC/SD cases in our experiments.

XBench can generate databases with 4 kinds of database sizes: small (10MB), normal (100MB) and large (1GB) and huge (10GB). We use the first 3 sizes in our experiments for compressing; the last one is too time consuming (i.e. 45 minutes for compression).

XBench provides XML documents with different application characteristics and different sizes that are suitable for us to comprehensively evaluate XComp's performance on large documents. But we also need to test its performance over small documents, which are from several KB to 1MB. Therefore we also used several individual XML documents used in XBench to generate XML databases. All the documents used in the experiments are listed in Table 4.1.

Name	Size	Comment
provinces	2096Byte	contains names of states of US and provinces of Canada
emails	4546Byte	contains Web URLs
countriesDC	9555Byte	contains some information about countries
lnames	194.612KB	contains English names
words	347.479KB	contains English words
titles	1426.053KB	contains business titles
DC_10	10.298MB	10MB Data-centric database document
DC_100	103.469MB	100MB Data-centric database document
DC_1000	1033.689MB	1GB Data-centric database document
TC_10	10.467MB	10MB Text-centric database document
TC_100	103.097MB	100MB Text-centric database document
TC_1000	1038.891MB	1GB Text-centric database document

Table 4.1: XML Documents Used in the Experiments

4.1.5 Comparative Compression Tools

We compared XComp with three other compression tools: XMill, XGrind and Zlib, which were introduced in Chapter 2. XMill and XGrind are both XML-specific compression tools which are written in C/C++. Because XGrind's source code is only for Linux, we slightly modified it to enable it to run on Unix and Windows. For Zlib, we used a small program, called minigzip, which comes with this library. Minigzip is used to illustrate how to use the library. Therefore it only contains the core compression engine of Zlib without other unnecessary functions, i.e. complex user interface, which will make our comparison less meaningful. All these programs were compiled by VC++ 6.0 on Windows and by gcc 2.95.2 on Unix.

4.2 Parameter Tuning Experiments

Parameter tuning is the first step in our evaluation. The aim of these experiments is to find the optimal parameters for XComp. We only used Zlib as the compression engine in these experiments. There are four parameters, all of which concern memory management.

- **Memory Window Size (MWS)**: The parsed data are contained in a memory window before being compressed. The entire window is compressed when it is full. Therefore a bigger window means more data can be compressed at one time, possibly leading to a better compression ratio. However memory space is not unlimited and a big window may also cause XComp to run slowly. We need to find a balance between these influences to achieve a good compression ratio as well as short compression time.

- **Minimal Block Size (MBS)**: As described in Chapter 3, when a container is constructed, it owns an initial block whose size is set to MBS. When the container requires a subsequent block, new one will be allocated from the system whose size is determined by Formula 3.4.1 given in Chapter 3. If MBS is too small to begin with, subsequent blocks will grow slowly and the system will have to perform many memory allocations.

- **Block Increase Ratio (BIR)**: BIR decides how fast the memory block size increases. It has a close relationship with MBS. The problem here is again finding a balance between memory allocation cost and memory waste.

- **Input/Output Buffer Size (IBS/OBS)**: An input buffer contains the current document content to be parsed. The parser works over this buffer. The parsed data will be copied to the containers. The input buffer will be reloaded when the current file pointer comes to its end. Similarly, when writing the output, XComp also keeps an output buffer. IBS and OBS don't influence compression ratio, but they affect the compression time.

In Figures 4.1, 4.2, and 4.3 we show compression times and compression ratios of XComp with different memory window sizes running on DC_10, TC_100, TC_1000[1]. For each data source, we used four sets of other parameter values while varying MWS. Each line in the figures represents the experimental results of one set of these parameters. We choose large documents as samples because MWS does not affect compression performance when the compressed documents are smaller than memory window.

[1]DC or TC indicates data-centric or text-centric, while the numeric values indicate database size in MB.

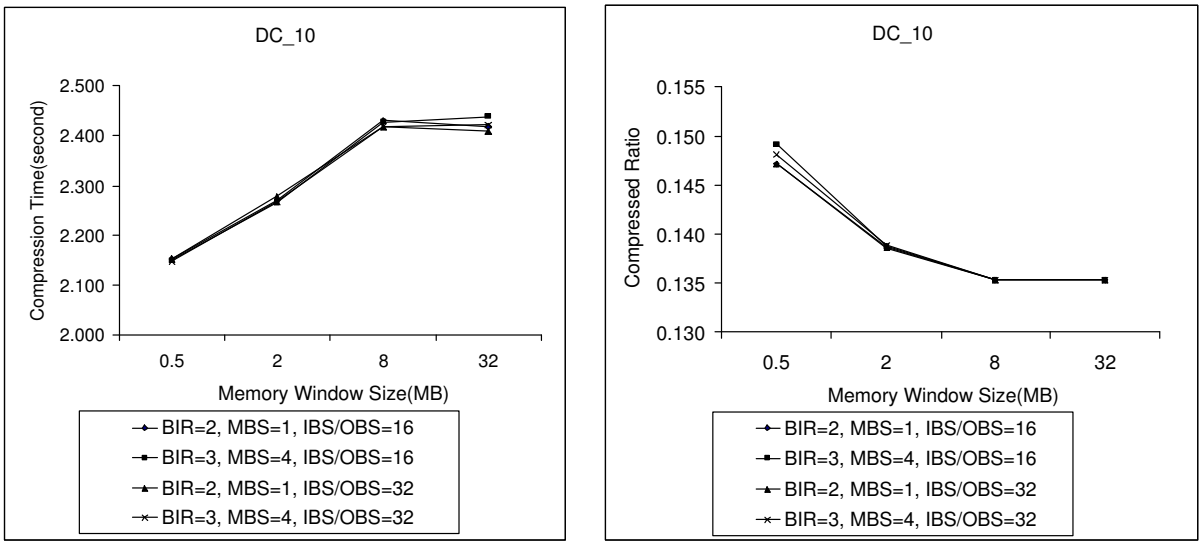

Figure 4.1: The Effects of Varying Memory Window Size (a)

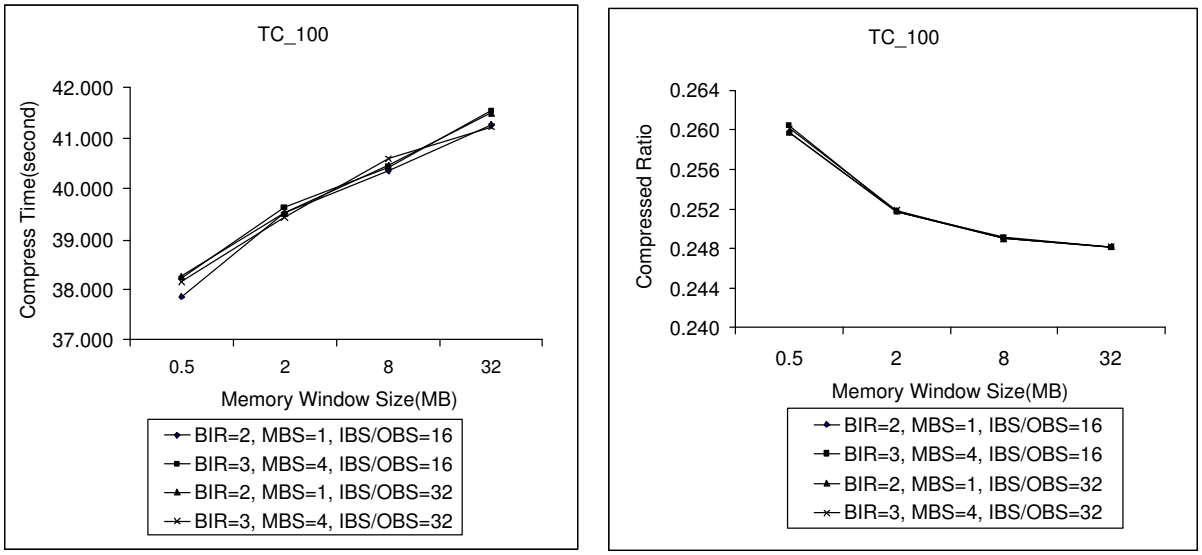

Figure 4.2: The Effects of Varying Memory Window Size (b)

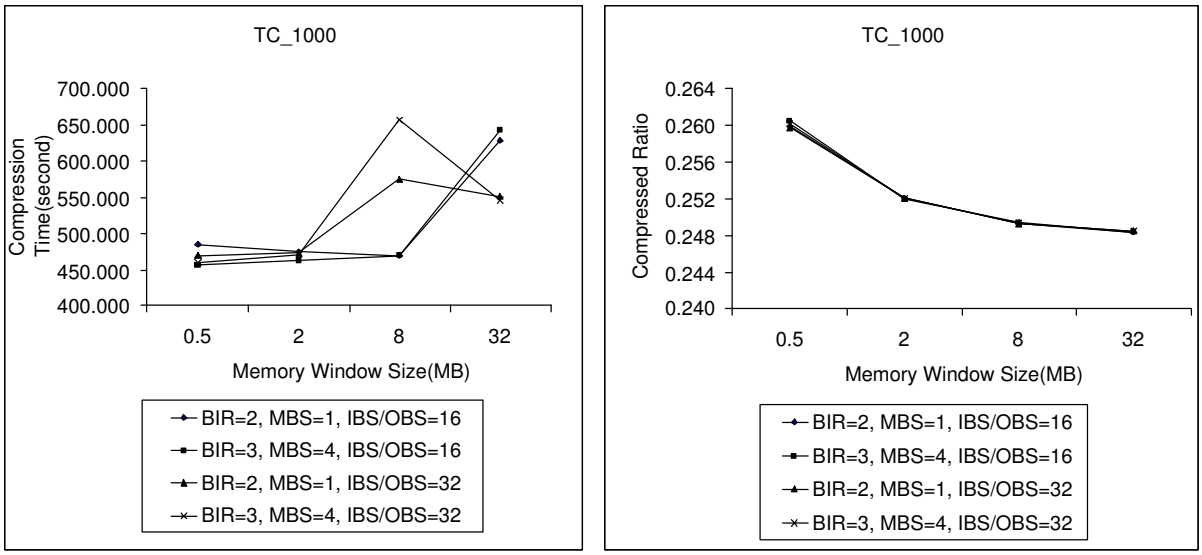

Figure 4.3: The Effects of Varying Memory Window Size (c)

The left charts show the change in compression time as MWS increases, while the right charts show the change in compression ratio as MWS increases. In most cases, there is an obvious trend that compression time increases and compression ratio decreases when MWS increases. Nearly all of the curves follow this trend. That means that MWS is independent of other parameters. It is interesting that in Figure 4.1 the curves become flat when MWS increases from 8MB to 32MB. That can be explained by the fact that all of the parsed data of DC_10 can fit in one memory window of size 8MB. Therefore, when the memory window continues to increase, the compression ratio and time won't change. Although DC_10's size (10.298MB) is bigger than 8MB, the parsed data are less than 8MB since we use a more compact representation to store them (for example, no duplicate tag or attribute names exist in the parsed data). We chose 8MB as the default value of MWS. The reason is

that when MWS is larger than 8MB, the marginal gain in compression ratio is not so significant while compression time continues to increase even with a higher rate (see Figures 4.2 and 4.3).[2] And main memory is a kind of precious resource. In real practice, it is impossible to allow an application to use most of main memory to do compression.

The experimental results for varying minimum block size is shown in Figure 4.4. For both TC_10 and DC_100, the compression time changes very little and randomly. Therefore, MBS is probably not a parameter that affects the compression time, and we choose 1KB as the default value for it in XComp. The situation is the same for block increase ratio and input/output buffer size. Figures 4.5 and 4.6 show that when BIR and IBS/OBS vary, compression time does not change much. These experiments convinced us that MBS, BIR and IBS/OBS are not very related to XComp's performance. We can assign any reasonable values for them, and we have chosen 1KB, 4 and 32KB/32KB, respectively, in the following experiments.

4.3 Comparison Experiments

4.3.1 Compression Ratio

Figure 4.7 shows the compression ratios for different compression tools over all data sources. For each document, the five connected bars represent XComp (using Zlib), XComp (using Huffman encoding), XMill, XGrind and Zlib compression ratios. In these five bars, XComp (Zlib) and XMill have similar performance and XComp(Huffman) and XGrind have similar performance and Zlib's performance is between them. The

[2]We note that in the left figure of Figure 4.3 two curves show a pattern different from others at 8 MB. It is not clear what the reason for this is. Since, in all other experiments, the curves follow the described pattern, we decided to ignore this anomaly.

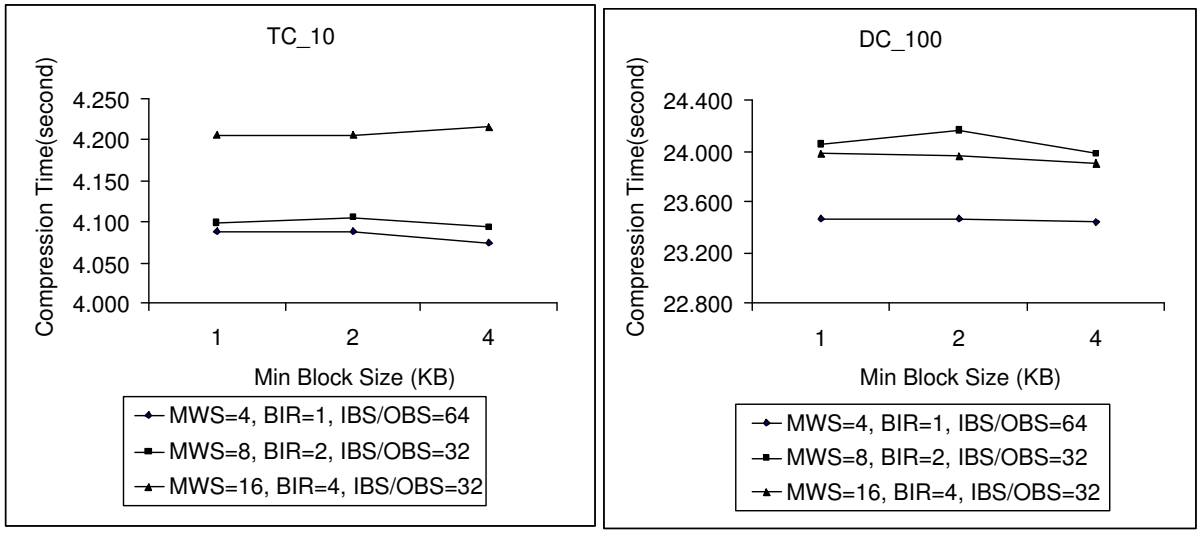

Figure 4.4: The Effects of Varying Minimum Block Size

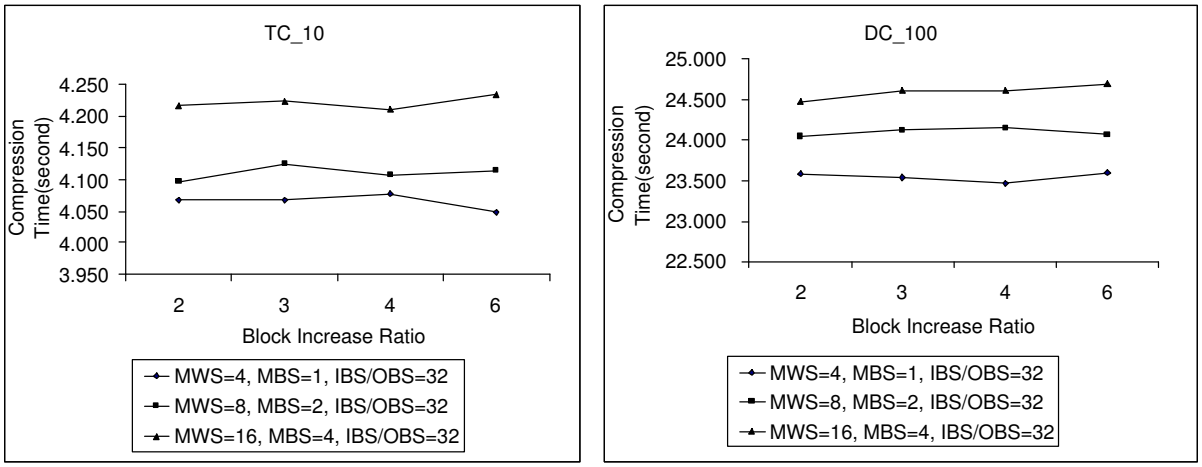

Figure 4.5: The Effects of Varying Block Increase Ratio

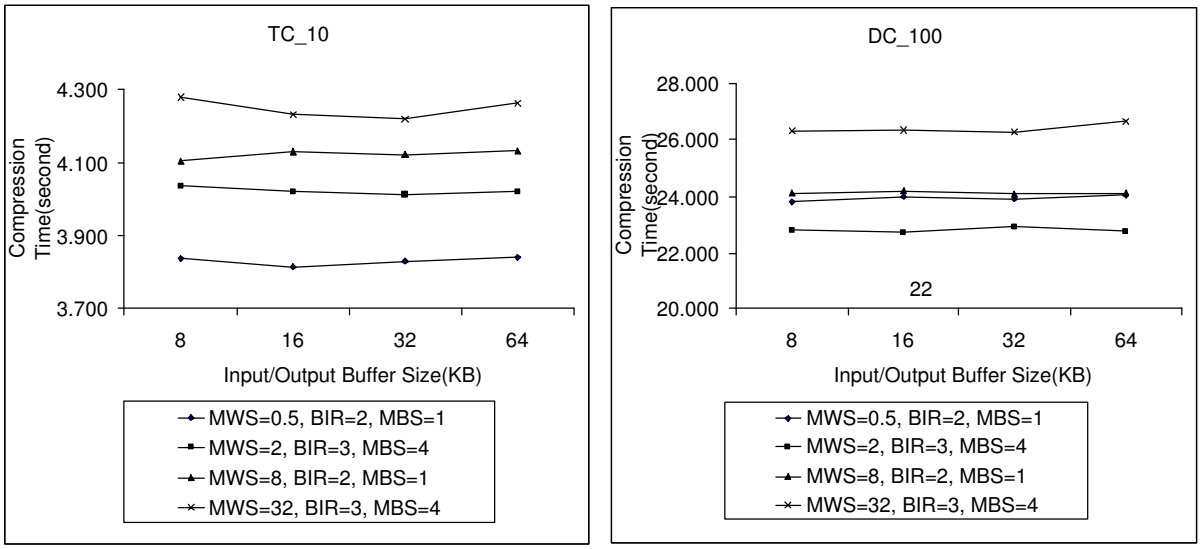

Figure 4.6: The Effects of Input/Output Buffer Size

compression ratios of XComp (Zlib) and XMill are very close for most documents. Except for DC_10 (13.53% vs 13.47%), XComp (Zlib) always outperforms XMill, albeit by a small margin, in compression ratio. The XComp (Huffman) and XGrind are much worse than these two. They compressed documents 25% larger than XComp (Zlib) and XMill. This is expected as the cause of their different compression engines — LZ algorithm and Huffman encoding. Huffman encoding assumes the sources as i.i.d. It can not exploit the fact that most data sources are not memoryless.

For the first six documents, which are all smaller than 1.5MB, Zlib's compression ratios are close to XComp(Zlib) and XMill. It's even better than XComp(Zlib) when the document is extremely small (Provinces in Table 4.1, which is 2KB). However, with the increase in document size, its performance gets worse. For data-centric XML database documents, Zlib is even worse than XComp(Huffman) and XGrind. This can be explained by the fact that XComp(Huffman) and XGrind can significantly improve

their work by re-organizing XML data, so that they can beat Zlib even if they have a less efficient compression engine. Zlib has an advantage over XComp(Zlib) and XMill only when the document size is small, i.e. several KB. XMill can not compete with Zlib when the document is smaller than 20KB [13]. XComp(Zlib) improved this boundary to 4KB (emails with 4546Byte in Table 4.1).

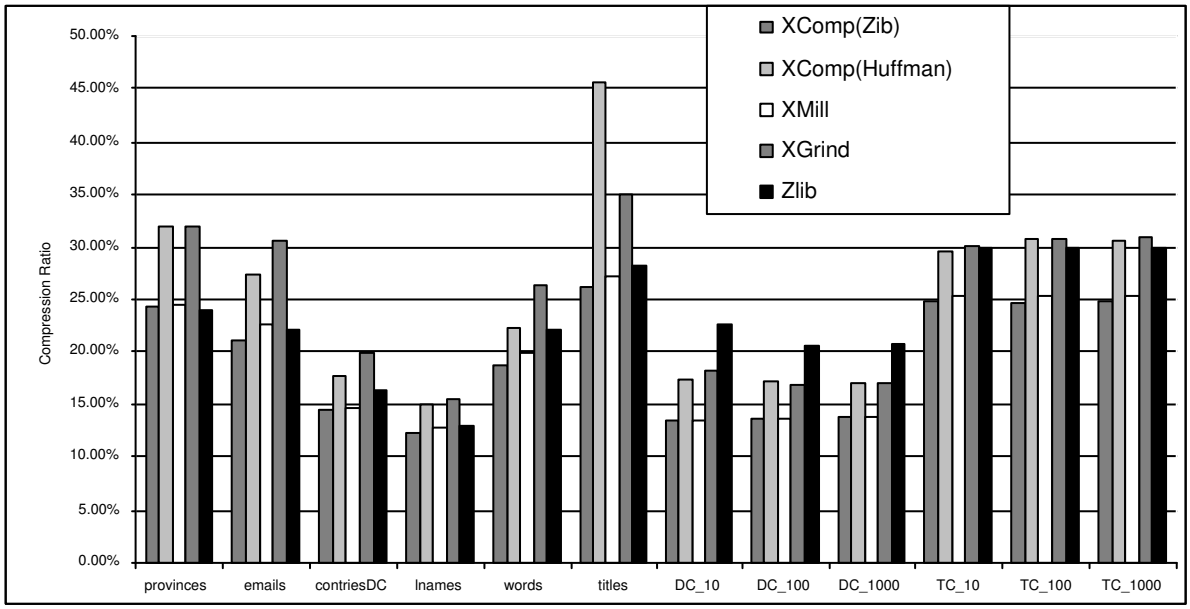

Figure 4.7: Compression Ratio

4.3.2 Compression Time Comparison

The compression times were measured as in Table 4.2. Similar to the results of compression ratio experiments, we still can classify the compression tools into two groups based on their performance. However, this time, XGrind and XComp(Huffman) worked better than XComp(Zlib), XMill, and Zlib. Obviously, compared with Huffman encoding, we pay the price of time to get better compression of LZ algorithm.

XComp(Zlib) and XMill have nearly the same speed. One did a little better on some documents while the other did better on the others. Their speeds are comparable to Zlib's. For XComp(Zlib) and XMill, the compression work split into two steps: (1) parsing and re-grouping the data, and (2) applying Zlib to compress these data. Why are they still nearly as fast as Zlib? The reason is that the first step reduced the size of the data to be compressed. So they do less work in step 2 than Zlib. The total compression time, which is the sum of steps 1 and 2, is close to that of compressing the original document directly.

	XCompr(Zlib)	XComp(Huffman)	XMill	XGrind	Zlib
provinces	0.011	0.005	0.011	0.010	0.010
emails	0.011	0.011	0.011	0.011	0.011
countriesDC	0.011	0.010	0.011	0.011	0.013
lnames	0.071	0.039	0.073	0.058	0.055
words	0.116	0.068	0.115	0.077	0.093
titles	0.532	0.229	0.115	0.326	0.526
DC_10	2.361	1.310	2.389	1.264	2.100
DC_100	23.557	13.000	23.479	13.069	23.299
DC_1000	259.081	150.073	260.195	140.321	252.161
TC_10	4.042	2.016	4.039	2.035	3.795
TC_100	39.086	19.560	39.771	19.085	37.043
TC_1000	431.753	227.800	432.791	221.589	397.310

Table 4.2: Compression Time

For these experiments, we conclude that XComp(Zlib) and XMill significantly improve compression ratio over Zlib while generally performing as fast as Zlib. They have a better overall performance than the general purpose compression tool. Compared to them, XComp(Huffman) and XGrind have better speed but worse compression ratio.

4.3.3 Internet Transmission

For a given network bandwidth, the transmission time is the sum of (a) time to compress/decompress, and (b) time to transfer the compressed document. The second component is decided by the compression ratio. From the above experiments we know that XComp(Huffman) and XGrind use less compression time and XComp(Zlib) and XMill generate smaller compressed data. In Tables 4.3 and 4.4, we report their performance in the Internet transmission. We only run the experiments on five documents. For the small ones, the bandwidth is continuously varying so that it is impossible to accurately measure the transmission time for them. We did not try the two largest documents as transferring them is too time consuming.

	XComp(Zlib)	XComp(Huffman)	XMill	XGrind	Zlib	No Compressoin
titles	1.158	2.025	1.209	1.987	1.777	3.488
DC_10	6.629	8.113	6.590	7.943	7.235	25.595
DC_100	68.897	80.145	69.594	77.212	81.093	253.712
TC_10	13.097	15.970	13.309	16.180	14.160	25.780
TC_100	135.415	162.097	138.129	252.436	140.128	252.436

Table 4.3: Internet Transmission (From University of Waterloo to University of Alberta

	XComp(Zlib)	XComp(Huffman)	XMill	XGrind	Zlib	No Compressoin
titles	0.891	1.245	0.905	1.179	1.044	3.350
DC_10	3.743	5.103	3.700	5.283	4.275	24.781
DC_100	35.889	50.174	39.551	51.882	40.197	255.770
TC_10	7.059	10.013	7.211	9.813	9.204	25.183
TC_100	71.654	100.194	74.183	103.105	89.724	254.195

Table 4.4: Internet Transmission (From University of Alberta to University of Waterloo

All the compression tools transferred documents faster than raw transmission (e.g. without compression). Among them, XComp(Zlib) did best in 8 cases and XMill did best in 2 cases. The performance of XComp(Huffman) and XGrind and Zlib are 30% to 80%, respectively, slower than these two. In this set of experiments, the compression tools with higher compression ratio performed better than the ones with faster speed.

The following analysis can be made for this experiment's results. The transmission work of XML compression tools can be divided into 5 parts: (1) reading and parsing (2) compressing (at the client side), (3) transferring the data, (4) decompressing (at the server side), and (5) writing. Step (2) only can be done when step (1) completes. However, steps (2) and (3) can be done in parallel. So, if the compression time is far less than transfer time, it does not contribute to much to the total time. For example, DC_10's compressed size by XComp(Zlib) is 2.603MB. So its transfer needs 6.533 seconds, which is longer than the time it takes to compress it. Therefore the total time is determined by the transfer time and time for steps (1), (4), and (5). Recall that our results are based on the rate 408KB/s. If we can greatly improve the bandwidth, the compression ratio will become less important and the compression time will become more critical. For example, if the transfer rate is 2MB/s, XComp(Zlib) compresses DC_10 will be transmitted in only 1.302 seconds which is much shorter than the compression time (4.042 seconds). In this case, the compression time will play an important role. In the extreme case, if we can improve the transfer rate to 10MB/s while the CPU speed remains unchanged, we will find that the uncompressed data are transferred the fastest.

Another interesting phenomenon is that sending data from University of Alberta

to University of Waterloo is much faster than doing it in the reverse direction if we use compression tools. As stated earlier, the total time is decided by transfer time and time for steps (1), (4), and (5). The PC (at University of Waterloo) is faster than the Unix server (which is a public server at University of Alberta). Therefore, step (4) takes more time on this Unix server than done on the PC. Steps (1) and (5) takes nearly same time if done on the same machine. Therefore doing (1) on the PC and (5) on the Unix server is equivalent to doing (1) on the Unix server and (5) on the PC.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

In this thesis, we develop an XML compression tool, called XComp, whose target applications are data archiving and data exchange. For the first type of applications, XComp has a competent compression ratio to reduce the data volume. For the data exchange applications, it helps to improve the transmission speed by its high compression ratio with considerable compression time. We have presented the architecture and implementation of XComp, and discussed the theoretical justification for our approach to XML compression. In our experiments, we compared XComp with Zlib and other XML compression tools. The results validate XComp as a generally better tool. Using Zlib compression engine, XComp gets the best compression ratio in nearly all cases. Compared to Zlib, it significantly improves the compression ratio while using similar time. It has similar performance as XMill, but outperform it (not in a major way) both in compression ratio and compression time. Consequently, XGrind is good at compression speed but worse in result of compression ratio. In general, XComp(Zlib) clearly has an overall advantage over others as a tool for compression XML data. For data exchange, it is also the winner in our experimental

circumstances. The performance of these compression tools is affected by CPU speed and network speed. The powerful machines with slow network favor the compressor with good compression ratio but slow compression speed, like XComp. In contrast, the reverse will favor the compressor with lower compression ratio but high speed. However, unless the bandwidth has a substantial improvement, i.e. reaching 5MB/s[1], while other conditions keep unchanged, we expect XComp(Zlib) to still outperforms XGrind or Zlib as it did in our experiments. Moreover, XComp has its potential in speed improvement, which is discussed in the section of future work.

The contributions of our research can be summarized as follows:

First, we proposed an architecture that follows the principles of separating data from structure and grouping data based on semantics. In this architecture, data are parsed before being compressed. Two ideas concerning memory management are used here. We use a memory window to confine the maximum memory usage and containers, which can efficiently increase its memory space, to hold the data with the same semantics. This architecture also allows consuming XML data and producing output in a stream model.

Second, XComp groups data not only based on their tag/attribute names but also based on their document levels. We believe that the document level also implies XML data's semantics. This grouping method enables XComp to distinguish data with the same name but with different information statistics, since they are at different levels and have different values or domains or formats.

Third, we designed our own way to encode XML structure. Integers are used in it to represent XML markups, while the length information of string data are stored

[1]Note that we are talking about sustained wide-area network speed, which is generally slower than this.

separately. This feature, different from other XML compression tools, guarantees our encoding efficiency for both small and large documents. We also have two models in our representation: preserving white space or not.

Finally, XComp can handle XML documents in Unicode. Unlike XMill, which only handles ASCII documents, it strictly follows XML specification of character set.

5.2 Future Work

In some data exchange applications, there is a need to exchange complete documents, while others only require data formatted in XML. The latter ones may still use a SAX parser to read XML documents. For these applications, we can save the time to compose the output documents by directly generating SAX events. Such a decompressor will provide a set of API following SAX standards. It directly translates the compressed data to SAX events. To implement such a decompressor, we do not make architectural changes to our existing decompressor, since the original document structure is encoded as a sequence of markups that follows the SAX events' order in the original document. By saving output, this approach will shorten the data transfer time.

Parallel computation usually helps to improve program speed, and XComp can be parallelized. In XComp's architecture, the containers are compressed one by one when the memory window is full or the end of file is met. The parsing and the compressing are done sequentially. This is necessary for Huffman coding, because we need to know all the statistics about the data in one container before we can compress them. However, Zlib compression does not require that. The parallel approach for Zlib compression can be done as follows. The parser and each container can have

its own process. While parsing, any container process begins to work when there are substantial data, i.e. 64KB, in its container. It can write the output to its individual internal buffer. When internal buffer is full, the content is written to the output stream. Since there is only one output stream, only one container process can write its output at one time. The benefit of this approach is that the parsing and the compression can be done in parallel. This will save time, especially when the parser does lots of I/O. An attractive side-effect of this approach is that we are able to reduce our memory use. We do not need to keep all data in the memory window before compressing, but only a fixed size memory for each container.

Another future work is to extend XComp(Huffman) to enable direct querying of compressed documents. Huffman coding encodes an alphabet with fixed codes. That allows us to directly search keywords in the compressed data. Since the data volume is reduced, such querying may be even faster than querying the original data. Related work has been done in XGrind.

Appendix A

XML Document Definition

XML Document	::=	**PreText Prolog Element Misc***
Pretext	::=	*%string without '<'%*
Prolog	::=	**XMLDecl**? **Misc*** (**DocTypeDecl Misc***)?
XMLDecl	::=	**PI**
DecTypeDecl	::=	('<!DOCTYPE' *%string without '[' and '>'%* '>') | ('<!DOCTYPE' *%string without '[' and '>'%* '[' *%string without ']'%* ']' **S** '>')
Misc	::=	**Comment** | **PI** | **S**
Comment	::=	'<!--' *%string without '--'%* '-->'
PI	::=	'<?' *%string without '?>'%* '?>'
S	::=	*%whitespace chars in XML%*
Element	::=	**EmptyElemTag** | (**StartTag Content EndTag**)
EmptyElemTag	::=	'<' **Name** (**S Attribute**)* **S**? '/>'
StartTag	::=	'<' **Name** (**S Attribute**)* **S**? '>'
EndTag	::=	'</' **Name S**? '>'
Content	::=	**S**? **CharData**? **S**? ((**Element**|**CDSect**|**PI**|**Comment**) **S**? **CharData**?)*
CDSect	::=	'<![**CDATA**[' *%string without ']]>'%* ']]>'
Attribute	::=	**Name S**? '=' **S**? **AttValue**
Name	::=	*%string without white space, '/' and '>'%*
CharData	::=	*%string without '<'%*
AttValue	::=	("' *%string without "'%* "') | ('"' *%string without '"'%* '"')

Appendix B

Light-Weight SAX Parser Interface

/** from: The start position of the data content or the tag/attribute name. It is in the buffer of input file.

length: The length of the data content or the tag/attribute name.

hasFound: Because the buffer for the input file is limited (typically 32KB), the length of the data content or the tag/attribute name may exceed this size. hasFound indicate whether the end of string is met. If it is false, the next event will still contain the rest of it. **/

- /*The event that Parser meets PreText before the beginning of first '<' in the XML file. */
  void handlePreText(const char hasFound, const char *from, const int length);
- /*The event that Parser meets PI in the XML file. */
  void handlePI(const char hasFound, const char *from, const int length);
- /*The event that Parser meets DTD(Document type declaration) in the XML file. */
  void handleDTD(const char hasFound, const char *from, const int length);
- /*The event that Parser meets Comment in the XML file. */
  void handleComment(const char hasFound, const char *from, const int length);
- /*The event that Parser meets CDADA Section in the XML file. */
  void handleCDSect(const char hasFound, const char *from, const int length);
- /*The event that Parser meets white spaces in the XML file. */
  void handleWhiteSpaces(const char hasFound, const char *from, const int length);
- /*The event that Parser meets a start tag(following '<') in the XML file. */
  uInt32** handleStartTag(const char *from, const int length);
- /*The event that Parser meets an end tag in the XML file. */
  bool handleEndTag(const char *from, const int length);
- /*The event that Parser meets end of an empty element. */
  void handleEndOfEmptyElement();
- /*The event that Parser meets an attribute name in the XML file. */
  uInt32** handleAttributeName(const char *from, const int length);

- /*The event that Parser meets an attribute value in the XML file. */
  void handleAttributeValue(const char hasFound, const char *from, const int length, const bool quoteMark);
- /*The event that Parser meets content data in the XML file. */
  void handleContentData(const char hasFound, const char *from, const int length);
- /*The event that Parser meets an equal mark '=' (for attribute) in the XML file. */
  void handleEqualMark();
- /*The event of '>' (the end of tag) only if WhiteSpaceContainer is perserved. */
  void handleGTChar();

Bibliography

[1] K. Challapali, X. Lebegue, J. S. Lim, W. Paik, R. Girons, E. Petajan, V. Sathe, P. Snopko, and J. Zdespki. The Ground Alliance System for US HDTV. *Proc. IEEE*, 83:158–174, Feburary 1995.

[2] N. Faller. An Adaptive System for Data Compression. In *7th Asilomar Conf. on Circuits, Systems and Computing*, pages 593–597, 1973.

[3] J. Gailly and M. Adler. A Massively Spiffy Yet Delicately Unobtrusive Compression Library. Available at http://www.gzip.org/zlib, January 2003.

[4] R. G. Gallager. Variations on a theme by huffman. *IEEE Trans. Information Theory*, IT-24:668–674, November 1978.

[5] Jerry D. Gibson, Toby Berger, Tom Lookabaugh, Dave Lindbergh, and Richard L. Baker. *Digital Compression for Multimedia*. Morgan Kaufmann, 2 edition, 2001.

[6] Marc Girardot and Neel Sundaresan. Millau: An Encoding Format for Efficient Representation And Exchange of XML over the Web. Available at http://www9.org/w9cdrom/154/154.html, 2001.

[7] D. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Proc. IRE*, 40:1098–1101, September 1952.

[8] R. Hunter and A. H. Robinson. International Digital Facsimile Coding Standards. *Proc. Inst. Elec. Radio Eng.*, 68:854–867, July 1980.

[9] International Organization for Standardization: 8879. Information processing – text and office systems – standard generalized markup language (sgml), 1986.

[10] D. E. Knuth. Dynamic huffman coding. *Journal of Algorithms*, 1985:163–180, 1985.

[11] A. K. Kolmogorov. Three Approaches to the Quantitative Definition of Information. *Probl. Inform. Transm.*, 1:1–7, 1965.

[12] A. Lempel and J. Ziv. On the Complexity of An Individual Sequence. *IEEE Trans. Information Theory*, IT-22:75–81, January 1976.

[13] H. Liefke and D. Suciu. XMill: An Efficient Compressor for XML Data. *ACM SIGMOD Record*, 29, June 2000.

[14] David Megginson. SAX Simple API for XML version 2. Available at http://www.saxproject.org/, May 2000.

[15] A. Salminen and F. W. Tompa. Requirements for XML document database systems. In *Proceedings of the 2001 ACM Symposium on Document Engineering*, pages 85–94, 2001.

[16] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Tech. J.*, 27:379–423, July 1948.

[17] The Unicode Consortium. The Unicode Standard, Version 4.0. Available at http://www.unicode.org/standard/versions/enumeratedversions.html, April 2003.

[18] P. Tolani and J. R. Haritsa. XGRIND: A Query-friendly XML Compressor. In *Proc. 18th Int. Conf. on Data Engineering*, pages 225–234, 2002.

[19] J. S. Vitter. Dynamic huffman coding. *ACM Trans. Mathematical Software*, 15:158–167, June 1989.

[20] T. A. Welch. A Technique for High Performance Data Compression. *Computer*, 17:8–19, June 1984.

[21] World Wide Web Consortium(W3C). HTML 4.01 Specification. Available at http://www.w3.org/TR/html4/, December 1999.

[22] World Wide Web Consortium(W3C). WAP Binary XML Content Format. Available at http://www.w3.org/TR/wbxml/, June 1999.

[23] World Wide Web Consortium(W3C). XML Path Language (XPath) Version 1.0. Available at http://www.w3.org/TR/xpath, November 1999.

[24] World Wide Web Consortium(W3C). Extensible Markup Language (XML) 1.0 (second edition). Available at http://www.w3.org/TR/REC-xml, October 2000.

[25] World Wide Web Consortium(W3C). XML Schema. Available at http://www.w3.org/XML/Schema, April 2000.

[26] World Wide Web Consortium(W3C). XML Information Set. Available at http://www.w3.org/TR/2001/PR-xml-infoset-20010810, August 2001.

[27] World Wide Web Consortium(W3C). XQuery 1.0 and XPath 2.0 Data Model, W3C Working Draft 7 June 2001. Available at http://www.w3.org/TR/2001/WD-query-datamodel-20010607, June 2001.

[28] World Wide Web Consortium(W3C). Document Object Model (DOM) Technical Reports. Available at http://www.w3.org/DOM/DOMTR, 2003.

[29] B. B. Yao, M. T. Özsu, and J. Keenleyside. *XBench – A Family of Benchmarks for XML DBMSs*, pages 162–164. 2002.

[30] J. Ziv and A. Lempel. A Universal Algorithm for Sequentail Data Compression. *IEEE Trans. Information Theory*, IT-24:337–343, May 1977.

[31] J. Ziv and A. Lempel. Compression of Individual sequences via Variable-rate Coding. *IEEE Trans. Information Theory*, IT-24:530–536, September 1978.