Faculty of Mathematics and Physics
Charles University, Prague

Master Thesis

Katsiaryna Chernik

Syllable-based compression for XML documents

Department of Software Engineering
Supervisor: Jan Lánský
Study program: Computer Science

I would like to thank Jan Lánský for his helpfulness and patience while supervising this work.

I declare that I have worked out this thesis on my own, using only the resources stated. I agree that the thesis may be publicly available.

Prague, April 23, 2006 Katsiaryna Chernik

Contents

1 Introduction **5**

2 XML: An Overview **6**
2.1 XML Document . . . 6
2.2 XML Processing . . . 8
2.2.1 SAX . . . 8
2.2.2 DOM . . . 8

3 Text compression algorithms **10**
3.1 Character-based text compression . . . 10
3.1.1 LZW . . . 10
3.1.2 Huffman Coding . . . 11
3.1.2.1 Static Huffman Coding . . . 12
3.1.2.2 Adaptive Huffman coding . . . 14
3.2 Syllable-based text compression . . . 14
3.2.1 Decomposition of word into syllables . . . 14
3.2.2 LZWL . . . 16
3.2.3 HufSyl . . . 18

4 XML compression **20**
4.1 XMill . . . 20
4.2 XMLPPM . . . 23
4.3 XGrind . . . 24
4.4 XPress . . . 26
4.5 Other XML compressors . . . 28

5 Experimental methodology **30**
5.1 Comparative compression tools . . . 30
5.2 XML data sources . . . 30
5.3 Compression performance metrics . . . 30

6 XMLSyl **32**
6.1 Motivation . . . 32
6.2 Architecture and principles of XMLSyl . . . 32
6.3 Parser . . . 33

6.4 Encoding the structure of XML document 34
6.5 Containers 36
6.6 Syllable Compressor 36
6.7 Syllable Dictionaries 37
6.8 Conversion between different character encoding 37
6.9 Comparison experiments 37
6.9.1 Performance of XMLzwl 37
6.9.2 Performance of XMLhuf 39

7 XMillSyl **40**
7.1 Motivation 40
7.2 Implementation 40
7.3 Comparison Experiments 41

8 Conclusions and further work **44**

Abstract

Title: Syllable-based compression of XML
Author: Katsiaryna Chernik
Department: Department of Software Engineering
Supervisor: Jan Lánský
Supervisor's e-mail address: zizelevak@gmail.com
Abstract: Syllable-based compression achieves sufficient results on small or middle-sized text documents. Since the majority of XML documents are that size, we suppose that the syllable-based method can give good results on XML documents, especially on documents that have a simple structure (small amount of elements and attributes) and relatively long character data content.
In this paper we propose two syllable-based compression methods for XML documents. The first method, XMLSyl, replaces XML tokens (element tags and attributes) by special codes in input document and then compresses this document using a syllable-based method. The second method, XMillSyl, incorporates syllable-based compression into the existing method for XML compression XMill. XMLSyl and XMillSyl are compared with other XML-conscious compression methods as well as with a non-XML syllable-based compression methods.
Keywords: XML, data compression, syllable-based compression.

Název práce:Slabiková komprese XML
Autor: Katsiaryna Chernik
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce:: Jan Lánský
e-mail vedoucího:: zizelevak@gmail.com
Abstrakt: Slabiková komprese prokazuje dobré výsledky na malých a sřtedně velkých textových dokumentech. Protože většina XML dokumentů je středně velká, domníváme se, že slabiková komprese muže byt vhodná pro XML, zvláště pak pro dokumenty, které mají jednoduchou strukturu (máły počet elementu a atributu) a poměrně dlouhý znakový obsah.
V této práci jsme navrhli dvě slabikové kompresní metody pro textová data ve formátu XML. První metoda, XMLSyl, nahrazuje XML značky (elementy a atributy) ve vstupním dokumentu speciálními kódý a pak komprimuje dokument pomoci slabikové komprese. Druhá metoda, XMillSyl, spojuje slabikovou kompresi a kompresní metodu XMill. XMLSyl a XMillSyl porovnáváme s již existujícími kompresními metodami pro XML a s obecnými slabikovými kompresními metodami.
Klíčová slova: XML, komprese dat, syntaktická komprese, pravděpodobnostní modelování.

Chapter 1

Introduction

The Extensible Markup Language (XML) [6] is a simple text format for structured text documents. XML provides flexibility in storing, processing and exchanging data on the Web. However, due to their verbosity, XML documents are usually larger in size than other exchange formats containing the same data content. One solution to this problem is to compress XML documents. Because XML is a text format, it is possible to compress XML documents with existing text compression methods. These methods are more effective when XML documents have simple structure and long character data content. There are different types of text compression: text compression by characters and text compression by words. There is also a novel method: text compression by syllables [13]. In our work an application of this method to XML documents was developed. Since single text compression is not able to discover and utilize the redundancy in the structure of XML, we modify the syllable-based compression method for XML.

At the beginning we made the hypothesis that XML syllable-based compression will be suitable for middle-sized textual XML documents. There are many XML documents that meet these conditions, for example documentation written in DocBook [17] format or news in RSS format [19]. Moreover we suppose that our compression would be more suitable for documents in languages with rich morphology (for example Czech or German [13]).

Chapter 2

XML: An Overview

XML [6] stands for eXtensible Markup Language, and it is a standard for structured text documents developed by the World Wide Web Consortium (W3C) [23]. XML can be used to structure text in such a way that it is readable by both humans and machines, and it presents a simple format for the exchange of information across the internet between computers.

XML is a simplification (or subset) of the Standard Generalized Markup Language (SGML) which was developed in the 1970s for the large-scale storage of structured text documents.

2.1 XML Document

This section covers the basics of XML [24]. An XML document contains a prolog and a body. The minimal prolog contains a declaration that identifies the document as an XML document:

```
<?xml version='1.0' encoding='UTF-8' standalone='yes'?>
```

The XML declaration may contain the attributes:

- *version*: Identifies the version of the XML markup language used in the data. This attribute is required.
- *encoding*: Identifies the character set used to encode the data. (The default is 8-bit Unicode: UTF-8.)
- *standalone*: Tells whether or not this document references an other external files. If there are no external references, then ”yes” is appropriate.

The prolog can also contain definition of entities and a Document Type Definition (DTD). DTD is a set of rules that specify how the different tags in an XML document can be used together and the attributes that may belong to each tag. DTD can be defined directly within the prolog, as well as with pointers to external specification files.

An entity is an individual XML item that has a name. Referencing the entity by name causes it to be inserted into the document in place of the entity reference. To create an entity reference, the entity name is surrounded by an ampersand and a semicolon, like this:

```
&entityName;
```

An entity reference can reference an entire document, or external entity, or a collection of DTD definitions (a parameter entity).

The body of an XML document contains the actual marked up document wich is comprised of one or more named elements organized into a nested hierarchy. An element is an opening tag, data, and a closing tag. An opening tag is an element name preceded by a less-than symbol (`<`) and followed by a greater-than (`>`) symbol. For any given element, the name of the opening tag must match that of the closing tag. A closing tag is identical to an opening tag except that the less-than symbol is immediately followed by a forward-slash (`/`).

```
<tag>Some Data</tag>
```

If an element does not contain any data, the opening and closing tags can be combined. Observe the location of the forward slash just prior to the greater-than (`>`) symbol.

```
<emptyTag/>
```

An element may include zero or more attributes. An attribute specifies properties of the element that you modify and consists of a name/value pair. Attribute values must be contained in matching single or double quotes.

```
<tag name="value">Some Data</tag>
```

The elements may be arranged in an infinitely nested hierarchy, but only one element in the document can be designated as the root document element. The root document element is the first element that appears in the document. Also the elements must be properly nested within each other, i.e. if the start tag of element is in the content of another element, the end tag is in the content of the same element.

A document author can place comments in XML documents. A comment begins with the combination of characters `"<!--"` and ends with the combination of characters `"-->"` . Comments may appear as a child of any element in an XML document. They can also appear before or after the root element.

An XML file can also contain processing instructions that give commands or information to an application that is processing the XML data. Processing instructions have the following format:

```
<?target instructions?>
```

where *target* is the name of the application that is expected to do the processing, and *instructions* is a string of characters that embodies the information or commands for the application to process.

There are occasions when you need handle large blocks of XML or HTML that include many of the special characters. It would be inconvenient to replace each of them with the appropriate entity reference. For those situations, you can use a CDATA section:

```
<![CDATA[ Text to be ignored ]]>
```

CDATA is predefined XML tag for "Character DATA" and characters in CDATA section are not interpreted as XML.

2.2 XML Processing

There are two basic ways to work with XML [25]. One is called SAX ("Simple API for XML"), and it works by reading the XML data in parts and calling a method for each element it finds. The other is called DOM ("Document Object Model"), and it works by reading in the entire XML document at once and creating an internal representation of it.

2.2.1 SAX

SAX [8], the Simple API for XML, is a traditional, event-driven parser. SAX obtains the content of an XML document as a sequence of events. For example [21], in parsing the following sample document

```
<?xml version="1.0"?>
  <document>
    <para>This is a very simple example</para>
  </document>
```

a SAX parser would report events

```
startDocument
startElement: document
startElement: para
characters: This is a very simple example
endElement: para
endElement: document
endDocument
```

This approach is very efficient because the parser does not hold the information in memory and the processing is minimal at the parser's level [21]. However, in most cases it is not enough to possess the information of a single SAX event. For example, when the SAX parser reports a character event the parent element is not known unless the name of the last start tag has been kept in memory. Therefore, relevant information should be stored in memory for further usage. The good thing is that an application knows exactly what it needs and it also knows when each piece of information is not necessary anymore. SAX lets build efficient parsing mechanisms, but it is difficult to use.

2.2.2 DOM

Document Object Model (DOM) [27] is an application programming interface (API) for valid HTML and well-formed XML documents. It builds a tree structure in memory containing the information of the entire XML document.

Consider following XML document [27]:

```
<TABLE>
  <TBODY>
   <TR>
    <TD>Shady Grove</TD>
    <TD>Aeolian</TD>
   </TR>
   <TR>
    <TD>Over the River, Charlie</TD>
    <TD>Dorian</TD>
   </TR>
  </TBODY>
</TABLE>
```

A graphical representation of the DOM of the XML document is shown in Figure 2.1.

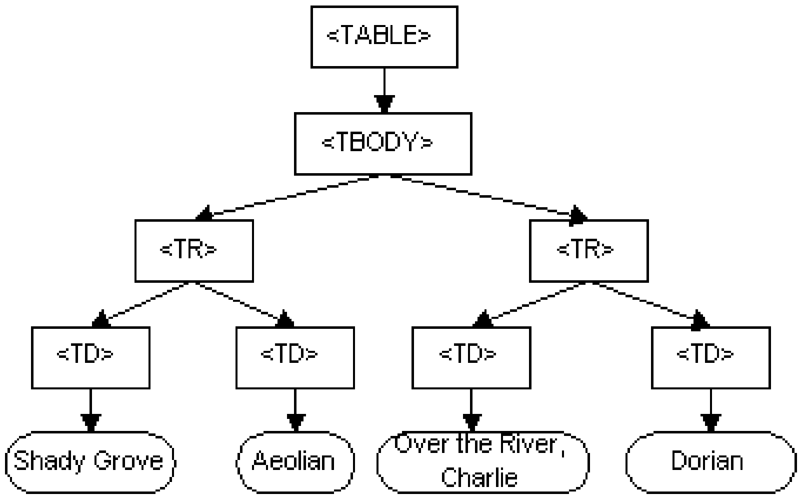


Figure 2.1: DOM Tree

The DOM API defines interfaces whose instances are linked in a tree and maintain information about elements, attributes, character data sections, processing instructions, etc. DOM is not the easiest way to manipulate the information of a document, but it has a huge advantage: a DOM tree is a mirror of the original document's structure and content [21]. Problems start to occur when you need to process large documents. Some documents will simply not fit into the computer's memory. Therefore, DOM is much easier to use than SAX, but has scalability issues.

Chapter 3

Text compression algorithms

It is possible to divide text compression algorithms into three types according to their basic compression unit: character-based, word-based and syllable-based algorithms. In this chapter we describe two character-based compression methods – LZW and Huffman. Then we introduce methods for decomposition of words into syllables. Finally we describe two syllable-base text compression methods – LZWL and HufSyl.

3.1 Character-based text compression

3.1.1 LZW

LZW [29] is short for Lempel-Zif-Welch, a data compression technique developed in 1977 by J. Ziv and A Lempel, and later refined by Terry Welch.

LZW is a lossless dictionary based encoding algorithm. It builds a dictionary of data occurring in the input stream that is being compressed and indexes that point to entries in the dictionary which are subsequently sent to the output stream. The dictionary does not have to be transmitted with the compressed text since the decompressor can build it the same way the compressor does.

In the initialization step the dictionary is filled with all single-character strings occurring in the input stream. In the following steps, LZW searches the input stream for maximal `STRING`, which corresponds with one from the dictionary, and generates new entries by appending the single character following `STRING` in the text to the end of the `STRING`. The index of the `STRING` in the dictionary is sent to the output stream.

The algorithm for LZW compression is as follows [28]:

```
Initialize table with single-character strings
STRING = first input character
WHILE not end of input stream
   CHARACTER = next input character
   IF STRING + CHARACTER is in the string table
       STRING = STRING + CHARACTER
   ELSE
       output the code for STRING
```

```
            add STRING + CHARACTER to the string table
            STRING = CHARACTER
        ENDIF
    ENDWHILE
    output code for string
```

Decoding LZW data is the reverse of encoding. In the initialization step the dictionary is filled with all single-character strings. The input stream of codes is translated into an output stream using the dictionary. The dictionary is updated for each code in the input stream excluding the first one. After the code has been translated into its corresponding string, the first character of the string is appended to the previous string. This new string is added to the dictionary in the same location as in the compressor's dictionary.

It is possible to have a situation where the code in input stream is not from the dictionary. In this case, the output string is created by concatenation of the last added string with its first character.

The decompression algorithm is as follows:

```
Initialize table with single character strings
CODE = first input code
OLDSTRING = translation of CODE
output OLDSTRING
WHILE not end of input stream
      CODE = next input code
      IF CODE is not in the string table
          STRING = STRING + CHARACTER
          add STRING to string table
          output STRING
      ELSE
          STRING = translation of CODE
          output STRING
          CHARACTER = first character of STRING
          add OLDSTRING + CHARACTER to the string table
      ENDIF
      OLDSTRING = STRING
END WHILE
```

3.1.2 Huffman Coding

Huffman coding is a coding algorithm presented by David Huffman in 1952. It is based on the frequency of occurrence of a data item. The basic idea is to assign short codes to symbols that have a higher probability of occurring and long codes to those that occur less often. This assignment can made in to variants - static and adaptive.

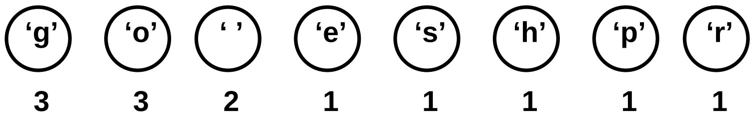
The static variant of the Huffman coding requires that the data items are not changed and their frequencies of occurrence are known in advance. These frequencies are received by static analysis of data. That why the static method requires two passes over the data.

In the adaptive variant, frequencies are changed during passing the data. The data are encoded in one pass and we do not need to store the mapping for the Huffman code, which is required by the static variant.

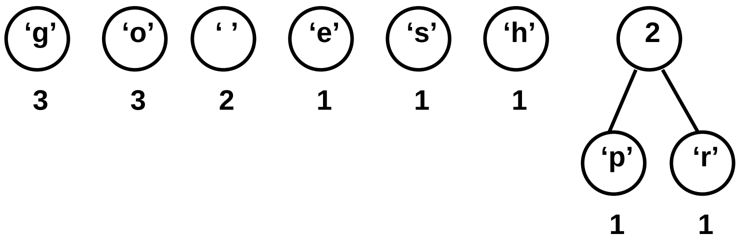
3.1.2.1 Static Huffman Coding

The Static Huffman [30] algorithm takes as input a list of nonnegative weights w(1), ... ,w(n) and constructs a binary tree. All the leaves of this tree are marked with the weights. The weights represent the frequency of occurrence of a particular data item. At each step in the algorithm the trees corresponding to the two smallest weights, w(i) and w(j), are merged into a new tree whose weight is w(i)+w(j) and whose root has two children which are the subtrees represented by w(i) and w(j). The weights w(i) and w(j) are removed from the list and w(i)+w(j) is inserted into the list. This process continues until the weight list contains a single value. If, at any time, there is more than one way to choose a smallest pair of weights, any such pair may be chosen.

For example [31], for string ”go go gophers” the list of input weights is `{3,3,2,1,1,1,1,1}` and the trees corresponding to these weights are shown below:



In first step two minimal nodes are merged. There are five nodes with the minimal weight of one, it does not matter which two are merged. The first step is shown here:



The final tree for string ”go go gophers” is shown in Figure 3.1.

The Huffman algorithm determines the length of the codewords to be mapped to each of the source letters. There are many alternatives for specifying the actual digits; it is necessary only that the code have the prefix property. The usual assignment entails labeling the edge from each parent to its left child with the digit `0` and the edge to the right child with `1`. The codeword for each source letter from our example is shown in Table 3.2.

The decoding Huffman algorithm is very simple. In the initialization step the Huffman binary tree is retrieved from the input stream (the tree was stored there by the encoding process). At the start of the decoding process we are in the root of the Huffman binary tree. Then we get a bit, and depending if it is `0` or `1` we go to the left or the right respectively. This process stops when we hit a leaf, which contains the symbol to decode. This symbol is sent to the output stream and we go to the root to decode a new symbol. This process is repeated until the end of input stream is not reached.

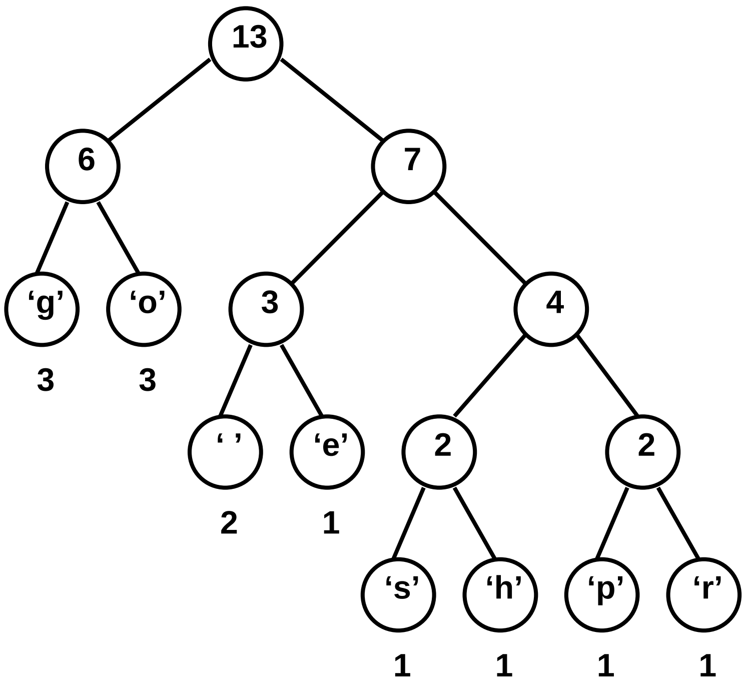


Figure 3.1: The Huffman tree for string ”go go gophers”.

char	codeword
’g’	00
’o’	01
’p’	1110
’h’	1101
’e’	101
’r’	1111
’s’	1100
’ ’	100

Figure 3.2: The Huffman codeword”.

3.1.2.2 Adaptive Huffman coding

The basis for the Adaptive Huffman [32] algorithm is the sibling property: A binary code tree suffices the sibling property if each node (except the root) has a sibling and if it is possible to assign the node number to each node in such a way:

1. A node with a higher weight will have a higher node number.

2. A parent node will always have a higher node number than its children.

Initially, the code tree consists of a single leaf node known as the Not-Yet-Transmitted (NYT) or escape code. NYT node has the maximum number. For each source unit the code tree is first checked to see if it already contains that units. If it does not, the code for path from the root to the NYT node is sent to the output stream to alert the decoder that a new source unit follows. Then the ASCII encoding of the new unit is sent to output. The NYT node spawns two new nodes. The node to its right is a new node containing the source unit and the new left node is the new NYT node. If the source unit is already in the code tree, the code for the path from root node to the corresponding tree node is sent to the output stream. The weight of that particular node is incremented. The increments cause possible reorganizations of the code tree by swapping the nodes to maintain the sibling property.

The reorganization process is done with the following steps. Before the node's weight is updated, the tree is searched for all nodes of equal weight. The soon-to-be updated value is swapped with the highest ordered node of equal weight and only then the weight is updated. In both cases for inserting values, weights are changed for a leaf and this change will effect all nodes above it. Therefore, after a node is inserted, the parent above must be checked and reorganized in the same way as the current node. In

The Figure 3.3 shows a flowchart of the tree manipulation process [32].

3.2 Syllable-based text compression

3.2.1 Decomposition of word into syllables

A syllable is a sequence of sounds which contains exactly one maximal subsequence of vowels [13]. This is a simplified definition of syllables which is not equivalent with the grammatically correct definition but suffices for the purpose of compression.

We recognize five types of syllables: small, capital, mixed, number and other. A small syllable is one that consists of small letters. A capital syllable consists of capital letters. A mixed syllable has the first letter capital and the other are small. A number syllable consists of digits and an other syllable consists of special symbols.

According to the syllable definition, decomposition of words into syllables is not always unique. There are four different algorithms of decomposing words into syllables to be used in syllable-based compression: universal left P_{UL}, universal right P_{UR}, universal middle-left P_{UML}, universal middle-right P_{UMR} [13]. These algorithms are composed of two parts. The first part is an initialization part and this is common for all algorithms. The second part is different for each algorithm.

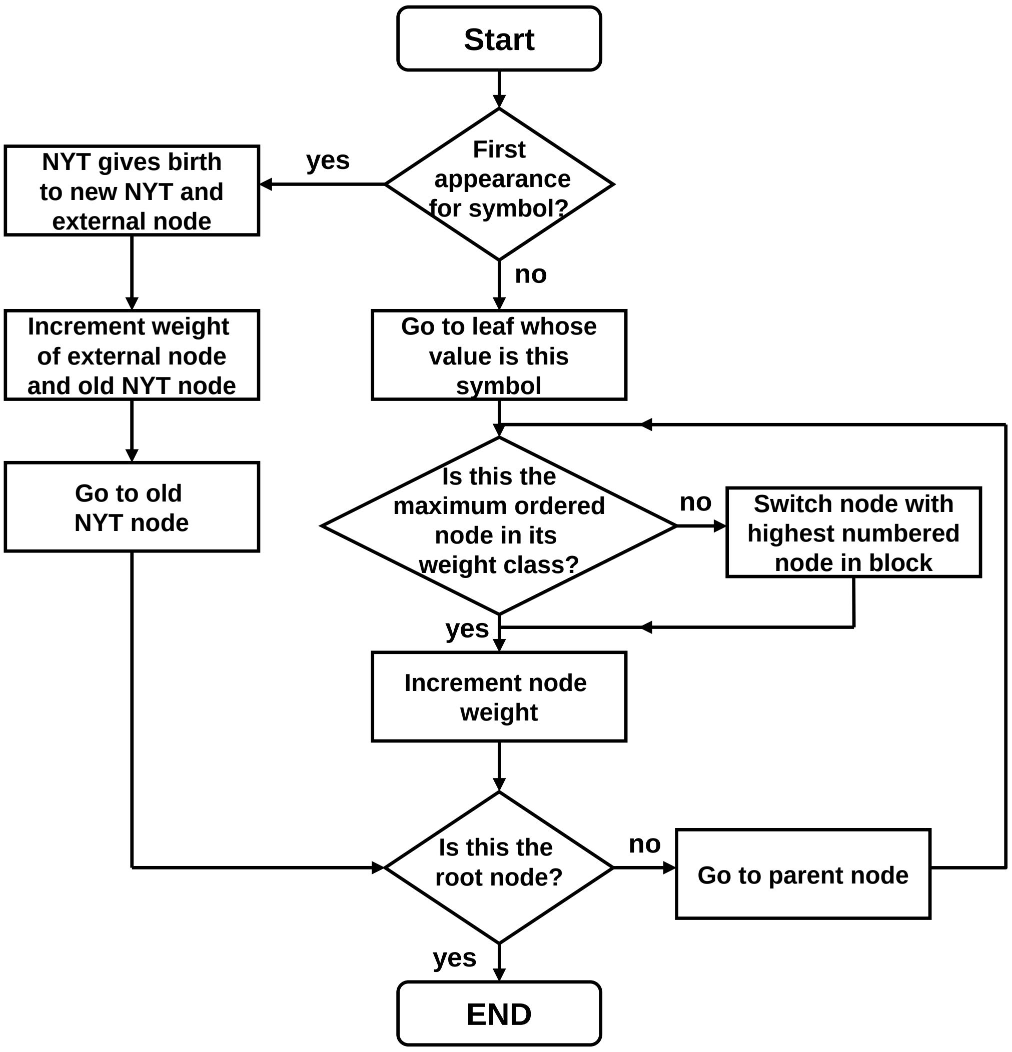

Start
First appearance for symbol?
yes
NYT gives birth to new NYT and external node
no
Increment weight of external node and old NYT node
Go to leaf whose value is this symbol
Go to old NYT node
Is this the maximum ordered node in its weight class?
no
Switch node with highest numbered node in block
yes
Increment node weight
Is this the root node?
no
Go to parent node
yes
END

Figure 3.3: The adaptive Huffman tree manipulation flowchart.

In the initialization part, text words are decomposed into maximal sequences (blocks) of vowels and consonants. The blocks of vowels will create bases of syllables. Consonants which are in the word before the first block of vowels are added to the first block. Consonants which are in the word after the last block of vowels are added to the last block.

Single algorithms of class P_U differ in the way of adding consonants, which are between two blocks of vowels. These algorithms are named for their method of addition of consonants:

Algorithm universal left P_{UL} adds all consonants between blocks of vowels to the left block.

Algorithm universal right P_{UR} adds all consonants between blocks of vowels to the right block.

Algorithm universal middle-right P_{UMR} in the case of $2n$ (even count) consonants between blocks adds to both blocks n consonants. In the case of $2n + 1$ (odd count) consonants between blocks it adds to the left block n consonants and to the right block $n + 1$ consonants.

Algorithm universal middle-left P_{UML} in case of $2n$ (even count) consonants between blocks adds to both blocks n consonants. In the case of $2n$+1 (odd count) consonants between blocks it adds to the left block n+1 consonants and to the right block n consonants.

Example 3.1. We will decompose the word *priesthood* into syllables. Blocks of vowels are: *ie*, *oo*.

`correct decomposition into syllables:`	`priest-hood`
`universal left` P_{UL}:	`priesth-ood`
`universal right` P_{UR}:	`prie-sthood`
`universal middle-left` P_{UML}:	`priest-hood (correct form)`
`universal middle-right` P_{UMR}:	`pries-thood`

3.2.2 LZWL

LZWL [14] is syllable-based version of compression algorithm LZW. It works over an alphabet of syllables obtained by any algorithm of decomposition into syllables. This algorithm can also be used for words.

In the initialization step an empty syllable and frequent syllables of the given language are added to the dictionary. In each of the following steps LZWL, in a similar manner as LZW, searches the input stream for the maximal string of syllables which the corresponds with one from the dictionary and sends indexes of the string in the dictionary to the output stream.

During compression unknown syllables can be encountered. In this case LZWL encodes unknown syllable by the unknown-syllable coding algorithm and sends to the output stream the code of the empty syllable followed by the code of unknown syllable.

The unknown-syllable coding algorithm encodes syllables as code of syllable type (there are five different types of syllables and each has special code) followed by code of syllable length and codes of individual characters of syllable.

Updating the dictionary in LZWL is analogical with character-based LZW: the first syllable of the string is appended to the previous string and the new string is added to the dictionary. Only, in contrast to LZW where the dictionary is updated in each step, with

LZWL the dictionary is updated only if no unknown syllables were detected in the current or previous step. This solution has two advantages: The first advantage is that strings are not created from syllables that appear only once. The second advantage is that we cannot receive in the decoder the code of the string that is not from the dictionary.

The algorithm for LZWL compression is as follows:

```
Initialize dictionary with empty syllable
and frequent syllables of given language
OLDSTRING = empty syllable
NEWSTRING = empty syllable
WHILE not end of input stream
      SYLLABLE = next input syllable
      IF NEWSTRING + SYLLABLE is in the dictionary
        NEWSTRING = NEWSTRING + SYLLABLE
      ELSE
        IF NEWSTRING is empty syllable
            output the code of empty syllable
            encode SYLLABLE by unknown-syllable coding algorithm
            output the code
            add SYLLABLE to the dictionary
        ELSE
            output the code for NEWSTRING
            IF OLDSTRING is not empty syllable
                SYLLABLE = first syllable of NEWSTRING
                add OLDSTRING + SYLLABLE to the dictionary
            ENDIF
        ENDIF}
      ENDIF
      OLDSTRING = NEWSTRING
ENDWHILE}
```

Decoding LZWL data is the reverse of encoding and analogous with LZW. There are only two differences: the dictionary is updated only if both this and the previously translated string are not the empty syllable, and when the translated string is the empty syllable, the unknown-syllable decoding algorithm should be applied to the input stream.

The unknown-syllable decoding algorithm is reverse of unknown-syllable encoding algorithm.

The decompression algorithm is as follows:

```
Initialize dictionary with empty syllable
and frequent syllables of given language;
WHILE not end of input stream
    CODE = get code from the input stream;
    NEWSTRING = translation of CODE
    IF NEWSTRING is not empty syllable
      output NEWSTRING
      IF OLDSTRING in not empty syllable
          SYLLABLE = first character of NEWSTRING;
          add OLDSTRING + SYLLABLE to the dictionary;
      ENDIF
    ELSE /* NEWSTRING is empty syllable */
      decode SYLLABLE by unknown-syllable decode algorithm
      add SYLLABLE to the dictionary
      output SYLLABLE
    ENDIF
    OLDSYLLABLE = NEWSYLLABLE;
ENDWHILE
```

3.2.3 HufSyl

HufSyl [14] is a statistical compression method based on adaptive Huffman coding. The HufSyl algorithm can work with syllables obtained by all algorithms of decomposition into syllables mentioned above. This algorithm can be used for words also.

For each type of syllable an adaptive Huffman tree is built which codes syllables of a given type. In the initialization step of the algorithm we add to Huffman tree for small syllables all syllables and their frequencies from database of frequent syllables.

In each step of the algorithm the expected type of actually processed `SYLLABLE` is calculated. If `SYLLABLE` has a different type than it is expected then an escape sequence is generated and `SYLLABLE` is encoded by the Huffman tree corresponding to the syllable type.

The algorithm for HufSyl compression is show below:

```
Initialize data structures
WHILE not end of input stream
    SYLLABLE = next input syllable
    EXPECTEDTYPE = expected type of SYLLABLE
    TYPE = type of SYLLABLE
    IF EXPECTEDTYPE != TYPE
        output escape sequence
    ENDIF
    if SYLLABLE is unknown
        CODE = code of NYT node in TYPE Huffman tree
        output CODE
        encode SYLLABLE by unknown-syllable coding algorithm
        output the code
        insert SYLLABLE to the TYPE Huffman tree
    ELSE
        CODE = code of SYLLABLE in  TYPE Huffman tree
        output CODE
    ENDIF
```

```
            increment weight of SYLLABLE
            if necessary reorganize the TYPE Huffman tree
        ENDWHILE
```

Reorganization of the Huffman trees in HufSyl is the same as in the adaptive variant.

The expected type of syllable is calculated from knowledge of types of previous syllables and knowledge of the structure of a sentence in natural language.

In Table 3.6 the expected types of syllables according type of previous syllable are shown.

previous / expected type of syllable	Expected syllable
small	small
capital	capital
mixed	small
number	other
other syllable without dot, last syllable from letters is not capital	small
other syllable with dot, last syllable from letters is not capital	mixed
other, last syllable from letters is capital	capital

Table 3.6: Expected types of syllables according type of previous syllable.

Chapter 4

XML compression

In this chapter, we introduce several XML specific compression technologies. We discuss their main principles and performance in the context of other compressors.

4.1 XMill

XMill [9] was developed by Hartmut Liefke and Dan Suciu at AT&T Labs. XMill is based on gzip and achieves about twice the compression rate of gzip.

XMill applies three principles to compress XML data:

- **Separate structure from data** The structure consists of XML tags and attribute names. The data consists of a sequence of items (strings) representing element contents and attribute values. Both the structure and the data are compressed separately.

- **Group data items with related meaning** Data items are grouped into containers and each container is compressed separately. For example, all `<name>` data items form one container, while all `<phone>` items form a second container.

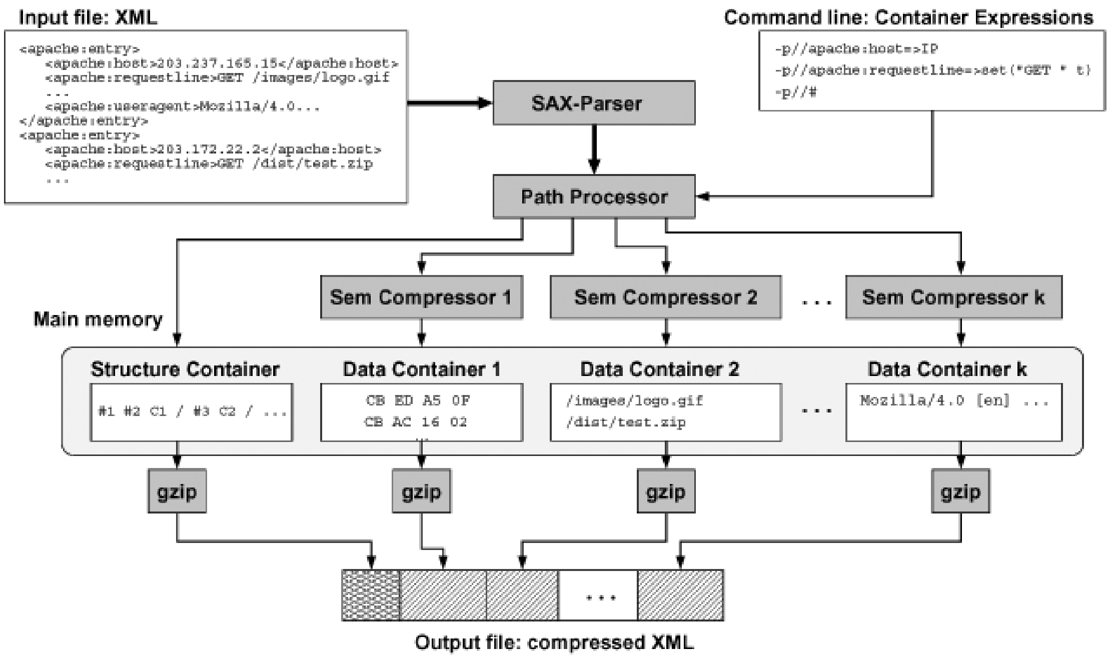
- **Apply different compressors to different containers** XMill applies different specialized compressors (semantic compressors) to different containers.

The architecture of XMill (shown in Figure 4.1) is based on the three principles discussed below. The XML document is parsed by a SAX parser. The parser then sends SAX events to the path processor. The path processor sends tags and attributes to the structure container. Data values are sent to various data containers, according to the container expressions. Finally all containers are compressed independently using gzip and sent to the output.

Before entering the container, a data value may be compressed with an additional semantic compressor.There are three types of semantic compressors available:

- Atomic compressors for the basic data types (like binary encoding of integers, differential compressors, etc.)

- Combined compressors (this is useful when data values have lexical structure, e.g. integers, separated by commas) and



```
<apache:entry>
   <apache:host>203.237.165.15</apache:host>
   <apache:requestline>GET /images/logo.gif
   ...
   <apache:useragent>Mozilla/4.0...
</apache:entry>
<apache:entry>
   <apache:host>203.172.22.2</apache:host>
   <apache:requestline>GET /dist/test.zip
   ...
```

```
-p//apache:host=>IP
-p//apache:requestline=>set("GET " t)
-p//#
```

```
#1 #2 C1 / #3 C2 / ...
```

```
CB ED A5 0F
CB AC 16 02
...
```

```
/images/logo.gif
/dist/test.zip
```

```
Mozilla/4.0 [en] ...
```

Figure 4.1: Architecture of XMill compressor

- User-defined compressors (this is useful when the XML data contains highly specialized types, like DNA sequences, for which special purpose compressors exist)

The default text semantic compressor simply copies its input to the container, without any compression.

In order to extract the structural item from the documents, XML documents are tokenized. Start-tags and attribute names are dictionary-encoded, while all end-tags are replaced by the token /. Data values are replaced with their container number. The received compact document skeleton is then stored in a structure container.

To illustrate, consider the following small XML file:

```
<paper>
  <entry year="2003">
      <journal>
          <title>Secret Sharing</title>
      </journal>
  </entry>
  <entry year="2004">
      <conference>
          <title>XML Water Mark</title>
      </conference>
  </entry>
</paper>
```

After the document is processed, the following dictionaries are created:

Element dictionary

paper	T0
entry	T1
journal	T2
title	T3
conference	T4

Attribute dictionary

year	A0

And following containers are created:

Structure container

```
T0
 T1 A0 C0
  T2
   T3 C1 /
  /
 /
 T1 A0 C0
  T4
   T3 C1 /
  /
 /
/
```

Data container

C0
2003
2004

C1
Secret Sharing
XML Water Mark

The core of XMill is the path processor that determines how to map data values to containers. The user can control this mapping by providing a series of container expressions on the command line.

Consider the following regular expression derived from XPath:

```
e ::= label | * | # | e1/e2 | e1//e2 | (e1|e2) | (e)+
```

Except for `(e)+` and `#`, all are XPath constructs: `label` is either tag or an `@attribute`, `*` denotes any tag or attribute, `e1/e2` is concatenation, `e1//e2` is concatenation with any path in between, and `(e1|e2)` is alternation. To these constructs, `(e)+` has been added, which is the strict Kleene closure. The construct `#` stands for any tag or attribute (much like `*`), but each match of `#` will determine a new container.

The container expression has the form:

```
c ::= /e | //e,
```

where `/e` matches `e` starting from the root of the XML tree while `//e` matches `e` at arbitrary depth of the tree. `//*` is abbreviated by `//`.

Example 4.1. The expression `//journal/title` creates a container for all journal's titles. `//` places all data items into a single container. The expression `//#` creates a family of containers: one for each ending tag or attribute (the default behavior of XMill).

XMill typically achieves much better compression rates than conventional compressors such as gzip.

4.2 XMLPPM

XMLPPM [4] is XML compressor based on SAX encoding and Prediction by Partial Match (PPM) encoding. It proposes a technique called Multiplexed Hierarchical Modeling (MHM), which employs two basic ideas: *multiplexing* several text compression models based on XML syntactic structure, and *injecting* hierarchical element structure symbols into the multiplexed models.

In XMLPPM, the input XML document is converted into a stream of SAX events. Element start tags, end tags, and attribute names are dictionary encoded and sent to corresponding PPM models for running predictions and encodings.

XMLPPM uses four compression models:

1. *the element and attribute name model (Syms),*
2. *the element structure model (Elts),*
3. *the attribute values model (Atts), and*
4. *the string value model (Chars).*

To illustrate the operation of XMLPPM, consider following XML fragment:

```
<elt att="abcd">XYZ</elt>
```

Assuming the tag `elt` has been seen before, and is represented by byte `10`, but attribute name `att` has not, and the next available byte for attribute name is `0D`, our XML fragment would be encoded as

Model	<elt	att=	"asdf"	>	XYZ	</elt>
Elts:	10				FE	FF
Atts:		<10> 0D	asdf 00	<10> FF		
Chars:					<10> XYZ 00	
Syms:		att 00				

XMLPPM *injects* in `Att` and `Char` models the enclosing token index `<nn>` in order to retain cross-model dependencies among the tokens in different contexts. `<nn>` indicate that a particular token has been seen, but these token indexes are not explicitly encoded in the models.

Compared to XMill, XMLPPM has the benefit of supporting the on-line processing of compressed documents. Most important is that XMLPPM does not rely on user intervention but is still able to achieve a better compression ratio than that of XMill (default mode)[2].

4.3 XGrind

XGrind [7] is a compression tool for XML documents developed by Pankaj M. Tolani, and Jayant R. Haritsa of the Indian Institute of Science in 2002. XGrind preserves the syntactic structure and semantic information of the original XML document. This implies that the compressed document can be parsed in the same way as any other XML document using the same SAX or DOM parser, without performing decompression, and allows execute queries on compressed XML documents.

XGrind uses different techniques for compressing meta-data (tags and attribute names), enumerated-type attribute values, and (general) element/attribute values. These techniques are described below:

1. **Meta-Data Compression** The method to encode meta-data is similar to that in XMill, and is as follows: Start tags and attribute names are dictionary encoded. All end-tags are encoded by `/`.

2. **Enumerated-type Attribute Value Compression** Enumerate-type attribute values are encoded using a simple encoding scheme to represent an enumerated domain of values. XGrind identifies such enumerated-type attributes by examining the DTD of the XML document.

3. **General Element/Attribute Value Compression** Element and attribute values are compressed using non-adaptive Huffman compression algorithm and each values are compressed *individually* to attain context-free coding.

To support the non-adaptive feature, two passes have to be made over the XML document: the first to collect the statistics and the second to do the actual encoding.

XGrind computes a separate frequency distribution table for each element and non-enumerated attribute. The motivation for this approach is that data belonging to the same element/attribute is usually semantically related and is expected to have similar distribution. For example, data such as telephone numbers or zip-codes will be composed exclusively of digits.

To illustrate the operation of XGrind, consider an XML document with its DTD in Table 4.2 and 4.3.

```
<STUDENT rollno = "604100418">
  <NAME>Pankaj Tolani</NAME>
  <YEAR>2000</YEAR>
  <PROG>Master of Engineering</PROG>
  <DEPT name = "Computer Science">
</STUDENT>
```

Figure 4.2: Sample XML document

A conceptual view of a compressed version of this XML document is shown in Figure 4.4. Here, `Tn` encodes a start tag, `/` encodes a closed tag, `An` encodes an attribute, and `nahuff(a)`

```
<!ELEMENT STUDENT (NAME,YEAR, PROG, DEPT)>
<!ATTLIST STUDENT rollno CDATA #REQUIRED>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT YEAR #PCDATA)>
<!ELEMENT PROG (#PCDATA)>
<!ELEMENT DEPT EMPTY>
<!ATTLIST DEPT name (Computer Science
   | Electrical Engineering
      .
      .
      .
   | Physics | Chemistry)
>
```

Figure 4.3: DDT of XML document

represents the non-adaptive Huffman code of a data value `a`. `enum(s)` denotes the output of the Enum-Encoder for an input data value `s`, which is an enumerated attribute.

```
T0 A0 nahuff(604100418)
  T1 nahuff(Pankaj Tolani) /
  T2 nahuff(2000) /
  T3 nahuff(Master of Engineering) /
  T4 A1 enum(Computer Science) /
/
```

Figure 4.4: Conceptual view of the XGrind compressed XML document

As the output compressed documents in XGrind are homomorphic transformations of their corresponding input documents, queries can be carried out over the compressed document without fully decompressing it. More precisely, exact-match (the search key is a specific data value) and prefix-match (the search key is a prefix of the data values) queries can be completely carried out directly on the compressed document, while range (the search key covers a range of data values) or 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.

XGrind provides a reasonably good compression ratio – on the average, about three-quarters that of XMill, and always at least two-thirds that achieved by XMill. Further, the compression time is always within a factor of two of that of XMill.

4.4 XPress

XPress [10] proposes an XML compressor that supports direct querying over compressed XML documents. Similar to XGrind, XPress transform an XML document into a compressed form that preserves the syntactic and semantic information of the original XML document.

```
<book>
  <author> author1 </author>
  <title> title1 </title>
  <section>
    <title> title2 </title>
    <subsection>
      <subtitle> title3 </subtitle>
          .
          .
          .
      </subsection>
    </section>
</book>
```

Figure 4.5: Sample XML document

XPress proposes a novel encoding method, called reverse arithmetic encoding, which is inspired by arithmetic encoding and uses for encoding *tree paths* of the elements.

Reverse arithmetic encoding operates as follows: The entire interval `[0.0, 1.0)` is partitioned into subintervals, one for each distinct element. An interval for element T is represented as $Interval_T$. The size of $Interval_T$ is proportional to the frequency (normalized by the total frequency) of element T. Consider XML document in Figure 4.5. The following example shows the intervals for elements in this document:

Example 4.2. Suppose that the frequencies of elements = {book, author, tile, section, subsection, subtitle} are {0.1, 0.1, 0.1, 0.3, 0.3, 0.1}, respectively. Then, based on the cumulative frequency, the entire interval [0.0, 1.0) is partitioned as follows:

element	frequency	cumulative frequency	$Interval_T$
book	0.1	0.1	[0.0, 0.1)
author	0.1	0.2	[0.1, 0.2)
title	0.1	0.3	[0.2, 0.3)
section	0.3	0.6	[0.3, 0.6)
subsection	0.3	0.9	[0.6, 0.9)
subtitle	0.1	1.0	[0.9, 1.0)

The intervals generated by reverse arithmetic encoding possess the suffix containment property. This property ensures that if an element path P is a suffix of an element path Q, then the interval that represents P, denoted as I_P, should contain the interval that represents Q, denoted as I_Q.

In Figure 4.6 is shown, how the interval [0.69, 0.699) for a simple path `/book/section/subsection` is obtained.

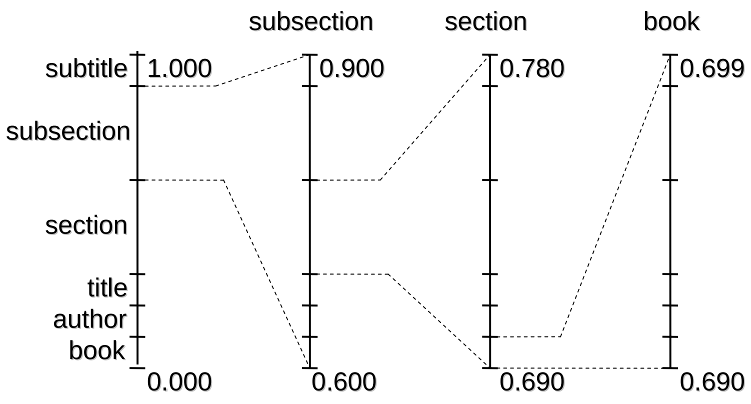


Figure 4.6: Behavior of reverse arithmetic encoding

Element and attribute values in the document are compressed individually using different context-free compression methods depending on their data types:

1. *Numerical* data values are transformed into their corresponding binary representations and differential encoding is applied on the transformed values.

2. *Enumerated-type* values are dictionary encoded.

3. *String* values are encoded using context-free Huffman encoding.

All end tags are replaced with a special symbol. Figure 4.7 shows a conceptual view of the XPress compressed XML document.

```
I(/book)
  I(/book/author) nahuff(author1) /
  I(/book/title) nahuff(title1) /
  I(/book/section)
    I(/book/section/title) nahuff(title2) /
    I(/book/section/subsection)
      I(/book/section/subsection/subtitle) nahuff(title3) /
          ...
    /
  /
/
```

Figure 4.7: Conceptual view of the XPress compressed XML document

The coding scheme in XPress improves the compression strategy adopted in XGrind in two aspects [2]. First, XPress encodes the tree paths of the elements using real number intervals that satisfy the suffixes containment property. Therefore, XPress is able to evaluate the path-based queries over compressed XML documents directly by checking the interval containment between the paths in an imposed query and the paths of the elements in the compressed document without decompression. Second, numerical domain data values in XPress compressed documents are encoded using order preserving context-free compression methods. This allows XPress to evaluate exact match and range queries concerning numerical data over compressed documents directly without decompressing the data values.

XPRESS achieves significantly improved query performance compared to XGrind and shows the reasonable compression ratio [10].

4.5 Other XML compressors

There are several other XML compressors available: XMLZip, Millau, XML-Xpress, XQueC, XQzip, XCQ, XComp. In subsequent paragraphs, the principles of the two compressors are briefly discussed.

XMLZip [2] compresses XML documents that are represented as DOM trees. XMLZip first parses XML data with a DOM parser, then breaks the structural tree into multiple components: a root component containing all data up to depth N from the root, and one component for each of the subtrees starting at depth N. The root component is then modified, references to each subtree are added onto the root, and finally components are compressed with gzip. The compression ratio achieved by XMLZip is usually worse than that achieved by other XML compressors. However, the advantage of XMLZip is that it allows limited random access to partially decompressed XML documents, since XMLZip supports decompressing the portions of the compressed components that are needed in query evaluation.

Millau [2] is an extension of the Wireless Binary XML format (WBXML). In WBXML, each element and each attribute is replaced by a compact binary token. Millau extends it with separation of the structure from the character data. The character data are compressed using conventional text compressors. In the structure data, special tokens are inserted to indicate the occurrences of compressed data.

XQueC [33] is a full-fledged XQuery query processor, which works entirely on compressed data. Like XGrind and XPress, XQueC compresses individual data items of an XML document. However, it differs from XGrind and XPress in that it separates the XML structure from the XML data items. Data items specified by the same root-to-leaf path are grouped into the same container. To support efficient query processing, XQueC constructs the structure tree of the input XML document and its structure summary that represents all distinct paths in the document. To allow efficient access of the compressed data items, XQueC links each individually compressed data item to its corresponding node in the structure tree and links each container to the corresponding path in the structure summary. Although XQueC achieves significantly improved query performance and query expressiveness, the structures might incur huge space overhead [2].

XQzip [36] is an XML compressor which supports querying compressed XML data by imposing an indexing structure, which is called the Structure Index Tree (SIT). XQzip avoids full decompression by compressing the data into a sequence of blocks which can be decompressed individually and at the same time allow commonalities of the XML data to be exploited to achieve a good compression. XQzip also uses a buffer pool for the decompressed blocks, which avoids repeated decompression in query evaluation if the data is already in the pool. With the use of the SIT index and the buffer pool, XQzip achieves a competitive querying time.

XML Compression and Querying System (XCQ) [37] is an XML compressor used for compressing XML documents that conform to a given DTD. It supports querying compressed documents without fully decompressing them. By exploiting the structural information in the input document and its associated DTD, XCQ restructures the input document into distinct data streams in a path-based manner. These data streams are then partitioned into indexed blocks that can be compressed and decompressed as an individual unit. A reasonable compression ratio, which is comparable to that of XMill, is achieved by XCQ. However, it requires longer compression and decompression times when forming the XCQ partitioned data streams.

Chapter 5

Experimental methodology

5.1 Comparative compression tools

To show the effectiveness of the new compression methods, we compare them with the existing compressors: XMill [9], LZWL and HufSyl [13], which were introduced in Chapter 3 and Chapter 4. XMill is XML-specific compression tools, LZWL and HufSyl are syllable-based compression tools.

5.2 XML data sources

The performance of the syllable-based compression tools for XML documents will be evaluated on three data sets. The first data set contains English XML documents with different inner structure. It includes regular data that has regular markup and short character data content (elts, stats, weblog, tpc). It also includes irregular data that has irregular markup (pcc, tall). The data in this set was distributed with the XMLPPM [4] and the Exalt [5] compressors.

The next two data sets contain textual XML documents, that have a rather simple structure (small amount of elements and attributes) and relatively long character data content. One of these data sets contains XML documents in English and the remainder contains documents in Czech. The documents in these sets are divided into three parts according their size: small (2-10KB), medium (10-50KB) and large (50-350KB). These data sets contain data in DocBook [16] and in RSS [19] format as well as five stage plays marked up as XML in English and one stage play in Czech. Some documents in these sets were distributed with the XMLPPM [4] and the Exalt [5] compressors, others were found on Internet.

5.3 Compression performance metrics

The compression ratio is expressed as the number of bits required to represent a byte and is defined as follows:

$$CR = \frac{sizeof(compressed\,file) \times 8}{sizeof(original\,file)}\,bits/byte.$$

We compare the compression ratios of syllable-based compression methods for XML (denoted as XSyl) with those of XMill, LZWL and HufSyl.

$$CRF_{XMill} = \frac{CR_{XSyl}}{CR_{XMill}},$$

normalizes the compression ratio of XSyl with respect to XMill.

$$CRF_{LZWL} = \frac{CR_{XSyl}}{CR_{LZWL}},$$

normalizes the compression ratio of XSyl with respect to LZWL.

$$CRF_{HufSyl} = \frac{CR_{XSyl}}{CR_{HufSyl}},$$

normalizes the compression ratio of XSyl with respect to HufSyl.

Chapter 6

XMLSyl

In this chapter, we introduce our XML compressor: XMLSyl. We present the motivation and the principles on which XMLSyl is built. We discuss the architecture of it and describe how it works by giving a compression example in detail. Finally, we present the results of our experiments that compare XMLSyl with XMill, LZWL and HufSyl.

6.1 Motivation

XML tokens (elements and attributes) inflate XML documents. They make documents larger in size than other specifications containing the same data content. To achieve good compression results, we attempted to minimize XML tokens size.

The existing syllable-based compressors divide XML tokens into many syllables and hence the size of the XML token is not minimized. In our method, we replace XML tokens with single bytes in the input document and add them to the syllable dictionaries. It causes the syllable-based compressor to treat XML tokens as single syllables and assign them the shortest possible codes during syllable-based compression.

The encoding of XML tokens is inspired by existing XML compression methods like XMLPPM [4], XGrind [7], XPress [10], XMill [9].

6.2 Architecture and principles of XMLSyl

The architecture of XMLSyl is shown in Figure 6.1. It has four major modules: the *SAX Parser*, the *Structure Encoder*, the *Containers* and the *Syllable Compressor*.

First, the XML document is sent to the SAX Parser. Next the parser decomposes the document into SAX events (start-tags, end-tags, data items, comments and etc.) and forwards them to the Structure Encoder. The Structure Encoder encodes the SAX events and routes them to the different Containers. There are three containers in our implementation:

1. **Element Container:** The Element Container stores the names of all elements that occur in an XML document. The Structure Encoder also uses the Element Container as the dictionary for encoding XML structure.

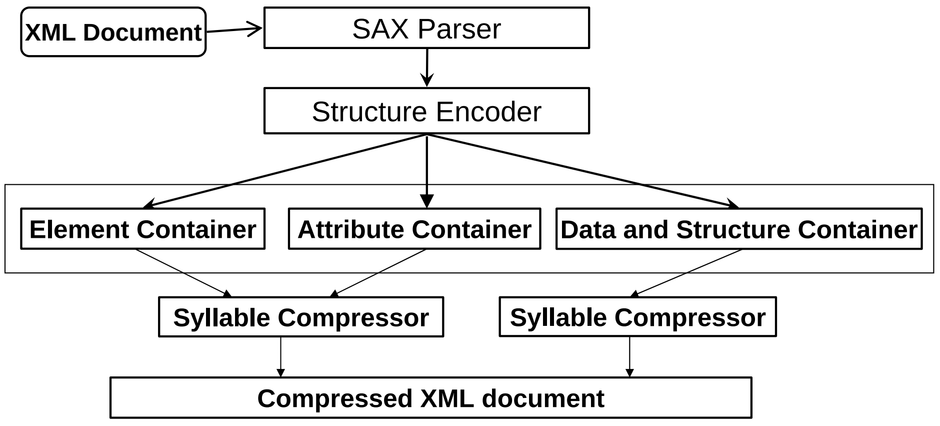


Figure 6.1: The Architecture of XMLSyl Compressor

2. **Attribute Container:** The Attribute Container stores the names of all attributes which occur in an XML document. The Structure Encoder also uses the Attribute Container as the dictionary for encoding XML structure.

3. **Structure and Data Container:** The Structure and Data Container stores an XML document, in which all meta-data are replaced with special codes. The encoding process is presented in Section 6.4.

When a document is parsed and separated into the containers, the contents of the containers are sent to the Syllable Compressor. It compresses the content of each container separately using syllable-based compression and sends the result to the output.

The decompression process is the reverse of the compression process. The architecture of the XMLSyl decompressor is shown in Figure 6.2. The decompressor has three major modules: the *Syllable Decoder*, the *Containers* and the *Structure Decoder*. The compressed document is decoded with the Syllable Decoder and decomposed into three parts. From the first two parts the Element and Attribute Dictionary is created. The third part is sent to the Data and Structure containers. The Structure Decoder decodes XML tokens using the dictionaries and send the result to the output.

6.3 Parser

The compression process begins with the XML parser. We did not write the parser by ourselves, rather we use the Expat parser. The Expat is a open-source SAX parser, written in C++.

The Expat works in the following way: Callback (or handler) functions are registered with the parser. Then the document is fed into the parser. As the parser recognizes *SAX event* of the document, it will call the appropriate handler for that event. The document is fed to the parser in pieces, so it is possible to start parsing before all the document is available. This also allows to parse very large documents that will not fit into memory.

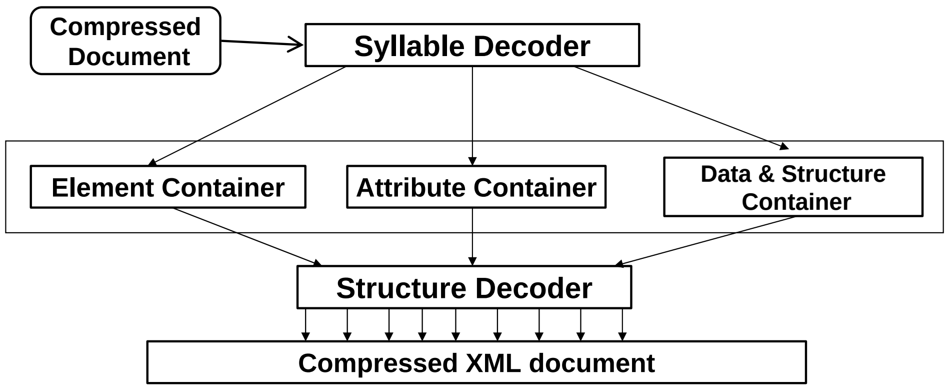


Figure 6.2: The Architecture of XMLSyl Decompressor

We decided to use the Expat because of its ease of use, its speed and also its possibility to parse large documents.

One disadvantage of the Expat is it supports few character encodings. There are four built-in encodings in the Expat:

- UTF-8
- UTF-16
- ISO-8859-1
- US-ASCII

For other encodings, the Expat calls the `UnknownEncodingHandler`. Users have to define this handler to deal with unknown encoding. Our `UnknownEncodingHandler` converts data from the original encoding to `UTF-8` encoding.

6.4 Encoding the structure of XML document

The structure of the XML document is encoded in XMLSyl as follows. Whenever a new element or attribute is encountered, its name is sent to the dictionary and its index in the dictionary is sent to the Data and Structure Container. Two different dictionaries are used for attributes and elements: the Element Dictionary and the Attribute Dictionary. The Attribute Container operates as the Attribute Dictionary and the Element Container as the Element Dictionary. Whenever an end tag is encountered a token `END_TAG` is sent to the Data and Structure container. Whenever a character sequence is encountered, it is sent to the Data and Structure Container without changes. Start and end of character sequences are indicated by special tokens. We distinguish four different character sequences: value of attribute, value of element, comment, and white spaces between tags, if white spaces are preserved.

To illustrate the encoding process, consider the encoding of the following small XML document:

```
<book>
  <title lang="en">XML</title>
  <author>Brown</author>
  <author>Smith</author>
  <price currency="EURO">49</price>
</book>
<!-- Comment-->
```

First, the XML document is converted into a corresponding stream of SAX events:

```
startElement("book")
startElement("title",("lang","en"))
characters("XML")
endElement("title")
startElement("author")
characters("Smith")
endElement("author")
startElement("author")
characters("Brown")
endElement("author")
startElement("price","currency","EURO")
characters("49")
endElement("price")
endElement("book")
comment("Comment")
```

The tokens in the SAX event stream are sent to the Structure Encoder. It encodes them and sends to their corresponding containers. When the start element *book* is encountered, the string value *"book"* is sent to the Element Container. An index `E0` is assigned to this entry. This index is sent to the Data and Structure Container. The same operation is executed for the element *title*. The attribute *lang* is encoded using the Attribute Container: the attribute name *"lang"* is sent to the Attribute Container and the index `A0` is assigned to it. The index `A0` is sent to the Data and Structure Container. The attribute value *"en"* and the token `END_ATT` are sent to the Data and Structure Container. The token `END_ATT` signals the end of the attribute value. When an element value *"XML"* is encountered, the token `CHAR`, the data value and then the token `END_CHAR` are sent to the Data and Structure Container. When a comment event is encountered, the code `CMNT` is put into the Data and Structure Container. The comment value is sent to the container and is enclosed by `END_CMNT` code. The final state of all containers is shown in Figure 6.3.

In this example we have ignored white spaces between tags, e.g. `<book>` and `<title>`, so the decompressor will produce a standard indentation. Optionally, XMLSyl can preserve the white spaces. In that case, it stores the white spaces as the sequence of characters in the Data and Structure Container between tokens `WS` and `END_WS`. In XMLSyl additional XML information like DTD, CDATA, Procession Instruction is also compressed as the common sequence of characters. It stores in the Data and Structure Container between tokens `CHAR` and `END_CHAR`. Only XML declaration is parsed.

Element Container

element	index
book	E0
title	E1
author	E2
price	E3

Attribute Container

attribute	index
lang	A0
currency	A1

Data and Structure Container

`<book>`	`<title`	`lang="en">`	XML	`</title>`	`<author>`
E0	E1	A0 en `END_ATT`	`CHAR` XML `END_CHAR`	`END_TAG`	E2

Brown	`</author>`	`<author>`	Smith	`</author>`	`<price`
`CHAR` Brown `END_CHAR`	`END_TAG`	E2	`CHAR` Smith `END_CHAR`	`END_TAG`	E3

`currency="EURO">`	49	`</price>`	`</book>`	`<!--Comment-->`
A1 Euro `END_ATT`	`CHAR` 49 `END_CHAR`	`END_TAG`	`END_TAG`	`CMNT` Comment `END_CMNT`

Figure 6.3: Content of containers

6.5 Containers

The containers are the basic units for grouping XML data. The Attribute Container holds attribute names and the Element Container holds element names. Since the number of all element and attribute names in any XML document is not high, these two containers are kept in main memory. During parsing, the container's size increases as the container is filled with entries. Each entry in the Element container is assigned a byte in the range `04-AE`. These bytes are used for encoding the element names. Each entry in the Attribute container is assigned a byte in the range `AF-FE`. These bytes are used for encoding the attribute names. The residual 6 bytes are reserved for special codes like `CHAR`, `END_TAG`. In most cases, 170 (or 80) bytes are enough to encode element (or attribute) names. If the number of elements (or attributes) is greater than 170 (or 80), entries are encoded with two bytes (ESC-symbol and byte from corresponding range) then three bytes and so on.

There is another situation with the Data and Structure Container. We do not know the size of the input XML document. The size of XML document can be so big that document will not fit into memory and it is not possible to increase the size of container endlessly. Therefore, the container consists of two memory blocks of constant size. The content of the first memory block is compressed as soon as the container is filled. We don't compress two blocks at once because the content of the second memory block is used for compression of the first one. After the compression, the compressed content of the first block is sent to the output and the first block swaps its purpose with the second one. Now the first block is filled with data. When it is full, the second block is compressed, and so on.

6.6 Syllable Compressor

The Syllable Compressor compresses the content of the Structure and Data Container and sends the result to the output file. Then the content of the Attribute Containers and the content of the Element Container are compressed and sent to the output file.

The data are compressed with an existing syllable-base compressor: LZWL or HufSyl. The user can set up which compressor will be used, from the command line. XMLSyl in combination with LZWL is denoted as XMLzwl and XMLSyl in combination with HufSyl is denoted as XMLHuf.

We did not write syllable-based compressors by ourselves, instead using existing sources of LZWL and HufSyl, which was slightly. The syllable-based compressors were slightly modified to be able to work with the containers of XMLSyl implementation instead of a file stream and compress the content of container in pieces in case of compressing the large document.

As LZWL and HufSyl can perform word-based compression, XMLSyl has this ability as well.

6.7 Syllable Dictionaries

In the initialization step, LZWL [13] initializes the syllable dictionary with an empty syllable and frequent syllables of the given language. To improve compression of XML documents, we add to the syllable dictionary also the XML token codes in the initialization step. This causes the syllable-based compressors to encode XML token as already known syllables, i.e. XML tokens are received the shortest possible codes.

6.8 Conversion between different character encoding

After the Exalt parses the document, all character sequences are converted to UTF-8. It does not depend on the input encoding. XMLSyl converts the character sequenced back to the original encoding using **libiconv** library [35].

If the input XML document is in Czech, after parsing it is converted to Windows-1250 encoding, because the existing syllable-based tools compress correctly only the documents in this encoding.

6.9 Comparison experiments

In this section, we evaluate the compression results achieved by XMLSyl, and compare its performance to the syllable-based compressors LZWL, HufSyl and the XML compressor XMill. Also we compare the word-based and syllable-based versions of XMLSyl.

6.9.1 Performance of XMLzwl

The compression ratio statistics of XMill, LZWL and XMLzwl over all data sets (see Chapter 5) are shown in Table 6.1.

Compared to LZWL, XMLlzwl exhibits a very good performance, especially on the first data set: the compression ratio is about 34% better than that of LZWL. On the second and the third data sets, the compression ration is about 5-10% better than that of LZWL.

		CR_{XMLzwl}	**CRF_{XMill}**	**CFR_{LZWL}**
1 set	all	0,67	**1,65**	**0,66**
2 set	all	2,99	**1,13**	**0,94**
	small	3,80	**1,17**	**0,95**
	medium	2,88	**1,17**	**0,94**
	large	1,90	**1,03**	**0,93**
3set	all	3,59	**1,15**	**0,93**
	small	3,89	**1,18**	**0,91**
	medium	3,34	**1,14**	**0,96**
	large	2,66	**1,07**	**0,97**

Table 6.1: Performance of XMLzwl

The reason of this sharp difference between the compression ratios is that the performance of XMLSyl is dependent on the amount of markup in the compressed document. The documents in the first data set have high markup percentages, while the other documents have relatively low percentages of markup.

Compared to XMill, XMLlzwl exhibits a poor performance on the first data set. XMLzwl compresses about 65% worse than XMill. On the second and the third data sets, XMLzwl yields much better results. It is, on average, only 14-18% worse than XMill on the small and the medium documents. On the large XML documents, the compression ratio of XMLzwl is very close to that of XMill. On some documents XMLzwl even outperformed XMill, specifically on the English language stage play documents. The performance of XMLzwl on English large documents is shown in detail in the Table 6.2.

size	CR_{XMill}	CR_{XMLzwl}	**CRF_{XMill}**	**CFR_{LZWL}**	**Description**
54765	2,15	2,39	**1,09**	**0,91**	"DocBook: The Definitive Guide" in DocBook format (1)
56440	2,34	2,80	**1,18**	**0,95**	Technology news for enterprise IT from InfoWorld
67142	1,70	1,99	**1,12**	**0,88**	"DocBook: The Definitive Guide" in DocBook format (5)
136807	2,10	2,02	**0,98**	**0,96**	"The Comedy of Errors" marked up as XML
160728	1,32	1,51	**1,07**	**0,86**	"DocBook: The Definitive Guide" in DocBook format (3)
217542	2,37	2,47	**1,03**	**0,99**	"ROMEA - Romany Information Service"
220497	1,84	1,70	**0,94**	**0,95**	"Much Ado about Nothing" marked up as XML
256393	2,08	1,88	**0,93**	**0,94**	"The Tragedy of Antony and Cleopatra" marked up as XML
314677	1,95	1,80	**0,94**	**0,96**	"The Tragedy of Hamlet, Prince of Denmark" marked up as XML

Table 6.2: Performance of XMLzwl on large data

The poor compression performance of XMLzwl on the first data set was expected and can be explained. XMill utilizes the structure information in order to restructure the XML document, which makes it more amenable to compression. This strategy is very fruitful on non-textual documents (which have more data and less text). XMLSyl does not utilize the structure information, it attempt only to reduce markup.

The word-based version performs worse than the syllable-based version on the documents in Czech. On the documents in English, the compression ratio of two versions are very close to each other. These results are similar to that of LZWL [13].

6.9.2 Performance of XMLhuf

The compression ratio statistics of XMill, HufSyl and XMLhuf are shown in Table 6.3.

		CR_{XMLhuf}	$\mathbf{CRF_{XMill}}$	$\mathbf{CFR_{HufSyl}}$
1 set	all	0,93	**2,64**	**0,37**
2 set	all	3,14	**1,21**	**0,85**
	small	3,65	**1,13**	**0,88**
	medium	3,18	**1,27**	**0,86**
	large	2,20	**1,20**	**0,79**
3set	all	3,57	**1,16**	**0,85**
	small	3,63	**1,10**	**0,83**
	medium	3,54	**1,22**	**0,91**
	large	3,06	**1,24**	**0,85**

Table 6.3: Performance of XMLhuf

Similar to XMLzwl, XMLhuf yields significant improvement compared to HufSyl on all data sets. It performed about 63% better on the first data set and about 15% better on the second and third data sets. This improvement is even better than that of XMLzwl compared to LZWL.

Compared to XMill, XMLhuf exhibits a poor performance on the first data set and sufficiently good on other sets. In contrast to XMLzwl, XMLhuf performed better on the small textual XML documents and worse on the medium and the large documents. The compression ratio of XMLhuf is on average 20-27% worse than that of XMill on the medium and the large documents and 10-15% worse on the small documents.

Compared to the syllable-based and the word-based versions of XMLhuf, the syllable-based version achieves expectedly worse results than the word-based version in both languages. This results are similar to that of HufSul [14].

Chapter 7

XMillSyl

In this chapter we introduce our second syllable-based compressor for XML: XMillSyl. First we present motivation, then implementation. At last we present the results of our experiments that compare XMillSyl with XMill, LZWL and HufSyl.

7.1 Motivation

There are several XML compression method available. We scrutinized closely existing XML compression methods (all this methods are described in the Chapter 4) and decided to incorporate one of these methods with syllable-based compressor.

The queriable XML compression methods are not suitable for our purpose. The queriable methods preserve the syntactic structure and semantic information of the original XML document. They compress individual data items (XGrind, XPress, XQuec) or decompose the data into a sequence of blocks which are compressed individually (XQzip, XCQ). It allows querying over compressed XML documents without full decompression. However, this makes it impossible to incorporate these compressors with the syllable-base method. The syllable-base method is the context method and oriented on data of sufficiently large size.

Other XML compressors use the principles that were for the first time implemented in XMill: separating structure from data and grouping data items based on semantic. This principles are very successful and XMill achieves very good results. Therefore we decided to incorporate the syllable-base method with XMill. We called this method XMillSyl.

We do not suppose that XMillSyl method achieves better results than XMill because XMill based on gzip and gzip performs better than syllable-based compressors. We have implemented XMillSyl in order to examine the power of the main principles of XMill.

7.2 Implementation

We did not write the XMill compressor. We decided to use existing sources of XMill.

XMill operates as follows: a SAX parser parses the XML file and the SAX events are sent to the core module of the XMill called the path processor. It determines how to map tokens to containers: element and attribute names are encoded and sent to the structure container, while the data values are sent to various data containers, according to their semantic. Finally, the containers are gzipped independently and stored on the disk.

In XMillSyl the containers are compressed with LZWL (XMillzwl) or with HufSyl (XMill-huf). The architecture of XMillSyl is shown in Figure 7.1

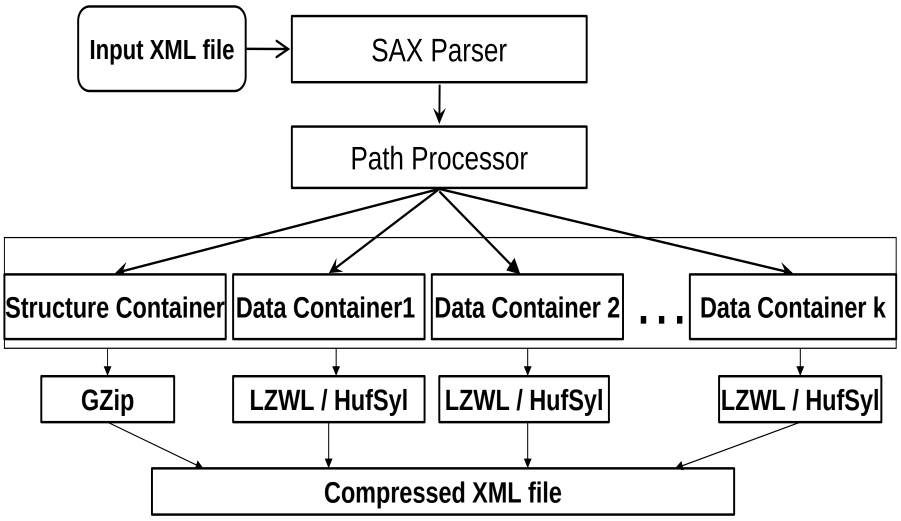


Figure 7.1: Architecture of XMillSyl

We have modified XMill compression and decompression functions to be able to compress and decompress the data containers with the syllable-based compressors (see Figure 7.1). We have also modified the syllable-based compressors so that they can work with the containers of XMill implementation instead of a file stream.

As LZWL and HufSyl can perform word-based compression, XMLSyl has this ability as well.

7.3 Comparison Experiments

In this section, we evaluate the compression results achieved by XMillSyl (both XMillzwl and XMillhuf) and compare its performance with LZWL, HufSyl and XMill. We also compare the word-based and the syllable-based version of XMLSyl. Finally, performances of XMillSyl and XMLSyl are compared.

Tables 7.1 and 7.2 show the compression ratio statistics of XMillzwl and XMillhuf respectively.

XMillSyl yields significant improvements compared to LZWL and HufSyl on the first data set. On the second and the third data sets, the compression ration of XMillhuf is about 5-15% better than that of HufSyl, but XMillzwl performs worse than LZWL. LZWL performance gets worse with the decrease in data size. XMillSyl splits data into containers and then compresses it with LZWL. This makes the performance of XMillzwl worse and XMillzwl achieves even worse results than LZWL.

		CR_{XMLzwl}	**CRF_{XMill}**	**CFR_{LZWL}**
1 set	all	**0,69**	**1,56**	**0,66**
2 set	all	**3,31**	**1,28**	**1,05**
	small	**4,32**	**1,36**	**1,07**
	medium	**3,19**	**1,29**	**1,03**
	large	**2,31**	**1,18**	**1,05**
3set	all	**3,98**	**1,29**	**1,03**
	small	**4,37**	**1,34**	**1,02**
	medium	**3,63**	**1,26**	**1,05**
	large	**2,80**	**1,15**	**1,03**

Table 7.1: Performance of XMillzwl

Compared with XMill, XMillSyl compress about 50-80% worse on the first data set and about 15-35% on the second and the third data sets.

The word-based version of XMillzwl performs worse than the syllable-based version on the documents in Czech. On the documents in English, the compression ratio of the two versions are very close to each other. Compared to the syllable-based and the word-based versions of XMillhuf, the syllable-based version achieves worse results than the word-based in both languages.

		CR_{XMLhuf}	**CRF_{XMill}**	**CFR_{HufSyl}**
1 set	all	**0,71**	**1,83**	**0,29**
2 set	all	**3,27**	**1,28**	**0,88**
	small	**3,96**	**1,25**	**0,95**
	medium	**3,20**	**1,29**	**0,86**
	large	**2,56**	**1,29**	**0,84**
3set	all	**3,78**	**1,24**	**0,90**
	small	**3,93**	**1,20**	**0,89**
	medium	**3,86**	**1,31**	**0,95**
	large	**3,06**	**1,25**	**0,85**

Table 7.2: Performance of XMillhuf

Compared to XMLSyl, XMillSyl outperforms XMLSyl on the first data set. On all textual XML documents XMLSyl yields better performance then XMillSyl.

We have also compared the performance of XMillSyl in default mode (grouping data items based on semantic) and in the no-grouping mode (option `-p //`). On the textual XML documents, the XMillSyl in default mode yields worse results than XMillSyl in no-grouping mode.

We can conclude from these experimental results that the utilization of structure information in the input document in order to restructure the document has no effect in combination with the syllable-based compression. On several documents the compression ratio is even worse than that of single syllable-based compression.

Chapter 8

Conclusions and further work

In this thesis, we proposed two syllable-based compression methods for XML data, called XMLSyl and XMillSyl. We presented the architecture and implementation of these methods and tested their performance on a variety of XML documents. XMLSyl and XMillSyl were compared with LZWL, HufSyl and XMill.

Our experimental results indicate that XMLSyl provides a reasonably good compression ratio compared to XMill on the textual XML documents and on some data it even outperforms XMill. Since XMill is based on gzip, and gzip outperforms the syllable-based compression, these results validate XMLSyl as very successful syllable-based XML compression method.

In contrast, our experimental results validate XMillSyl as an unsuccessful method. It performs worse than XMLSyl, and on some data even worse than the single syllable-based compressor. Moreover, on textual XML documents the compression ratio of XMillSyl in no-grouping mode (all data items are placed into single container) is better than that of XMillSyl in grouping mode (data items are grouped based on semantic). Thus, we can conclude that the reduction of markup size by treating it as single syllable is more efficient than separating structure from data and grouping semantically related data values.

At the beginning we supposed that our methods will be suitable for XML documents in languages with reach morphology (Czech). But we can not conclude from our experiments if our methods are more suitable for English or Czech language. The documents in our data sets are of very different structures so it difficult to come to a conclusion.

We also supposed that the syllable-based compression for XML documents will be suitable for small or middle-sized documents. Our experiments show that the XMLhuf is more suitable for the small-sized files while XMLzwl for the large-sized files.

In the future we would like to implement some modifications to enhance the compression ratio of XMLSyl. We plan to extract and utilize the information in the DTD section, create a special syllable dictionary for elements and attributes in XMLhuf. We also plan modify this method, that it can compress HTML data.

On the base of this thesis was created the article K. Chernik, J. Lánský, L. Galamboš: Syllable-based Compression for XML Documents [1].

Bibliography

[1] K. Chernik, J. Lánský, L. Galamboš: Syllable-based Compression for XML Documents. *V: V. Snášel, K. Richta, J. and Pokorný: Proceedings of the Dateso 2006 Annual International Workshop on DAtabases, TExts, Specifications and Objects.*, 2006

[2] Wilfred Ng, Lam Wai, Yeung James Cheng. Comparative Analysis of XML Compression Technologies. World Wide Web Journal, 2005

[3] Smitha S. Nair. XML Compression Techniques: A Survey.
www.cs.uiowa.edu/~rlawrenc/research/Students/SN_04_XMLCompress.pdf

[4] J. Cheney. Compressing XML with Multiplexed Hierarchical PPM Models *In Proc. Data Compression Conference*, 2001.

[5] V. Toman. Compression of XML Data. MFF UK, 2003

[6] World Wide Web Consorcium. Extensive Markup Language (XML) 1.0.
http://www.w3.org/XML/

[7] P. Tolani, J. R. Haritsa. XGrind: A Query-friendly XML Compressor. *In Proc. IEEE International Conference on Data Engineering*, 2002.

[8] SAX: A Simple API for XML.
http://www.saxproject.org

[9] H. Liefke, D. Suciu. XMill: an Efficient Compressor for XML Data. *In Proc. ACM SIGMOD Conference*, 2000.

[10] Jun-Ki Min, Myung-Jae Park, Chin-Wan Chung, XPRESS: A Queriable Compression for XML Data *SIGMOD 2003, June 912, 2003, San Diego, CA*, 2000.

[11] Expat XML Parser.
http://expat.sourceforge.net

[12] T. A. Welch. A technique for high performance data compression. *IEEE Computer*, 1984.

[13] J. Lánský, M. emlička. Text Compression: Syllables. *DATESO*, 2005.

[14] J. Lánský, Slabiková komprese. MFF UK, 2005

[15] V. Toman. Komprese XML dat.
http://kocour.ms.mff.cuni.cz/~mlynkova/prg036/

[16] J. Kosek. Inteligentní podpora navigace na WWW s využitím XML.
http://www.kosek.cz/diplomka/, 2002

[17] DocBook http://www.docbook.org/

[18] A Quick Introduction to XML.
http://www.cellml.org/tutorial/xml_guide

[19] M. Pilgrim. What Is RSS.
http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html

[20] XML Processing.
http://diveintopython.org/xml_processing/

[21] SAX And DOM Overview.
http://www.jezuk.co.uk/cgi-bin/view/arabica/SAXandDOMIntro

[22] The gzip home page.
http://www.gzip.org/

[23] A Quick Introduction to XML.
http://www.cellml.org/tutorial/xml_guide

[24] Introduction to XML.
http://java.sun.com/j2ee/1.4/docs/tutorial/doc/IntroXML2.html

[25] XML Processing.
http://diveintopython.org/xml_processing/

[26] SAX + DOM Mix = SAXDOMIX.
http://www.devsphere.com/xml/saxdomix/index.html

[27] What is the Document Object Model?
http://www.w3.org/TR/DOM-Level-2-Core/introduction.html

[28] LZW
http://www.netnam.vn/unescocourse/computervision/106.htm

[29] LZW, From Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/Lempel-Ziv

[30] Data Compression.
http://www.ics.uci.edu/~dan/pubs/DC-Sec3.html

[31] Huffman Coding: A CS2 Assignment.
http://www.cs.duke.edu/csed/poop/huff/info/

[32] Adaptive Huffman Coding.
http://www.cs.duke.edu/csed/curious/compression/adaptivehuff.html

[33] XQueC Project.
http://www.icar.cnr.it/angela/xquec/

[34] Millau: an encoding format for efficient representation and exchange of XMLover the Web.
http://www9.org/w9cdrom/154/154.html

[35] Introduction to libiconv.
http://www.gnu.org/software/libiconv/

[36] J. Cheng and W. Ng. XQzip: Querying Compressed XML Using Structural Indexing. *In Proc. EDBT*, 2004.

[37] W. Y. Lam, W. Ng, P. T. Wood, and M. Levene. XCQ: XML Compression and Querying System. *Poster Proceedings, 12th International World-Wide Web Conference (WWW2003)*, 2003.