Syllable-Based Burrows-Wheeler Transform*

Jan Lánský, Katsiaryna Chernik, and Zuzana Vlčková

Charles University, Faculty of Mathematics and Physics
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
{zizelevak, kchernik, zuzana.vlckova}@gmail.com

Abstract. The Burrows-Wheeler Transform (BWT) is a compression method which reorders an input string into the form, which is preferable to another compression. Usually Move-To-Front transform and then Huffman coding is used to the permutated string. The original method [3] from 1994 was designed for an alphabet compression. In 2001, versions working with word and n-grams alphabet were presented. The newest version copes with the syllable alphabet [7]. The goal of this article is to compare the BWT compression working with alphabet of letters, syllables, words, 3-grams and 5-grams.

1 Introduction

The Burrows-Wheeler Transform is an algorithm that takes a block of text as input and rearranges it using a sorting algorithm. The output can be compressed with another algorithms such as bzip. To compare different ways of document parsing and their influence on BWT, we decided to deal with natural units of the text: letters, syllables and words; and compare these approaches with each other and also with the methods that divide the text into unnatural units – N-grams. In our measurements we found out that depending upon the language, words have 5–6 letters and syllables 2–3 letters in average. Therefore 3-grams were chosen to conform to the syllable length and 5-grams to correspond to average words length.

If we want to compare different BWT for various ways of text file parsing, it is necessary to use an implementation modified only in document parsing; the rest of compression method will stay unchanged.

Very important BWT parameter is a block size the document is compressed into; the larger block, the better results. For example, in bzip2 algorithm [14], the maximum block size is 900 kB. We decided to choose the block size so that any document up to 5MB could be considered as a single block. This approach is fairly-minded since e.g. word methods will not be favoured by considering the document as one block whilst letter methods would split the text into several blocks.

For our purposes, we had to alter two method's properties: We modified the XBW method [3] to be able to compress above all required entities. The XBW

* This research was partially supported by the Program "Information Society" under project 1ET100300419.

transform represents a labeled tree using two arrays, and supports navigation and search operations by means of standard query operations on arrays. Since this method was designed for syllable and word compression, it was easy to adjust it to compress also above remaining entities (letters, 3-grams and 5-grams). Since this method was intended as XML compression algorithm, we had to suit it on plain text documents - it was the second modification.

XBW method does not use syllable or word models attributes which predict a word or syllable type alternation.

In section 2 we try different strategies of document parsing. Section 3 details XBW method, section 4 describes experiments and their results. Section 5 contains the conclusion.

2 Parsing strategies

We parse an input document into words, syllables, letters, 3-grams and 5-grams. All above mentioned entity type division examples of string "consists of" are introduced in the table 1.

The word-based methods require parsing the input document into a stream of words and non-words. Words are usually defined as the longest alphanumeric strings in the text while non-words are the remaining fragments.

In [17] the syllable is defined as: "a unit of pronunciation having one vowel sound, with or without surrounding consonants, and forming all or part of a word." We do not need to follow this definition strictly. Therefore we will use simplified definition [11]: "Syllable is a sequence of speech sounds, which contains exactly one vowel subsequence." Consequently, the number of syllables in certain word is equal to the number of word's maximum sequences of vowels.

Example 1. For example, the word "wolf" contains one maximal vowel subsequence ("o"), so it is one syllable; while word "ouzel" consists of two maximal vowel sequences - "ou" and "e" – there are two syllables, "ou" and "zel".

N-grams are sequences of n letters, then 3-gram contains 3 letters, 5-gram is created by 5 letters, 1-gram is one letter.

Table 1. Examples of string parsing into words, syllables, letters, 3-grams and 5-grams

parsing type	parsed text elements
orginal	`"consists of"`
letters	`"c", "o", "n", "s", "i", "s", "t", "s", " ", "o", "f"`
3-grams	`"con", "sis", "ts ", "of"`
5-grams	`"consi", "sts o", "f"`
syllables	`"con", "sists", " ", "of"`
words	`"consists", " ", "of"`

3 XBW

We designed an XBW method based on the Burrows-Wheeler transform [3]. This XBW method was partially described in [7]; in this paper, we give a more detailed description.

The XBW method was not named very appropriately, because it can be easily mistaken by name xbw used by the authors of paper [5] for XML transformation into the format more suitable for searching. In another article these authors renamed the transformation from xbw to XBW. Moreover they used it in compression methods called XBzip and XBzipIndex.

XBW method we designed consists of these steps: 1. replacement of the tag names, 2. division into the elements (words, syllables or n-grams), 3. encoding of the dictionary of used elements [10], 4. Burrows-Wheeler transform (BWT), 5. Move to Front transform (MTF), 6. Run Length Encoding of null sequences (RLE), and 7. Canonical Huffman. The steps are all described also by examples. Our tests were performed on plain texts, therefore we will not detail this method with XML support.

3.1 Replacement of the tag names

SAX parser produces a sequence of SAX events processed by a structure coder. This coder builds up two separate dictionaries for elements and attributes of encoding.

If a corresponding dictionary contains the processed element or attribute, it is substituted by an assigned identifier. Otherwise it will be substituted by the lowest available identifier and put into the proper dictionary. Moreover the name of the element or attribute will be written to the output just after the new identifier. It ensures that the dictionaries do not need to be coded explicitly and can be reconstructed during the extraction using the already processed part.

Example 2. Suppose the input

```
<note importance="high">money</note>
<note importance="low">your money</note>
```

Then the dictionaries look like

Element dictionary		Attribute dictionary	
EA	End-of-Tag	E1	importance
A1	note	E2	*empty*

and output is *A1 E1 high EA money EA A1 E1 low EA your money EA.*

3.2 Division into words or syllables

Output of the previous step is divided into words or syllables as described in [7]. The coder then creates a dictionary of basic units (syllables or words). In this phase, the coder creates a syllable (word) dictionary. If the dictionary contains a processed basic unit, it is substituted by its identifier. Otherwise, it is added to the dictionary and assigned a unique identifier. Then all occurrences in the code are replaced by this identifier. Resulting stream is denoted as S-stream. This part has two outputs: the S-stream and the dictionary of used basic units. The dictionary has to be also encoded because of the document reconstruction.

Example 3. Let us continue in the previous example where the syllables in the dictionary could have the following associations: 01 – high, 02 – mo, 03 – ney, 04 – low, 05 – your. The S-stream is then *A1 E1 01 EA 02 03 EA A1 E1 04 EA 05 06 02 03 EA*.

3.3 The dictionary encoding

The previous step also generates a dictionary of words or syllables which are used in the text. TD3 [10] is one of the most effective dictionary compression methods. It encodes the whole dictionary (represented as a trie) instead of the separate items stored inside.

Trie compression of a dictionary (TD) is based on the trie representing the dictionary coding. Every node of the trie has the following attributes: *represents* — a boolean value stating whether the node represents a string; *count* — the number of sons; *son* — the array of sons; *extension* — the first symbol of an extension for every son.

The latest implementation of TD3 algorithm employs a recursive procedure *EncodeNode3* (Figure 1) traversing the trie by a depth first search (DFS) method. For encoding the whole dictionary we run this procedure on the root of the trie representing the dictionary.

An example is given in Figure 2. The example dictionary contains strings `".\n"`, `"ACM"`, `"AC"`, `"to"`, and `"the"`.

In procedure *EncodeNode3* we code only the number of sons and the distances between the sons' extensions. For non-leaf nodes we must use one bit to encode whether that node represents a dictionary item (e.g. syllable or word) or not. Leafs always represent dictionary items, so it is not necessary to code them. Differences between extensions of sons are defined as distances between `ord` function values of the extending characters. For coding the number of sons and the distances between them we use gamma and delta Elias codes [4]. (We have tested other Elias codes too, but the best results were achieved for the gamma and delta codes). The number of sons and distances between them can reach the value 0 but standard versions of gamma and delta codes start from 1 — it means that these codings do not support value 0. Therefore we use slight modifications of Elias *gamma* and *delta* codes: $gamma_0(x) = gamma(x+1)$ and $delta_0(x) = delta(x+1)$.

Using the `ord` function we reorder the symbols alphabetically according to the symbols' types and their occurrence frequencies typical for a certain language. While the distances between the sons are smaller than distances coded for example by ASCII, they can be represented by shorter Elias delta codes.

```
EncodeNode3(node) {
  /* encode the number of sons */
  output->WriteGamma0(count + 1);
  if (count = 0) return;
  /* is the node a string? */
  if (represents)
    output->WriteBit(1);
    else output->WriteBit(0);
  if (IsKnownTypeOfSons)
    previous = first(TypeOfSymbol(This->Symbol));
    else previous = 0;
  /* iterate and encode all sons of this node */
  for(i = 0; i < count; i++) {
    /* count and encode the distances between sons */
    distance = ord(son[i]->extension) - previous;
    output->WriteDelta0(distance + 1);
    /* call the procedure on the given son */
    EncodeNode3(son[i]);
    previous = ord(son[i]->extension);
  }
}
```

Fig. 1. Procedure EncodeNode3

In our example (Figure 2) the symbols 0–27 are reserved for lower-case letters, 28–53 for upper-case letters, 54–63 for digits and 64–255 for other symbols.

Additional improvement is based on the knowledge of a syllable type that is determined by the first one or two letters of the syllable. If a string begins with a lower-case letter (lower-word or lower-syllable), the following letters must be lower-case too. In a trie every son of a node representing lower-case letter must be lower-case letter as well.

The same situation comes on for other-words and numeric-words: if a string begins with an upper-case letter, we must examine the second symbol to recognize the type of the string (mixed or upper). In our example (Figure 3.3) we know that all sons of nodes `'t'`, `'o'`, `'h'`, and `'e'` are lower-case letters.

In this ordering (described by the function `ord`), each symbol type is given an interval of potential order. Function `first` returns the lowest orders available for each given symbol type.

Each first node (son) has its imaginary left brother having default value 0. If the syllable type is defined, it is possible to set the imaginary brother's value to the corresponding value of `first`. It lowers the distance values (and shortens their codes) of the mentioned nodes.

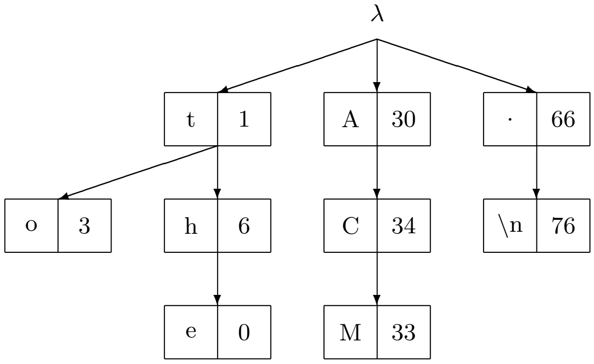

Fig. 2. Example of a dictionary for TD3

Take the node labeled ‘t’ in Figure 2 to describe the coding procedure EncodeNode3. First we encode the number of its sons. The node has two sons therefore we write $gamma_0(2) = 011$. By writing a bit 0 we denote that the dictionary does not containt the processed word (string “t”). The value of the first son of `'t'` is encoded as a distance between its value 3 and zero by $delta_0(3 - 0) = 01100$. Then the first subtrie of node `'t'` is encoded by a recursive call of the encoding procedure on the first son of the actual node.

Table 2. The output: 8 / E1 06 01 02 02 E1 04 05 EA EA A1 A1 03 03

01	02	03	EA	A1	E1	04	05	06	02	03	EA	A1	**E1**
02	03	EA	A1	E1	01	02	03	EA	A1	E1	04	05	**06**
02	03	EA	A1	E1	04	05	06	02	03	EA	A1	E1	**01**
03	EA	A1	E1	01	02	03	EA	A1	E1	04	05	06	**02**
03	EA	A1	E1	04	05	06	02	03	EA	A1	E1	01	**02**
04	05	06	02	03	EA	A1	E1	01	02	03	EA	A1	**E1**
05	06	02	03	EA	A1	E1	01	02	03	EA	A1	E1	**04**
06	02	03	EA	A1	E1	01	02	03	EA	A1	E1	04	**05**
A1	**E1**	**01**	**02**	**03**	**EA**	**A1**	**E1**	**04**	**05**	**06**	**02**	**03**	**EA**
A1	E1	04	05	06	02	03	EA	A1	E1	01	02	03	**EA**
E1	01	02	03	EA	A1	E1	04	05	06	02	03	EA	**A1**
E1	04	05	06	02	03	EA	A1	E1	01	02	03	EA	**A1**
EA	A1	E1	01	02	03	EA	A1	E1	04	05	06	02	**03**
EA	A1	E1	04	05	06	02	03	EA	A1	E1	01	02	**03**

3.4 Burrows-Wheeler transform

In a BWT step we transform the S-stream into a “better” stream. The “better” stream means achieving some better final compression ratio. Obviously the transform should be reversible, otherwise we could lose some information. Specifically we achieve a partial grouping of the same input characters. This process requires sorting of all the step input permutations. In this certain prototype we do not use an effective algorithm described in [13] but a simpler C/C++ qsort function. The use of more sophisticated algorithm would lower the compression time but would not affect the compression ratio. We realize that time and spatial complexity is markedly worse as in optimal case. Since in optimal case the order of single elements can be different, we do not mention the time complexity in the text, it would be confusing.

Suppose we have a sorted matrix of all input permutations: the transform output is then composed by its last column and by the column index of input in this matrix (Table 2).

3.5 Move to Front transform

Then the output stream of BWT is transformed by another transformation step – MTF [1]. This step translates text strings into a sequence of numbers. Suppose a numbered list of alphabet elements, then MFT reads these input elements and writes their list order. As soon as the element is processed it is moved up to the front of the list.

Example 4.

```
alphabet: 01 02 03 04 05 06 A1 E1 EA
string: E1 06 01 02 02 E1 04 05 EA EA A1 A1 03 03
output: nothing

alphabet: 03 A1 EA 05 04 E1 02 01 06
string:
output: 76230356808080
```

The output of MTF phase is “76230356808080”.

3.6 Run Length Encoding of null sequences

One MFT step may generate long sequences of zeroes (null sequences). If successful, the RLE step shrinks these null sequences and replaces them with a special symbol representing a null sequence of a given length. Finally the output is a stream of numbers and the special symbols.

Unfortunately, our example does not show a proper use of RLE. Therefore we will alter the problem and replace the string “02 01 00 00 00 03 00 07 00 00” by “02 01 N3 03 00 07 N2”.

3.7 Canonical Huffman

After the RLE step the stream is encoded by a canonical Huffman code [12]. Huffman coding is based on assigning shorter codes to more frequent characters then to characters with less frequent appearance. In our example, “76230356808080” will have assigned the following codes:

Example 5.
`0 - 00, 2 - 110, 3 - 010, 5 - 1110, 6 - 011, 7 - 1111, 8 - 10`
The output is then: 1111 011 110 010 00 010 1110 011 10 00 10 00 10 00.

4 Experiments

We compared the compression effectivity using BWT of letters, syllables, words, 3-grams and 5-grams. Testing procedures proceeded on commonly used text files of different sizes (1 kB - 5 MB) in various languages: English (EN), Czech (CZ), and German (GE).

4.1 Testing data

Each of tested languages (EN, CZ, GE) had its own plain text testing data. Testing set for Czech language contained 1000 random news articles selected from PDT [20] and 69 books from eKnihy [15]. Testing set for English contained 1000 random juridical documents from [19] and 1094 books from Gutenberg project [16]. For German, we used 184 news articles from Sueddeutche [18] and 175 books from Gutenberg project [16].

4.2 Results

The primary goal was to compare the letter-based compression, syllables and words. For files sized up to 200 kB, the letter-based compression appears to be optimal; for files 200 kB - 5 MB syllable-based compression is the most effective. Also used language affects the results. English has a simple morphology: in large documents the difference between words and syllables is insignificant. In languages with rich morphology (Czech, German) words are still about 10% worse then syllables, even on the largest documents. Language type influence on compression is detailed in [11].

As the second aim we tried to compare the syllable-based compression with 3-gram-based and word-based compression with 5-gram-based compression. Syllables as well as words are natural language units therefore we supposed using it to be more effective than using 3-grams and 5-grams. These assumptions were confirmed. Natural units were the most effective for small documents (20 - 30 %), by increasing the document size efficiency falls to 10 - 15 % for documents of size 2 - 5 MB.

5 Conclusion

In this paper, we compare experimentally the compression efficiency of Burrows-Wheeler transform by letters, syllables, words, 3-grams and 5-grams, using the XBW compression algorithm. We test common plain text files of different sizes (1kB – 5MB) in various languages (English, Czech, German). BWT block contains the whole document.

Table 3. Comparision of different input parsing strategies for Burrows-Wheeler transform. Values are in bits per character.

—	File size	100 B	1 kB	10 kB	50 kB	200 kB	500 kB	2 MB
Lang.	Method	1 kB	10 kB	50 kB	200 kB	500 kB	2MB	5 MB
CZ	Symbols	**5.715**	**4.346**	**3.512**	**3.200**	**2.998**	2.846	—
CZ	Syllables	6.712	4.996	3.765	3.280	3.003	**2.825**	—
CZ	Words	7.751	6.111	4.629	3.871	3.476	3.149	—
CZ	3-grams	8.539	6.432	4.629	3.851	3.463	3.166	—
CZ	5-grams	10.104	8.415	6.566	5.448	4.796	4.265	—
EN	Symbols	**5.042**	**3.018**	**2.552**	**2.647**	2.513	2.336	2.066
EN	Syllables	5.974	3.267	2.647	2.685	**2.486**	**2.282**	**1.996**
EN	Words	6.323	3.651	2.969	2.944	2.668	2.382	2.014
EN	3-grams	7.740	4.571	3.421	3.148	2.823	2.530	2.136
EN	5-grams	9.358	6.293	4.877	4.367	3.769	3.246	2.530
GE	Symbols	**4.545**	**3.853**	**2.914**	**2.629**	**2.491**	2.323	2.416
GE	Syllables	5.591	4.671	3.201	2.724	2.505	**2.295**	**2.354**
GE	Words	6.343	5.491	3.679	3.119	2.865	2.545	2.608
GE	3-grams	6.813	5.583	3.760	3.117	2.820	2.525	2.519
GE	5-grams	8.545	7.429	5.324	4.320	3.744	3.237	3.004

Comparing the letter-based, syllable-based and word-based compression, we find out that character-based compression is most suitable for small files (up to 200 kB) and syllable-based compression for files of size 200 kB – 5 MB (see Table 3, the compress ratio in the table is presented in bites per byte (bpc)).

The compression using natural text units like words or syllables is 10–30% better than compression with 5-grams and 3-grams.

To achieve the results introduced in this article, we use compression algorithm XBW implemented mainly for testing purposes. Our next goal is to improve this program for practical use, e.g. time complexity advance (faster sorting algorithm in BWT [13], splay trees for MTF [2]) or UTF-8 encoding support.

References

1. Arnavut, Z.: Move-to-front and inversion coding. Data Compression Conference, IEEE CS Press, Los Alamitos, CA, USA (2000) 193–202.
2. Bentley, J. L., Sleator, D. D., Tarjan, R. E., Wei, V. K.: A locally adaptive data compression scheme. Communications of the ACM, 29(4):320-330, April 1986.
3. Burrows, M., Wheeler, D. J.: A Block Sorting Loseless Data Compression Algorithm. Technical report, Digital Equipment Corporation, Palo Alto, CA, U.S.A (2003).
4. Elias, P.: Universal codeword sets and representation of the integers. IEEE Trans. on Information Theory, Vol. 21, (1975) 194–203.
5. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. Proc. 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05), (2005) 184-193.
6. Ferragina, Manzini, G., Muthukrishnan, S.: Compressing and Searching XML Data Via Two Zips. Proc. WWW 2006, Edinburgh, Scotland. (2006) 751–760.

7. Galambos, L., Lansky, J., Chernik, K.: Compression of Semistructured Documents. In: International Enformatika Conference IEC 2006, Enformatika, Transactions on Engineering, Computing and Technology, Volume 14, (2006) 222–227
8. Isal, R.Y.K., Moffat, A.: Parsing Strategies for BWT Compression. Data Compression Conference, IEEE CS Press, Los Alamitos, CA, USA (2001) 429-438.
9. Isal, R.Y.K., Moffat, A.: Word-based Block-sorting Text Compression. Proc. 24th Australasian Computer Science Conference, Gold Coast, Australia, (2001) 92–99
10. Lansky, J., Zemlicka, M.: Compression of a Dictionary. In: Snasel, V., Richta, K., and Pokorny, J.: Proceedings of the Dateso 2006 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS, Vol. 176, (2006) 11-20
11. Lansky, J., Zemlicka, M.: Text Compression: Syllables. In: Richta, K., Snasel, V., Pokorny, J.: Proceedings of the Dateso 2005 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS, Vol. 129, (2005) 32–45
12. Moffat, A., Turpin, A.: On the implementation of minimum redundancy prefix codes. IEEE Trans.Comm. 45(1997), 1200–1207
13. Seward, J.: On the Performance of BWT Sorting Algorithms. DCC, IEEE CS Press, Los Alamitos, CA, USA (2000) 173.
14. Seward, J.: The bzip2 and libbzip2 official home page. `http://sources.redhat.com/bzip2/`
15. e-books. `http://go.to/eknihy`
16. Project Gutenberg. `http://www.promo.net/pg`
17. Compact Oxford English Dictionary. `http://www.askoxford.com`
18. Sueddeutsche. `http://www.sueddeutsche.de`
19. California Law. `http://www.leginfo.ca.gov/calaw.html`
20. The Prague Dependency Treebank. `http://ufal.mff.cuni.cz/pdt/`