International Journal of Computer Sciences and Engineering Systems
Vol. 13 No.1 (June, 2019)

Browsing Compressed Collection of Semi-Structured Documents

Ashutosh GUPTA[1] and Suneeta AGARWAL[2]

[1] Computer Science and Engineering Department, Institute of Engineering and Rural Technology
Allahabad, UP 211002, India, *E-mail:ashutosh333@rediffmail.com*

[2] Computer Science and Engineering Department, Motilal Nehru National Institute of Technology
Allahabad, UP 211002, India, *E-mail: suneeta@mnnit.ac.in*

Manuscript received October 25, 2018 Manuscript revised December 20, 2018

Abstract: *Semi-structured document has high redundancy that wastes disk space, bandwidth and disk I/O. To overcome the verbosity problem, the research is going on to develop efficient compressors for semi-structured data. For the efficiency of storage and pattern searching, it is necessary to compress document. We describe a compression model for semi-structured document. The novelties are; pattern searching can be done on the compressed data directly, using any known sequential pattern-matching algorithm; decompression of any selected portion can also be done very efficiently; insertions, deletions and modifications in the original XML data can be incorporated directly in earlier compressed XML data. A special feature of the compressor is that the compressed document retains the structure of the original document. We have compared our model with the sate-of-art compressors and achieved more than 82% compression ratio. The compression speed is approximately 67% faster than XMLPPM and bzip2 and decompression speed is 4-2 times faster than XMLPPM and bzip2.*

Keywords: *Text Compression, Semi-Structured Compression, Compressed documents, Pattern matching*

1. INTRODUCTION

In the last twenty years, we have seen a vast explosion of textual information flow over web through electronic mail, web browsing and information retrieval systems etc. The importance of data compression is likely to be enhancing in the future, as there is continuous increase in amount of data that need to be transformed or archived. The aim of data compression is to exploit the redundancies in the data to reduce its space usage. The most widely used data compression algorithms are based on the sequential data compressors of Lempel and Ziv [24, 25]. Statistical modeling techniques may produce superior compression [34], but are significantly slower.

Text compression is about finding ways to represent the text in less space. This is accomplished by substituting the symbols in the text by equivalent ones that are represented using a smaller number of bits or bytes. For large text collection, text compression appears as an attractive option for reducing costs. The gain obtained from compressing text is that it requires less storage space, it takes less time to be read from disk or transmitted over a communication link, and it takes less time to search. The savings of space obtained by a compression method is measured by the compression ratio. There are other important aspects to be considered, such as compression and decompression speed. In some situations, decompression speed is more important than compression speed. For instance, this is the case with textual databases and documentation systems in which it is common to compress the text once and to read it many times from disk

In the field of data compression, Researchers developed various approaches such as Huffman encoding [9], arithmetic encoding [23, 33], Ziv-Lempel family [24, 25, 33,], Dynamic Markov compression, Prediction with partial matching [2] and Burrows Wheeler Transform [26, 27, 31, 32] based algorithms, etc. BWT permutes the symbol of a data sequence that share the same unbounded context by cyclic rotation followed by lexicographic sort operations. BWT uses move-to-front and an entropy coder as the backend compressor. PPM is slow and also consumes large amount of memory to store context information but PPM achieves better compression that almost all existing compression algorithms.

As opposed to the HTML, which is a markup language for a specific kind of hypertext, SGML and XML [7] becomes the universal format for structured documents and data on web. According to [38] Extensible Markup language is a standardized language that “describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them”. XML language [35] has now become the de facto standard for data exchange and storage, especially on the Internet, due to its self-describing and textual nature. The storage, exchange,

and manipulation of semi-structured data are spreading across all kinds of applications, ranging from web-services and electronic commerce to text database and digital libraries.

The semi-structured documents (e.g. SGML, XML) have high redundancy due to the repeated nature of tags, which lend themselves naturally to compression [6, 11, 12, 13]. After getting the compressed document, one still require query in compressed domain. The more recent work in [12, 20, 21] proposes techniques that allow the query processor to decompress a small unit of data at a time: one column value in the table or one row. There are several advantages of performing query under compressed domain. First, access to compressed data lead to less disk I/Os and reduce the query processing time. Second; the memory requirement in processing compressed data is lower than those for uncompressed data, finally, the result of compressed query uses less bandwidth in network while sending the result to remote machine.

Our contribution in this paper is to combine the text structure. We take the advantage of the document structure, while maintaining the homomorphism feature. Our idea is to make the separate containers for start/end tag, attribute name / attribute value and textual words. We will show that this separation will give lower entropy as compared to without separating the tags.

The paper is organized as follows. In Section 2, we discuss some basics of general-purpose compression methods and compression algorithms for XML data. In Section 3, we present the description of our compressor model. Section 4 describes how the partial decompression and searching is performed. The experimental framework for the compression model is described in section 5. Section 6 describes the performance results. Finally we conclude in Section 7.

2. BASICS AND RELATED WORK

Text compression [1, 36] is usually categorized into three groups: First group belongs to static compression, which uses a fixed statistics or does not use any statistics for compression. Examples of the static encoding methods are dictionary encoding, binary encoding, differential encoding and [4, 5]. Second category belongs to Statistical compression. This category belongs to estimating source symbol probabilities and assigning them codes according to the probability. Examples of this category are: Huffman coding [9], Arithmetic coding [20] etc. Third category is dictionary methods, which are, consist in replacing text sub string by identifiers, so as to exploit repetitions in the text. Adaptive compression cannot start decompression at arbitrary file positions, because all the previous text must be processed so as to learn the model that permits decompressing the text that follows.

Lempel-Ziv compression is a dictionary method based on replacing text substrings by previous occurrences thereof. The two well-known algorithms of this family are called LZ77 [24] and LZ78 [25]. A well-known variation of the LZ78 is called LZW [3, 33]. GNU's gzip is an example LZ77 compression. A well-known representative of LZW is Unix's *compress*. Lempel-Ziv compressed text cannot be decompressed at random positions, because one must process all the text from the beginning in order to learn the window that is used to decompress the desired portion.

XMill[14] is an compressor that make use of the similarities between the semantically related XML data to eliminate data redundancy. XGrind and XPRESS [22, 30] compressor has the ability that it supports queries directly over the compressed XML data and adopts homomorphic transformation to preserve the structure of the XML data. We advised readers to refer [29] or standard textbooks [10, 36] for a general discussion.

3. MODEL FOR COMPRESSOR

Today, the information retrieval is one of the most important aspects. We propose a compressor model that permits access and direct searching on the compressed semi-structured document. In this section, we describe the features of our compressor. A useful feature of the model is that it retains the structure of the original XML document in the compressed format also. The model uses similar technique as used by XGrind [30] for compressing meta-data but differing in compressing the element/value and textual information in document. The working of model is as follows: Each start-tag of an element is encoded by 'T' followed by a uniquely assigned element-ID. All end-tags are encoded by '/'s. The pair (element, element-ID) is stored in a container (C_1) except for end-elements. Since each start element is closed by a corresponding end-element, there is no need for storing end-element code i.e. '/'. Each Attribute Name/Attribute Value is encoded by a string "AV" followed by a uniquely assigned element-ID. The pair (attributeName/ AttributeValue, element-ID) is stored in a container (C_2). We use a novel approach for compressing the text. The text is seen as an alternating sequence of words and separators, where a word is a maximal sequence of alphanumeric characters and a separator is a maximal sequence of non-alphanumeric characters. Each word from the document is extracted and is stored in a separate container. In first phase, the frequency distributions for all the words in text are gathered. In second phase, we actually assign the codes to the words in a sequential manner. If container contains *m* words, then the i^{th} word w_i together with its code *i* is stored in another container for all $0 <= i < m$.

In first phase, the routine *gather_statistics* computes the frequency for each word of text. The Attribute Name/ Attribute Value pair is encoded by *av_routine*. These routines create C_1 and C_2 containers. In second phase, the output of *gather_statistics* routine is used for encoding the textual information in document with the help of *text_coding* routine. The *text_coding* routine produces another container, which

we called text container (C_3). The resulting containers are independently compressed with gzip producing Compressed Container Repository. These compressed containers can be regarded as a compressed index. During second pass, start/ end element, attribute name, attribute value and textual words as fetched and their corresponding codes are written the the output file. The final compressed document is consists of Compressed Container Repository (Index) and the code assignment of document tags, attribute Name/Attribute Value and textual words.

The important feature of the compressor is that its output is similar to input in nature i.e. it is in semi-structured format. The compressed XML document can be viewed as the original XML document with its tags, attribute Name/ Attribute Value and textual words replaced by their corresponding codes.

Proposition 1: The compressed XML document generated by compressor is homomorphism with respect to original XML document.

3.1 Estimation of Entropy

Assume we have a text *T* of *m* terms partitioned into *M* texts $T_1 \ldots T_M$, so that T_c has m_c terms. The idea is that each T_c corresponds to the text will be encoded using its own container. We define the raw frequency relative to a given container *c*.

Definition 1 (Frequency) *The frequency fc(j) of term j is given by*

$$f_c(j) = \frac{noc_c(j)}{m_c}$$

where $\text{noc}_c(j)$ *is the number of occurrences of term j in Tc.*

Definition 2 (Estimating Zero-order entropy) *Let W_c be the number of word terms for text T_c. The zero-order entropy H_c of text T_c is estimated as*

$$H_c = \sum_{j=1}^{Wc} f_c(j) \ln_2 \frac{1}{f_c(j)} \quad (1)$$

We can now define the overall entropy of a text *T* partitioned into multiple texts. This is a lower bound to the average codeword length obtained by applying any zero-order compressor to the text under each container.

Definition 3: (Estimating Zero-order entropy with multiple containers) *The zero-order entropy H for text $T = \{T_1, \ldots, T_M\}$ is computed as the weighted average of zero-order entropies H_c contributed by each text T_c:*

$$H = \frac{\sum_{c=1}^{M} m_c H_c}{m} \quad (2)$$

Definition 4: (Estimating container size contribution) *Let W_c be the size, in bits of the text words that forms containers c, and H_c its estimation of zero-order entropy. Then the estimation of container(c) size contribution is given by*

$$J_c = W_c + m_c H_c \quad (3)$$

3.2 Example

Figure 1 shows an example XML file. Let us consider a single model for compressing all the words. The text has $W_{all} = 44$ different words and $m_{all} = 69$ total words. The words and their frequencies follow:

nutrition(1), daily-values(1), total-fat(2), units(8), g(6), 20(1), cholesterol(2), saturated-fat(2), 65(1), mg(2), 300(2), sodium(2), 2400(1), carb(2), fiber(2), 25(1), protein(2), 50(1), food(1), name(1), Avocado(1), Dip(1), mfr(1), Sunnydale(1), serving(1), 29(1), calories(1), total(1), 110(1), fat(1), 100(1), 11(1), 3(1), 5(1), 210(1), 2(1), 0(5), 1(1), a(1), c(1), vitamins(1), minerals(1), ca(1), fe(1).

Following Eq. (1) the entropy of this text is $H_{all} = 5.14$ bits per word. To account for the cost of encoding the container, let us assume that we need 8 bits per different container word, thus $W_{all} = 8 \cdot W_{all}$. According to Eq. (3), the overall number of bits to represent the text is

$T_{all} = W_{all} + m_{all} H_{all} = 8 \cdot 44 + 69 \cdot 5.14 = 707$ bits.

Let us now separate the elements, attribute name/ attribute value, textual words and compute the raw frequencies of the words within each container.

<element>: nutrition(1), daily-values(1), total-fat(2), cholesterol(2), saturated-fat(2), sodium(2), carb(2), fiber(2), protein(2), food(1), name(1), mfr(1), serving(1), calories(1), a(1), c(1), vitamins(1), minerals(1), ca(1), fe(1).

<attribute name/attribute value>: units/g(6), 20(1), units/ mg(2), total/110(1), fat/100(1).

<textual words>: 20(1), 65(1), 300(2), 2400(1), 25(1), 50(1), Avocado(1), Dip(1), 29(1), 110(1), 100(1), 11(1), 3(1), 5(1), 210(1), 2(1), 0(5), 1(1).

The entropies for each container are computed using Eqs. (1) and (3).

Containers	*Entropy (bits / word)*	*Number of Words*	*Different words*	*Total bits in container*
Element (C_1)	$H_{element}$ = 4.03	$m_{element}$ = 29	$V_{element}$ = 20	$J_{element}$ = 276
Attribute name/attribute value (C_2)	$H_{an/av}$ = 1.56	$m_{an/av}$ = 10	$V_{an/av}$ = 4	$J_{an/av}$ = 48
textual words (C_3)	$H_{textword}$ = 3.83	$m_{textword}$ = 22	$V_{textword}$ = 17	$J_{textword}$ = 221

If we consider the text words in separate form we obtain, a total entropy of $H = 3.53$ bits per word (using Eq. 2). This is much improved (31% less) than if we consider all text

words together. This is not unexpected, as the entropy is always lower when we divide a text. However, we must also consider the cost of maintaining separate containers for each text word. In this case, even if we consider the container storage, we obtain a total of 545 bits when we sum the values in the last column of the previous table. This is still lower (23% less) than the 707 bits used when the text words are not separated.

The above result shows that our heuristic for combining the attribute name with attribute value reduces the entropy. The important feature of the model is that it retains the structure of original document. This feature permits for faster decompression as attribute name and attribute value are coded by single code.

```
<nutrition>
<daily-values>
    <total-fat units="g">65</total-fat>
    <saturated-fat units="g">20</saturated-fat>
    <cholesterol units="mg">300</cholesterol>
    <sodium units="mg">2400</sodium>
    <carb units="g">300</carb>
    <fiber units="g">25</fiber>
    <protein units="g">50</protein>
</daily-values>
<food>
    <name>Avocado Dip</name>
    <mfr>Sunnydale</mfr>
    <serving units= "g">29</serving>
    <calories total= "110" fat="100"/>
    <total-fat>11</total-fat>
    <saturated-fat>3</saturated-fat>
    <cholesterol>5</cholesterol>
    <sodium>210</sodium>
    <carb>2</carb>
    <fiber>0</fiber>
    <protein>1</protein>
    <vitamins>          <a>0</a>          <c>0</c>
    </vitamins>
    <minerals>          <ca>0</ca>        <fe>0</fe>
    </minerals>
</food>
</nutrition>
```

Figure 1: Example XML file.

4. SEARCHING AND RANDOM ACCESS

The method we suggested for compression permits local decompression of the text from random positions, as well as efficient direct search of the compressed text. In this section we show how these tasks can be carried out.

4.1 Searching

The search for a pattern on a compressed text is made in two phases. In the first phase we compress the pattern using the same structures used to compress the input data. In the second phase we search for the compressed pattern. In an exact pattern search, the first phase generates a unique code word for pattern that can be searched with any conventional searching algorithm. In a set of pattern search, the first phase generates all the code words that match with the original patterns in the Text container (C_3). In the second phase we obtain the list of code words that match the pattern set and use a multi-pattern search algorithm.

4.2 Partial Decompression

The partial decompression of compressed text in this scheme is very simple. The Algorithm 1 is used for decompression from given position *pos*. As the compressed document generated by compressor is homomorphism with respect to original document, this property permits one to perform partial decompression from any given position in compressed text. Once we wish to start decompression at some position of the compressed text, we check three things. First, if the compressed code starts with 'T', then we switch to container C_1 and retrieve the corresponding start-tag and push the start-tag to the stack. This way we generate corresponding start-tag, attribute name/attribute value and textual words. Whenever a '/' is seen, we pop the start-tag, prepend "</" and append ">" to the start-tag (i.e. forming end-tag). In this way we continue to decompress the rest of the compressed text. Second, if the compressed code starts with 'AV', then we switch to container C_2 and retrieve the corresponding attribute name/attribute value. Next time if 'T' appears in compressed text, we do in the same way, as we did in first case. At last, if position in compressed text starts with code of textual word, then we decompress using container C_3 and rest of the decompression is performed using case first and second.

Algorithm 1 (Partial Decompression)

// *top* is top index of stack. *pos* is position in compressed text from where partial decompression starts.

1. $S \leftarrow \$$ // '$' is used as a marker in stack
2. while (stack not empty)
3. if (symbol is 'T_{pos}')
4. push T_{pos} to stack
5. retrieve string representing code of T_{pos} and write to output file
6. if(symbol is 'AV_{pos}')
7. retrieve string representing code of AV_{pos} and write to output file
8. if (symbol is '/') and (S_{top} !=$)
9. pop top element from the stack (T_{top})
10. retrieve string representing code of T_{top}, form end-tag and write to output file
11. end while

5. EXPERIMENTAL SETUP

To show the efficiency of compressor, we empirically compared the performance of compressor with variety of documents ranging from small file size to large file size using real-life data sets. We used g++ compiler with full optimization. The experiment was carried out on a 2.6GHZ Pentium IV machine housing Fedora Core 2.

5.1 Data Sets

The details of the XML documents used in our experiment are summarized in Table 1. The Size refers to the total disk space occupied by the document in MBs;

Table 1
Details of XML Documents

Document	*Size*
mondial [18]	1.7
play [19]	7.8
tpc-h [17]	42
treebank [16]	86
xmark [15]	111
dblp [15]	207

The reason for choosing small to large set of document is to ensure that our compressor framework is well suited for different document statistics.

5.2 Category of Experiments

We carry out two categories of experiments. We run each experiment four times and take the average of four runs. The compression ratio is defined as follows:

$$CR = 1 - \frac{\text{Size of compressed XML data}}{\text{Size of original XML data}}$$

Second we measure the compression and decompression speed for all variety of data.

6. PERFORMANCE RESULTS

The efficiency of our compression method is evaluated with two measures defined in subsection 5.2. We compare our heuristic with several classical compression tools: (1) *bzip2*[1], which uses the Burrows Wheeler block sorting text compression algorithm and Huffman coding; (2) GNU *gzip*[2], which uses LZ77 and a variant of Huffman algorithm; (3) *LZW* algorithm, which is a variant of LZ78; (4) *XMLPPM*[3], a XML specific compressor based on adaptive PPM over the structural contexts; (5) *XMill*[4] is an XML specific compressor based on Huffman and Lempel-Ziv. We used the maximum compression option whenever possible and also standard options for all. Compression ratios for each data set is shown in Figure 2.

Let us compare the compression ratio for each compressor. The LZW algorithm obtained the worst compression ratio, and hence not included in the experiment. Then gzip compression ratio is still lower, roughly 77%. The best and default compression options in gzip has nearly no difference. Next to gzip is, XMill, which obtains average

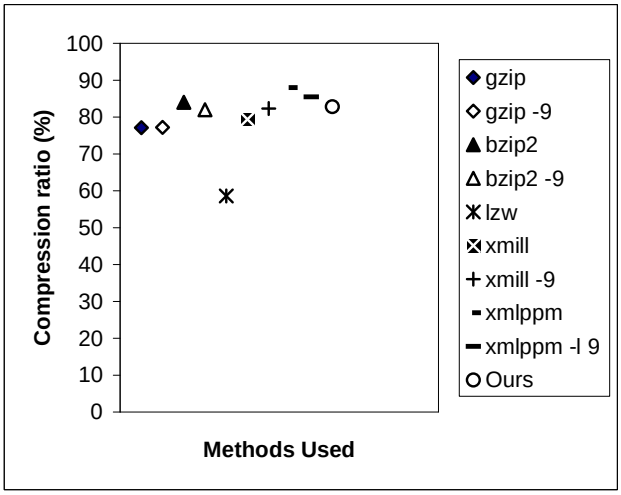

Figure 2: Comparison between other Compressors

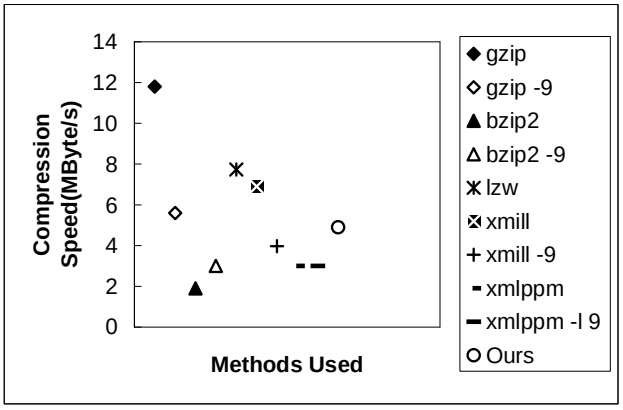

Figure 3: Comparison between Compression Speed (MB/s) for all Methods

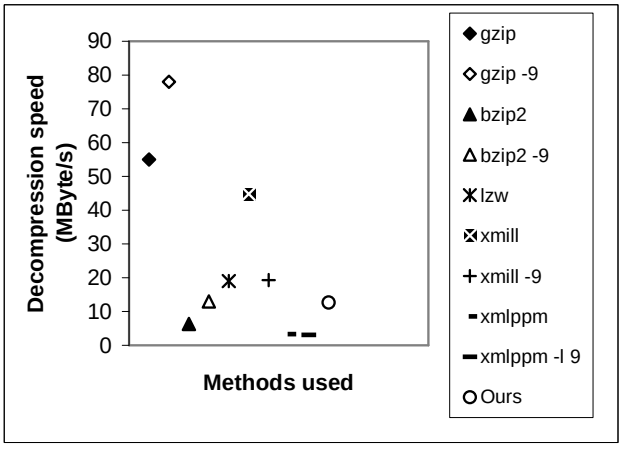

Figure 4: Comparison between Decompression Speed (MB/s) for all Methods

compression ratio between 79-82%, and is lower than our approach. Then bzip2 gives the compression ratio between 82-84% which is nearly same to our approach. Finally the best standard compressor among the set of compressors is XMLPPM[8], which achieves 85-88% compression ratio. We have excluded the XGrind from our experiment, as the compression ratio achieved by it is lower than XMill.

Figure 3 and Figure 4 shows the overall average values for compression and decompression speed with respect to the methods described above. We have taken the average values of all the data collection. The compression speed of gzip is fastest, followed by LZW and XMill. Our model is slightly slower than these but faster than bzip2 and XMLPPM. The possible reason is that our implementation is in prototype stage and is not fully optimized. The decompression is faster for gzip followed by XMill and LZW. This is not surprising, as gzip is known for its faster decompression. As XMill uses gzip, it also has faster decompression. Next bzip2 comes and XMLPPM is the slowest at decompression, as expected from this family. Our model comes in between bzip2 and XMLPPM.

All the methods described above do not permit direct access to the compressed document. One of the methods that permit direct access is MG System. We have excluded the MG[5] System due to two reasons: first, we could not make it work properly in our machine and second, it does not permit decompressing the whole collection. In our method, the partial decompression is possible and described in subsection 4.2 (Algorithm 1). The decompression speed for documents is low, but this is compensated, if we perform the query directly over the compressed data instead of decompressing it. This makes query processing fast as the query processor has to do process less data.

7. CONCLUSIONS AND FUTURE WORK

We have described a compressor for semi-structured document based on the idea that by separating the start tag, attribute name/attribute value and textual words will reduce the entropy. We also combine the attributes with their values and use the separate containers for them. The heuristic and proposed model is useful in the information retrieval system. One of the applications is XML data archiving, where compression rate counts alone. The main results are that we can handle the compressed text throughout the whole process with improved time. The documents can be compressed to 82% (average) of its original size.

The compression process is considered as a single time investment. After that one can perform the local decompression and pattern searching directly over the compressed data, which reduces the CPU processing time as CPU has to process less data. Random access of file is also very low as the disk seek time is depend upon number of tracks, and is a linear function of size of file.

This method can be applied to meet other requirements. For example, it is not hard to mix it with the idea of [28] to allow searching for regular expressions, approximate patterns, etc.

Notes

1. *http://www.bzip.org*
2. *http://www.gnu.org*
3. *http://sourceforge.net/projects/xmlppm*
4. *http://sourceforge.net/projects/xmill*
5. *http://www.cs.mu.oz.au/mg*

REFERENCES

[1] Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers. 2002.

[2] Moffat. Implementing the PPM Data Compression Scheme. *IEEE Transactions on Communications*, 38(11): 1917-1921, 1990.

[3] Ashutosh Gupta and Suneeta Agarwal. "Block Based LZW Compression of Dynamic Documents". In *Proc. of International Conference on Information and Communication Technology* (IICT). p. 261-263, 2007.

[4] Ashutosh Gupta and Suneeta Agarwal. "New Transform for Improving Compression Performance in Natural Language Text". In *Proc. of International Multi-Conference for Engineers and Computer Scientists* (IMECS). p. 564-567, 2007.

[5] Ashutosh Gupta and Suneeta Agarwal. "Word Based Text Compression Using Encyrption". In *Proc. of International Multi-Conference for Engineers and Computer Scientists* (IMECS). p. 471-473, 2007.

[6] Iyer and D. Wilhite. "Data Compression Support in Database". *Proc. of VLDB*, 1994.

[7] Charles F. Goldfarb and Paul Prescod. *The XML Handbook*. Prentice-Hall, third edition, 2001.

[8] Cheney, J. and I. Witten I. Data Compression using Adaptive Coding and Partial String Matching. *IEEE Trans. on communication*, 32: 396-402.

[9] A. Huffman. A Method for the Construction of Minimum Redundancy Codes. In *Proceedings of the Institute of Radio Engineers* 40, 1098-1101, 1952.

[10] Salomon. "Data Compression". *The Complete Reference*. Springer, New York, 1997.

[11] G. Graefe and L. Shapiro. "Data Compression and Database Performance". *Proc. of ACM/IEEE CS Symp. on Applied Computing*, 1991.

[12] G. Graefe. "Options in Physical Database". *ACM SIG-MOD Record*, 1993.

[13] G. Ray, J. Haritsa and S. Seshadri. "Database Compression: A Performance Enhancement Tool". *Proc. of 7th Intl. Conf. on Management of Data* (COMAD), 1995.

[14] H. Liefke and D. Suciu. XMill: An Efficient Compressor for XML Data. In *Proc. of SIGMOD 2000*.

[15] http://cs.washington.edu/research/xmldatasets/www/repository.html

[16] http://monetdb.cwi/nl/xml/downloads.html

[17] http://www.cs.wisc.edu/niagara/data/tpc-h/

[18] http://www.informatik.uni-freiburg.de/~may/lopix/lopix-mondial.html

[19] http://www.oasis-open.org/cover/bosakShakespeare200.html

[20] I. H. Witten, A. Moffat and J. G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6): 520-541, 1987.

[21] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *Proc. IEEE Conf. on Data Engineering*, 1998.

[22] J. K. Min, M. J. Park, C. W. Chung. XPRESS: A Queriable Compression for XML Data. *In Proceedings of SIGMOD*, 2003.

[23] J. Rissannen and G. G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 2 3: 149-162, 1979.

[24] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*. Vol. IT-23, 3: 337–343, 1977.

[25] J. Ziv and A. Lempel. Compression of Individual Sequences Via Variable-Rate Coding. *IEEE Trans. on Information Theory*, IT-24(5): 530-536, 1978.

[26] K. Sadakane. *Unifying Text Search and Compresion : Suffix sorting, Block sorting and Suffix Array*. PhD. Thesis, The university of Tokyo, December 1999.

[27] M. Burrows and D. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Technical Report, *SRC Research Report 124*, Digital Systems Research Center, Palo Alto, CA, 1994

[28] M. D. Araujo, G. Navarro and N. Ziviani. Large Text Searching Allowing Errors. In *R. Baeza-Yates*, editor, Proc of WSP'97.

[29] M. A. Roth and S. Van Horn. "Database Compression". *ACM SIGMOD Record*, 22(3): 31-39, 1993.

[30] P. M. Tolani and J. R. Haritsa. XGRIND; A Query-friendly XML Compressor. In *Proceedings of ICDE*, 2002.

[31] R. Yugo Kartono Isal and Alistair Moffat and Alwin C. H. Ngai. Enhanced Word-Based Block-Sorting Text Compression. *Twenty-Fifth Australasian Computer Science Conference* (ACSC2002). Melbourne, Australia, 2002.

[32] S. Grabowski. Text Preprocessing for Burrows-Wheeler Block Sorting Compression. VII Konferencja "Sieci i Systemy Informatyczne–teoria, projekty, wdrożenia" Łódź, październik 1999.

[33] T. A. Welch. A Technique for High Performance Data Compression. *IEEE Computing* 17(6): 8-19, 1984.

[34] T. Bell, I. H. Witten, and J. G. Cleary. Modeling for Text Compression. *ACM Computing Surveys*, 21(4): 557–589, 1989.

[35] T. Bray, *et al.* "Extensible Markup Language (XML) 1.0", October 2000, *http://www.w3.org/TR/REC-xml*.

[36] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, New Jersy, 1990.

[37] W. K. Ng and C.V. Ravishankar. Block-Oriented Compression Techniques for Large Statistical Databases. TKDE, 9(2): 314-328, 1997.

[38] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Seocnd Edition). http://www.w3.org/TR/2000/REC-xml-20001006.