Working with Compressed Concordances

Miri Ben-Nissan and Shmuel T. Klein

Department of Computer Science
Bar Ilan University
Ramat-Gan 52900, Israel
Tel: (972–3) 531 8865
Fax: (972–3) 736 0498
Israel
tomi@cs.biu.ac.il
December 8, 2007

Abstract: A combination of new compression compression methods is suggested in order to compress the concordance of a large Information Retrieval system. The methods are aimed at allowing most of the processing directly on the compressed file, requesting decompression, if at all, only for small parts of the accessed data, saving I/O operations and CPU time.

1. Introduction

Research in Data Compression has recently dealt with the *compressed matching* paradigm, in which a compressed text T is searched directly for the occurrence of a given pattern P, rather than the usual approach of first decompressing the text and then searching for P in the original text. This has several advantages, such as enabling the search for some pattern on remote computers, saving time and space for the decoding, etc. Not every compression method is suitable for compressed matching, but many are, e.g. static Huffman coding [12], LZW [6] and many others [18, 15, 16].

But there are also other file types for which compressed matching could be advantageous. In a large full-text Information Retrieval System (IRS), a text is not searched directly, but by means of auxiliary files, such as a dictionary and a concordance. Searching in compressed dictionaries [13] means that the dictionary, which is the list of all the different terms in the database, is searched in its compressed form, by compressing the term to be looked up. The present work now extends the paradigm to the other large file of the IRS, namely the concordance.

The *concordance* gives for each word W in the database a list $\mathcal{L}(W)$ of indices to all its locations. We shall refer to such indices as the *coordinates* of W. In order to find all the places in which the words A and B occur, one has to intersect $\mathcal{L}(A)$ with $\mathcal{L}(B)$. Often, each keyword represents a family of linguistically different variants, which are all semantically equivalent to the given keyword. In this case, we first need to merge the coordinates of each variant in this list, before processing the query itself [4].

In fact, our approach is slightly different from the classical compressed matching framework. A simple search as in a compressed text does not make any sense, since one usually does not try to locate a single coordinate in the file. What is rather needed is a way to process entire lists of coordinates, which leads to our topic of working with compressed concordances.

There are several ways to build the concordance. Each depends on the needs of the system and the accuracy level that the system wants to support. Typically, a coordinate of a word W is a quintuple (b, r, p, s, w), where b is the index of the book in which the word W appears, considering the book as the highest level in the hierarchy describing the locations of the words in the database. The other elements of the coordinate are: r, the index of the part within the book; p, the index of the paragraph (within the part); s, the sentence number (in the paragraph), and w, which is the word number (in the sentence). However, there are Information Retrieval systems that do not support such an accuracy level, and the concordance keeps then only a document number and maybe also the paragraph number. In this work, we assume that the concordance consists of quadruples of the form (d, p, s, w), where we have merged the pair (b, r) to a single *document* field.

In order to achieve convenient computer manipulations, one may choose to keep a fixed length for each field of a coordinate. In this case, each field must be long enough to hold the representation of the maximal possible value. This, however, is extremely wasteful, since most of the values are small. On the other hand, if we decide to keep a variable length encoding for each field, some extra information is needed, to be able to read the concordance and identify the boundaries of each coordinate.

The problem with concordances is that their initial size may be 50%-300% of the size of the plain text itself. In addition, using the original concordance when processing a query often leads to a very large set of coordinates which need to be checked. It is therefore necessary to compress the concordance. However, if the database is queried frequently, there is a large time and space overhead for decompressing the necessary parts of the concordance. For a query of the form A AND B, we need to extract all the coordinates of term A, and intersect them with all the coordinates of term B, all at runtime. In large textual databases this may require a huge amount of data to decompress, and technically, large parts of the concordance will be in uncompressed form most of the time [16].

The present work is an extension of the compressed pattern matching problem, as it presents a new compression method for compressing the concordance, which is aimed to allow working directly with the compressed concordances, and accessing the extracted data only for final decision, while most of the work is done on the compressed part. All logical operations will be applied on the encoded concordance itself. The method to be presented is suitable for static databases, since it relies on statistical information, collected off line before the compression phase is done.

The paper is organized as follows: in the next section, we review previous and related work in the area of concordance compression methods, and discuss its applicability to compressed matching. Section 3 then presents the details of the new algorithm.

2. Previous and Related Work

2.1 The Prefix Omission Method

The most basic compression method that can be applied on concordances is the Prefix Omission Method (POM)[4]. This method is based on the observation that since the coordinates are ordered, consecutive coordinates may share the same b, r, p or even s fields (obviously, different coordinates cannot share all 5 fields). In that case, we can omit the repeated fields from the second coordinate. In order to maintain the ability of reconstructing the file, we need to adjoin a *header* to that coordinate, holding information on which fields are to be copied from the preceding coordinate. For a coordinate of the form (d, p, s, w), it is sufficient to keep a 2-bit header for the four possibilities: don't copy any field from the previous coordinate, copy the d-field, copy d and p fields, and copy fields d, p and s. For example, if the d field is the same as in the preceding coordinate, we shall save as coordinate only the triple (p, s, w), with header 01. Although POM yields quite good compression, it is not possible to work directly with the compressed file.

2.2 Variable Length Fields

As mentioned before, one may choose, for the ease of implementation, to represent each field in the coordinates by a fixed length code, at the cost of keeping a much larger file; turning to variable length codes may reduce its size considerably. But in this case, we need to save extra information to be able to read the file and identify the field boundaries. The idea is to add a fixed-length header to each field, which contains a codeword that represents either the length (in bits) of the stored value, or the value itself, which is useful for specially frequent values in a given field. A small table is kept translating those codewords to the corresponding values.

2.3 Numerical Compression

Most of the data stored in the concordance is numerical and traditional compression algorithms cannot provide sufficient compression for such data. In the concordances, all the values are sorted in non-descending order. For some of the fields, we would like to find a method that allows to compare the values of two compressed elements directly, without decoding first. For other fields we just need a good compression method, that will allow us to identify the element's boundaries fast. Studying the distribution of each of the concordance components can help us to choose the proper compression technique to encode it [2].

A well known technique for compressing lists of non-decreasing values is *Delta Encoding*, where each element, except the first, is replaced by difference from its predecessor [1, 7], resulting in smaller values to be stored. This can be done either directly, or, e.g., devising a Huffman code for the differences. Witten *et al.* [17] review some compression techniques that can be applied on those delta values and compare their performances. Linoff *et al.* [14] also presented some techniques for compressing numerical data, such as *n-s Coding*.

There are more techniques that use variable blocks with escape codes, such as the Elias codes, Fibonacci codes, etc. The idea is to set aside one bit per block as a *flag-bit*, indicating the lengths of the blocks. The γ *Elias Codes* [5] maps an integer x onto the binary value of x prefaced by $\lfloor log(x) \rfloor$ zeros. The binary value of x is expressed in as few bits as possible and therefore begins with a 1, which serves to delimit the prefix. Using this code, each integer with N significant bits is represented in $2N+1$ bits [8]. The *Fibonacci Code* is a universal variable length encoding of integers, based on the Fibonacci sequence, rather than on powers of 2. The advantage of this scheme is its simplicity, robustness and speed. In this code, there are no adjacent 1's, so that the string 11 represent a delimiter between two codewords [10].

3. An algorithm enabling work on the compressed concordance

3.1 General layout of the compressed file

In the suggested algorithm, the concordance will be compressed along the lines described in [4], with a few adaptations intended to facilitate the work directly within the compressed file. For the ease of description, we shall use the statistics of a real life concordance, that of a subset of the Responsa Retrieval Project [9], the relevant parameters of which appear below in Table 1. It should, however, be emphasized that the methods to be described are general in nature, and could straightforwardly be adapted to concordances of other natural language full text Information Retrieval Systems, of different sizes, for different languages and even with different hierarchical structure of the coordinates.

number of unique words	725,966
total number of words (coordinates)	80,928,240
number of documents	67,937
average number of coordinates per word	111.48
average number of documents per word	52.33
average number of coord. per word in each doc.	2.13
average size of coordinate	7.33 bits
average size of delta field	3.30 bits
maximum size of document field	17 bits
average size of document delta values	3.55 bits

Table 1: *Statistics of the Responsa Retrieval Project concordance*

Figure 1 shows the layout of a full coordinate, as it is handled by the retrieval procedures of the Responsa Project. The numbers under each field design their sizes, in bytes, for a total of 8 bytes per coordinate. The main idea of the compression is to represent the values in the different fields using a variable number of bits. This reduces the average number of bits needed, since most values stored in the coordinate are quite small, while the size of the fields in the full coordinate are chosen to accommodate the highest possible values, which occur only extremely rarely. The meta-information necessary for the decompression is kept, for each coordinate, in a fixed length *header*, which is itself partitioned into fields, each encoding the number of bits needed in the corresponding field of the compressed coordinate, as represented in Figure 2.

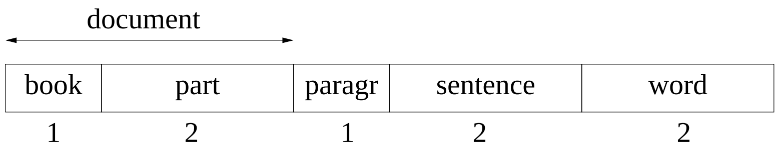

FIGURE 1: *Layout of a non-compressed coordinate*

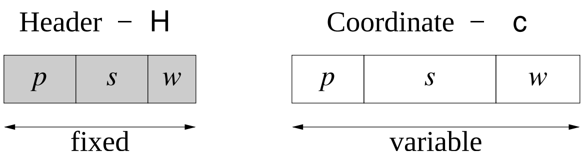

FIGURE 2: *Layout of a compressed coordinate*

In the original algorithm of [4], headers and coordinates were stored in an alternating sequence. This did not cause any problem, since for the processing of a query A AND B, the entire lists $\mathcal{L}(A)$ and $\mathcal{L}(B)$ were first fetched from the disk, then decompressed and finally intersected. To avoid the decompression of many coordinates, it is convenient to partition the compressed file into two parts: the variable length compressed coordinates in one file, which we call the *Coordinates* file, and the fixed length headers in a *Headers* file. A high level schematic representation of the layout of the compressed files is given in Figure 3.

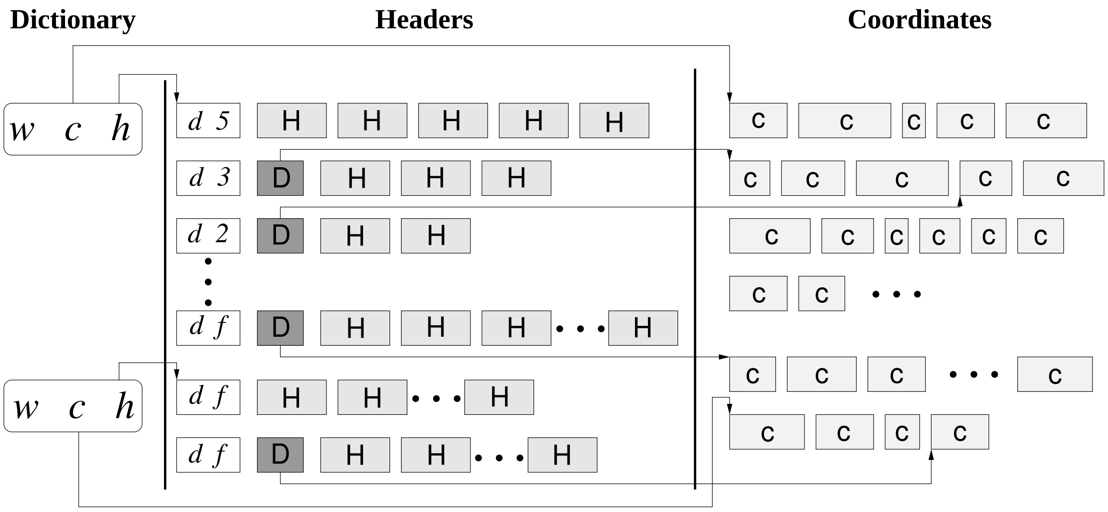

FIGURE 3: *Layout of the compressed concordance*

The concordance is accessed via the *Dictionary*, including all the different words of the text. The dictionary may be stored in a variety of ways, e.g., in lexicographic order of the terms, or sorted first by length and only internally in alphabetic order, or in form of a trie or even as a hash table. The main purpose is fast and direct access to the information stored for each word. The size of the dictionary is usually of a lower order of magnitude than that of the concordance, and by Heaps's Law [11, p. 206–208], its size is $\alpha\ N^{\beta}$, where α and β are suitable constants, $0.4 \le \beta \le 0.6$, and N is the number of words in the text.

Each word w_i in the dictionary is stored along with two pointers c_i and h_i, to the corresponding entries in the Coordinates and Headers files, respectively. In the Headers file, the items are grouped by document number: let n_i be the number of documents the i-th word, w_i, appears in, and let $d_{i,1}, d_{i,2}, \ldots, d_{i,n_i}$ be the indices of these documents; let $f_{i,j}$, $1 \le j \le n_i$ be the number of times w_i appears in document $d_{i,j}$. The entry in the Headers file corresponding to w_i then starts with the pair $(d_{i,1}, f_{i,1})$, followed by $f_{i,1}$ fixed length headers, followed then by $(d_{i,2}, f_{i,2})$, etc.

Note that by grouping the coordinates by documents, the document fields can be omitted, but this comes at the cost of having to add the corresponding number $f_{i,j}$ at the beginning of each group. This is very often a reasonable overhead, especially in Information Retrieval systems supporting, as many do, also a ranking of the passages to be retrieved. The ranking is performed by means of some scores that are usually based, among others, on the local frequencies $f_{i,j}$, so that these are needed anyway.

The Coordinates file consists of the sequence of the variable length (p, s, w) fields. There is no need to separate the elements belonging to different words, since the pointers c_i provide direct access to the elements belonging to w_i. In fact, if for a given word w_i one would always process the full list $\mathcal{L}(w_i)$, the data mentioned so far would be enough to restore the full list. However, since we wish to move as much of the processing as possible to the compressed domain, we would like to perform much of the work on the Headers file and access the Coordinates file only occasionally. We thus need, in addition to the pointer c_i, a sequence of pointers $D_{i,j}$, for $2 \le j \le n_i$ (the darker elements in Figure 3), pointing to the first element in the Coordinates file belonging to document $d_{i,j}$.

3.2 Compression of Coordinates

3.2.1 p, s and w fields

To compress the coordinates, one has first to collect statistics about the exact distribution on the values that appear in the different fields. These lengths are then partitioned into classes, e.g., according to the number of bits needed to represent each of the values. That is, the 1-bit class corresponds to the single value 1, the 2-bit class corresponds to 2 and 3, etc. On the basis of the frequencies of each of the classes, a Huffman code could be devised, for optimal compression. We prefer, however, to use a fixed length encoding of the classes, to facilitate the work directly on the compressed file. Table 2 gives the distribution of the values in the various fields for our test database. The columns correspond to the number of bits needed (up to and including the leading 1-bit) to represent the given numbers, and the values in the table are percents.

As can be seen in Figure 2, the fixed length header allocated to each coordinate consists of 8 bits: 3 bits for each of the p and s fields, and 2 bits for the w field. This allows eight options for p and s and four options for w, which are depicted in Table 2. Option 0 for the p and s fields represent the fact that one copies the corresponding values from the previous coordinate, extending thereby the POM technique. The last line of Table 3 shows the number of times (in percent) a value is equal to that of the preceding coordinate. Since

Bits	1	2	3	4	5	6	7	8	9	10	11-15
Range	1	2–3	4–7	8–15	16–31	32–63	64–127	128–511	512–1023	1024-2043	2048–65535
p-field	6	21	25	23	15	7	3	1			
s-field	25	30	20	12	7	4.5	1	0.5	0.01		
w-field	1.4	4	9.2	14	20	21	18	12	0.4		
delta	0.5	1.2	13	55.8	17.8	7.6	2.9	0.9	0.3	0.1	< 0.01

Table 2: *Distribution of values grouped by lengths (in bits) of the numbers*

for the w field, this happens only rarely, the copy option is not used for w. To understand the values in Table 3, consider for example the w-field: if the code in the corresponding 2-bit header is 10, the w-field in the coordinate itself will be encoded by 7 bits.

	p-field (3 bits)	s-field (3 bits)	w-field (2 bits)
lengths of coord. fields	0 1 2 3 4 5 6 **8**	0 1 2 3 4 5 6 **9**	**5** 6 7 **9**
possible omissions	36%	42%	3%

Table 3: *Interpretation of the codes in the header*

The following amendment should be mentioned, which saves actually quite a few bits: when one considers the number of bits *needed* to encode an integer, one usually refers to the number of significant bits, up to and including the leftmost 1-bit. So for the number 12, in binary 1100, one needs 4 bits. However, if one knows the *exact* number of bits needed, then the leftmost bit is in fact superfluous, since it must be a 1. That is, if the number to be encoded is 12, and we store the information that 4 bits are needed, then one needs only 3 additional bits, giving the relative index in the range $[2^3, 2^4 - 1]$. Returning to Table 3, one can save a bit in the coordinate fields in all cases where the header field gives the exact number of bits. Only for the others, indicated in bold-face in Table 3, one really needs all the bits as indicated by the value in the header. For example, the code 111 in the p-field header stands for a length of 8 bits, but since there is no code for a 7-bit integer, one cannot be sure that the highest bit is a 1, thus all 8 bits are needed; but if the code were 110, the corresponding length would be 6 and here we see that there is also a code for a length of 5, thus the encoded integer must be in the range $[32, 63]$, for which 5 bits are enough.

3.2.2 document, frequency and delta

The 3-byte document index is first translated into a running index. Since the number of documents is only about 68000, one can encode an element of the index in 17 bits. The first element $d_{i,1}$ for each word w_i will indeed be encoded that way, but the subsequent elements $d_{i,j}$ for $j > 1$ will be delta encoded, that is, we store actually $d_{i,j} - d_{i,j-1}$. These differences have a skew distribution, and need, on the average, only 3.55 bits for their encoding. The classes corresponding to the different lengths are Huffman encoded, yielding an average codeword length of 3.45 bits.

The best encoding for the the frequencies $f_{i,j}$ is a unary one. Table 4 lists for the first few values, the percentage of (d, f) pairs having that value in their $f_{i,j}$ field, indicating that there is a rapid exponential decrease. The value 1 can thus be encoded by a single bit 1, the value 2 by 01, 3 by 001, etc. This yields an average of 2.13 bits for each $f_{i,j}$ value.

$f_{W,j}$	1	2	3	4	5	6	7	8	9	$\cdots$
%	67	15	6	3	2	1	1	0.7	0.5	$\cdots$

Table 4: *Distribution of $f_{W,j}$*

The location pointers $D_{i,j}$, for $j > 1$, pointing into the Coordinates file, can also be encoded by their differences, similarly to the $d_{i,j}$ values. Partitioned into classes by their lengths in bits (refer to the last line of Table 3), one needs 3.3 bits on the average for each value, plus 1.96 bits for an average Huffman codeword indicating to which class it belongs.

Table 5 summarizes the average lengths for one word. The first column is the parameter we compute, the second column holds the average number of times the corresponding parameter appears for a single word in the concordance, and the third column holds the average number of bits needed to represent that parameter.

parameter	mult factor	number of bits
$d_{i,1}$	1	17 bits
$d_{i,j}$ for $j > 1$	51.33	3.55+3.45 = 7.0 bits
$f_{W,j}$	52.33	2.13 bits
$D_{i,j}$ for $j > 1$	51.33	3.3+1.96 = 5.26 bits
header block	111.48	8 bits
coordinate	111.48	7.33 bits

Table 5: *Components of the compressed coordinate*

To calculate the total number of bits needed for each coordinate in the suggested compressed concordance, we multiply the second and third columns, and then divide the total sum by the average number of coordinates per a word. Following the above statistics, we get an average of 22.12 bits = 2.77 bytes for each coordinate. Recall that the original, uncompressed coordinate was of length 8 bytes, so we get a compression ratio of 2.9, in spite of the fact that we have added also information on the local frequencies $f_{i,j}$.

3.3 The Algorithm

The algorithm below shows how to work with the compressed concordance. Given a query

$$\mathcal{Q} = q_1\ (l_1, u_1)\ q_2\ (l_2, u_2) \cdots q_{m-1}\ (l_{m-1}, u_{m-1})\ q_m,$$

where q_i $(1 \le i \le m)$ is a term. The couple (l_i,u_i) imposes a lower an upper limit on the distance from q_i to q_{i+1}, that is, an m-tuple of coordinates $(c_1, \ldots, c_m)$ is considered relevant if c_i belongs to the list of coordinates of q_i and

$$l_i \le dist(c_i, c_{i+1}) \le u_i \qquad \text{for } 1 \le i < m.$$

Negative distance means that q_{i+1} may appear before q_i in the text. The distance is measured in words. Note that this is a conjunctive query, and in real life applications, each terms q_i would in fact stand for a disjunction of several terms, all considered equivalent for the given query.

In the algorithm below, we use the following notations: *HVal(w)* and *CVal(w)* return the pointers h and c, respectively, of the word w from the dictionary. *fp* holds the position (in bits) in the Headers file. *index(w)* returns the index of the word w in the dictionary. The constant HS denotes the size of each header block (8 bits in our implementation).

```
AND Processing(q1, q2)
i ← 1,    j ← 1
a ← index(q1),    b ← index(q2)
fp1 ← HVal(q1),    fp2 ← HVal(q2)
Queue1 ← empty,    Queue2 ← empty
limit1 ← HVal(a + 1),    limit2 ← HVal(b + 1)
pos1 ← 0,    pos2 ← 0
mask ← 11111100
read ⟨d_{a,i}, f_{a,i}⟩ from H(q1)
read ⟨d_{b,j}, f_{b,j}⟩ from H(q2)
WHILE  fp1 < limit1  AND  fp2 < limit2  DO:
    IF  d_{a,i} < d_{b,j}  THEN:
        IF  i > 1  THEN:
            read deltaCode from H(q1)
            Push deltaCode into Queue1
        fp1 ← fp1 + HS · f_{a,i}
        i ← i + 1
        read ⟨d_{a,i}, f_{a,i}⟩ from H(q1)
    ELSE IF  d_{a,i} > d_{b,j}  THEN:
        IF  j > 1  THEN:
            read deltaCode from H(q2)
            Push deltaCode into Queue2
        fp2 ← fp2 + HS · f_{b,j}
        j ← j + 1
        read ⟨d_{b,j}, f_{b,j}⟩ from H(q2)
    ELSE: //the same document number
        ii ← 1
        jj ← 1
        WHILE  ii < f_{a,i}  AND  jj < f_{b,j}  DO:
            IF  h1_{ii} ∧ mask  <  h2_{jj} ∧ mask  THEN: //compare first 6 bits of the header
                ii ← ii + 1
            ELSE IF  h1_{ii} ∧ mask  >  h2_{jj} ∧ mask  THEN:
                jj ← jj + 1
```

33 ELSE: *// p and s fields in the header are equal*
34 *//compute the position of two corresponding coordinates*
35 $pos1 \leftarrow CVal(q_1)$ + sum of elements in Queue1
36 $pos2 \leftarrow CVal(q_2)$ + sum of elements in Queue1
37 *//compare the extracted coordinates*
38 WHILE $h1_{ii} \wedge mask = h2_{jj} \wedge mask$ DO:
39 read $Coord1.p, s$ from $pos1$
40 read $Coord2.p, s$ from $pos2$
41 IF $Coord1.p, s < Coord2.p, s$ THEN:
42 $ii \leftarrow ii + 1$
43 ELSE IF $Coord1.p, s > Coord2.p, s$ THEN:
44 $jj \leftarrow jj + 1$
45 ELSE: *// Coord1.p,s = Coord2.p,s*
46 *//make sure that the w fields are in the right range*
47 read $Coord1.w$ and $Coord2.w$
48 IF $l \leq Coord1.w - Coord2.w \leq u$ THEN:
49 return $Coord1$ and $Coord2$
50 ELSE IF $Coord1.w < Coord2.w$ THEN:
51 $ii \leftarrow ii + 1$
52 ELSE:
53 $jj \leftarrow jj + 1$
54 *//we compared all the headers that has the same p and s codes*
55 *//and didn't find a match*
56 return false
57 *//we finished to compare all $H(q_1)$ and $H(q_2)$ and found no match*
58 return false

Note that up to line 38, the processing deals only with the Headers file, which is used in its compressed form, the main idea of the suggested compression being that compressed headers can be compared directly as if they were numbers, that is, the compression methods preserve order.

4. Experimental design

It obviously makes no sense to try to compare empirically the retrieval time by the compressed algorithm to that of decompression and afterwards retrieval, using a set of "random" queries. A query consisting of random terms will most probably retrieve an empty set of locations. An empirical study should thus involve what could be called a "typical query", though this is hard to define. We therefore leave the present proposal on the theoretical level.

5. Conclusions

Although the compression methods presented do not necessarily give the most effective compression, they give a quite good compression ratio after all on the one hand, and on the other hand allow working directly with the compressed concordance, thus saving expensive I/O and CPU operations.

References

[1] Anh V., and Moffat A., Compressed inverted files with reduced decoding overheads. *Proc. 21st Annual SIGIR Conference on Research and Development in Information Retrieval*, pages 290-297, 1998.

[2] Bookstein A., Klein S.T., and Raita T., Model based concordance compression. *Proc. IEEE Data Compression Conference*, pages 82-91, Snowbird, Utah, March, 1992.

[3] Choueka Y., Fraenkel A. S., Klein S. T., and Segal E., Improved techniques for processing queries in full-text systems. *Proc. of ACM Conference on research and development in Information Retrieval*, pages 306-315, ACM press, NY, June, 1987.

[4] Choueka Y., Fraenkel A.S., and Klein S.T., Compression of concordances in full-text retrieval systems. *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 597-612, Grenoble, France, June , 1988.

[5] Elias P., Universal codeword set and representations of the integers, *IEEE Trans. Information Theory*, Vol **IT 21(2)**, pages 194-203, Mar, 1975.

[6] Farach, M. & Thorup, M. (1995). String Matching in Lempel-Ziv Compressed Strings, *Proc. 27th Annual ACM Symposium on the Theory of Computing*, 703–712.

[7] Engelson V., Fritzson D., Fritzson P., Lossless compression of high-volume numerical sata from simulations, *Data Compression Conference*, page 547, Snowbird, Utah, 2000.

[8] Fenwick P.M., Punctured Elias codes for variable-length coding of the integers, Report No 137, The University of Auckland, Department of Computer Science, Dec. 1996.

[9] Fraenkel A.S., All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, Expanded Summary, *Jurimetrics J.* **16** 149–156, 1976.

[10] Fraenkel A.S., and Klein S.T., Robust universal complete codes for transmission and compression, *Discrete Applied Mathematics*, vol. **64**, pages 31-55, 1996.

[11] Heaps J., *Information Retrieval : Computational and Theoretical Aspects.* New York, NY, Academic Press, Inc., 1978.

[12] Klein S.T., Shapira D., Pattern matching in Huffman encoded texts, *Information Processing & Management* **41**(4), pages 829–841. 2005.

[13] Klein S. T., and Shapira D., Searching in compressed dictionaries, *Proc. of Data Compression Conference*, pages 142-151, Snowbird, Utah, 2002.

[14] Linoff G., and Stanfill C., Compression of indexes with full positional information in very large text databases. *Proc. of the 16th Inter. ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 88-95, Pittsburgh, PA, June, 1993.

[15] Manber, U., A Text Compression Scheme That allows Fast Searching Directly in the compressed File, *ACM Transactions on Information Systems (TOIS)* **15**, 124–136, 1997.

[16] Moura E., Navarro G., Ziviani N., and Baeza-Yates R., Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, vol. **18(2)**, pages 113-139, 2000.

[17] Witten I.H., Moffat A., and Bell T.C., *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann, 2nd edition, 1999.

[18] Takeda, M., Miyamoto, S., Kida, T., Shinohara, A., Fukamachi, S., Shinohara, T. & Arikawa, S. Processing Text Files as Is: Pattern Matching over Compressed Texts, Multi-Byte character Texts, and Semi-Structured Texts, *Proc. of String Processing and Information Retrieval (SPIRE–02)*, 170–186, 2002.

[19] Westmann, T., and Kossmann, D.: The implementation and performance of compressed databases. *SIGMOD Record*, **29(3)**, pages 55-67, 2000.