g-binary: a new non-parameterized code for improved inverted file compression

Ilias Nitsos[1], Georgios Evangelidis[1], and Dimitrios Dervos[2]

[1] Department of Applied Informatics, University of Macedonia
156 Egnatia Str., 54006 Thessaloniki, Greece
{nitsos, gevan}@uom.gr
[2] Department of Information Technology, TEI
P.O. Box 14561, 54101 Thessaloniki, Greece
dad@it.teithe.gr

Abstract. The inverted file is a popular and efficient method for indexing text databases and is being used widely in information retrieval applications. As a result, the research literature is rich in models (global and local) that describe and compress inverted file indexes. Global models compress the entire inverted file index using the same method and can be distinguished in parameterized and non-parameterized ones. The latter utilize fixed codes and are applicable to dynamic collections of documents. Local models are always parameterized in the sense that the method they use makes assumptions about the distribution of each and every word in the document collection of the text database. In the present study, we examine some of the most significant integer compression codes and propose *g-binary*, a new non-parameterized coding scheme that combines the Golomb codes and the binary representation of integers. The proposed new coding scheme does not introduce any extra computational overhead when compared to the existing non-parameterized codes. With regard to storage utilization efficiency, experimental runs conducted on a number of TREC text database collections reveal an improvement of about 6% over the existing non-parameterized codes. This is an improvement that can make a difference for very large text database collections.

1 Introduction

Among the numerous schemes that have been developed for indexing text databases the most popular is the inverted file [4], [12], [11]. The success of the inverted file is closely related to the great advances in the field of integer compression codes [7], [3], [6], [8], [1], [9]. There exist several such codes that allow the average pointer (an integer value) inside an inverted file to be stored in less than 1 byte [11], thus producing very compact indexes. Compressed indexes are also fast, because the required number of disk accesses is small and the decompression CPU cost is low [10].

In the present paper we consider a number of compression codes and focus mainly on the Golomb code [8], [6], [12], [9] and the binary representation of integers. There exist two major categories of compression models for inverted files:

the *global* and the *local* models. Global models are further divided into *parameterized* and *non-parameterized*. The former involve some parameter that reflects the distribution of the integers inside the inverted files. Non-parameterized models produce fixed codes and are useful when dynamic collections of documents are stored. On the other hand, local models are always parameterized.

In Section 2 we examine several codes that have been developed for compressing integers inside the inverted files and the models they are based on. In Section 3 we introduce a new non-parameterized code, g-binary, that comprises a combination of the Golomb code and the binary representation of integers. The performance of the new code against some TREC text database collections is tested in Section 4. In Section 5, we perform an analysis on the length of the codes for all integers obtained by the coding schemes we examine in this paper. We show that g-binary is always equal or better in compression efficiency than the popular non-parameterized codes for certain integer intervals. Finally, in Section 6 we present our conclusions.

2 Codes for Compressing Inverted Files

In the inverted file index, each one of the N (say) text database documents is represented by a positive integer $d \in [1, \ldots, N]$. For each distinct word t, an inverted list is created that stores all the document numbers d containing t. Inverted lists are stored in a single file, known as the inverted file [4].

The integer numbers are stored in the inverted file in a way suitable for allowing compression by means of run length encoding [6]: instead of storing the absolute numbers d of the documents containing a word, inverted lists store their differences. For example, instead of storing the d list $\{2, 9, 10, 15, 16, 20\}$, for word t, one could store the corresponding d-gap list $\{2, 7, 1, 5, 1, 4\}$. This results in storing smaller (in value) and more frequently occurring integers, in general. The initial list may easily be reconstructed from the d-gap list.

In the subsections that follow, a number of popular compression codes are considered that utilize run length encoding to implement compression. The codes are based on models that take into consideration the probability distribution of d-gap sizes. In this respect, small bit codes are assigned to frequent d-gap sizes and larger bit codes to rare ones. Depending on whether they involve or not the storage of some kind of parameter, the models and their codes are categorized as being *parameterized* or *non-parameterized*, respectively.

2.1 Non-parameterized codes

Unary code. In the unary coding scheme each positive integer x is represented by x-1 ones followed by a zero. For example, number 5 is stored as 11110. This means that the bit length of integer x in unary is $len_u(x) = x$. A list of the unary codes for the first ten positive integers is shown in Table 1.

The unary code is equivalent to assigning a probability of 2^{-x} to gaps of length x [11].

Table 1. Examples of codes for integers

				Golomb			g-binary	
x	unary	γ code	δ code	b=2	b=3	b=4	b=2	b=3
1	0	0	0	00	00	000	00	00
2	10	100	1000	01	010	001	010	0100
3	110	101	1001	100	011	010	011	0101
4	1110	11000	10100	101	100	011	10000	01100
5	11110	11001	10101	1100	1010	1000	10001	01101
6	111110	11010	10110	1101	1011	1001	10010	01110
7	1111110	11011	10111	11100	1100	1010	10011	01111
8	11111110	1110000	11000000	11101	11010	1011	101000	100000
9	111111110	1110001	11000001	111100	11011	11000	101001	100001
10	1111111110	1110010	11000010	111101	11100	11001	101010	100010

Elias codes. Two popular non-parameterized codes for compressing integers, that result in significant savings, are the γ and δ codes introduced by Elias in [3]. In the γ coding scheme each positive integer x is encoded as follows:

- store number $1 + \lfloor log_2 x \rfloor$ in unary.
- store the remainder $x - 2^{\lfloor log_2 x \rfloor}$ in binary using $\lfloor log_2 x \rfloor$ bits.

It follows that the bit length of integer x in γ code is $len_\gamma(x) = 2\lfloor log_2 x \rfloor + 1$.
In the δ coding scheme each positive integer x is encoded as follows:

- store number $1 + \lfloor log_2 x \rfloor$ using γ code.
- store the remainder $x - 2^{\lfloor log_2 x \rfloor}$ in binary using $\lfloor log_2 x \rfloor$ bits.

It is shown [11] that the bit length of integer x in δ code is $len_\delta(x) = \lfloor log_2 x \rfloor + 2\lfloor log_2(1 + \lfloor log_2 x \rfloor) \rfloor + 1$.

A list of the γ and δ codes for the first ten positive integers is shown in Table 1. The γ code is equivalent to assigning a probability of $\frac{1}{2x^2}$ to gap x, whereas, the δ code assigns a probability of $\frac{1}{2x(logx)^2}$ [11].

2.2 Parameterized codes

Golomb code for the global Bernoulli model. Let us suppose, that we have an N-document text database that contains n distinct words and f index pointers (i.e., f distinct *"document, word"* pairs).

The global Bernoulli model assumes that large text databases are homogeneous, i.e. the words are distributed uniformly across the N documents. This, in turn, implies that the probability of a randomly selected word to appear in any one randomly selected document is $p = f/(N \times n)$. It has been shown that the d-gaps in this case can be efficiently represented by the Golomb code [6]. According to this code, for a given parameter b, each positive integer x is encoded as follows:

- store number $q + 1$ in unary, where $q = \lfloor (x-1)/b \rfloor$.
- store the remainder $r = x - q \times b - 1$ in binary using either $\lfloor log_2 b \rfloor$ or $\lceil log_2 b \rceil$ bits [11].

It follows that the bit length of integer x in Golomb code is at most $len_g(x) = \lfloor (x-1)/b \rfloor + 1 + \lceil log_2 b \rceil$.

A list of Golomb codes for the first ten positive integers, for some values of the parameter b, is shown in Table 1. At the decompression phase, the original number is computed as $x = r + q \times b + 1$. For a given value of p, b is calculated as follows: $b = \lceil \frac{log_2(2-p)}{-log_2(1-p)} \rceil$ [5].

Golomb code for the local Bernoulli model. The difference between the global and the local Bernoulli model is that the latter does not assume uniformity in the distribution of words across the N-document text database. This in turn implies that in general a different b parameter is associated to each one list [2]. The value of b for the d-gap list of word t is calculated as in the case of the global Bernoulli model, except that now $p = f_t/N$, where f_t is the number of documents in the text database containing t. This implies that for each distinct word t in the text database, the value for f_t must also be stored alongside with the inverted list of the word in order to be possible to compute the value of b at decoding time. The value for f_t is stored at the head of each list, using the γ code [11]. During the inverted list decompression phase, the f_t value is decoded first, then the b parameter is calculated and decompression continues with the rest of the list.

3 g-binary

In this section we introduce g-binary, a combination of the Golomb code and the binary representation of an integer. The aim is to produce a non-parameterized code that reduces the number of bits used to store a d-gap. The main idea behind the proposed new code is to store an integer using its exact binary representation. The problem is that the length of the binary representation must also be stored so that decoding is possible. After experimenting with all the popular codes for integers and examining the lengths of the bit sequences produced, we came down to the conclusion that the best code for storing the above lengths is the Golomb code for certain global values of b.

In g-binary, for a given b value, each positive integer x is encoded as follows:

- store the length m of the binary representation of x using the Golomb code for the globally selected parameter b.
- store the exact binary representation of x excluding its most significant bit (which is always 1).

Essentially, the g-binary code of an integer is the concatenation of the Golomb code that represents the length of the binary representation of the integer and the actual binary representation of that integer (without its most significant bit). Thus, the compression and decompression algorithms share the same complexity with the other codes (γ, δ and Golomb).

We use the notations len_b and len_{gb} for the lengths of the exact binary and g-binary representations respectively. The length of the exact binary representation for any integer $x > 0$ is $len_b(x) = \lfloor log_2 x \rfloor + 1$ bits. For example, the exact binary representation of integer 9 is 1001 and requires $\lfloor log_2 9 \rfloor + 1 = 4$ bits. In g-binary, we omit the first bit of this representation which is always 1, thus, we are left with $\lfloor log_2 x \rfloor$ bits. In order to store $m = \lfloor log_2 x \rfloor + 1$ (i.e., the length of the exact binary representation of x) using the Golomb coding scheme, we need at most $len_g(m) = \lfloor (m-1)/b \rfloor + 1 + \lceil log_2 b \rceil$ bits (see Section 2.2). This means that the total number of bits used by g-binary adds up to at most

$$len_{gb}(x) = len_g(m) + len_b(x) - 1 = \lfloor (m-1)/b \rfloor + 1 + \lceil log_2(b) \rceil + \lfloor log_2(x) \rfloor \quad (1)$$

or

$$len_{gb}(x) = \lfloor \lfloor log_2(x) \rfloor / b \rfloor + 1 + \lceil log_2(b) \rceil + \lfloor log_2(x) \rfloor \quad (2)$$

We will use an example to demonstrate the coding and decoding stages for the g-binary code. Let us suppose that we have to code the following integer list: 12 (1100), 19 (10011), 75 (1001011), 1 (1). We will use Golomb (b = 2) for the coding of the lengths of the binary representations of the integers.

The exact binary representation for integer 12 is 1100. The length of the binary representation is 4 and it is encoded using the Golomb code (b=2) as 101 (see Table 1). Eventually, 12 is stored as 101,100 where the first part is the length of the exact binary representation of 12 and the second part is the binary representation of 12 itself without its most significant bit. The comma between the length and the binary representation is not stored; we use it here to facilitate our presentation. The rest of the numbers are encoded in a similar way as [19: 1100,0011], [75: 11100,001011] and [1: 00]. In the case of the last number 1, it is only the length of the binary representation that is stored; the decoding process stage proceeds without any problem, since 1 is the only number that is represented with just one bit.

During decompression, the Golomb value is decoded first and reveals m, the length of the binary representation of the integer we try to decode. Next, the m-1 bits of the remaining bit sequence are retrieved. By adding 1 in the beginning of the m-1 bits we obtain the exact binary representation of the integer in question. Decompression continues with the rest of the bit sequence.

Several g-binary versions can be obtained by assigning different values to b. Any single version can be regarded as a non-parameterized code. Table 1 lists the g-binary codes for the first ten positive integers, when b=2 and b=3. In the next section, we examine whether some of these versions produce bit sequences for integers that result in better compression ratios, compared to the existing non-parameterized codes.

4 Experimental Results

The fifth volume of the TREC collection comprises the testbed for the storage utilization efficiency measuring test runs that we present in this section. The volume contains 475MB of articles from the Los Angeles Times (LAT), published during the January 1, 1989 – December 31, 1990 period, plus 470MB of documents of the Foreign Broadcast Information Service (FBIS) from 1994. The profiles for the LAT and FBIS collections as well as of their concatenation (LATFBIS) are shown in Table 2.

Table 2. Testbed collection profiles

Collection	Size (MB)	Distinct words (n)	Total documents (N)	Total pointers (f)
LAT	475	288823	130472	31193292
FBIS	470	247563	131167	34982316
LATFBIS	945	437864	261639	66175608

We implemented and compared the following codes:

– Elias γ code for all the gaps in all the lists of the inverted file. This code will be referred to as "γ-code".
– Elias δ code for all the gaps in all the lists of the inverted file. This code will be referred to as "δ-code".
– Golomb code for the local Bernoulli model. An inverted list is created for every word in the collection. Each list also stores the value for f_t (document frequency for term t) using the γ code. The f_t overhead for each word is included in the final results. This code will be referred to as "Golomb".
– The g-binary code for various values of the b parameter (see Section 3). This code will be referred to as "g-binary".

Figure 1 illustrates the results obtained for the LAT, FBIS, and LATFBIS collections. In each graph, the average number of bits per pointer is plotted as a function of b and this is why the curves for "γ-code", "δ-code" and "Golomb" are horizontal lines. For the "Golomb" code, in particular, it has to be stated that it is a local, parameterized code involving a number of b values that are calculated and stored alongside each inverted list inside the inverted file, according to the model discussed in Section 2. In this respect, the "Golomb" line represents the average bit size per pointer and it is not affected by the b values of the x-axis. The latter are used to differentiate the g-binary code versions and for each such version they are applied globally on the entire inverted list.

A first observation is that the "γ-code" line is always above the "δ-code" line. This means that the γ code produces larger indexes than the δ code, when applied globally on an entire inverted file. This finding confirms similar results presented in [8]. Another observation is that the "γ-code" compression ratio is

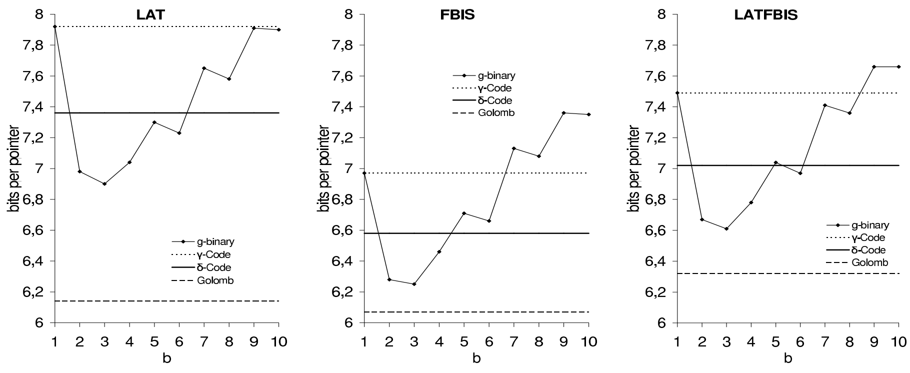

Fig. 1. Comparison of the γ, δ, Golomb and g-binary codes using the LAT, FBIS and LATFBIS collections

equal to the "g-binary" compression ratio when b=1. This is expected because for b=1, g-binary is identical to γ code: they both produce the same bit sequences for all integers. Another expected result is that the "Golomb" code achieves the best compression among the existing codes, because it is a local, parameterized code. Unfortunately, such codes are not suitable for dynamic document collections.

The results demonstrate that the compression efficiency of "g-binary" is maximized when b is 2 and 3 and that in general it decreases as b increases beyond 4. More specifically, "g-binary" performs better than the "γ-code" and "δ-code" when the value of the b parameter is set to 2 or 3. The gain achieved ranges from 0.3 to 0.4 bits per 7-bit (average) pointer. This is about half the gain achieved by the popular parameterized Golomb code. It represents a significant gain since the two "g-binary" code versions are non-parameterized and they can be used to compress dynamic text database collections. We have run experiments with many other text database collections and, in all cases, we have obtained results similar to the ones presented.

5 Analysis

In this section we follow an analytic approach and try to investigate the conditions under which g-binary for $b = 2$ and 3 achieves better compression than γ and δ code. To achieve this we examine the code lengths we obtain when we code all integers using these three methods.

For any integer $x > 0$ the length difference between the γ and g-binary codes is at least:

$$len_{\gamma}(x) - len_{gb}(x) = \lfloor log_2(x) \rfloor - \lfloor \lfloor log_2(x) \rfloor / b \rfloor - \lceil log_2(b) \rceil \quad (3)$$

For any integer $x > 0$ the length difference between the δ and g-binary codes is at least:

$$len_{\delta}(x) - len_{gb}(x) = 2\lfloor log_2(1 + \lfloor log_2 x\rfloor)\rfloor - \lfloor\lfloor log_2(x)\rfloor/b\rfloor - \lceil log_2(b)\rceil \quad (4)$$

We can substitute $\lfloor log_2(x)\rfloor + 1$, that is equal to the number of bits used for the binary representation of number x, with y. The above equations are respectively transformed into

$$len_{\gamma}(y) - len_{gb}(y) = y - 1 - \lfloor(y-1)/b\rfloor - \lceil log_2(b)\rceil \quad (5)$$

and

$$len_{\delta}(y) - len_{gb}(y) = 2\lfloor log_2 y\rfloor - \lfloor(y-1)/b\rfloor - \lceil log_2(b)\rceil \quad (6)$$

Apparently, the latter equations are functions of y, i.e., the number of bits that are used for the binary representation of the integers. Figures 2 and 3 are the graphical representation for equations 5 and 6 respectively.

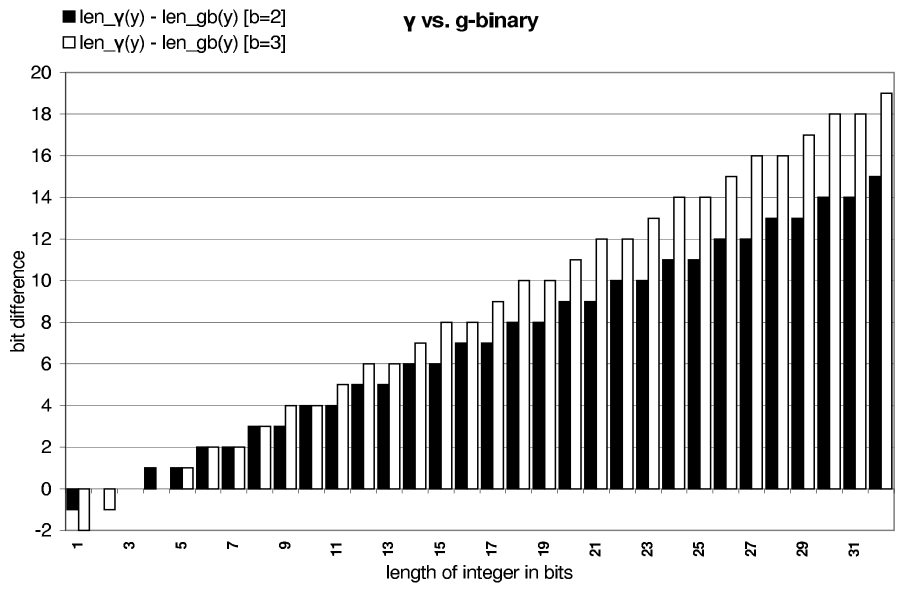

Fig. 2. Comparing γ code and g-binary

The results demonstrate (see Figure 2) that the g-binary code for $b = 2$ has equal or better compression efficiency than the γ code for all integers except 1. Also, the g-binary code for $b = 3$ has equal or better compression efficiency than the γ code for all integers except 1, 2 and 3. That explains the improved performance of g-binary when compared to γ code.

The results demonstrate (see Figure 3) that the g-binary code for $b = 2$ has equal or better compression efficiency than the δ code for all integers in

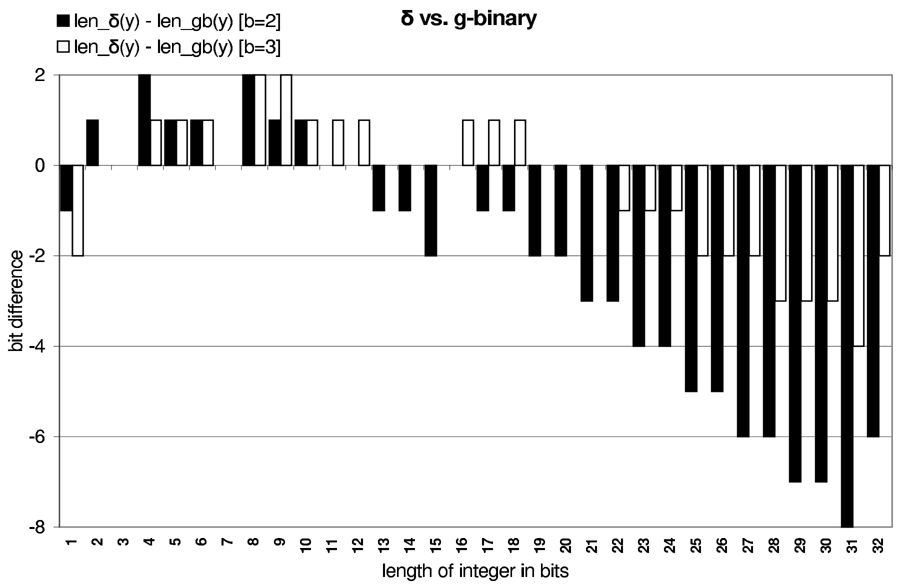

Fig. 3. Comparing δ code and g-binary

$[2 \ldots 2^{12} - 1]$ or $[2 \ldots 4095]$. Also, the g-binary code for $b = 3$ has equal or better compression efficiency than the δ code for all integers in $[2 \ldots 2^{21} - 1]$ or $[2 \ldots 2097151]$. This is an important observation for g-binary, because large gaps inside the inverted lists only occur in the case of rare terms. This means that large integers are not very frequent and that the majority of the integers stored are relatively small. Moreover, very small d-gap values (1 or 2) only occur in the case of very frequent terms. The latter may appear in stop-word lists, i.e., lists of words such as "the", "a", "of", etc., that are usually not indexed. In our experiments we did not take into consideration such lists, because we did not want to favor our method in any way. In other words, we indexed all words.

6 Conclusion

In the present study we consider the problem of integer number encoding in the context of index compression for dynamic document collections. The relevant research literature suggests that the γ and δ codes comprise the preferred choice for index compression.

We introduce g-binary, a new group of codes, involving a tunable parameter b that determines the Golomb code version used to store the length of the binary representation of an integer. A g-binary code consists of the length of the binary representation of an integer stored using Golomb code and the exact binary representation of the integer itself (without its most significant bit that is always 1). Experimental runs of the proposed group of codes conducted against two

TREC collections reveal that by setting $b = 2$ and 3 we obtain two global non-parameterized codes that improve the size of the encoded index file by nearly 6% when compared to the size obtained by δ codes. Moreover, by construction the g-binary code does not introduce any extra CPU overhead during the encoding and decoding phases when compared to the γ and δ codes.

References

1. Blandford, D., Blelloch, G.: Index Compression through Document Reordering. Proceedings of the Data Compression Conference. (2002)
2. Bookstein, A., Klein, S.T., Raita, T.: Model based concordance compression. In Storer and Cohn. (1992) 82–91
3. Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory, Vol. IT-21. (1975) 194–203
4. Fox, E., Harman, D., Baeza-Yates, R., Lee, W.: Inverted Files. In: Frakes, W., Baeza-Yates, R. (eds): Information Retrieval: Data Structures and Algorithms. Prentice-Hall, Englewood Cliffs, NJ, Chapter 3. (1992) 28–43
5. Gallager, R.G., van Voorhis, D.C.: Optimal source codes for geometrically distributed alphabets. IEEE Transactions on Information Theory, Vol. IT-21. (1975) 228–230
6. Golomb, S.W.: Run-length Encodings. IEEE Transactions on Information Theory, Vol. IT-21. (1966) 399–401
7. Huffman, D.A: A method for the construction of minimum redundancy codes. Procedures IRE, Vol.40(9). (1952) 1098–1101
8. Moffat, A., Zobel, J.: Paremeterised Compression for Sparse Bitmaps. 15th Ann Int'l SIGIR, Denmark (1992) 274–285
9. Scholer, F., Williams, H.E., Yiannis, J., Zobel, J.: Compression of Inverted Indexes for Fast Query Evaluation. SIGIR, Finland (2002) 222–229
10. Williams, H. E., Zobel, J.: Compressing Integers for Fast File Access. The Computer Journal, Vol. 42. (1999) 193–201
11. Witten, I.H., Moffat A., Bell T.C.: Managing Gigabytes. Compressing and Indexing Documents and Images. Academic Press (1999)
12. Zobel, J., Moffat, A., Ramamohanarao K.: Inverted Files Versus Signature Files for Text Indexing. ACM Transactions on Database Systems, Vol. 23. (1999) 369–410