VSEncoding: Efficient Coding and Fast Decoding of Integer Lists via Dynamic Programming

Fabrizio Silvestri
ISTI-CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
fabrizio.silvestri@isti.cnr.it

Rossano Venturini
ISTI-CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
rossano.venturini@isti.cnr.it

ABSTRACT

Encoding lists of integers efficiently is important for many applications in different fields. Adjacency lists of large graphs are usually encoded to save space and to improve decoding speed. Inverted indexes of Information Retrieval systems keep the lists of postings compressed in order to exploit the memory hierarchy. Secondary indexes of DBMSs are stored similarly to inverted indexes in IR systems. In this paper we propose *Vector of Splits Encoding* (VSEncoding), a novel class of encoders that work by optimally partitioning a list of integers into blocks which are efficiently compressed by using simple encoders. In previous works heuristics were applied during the partitioning step. Instead, we find the *optimal* solution by using a dynamic programming approach. Experiments show that our class of encoders outperform all the existing methods in literature by more than 10% (with the exception of Binary Interpolative Coding with which they, roughly, tie) still retaining a very fast decompression algorithm.

Categories and Subject Descriptors

H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*; E.4 [**DATA**]: CODING AND INFORMATION THEORY—*Data compaction and compression*

General Terms

Algorithms,Performance,Experimentation

Keywords

d-gap encoding, inverted index encoding, adaptive encoding, index compression

1. INTRODUCTION

Data management systems such as: DBMSs, Information Retrieval Systems, Search Engines and the alike, are continuously facing the problem of the so called data deluge[1]. At the petabyte scale we cannot think of data as something to be viewed but, instead, something that need to be first abstracted and then use this abstraction to extract knowledge from new (possibly fresher) data. To compute any result from (real) data it is necessary to store them on some sort of storage and this does not come for free, especially at the petabyte scale. Encoding data to save space is, therefore, of utmost importance to enable the effective exploitation of the very large datasets managed by today's systems. Consider that, even by changing scale, storage might be an issue. Personal devices size is constantly shrinking. Increasing data density in storage devices it is just not enough. We need to design technique enabling the efficient store of (relatively) large datasets in these devices. In these scenarios, data compression seems mandatory because it may induce a twofold advantage. On one hand, the obvious reduction in space occupancy allows more data to be stuffed within a single store unit. On the other hand, by fitting more data into faster memory levels, compression reduces the size of data to be transferred from the slower levels. This is a classic example of trading CPU cycles for decreased I/O latency and bandwidth. For example, given the amount of computing power on a modern multi-core CPUs, transferring a compressed payload from the disk and decompress its content into memory is still far cheaper than just transferring the uncompressed data. Not only data transfers from disk to memory benefit from compression, also data transfers from memory to CPU is also positively affected by compression as it is shown by IBM Memory Expansion Technology [1]. This is a very well know fact also in IR where many scientific results show hot to exploit this trade-off [19, 18, 20, 13].

In this paper we present *Vector of Split Encoding*, hereinafter VSEncoding, a novel class of encoders designed to efficiently represent lists of integers. Our encoders work by splitting the list in blocks and by encoding any integer in each block by using a fixed number of bits (namely, the number of bits required to represent the largest integer of the block). Obviously, we could have chosen to represent integers in a block by using codewords of variable length, this would have, perhaps, increased the compression efficiency but also the decoding time. The simpler choice is adopted to allow a very fast decoding algorithm. Thus, VSEncoding

[1]`http://glinden.blogspot.com/2006/10/advantages-of-big-data-and-big.html`

are, basically, block-based compression schemes where the critical difference with respect to previous block-based methods, e.g., P4D [21] and Simple9 [2], is the strategy used to choose the blocks. Differently from previous works, where heuristics were applied during the partitioning step [2, 21, 3], we optimally partition the lists in blocks via dynamic programming. The main contribution of our paper is the use of a *global* optimization technique that is able to discover the best possible block allocation given the encoding method. Our encoders are able to achieve very good compression performance and outperform all the existing methods in literature by more than 10% (with the exception of Binary Interpolative Coding with which they, roughly, tie). In particular, we report that our encoders are able to "*beat the entropy*" of the distribution of values in the lists. We observe that this is possible due to the fact that our encoders are able to exploit regularities in the lists that are not captured by the entropy. Regarding decoding speed, our encoders are faster than the state-of-the-art PForDelta-like encoders [21, 20], VBytes [19], Simple9 [2] and Simple16 [20].

The paper is organized as follows. Section 2 is used to fix useful notation. In Section 3 we describe some of the most popular encoding methods proposed in the literature. We present known techniques that are either suitable for encoding single integers, or specifically designed to compress lists of integers. In Section 4 we present our new class of integer list encoders. We also propose two of its instantiations that better exploit the skewness of the list to be encoded. Moreover, we show how to organize in memory the compressed representation of a list in order to achieve a very fast decompression algorithm. Section 5 show empirical comparison among our solutions and the most popular ones on three real datasets representing the posting lists of inverted indexes of three different document collections. We conclude the paper in Section 6 by presenting our plans for future work.

2. NOTATION

Let L denote a list of n strictly positive integers. For any list L, $L[i]$ denotes the i-th element, and $L[i:j]$ is the contiguous sublist of L ranging from position i to j, $0 < i \leq j \leq n$. Therefore, $L[1:n]$ denotes the entire list L. We say that L is sorted iff $L[i] < L[i+1]$, for any $0 < i < n$. Given an integer $L[i]$ we denote with $\text{bin}(L[i])$ its binary representation, and with $|\text{bin}(L[i])|$ its length in bits (namely, $|\text{bin}(L[i])| = \lfloor \log_2(L[i]) \rfloor + 1$).

Even if our encoders are able to compress any list of integers, in the experimental part of this paper we apply our solutions to lists of d-gaps [19] coming from inverted indexes. Given a sorted list of integers L, a list of d-gaps D is defined as follows: $D[1] = L[1]$, $D[i] = L[i] - L[i-1]$, $i > 1$. As an example, consider the list $L = \langle 1, 2, 12, 30, 32 \rangle$ we have the corresponding d-gap list $D = \langle 1, 1, 10, 18, 2 \rangle$. D-gap lists are be made up of smaller values than the original list[2]. Therefore, codes that represent small values with shorter codewords will result in more compact encodings for L.

For our purposes we are particularly interested in the distribution of the integers in the (d-gap) lists. The *skewness* of a list can be (informally) defined as the measure of the asymmetry of the distribution of its elements. In particular, a "positively skewed" distribution is a distribution where the mass of the distribution is concentrated on the left, i.e., an element of the list is rarely a large integer. It is worth noticing that the list distributions in which we are interested in practice are highly skewed. In particular, it has been shown that the distribution of d-gaps follows a power-law [19, 15, 20], which is an extremely positively skewed distribution, i.e., a large fraction of values are equal to '1'.

3. RELATED WORKS

The aim of section is that of introducing the most popular encoders that are going to be compared in the experiments section (Section 5). We divide known methods in two classes: *Integer encoder*s and *Integer List encoder*s. The former codes assign a distinct codeword to each possible integer. Thus, a list is compressed by replacing each integer with its corresponding codeword. Encoders in the second class, instead, are specifically designed to compress lists of integers and may encode any of them considering also its neighbors in the list. These methods are much more powerful than integer encoders since they can exploit regularities (e.g., clusters of almost equal integers) on the underlying list either to achieve higher compression or to provide faster decompression. As a consequence of this, methods in the second class may potentially be able to beat the entropy of the distribution of values in the underlying lists. Indeed, it is well-known that the compress size achievable by any of the former methods is lower bounded by entropy. Our methods belong to the class of Integer List encoders and are able to beat the entropy on the three tested datasets. Thus, we are sure that they achieve better compression than any integer encoder even without the need of an explicit experimental comparison.

3.1 Integer Encoders

In modern computer architectures, integers are usually represented (uncompressed) using 32 bits per integer. However, whenever the largest possible integer to be encoded, say $m = \max_{i \in [1,n]} L[i]$, is known, we can store each $L[i]$ as $L[i] - 1$ using only $\lceil \log_2 m \rceil$ bits[3]. This representation may result in a net saving of $32 - \lceil \log_2 m \rceil$ bits per integer with respect to the plain representation. This is the best compression we can hope to achieve whenever the underlying distribution of integers is uniform and m is an exact power of two. If m is not a power of two we can resort to *minimal binary code*s. Notice that, by assigning codewords of $\lceil \log_2 m \rceil$ bits, the fixed representation above wastes $2^{\lceil log_2 m \rceil} - m$ codewords. This implies that $2^{\lceil log_2 m \rceil} - m$ codewords can be shortened by one bit without loss of unique "decodability". This is done by using in the code all the prefixes of numbers in a given interval. If we use the regular binary numbers to encode the first six integers as $(000, 001, 010, 011, 100, 101)$, we miss '11' as a prefix. On the other hand the first six integers can be coded using a code $(00, 01, 100, 101, 110, 111)$. Note all possible prefixes of one bit $(0, 1)$ and all possible prefixes of two bits $(00, 01, 10, 11)$ appear in the code allowing the saving of one bit when encoding 0 and 1.

Fixed representation and minimal binary codes could be very inefficient for skewed distributions. This is the main

[2]Obviously, to recover L from D requires a second pass to "prefix-sum" up the values to have the original list back.

[3]We recall that L values are strictly positive and, thus, $\lceil \log_2 m \rceil$ bits suffices to represent a value from 0 to $m-1$.

motivation for integer encoders which assign to each integer a variable length codeword. The strategy adopted to assign codewords is crucial. Usually, each method is tuned to work (almost) perfectly on its "ideal" distribution of values. However, whenever the real distribution differs from the ideal one, the codewords lengths of various integers could be not suitable and the encoder could waste space. It is important to notice that each encoding is a prefix code: no valid codeword is prefix of another codeword and thus can be instantly decoded as it is read [14]. As a consequence, none of these kind of codes can beat the entropy of the underlying distribution of integers.

Unary (Unary). In the unary representation each integer value x is represented using $x-1$ bits equal to '1' followed by a '0' that acts as a terminator [19]. Therefore, the length of the encoding of an integer x is $|\mathsf{Unary}(x)| = x$. As an example, if $x = 5$ we have UN(5) = 11110.

Elias' Gamma (γ). In γ, an integer $x > 0$ is encoded by representing $|\operatorname{bin}(x)|$ (i.e. $\lfloor \log_2(x) \rfloor + 1$) in unary followed by $\operatorname{bin}(x)$ without its most significant bit [9]. Therefore, $|\gamma(x)|$ is equal to $2\lfloor \log_2(x) \rfloor + 1$. As an example, if $x = 5$ we have $\operatorname{bin}(x) = 101$ and thus $\gamma(5)$ is equal to 11001.

Elias' Delta (δ). In δ, an integer x is encoded by representing $|\operatorname{bin}(x)|$ by using γ followed by $\operatorname{bin}(x)$ without its most significant bit [9]. The length of $\delta(x)$ is $|\delta(x)| = \lfloor \log_2 x \rfloor + 2\lfloor \log_2 \log_2(x) \rfloor + 1$. For instance, $\delta(5) = 10101$.

Boldi&Vigna's Zeta (ζ_k). In a recent paper, Boldi and Vigna [4] propose a class of integer encoders that are suitable for lists of numbers drawn from a power-law distribution[4]. Given an positive integer parameter k, ζ_k encodes a positive integer x in the interval $\left[2^{hk}, 2^{(h+1)k} - 1\right]$ by writing UN$(h+1)$ followed by a minimal binary code of $x - 2^{hk}$ in the interval $\left[0, 2^{(h+1)k} - 2^{hk} - 1\right]$. Note that ζ_1 is equivalent to γ. As an example $\zeta_2(5) = 10001$, $\zeta_3(5) = 0101$, and $\zeta_4(5) = 00101$.

Others. In literature are known many other integer encoders [13]. Among them we recall Golomb [11] and its variation Rice. We do not enter into details of these two methods since it is well known that they are slower [20] than the previous encoders and. Also, since their space occupancy is bounded by the entropy, they cannot beat our encoders in compression.

The previous encoders are said to be bit-oriented encodings since their codewords may cross the boundary of a computer word. During decoding, this requires additional bitwise OR, mask, and shift operations that slows down the decoding phase. Other encoders are said to be byte/word-aligned codings since they try to find a workaround to this by aligning each codeword to byte (or word) boundary. Thus, usually they are faster but much less space efficient with respect to bit-oriented encodings.

Variable-Bytes (VBytes). A non negative integer x is represented in VBytes as a list of 7-bit entries. Each element of the list is prefixed with 0 except for the last one, which is prefixed with a 1 [13, 19]. The length in bits of $\mathsf{VBytes}(x)$ is given by $|\mathsf{VBytes}(x)| = 8\lceil(\lfloor \log_2 x \rfloor + 1)/7\rceil$ bits (or alternatively $\lceil(\lfloor \log_2 x \rfloor + 1)/7\rceil$ bytes). As examples, $\mathsf{VBytes}(5) = 10000101$, $\mathsf{VBytes}(129) = 00000001\ 10000001$.

[4]Recall that a discrete random variable Z is distributed as a power-law with parameter α whenever the probability of the event $Z = x$ is $\mathbb{P}(\{Z = x\}) = \frac{1}{\zeta(\alpha)x^{\alpha}}$.

3.2 Integer List Encoders

The main limitation of integer encoders is that they encode each integer in the list separately, without taking into consideration its neighbors. Instead, Integers list encoders may improve compression by, for example, exploiting clusters of almost equal integers in the underlying list.

Binary Interpolative Coding (Interpolative). A more sophisticated way of encoding a list of sorted integers is using the Binary Interpolative Coding of Moffat and Stuiver [12]. Starting from the assumption that in highly-skewed distributions integers usually appear *clustered* [5] within a list, Interpolative works by recursively splitting the interval of integers contained within a list and encoding the central element via minimal binary code. By doing this, whenever a (sub)list of consecutive numbers is found it is encoded using "*zero*" bits. Experiments performed throughout these years have shown that Interpolative is still the best encoding method for highly skewed lists of integers [19, 15, 20]. The major drawback of Interpolative is the poor performance exhibited at decoding time.

Simple9 (Simple9). It encodes groups of integers within a single 32-bit word. Basically, in Simple9 there are nine possible ways of encoding a list of positive integers: 28 1-bit integers, 14 2-bit integers, 9 3-bit integers (one bit unused), 7 4-bit integers, 5 5-bit integers (three bits unused), 4 7-bit integers, 3 9-bit integers (one bit unused), 2 14-bit integers, or 1 28-bit integer. The remaining four bits to complete the 32-bit word are used as status bits to represent which of the nine cases is used. Decompression is done by reading the status bits and, depending on their value, by applying a specific function that efficiently extracts all the integers in the word [2]. Simple9 wastes bits when encoding some combinations of integers. For instance, in encoding 5 5-bit integers we have three unused bits. To overcome this issue, Yan et al. have designed Simple16 [20], a different encoding schema for fitting sixteen different combinations of integers within a word. Experiments showed that Simple16 is more compact than Simple9 (from which it is inspired). Another variant of Simple9 , that reduces the wastage of bits of Simple9 is `slide` [3]. Since it incurs in a higher decoding complexity, we do not include `slide` in our experiments.

PForDelta (P4D). P4D encodes blocks of k consecutive integers (e.g. $k = 128$ integers). The method firstly finds the smallest b such that most (e.g. 90%) of the integers in the block are non greater than 2^b. Then, it performs the encoding by storing each integer as a b-bit entry. Each entry is then packed within a list of $\lceil k \cdot b \rceil$ bits. The parameter k is usually chosen to be a multiple of 32. This implies that the $k \cdot b$ bits list is always word aligned regardless of the value of b. Those integers that do not fit within b bits are treated as exceptions and stored differently [21]. We actually refer to a different representation of P4D by Yan et al. [20] (called OPT-P4D). In this variant the number of exceptions is not forced to be smaller that 10% of the block length. Instead, it is chosen to minimize the space occupancy. Moreover, exceptions are stored in a separate array that is merged to the original sequence of codewords during the decoding phase. According to Yan *et al.* [20], this representation is more compact and not significantly slower than the original P4D.

4. VSENCODING: A CLASS OF INTEGER LIST ENCODERS

State-of-the-art integer list encoders use predefined schemes for partitioning a list into blocks and encoding each block separately. For example, P4D and its variations divide the list in to blocks of fixed length, and, then encode each block with b-bit codewords possibly generating exceptions for integers greater than 2^b. Instead, Simple9 and its variations greedily partition the list into blocks of variable length and encode each of them accordingly to predefined possibilities. Finally, Interpolative represents the the middle value of the list encodes the remaining part recursively by dividing the list in two almost equal parts. These methods have inefficiencies either in achieved compression or decompression speed. By fixing the block length, P4D-based encoders are not allowed to adapt themselves to regularities present in the lists. For example, the block length should be smaller for some portion of the list and larger for the others. Exceptions serve to attenuate the effect of misplacing integers of different magnitude in the same block. However, we pay their effort at a cost of introducing significant complications in the decompression algorithm that affect decompression speed. Simple9, instead, is too limited in possible choices which inevitably led itself to miss some regularities in the list. For example, grouping a run of 1s into a single block and encoding each of them with one bit, is possible only if the run has length at least 28. Finally, the Interpolative strategy is very effective in term of compression but slow due to its recursive compression/decompression algorithms.

In what follows we present our class of integer list encoders that overcome the above limitations. Our class of encoders is similar in the spirit to P4D and Simple9 but partitioning and encoding steps are done in a more principled way in order to maximize the achieved compression still retaining very simple and fast decompression algorithm. Our encoders (called, VSEncoding) are parametric with respect to two given integer encoders $\mathcal{M}_1$ and $\mathcal{M}_2$. Informally, the general scheme works as follow. We partition each list into blocks of variable length, and we encode the integers inside of each block with the number of bits, say b, required to encode the largest one. Finally, we encode the above value of b with $\mathcal{M}_1$ and the length of the block with $\mathcal{M}_2$. Obviously, the partition step is crucial for achieving high compression. On one hand, if a block is too large, we may waste a lot of space by encoding all its elements with b bits. On the other hand, if the block is too small, we may waste too much space in writing the value of b and the block length. Our solution uses a Dynamic Programming approach to find the optimal partition (i.e., the one that maximizes compression) with respect to $\mathcal{M}_1$ and $\mathcal{M}_2$. The partition step is discussed in Subsection 4.1, for the moment, let us define more formally our class of encoders assuming that any partition is given.

Let L be the list of n positive integers to compress and let S be the list of $m < n$ integers (called *Vector of Splits*), with $S[1] = 1$, and $S[m] = n+1$, that induces the given partition of L: each two consecutive elements $S[i]$ and $S[i+1]$ induce a block, namely $s_i = L[S[i] : S[i+1]-1]$. For any block s_i, let b_i be the minimum number of bits required to represent any integer in the block s_i, namely $\lceil(\log_2 \max(a \in s_i)\rceil)$, and let k_i be the number of elements in s_i, i.e. $k_i = S[i+1] - S[i]$. Given the two integer encoders $\mathcal{M}_1$ and $\mathcal{M}_2$, VSEncoding[5] encodes each block s_i by encoding

1. value $b_i + 1$ with $\mathcal{M}_1$;
2. value k_i with $\mathcal{M}_2$;
3. the k_i elements of s_i using b_i bits each.

Let us make an example to show how VSEncoding works. Let $L = \langle 8, 1, 1, 8, 1, 1\rangle$ be the list to encode, $S = \langle 1, 3, 5, 7\rangle$ be the given vectors of splits, $\mathcal{M}_1$ be γ, $\mathcal{M}_2$ be Unary. From S we can devise the following partition: $s_1 = L[1:2] = \langle 8, 1\rangle$, $s_2 = L[3:4] = \langle 1, 8\rangle$, and $s_3 = L[5:7] = \langle 1, 1\rangle$. The three blocks are encoded as:

1. $\gamma(b_1 + 1 = 4) = 11000$, Unary$(k_1 = 2) = 10$, 101 000;
2. $\gamma(b_2 + 1 = 4) = 11000$, Unary$(k_2 = 2) = 10$, 000 101;
3. $\gamma(b_3 + 1 = 1) = 1$, Unary$(k_3 = 2) = 10$.

Notice that the encoding of elements of third block requires no bits, since we can infer that they are all 1s by knowing the value of b_3.

Given a list L and a vector of splits S, we can easily compute the number of bits required by VSEncoding to encode L using the partition induced by S (which is denoted by $|\text{VSEncoding }(L, S)|$). This quantity can be computed by summing up the costs of encoding all the blocks as follows:

$$|\text{VSEncoding }(L, S)| = \sum_{i=1}^{m-1} c(S[i], S[i+1]-1) \qquad (1)$$

where $c(S[i], S[i+1]-1) = |\mathcal{M}_1(b_i+1)| + |\mathcal{M}_2(k_i)| + k_i b_i$ is the cost (in bits) required to encode the i-th block[6].

In the previous example we have that $|\text{VSEncoding }(L, S)| = c(S[1], S[2]-1) + c(S[2], S[3]-1) + c(S[3], S[4]-1) = 2(|\gamma(3)| + |\text{Unary}(2)| + 2\cdot 3) + |\gamma(1)| + |\text{Unary}(2)| + 2\cdot 0 = 29$ bits.

As we said before, the choice of correct partition is crucial to achieve high compression. To make a concrete example consider the partition induced by $S' = \langle 1, 2, 4, 5, 7\rangle$ on the same list. The compress obtained with the same choices of $\mathcal{M}_1$ and $\mathcal{M}_2$ has size $|\text{VSEncoding }(L, S')| = 22$ bits, which is more than 30% better than the previous one. In the next subsection we show how to efficiently compute the optimal vector of splits for a list L fixed $\mathcal{M}_1$ and $\mathcal{M}_2$, that is, among all the possible vector of splits, we select one that achieves the best compression.

4.1 Finding an Optimal Vector of Splits

The problem of finding the optimal encoding for a list L is formulated as the problem of finding the vector of splits S^* that minimizes $|\text{VSEncoding }(L, S)|$ defined in Equation 1 among all the possible 2^n vectors of splits $\mathcal{S}$. More formally, S^* is such that

$$S^* = \arg\min_{S \in \mathcal{S}} |\text{VSEncoding }(L, S)|$$

Since it is useful for the choices of $\mathcal{M}_2$s used in the experiments, we consider the case in which one can also fix the maximum length of the blocks by specifying a value maxK. Notice that this is actually a generalization of the problem

[5] Actually, since VSEncoding is a class of encoders parametric in $\mathcal{M}_1$ and $\mathcal{M}_2$, it should be denoted as $\text{VSEncoding}_{\mathcal{M}_1,\mathcal{M}_2}$ to make more explicit this dependence. Since in the following the role of $\mathcal{M}_1$ and $\mathcal{M}_2$ is unambiguous, we decide to drop this more precise notation in favor of legibility.

[6] Notice that the cost depends on the choices of $\mathcal{M}_1$ and $\mathcal{M}_2$.

```
Algorithm Optimizer($L[1,n], \mathcal{M}_1, \mathcal{M}_2$, maxK)
1. $E[1] = 0$; $P[1] = 1$;
2. for($i = 2$; $i <= n + 1$; $i = i + 1$)
3.     $b = 0$; $E[i] = +\infty$;
4.     for($j = i - 1$; $j >= \max(0, i - \mathsf{maxK})$; $j = j - 1$)
5.         if($b < \lceil \log_2(L[j]) \rceil$ $b = \lceil \log_2 L[j] \rceil$;
6.         $c(j,i) = (i-j)b + |\mathcal{M}_1(b+1)| + |\mathcal{M}_2(i-j)|$
7.         if($E[j] + c(j,i) < E[i]$)
8.             $E[i] = E[j] + c(j,i)$;
9.             $P[i] = j$;
```

Figure 1: The algorithm to find the optimal partition of a list $L[1,n]$ using encoders $\mathcal{M}_1$ and $\mathcal{M}_2$ to encode values of b and block length respectively and allowing only blocks of length at most maxK.

above: to have no limits on blocks lengths, it is enough to set maxK equal to the length of the list.

It is easy to prove that this problem can be solved via Dynamic Programming paradigm using the following recurrence:

$$E[i] = \min_{\max(0,j-\mathsf{maxK})<=j<i} (E[j] + c(j,i)) \qquad (2)$$

where

- $E[j]$ is the already computed optimal cost for encoding list up to its j - 1-th element;
- $c(j,i)$ is called *cost function* and, as we said before, accounts for the cost of encoding the sublist $L[j : i-1]$ as a single block (recall that $c(j,i) = |\mathcal{M}_1(\lceil \log_2 max(L[j : i-1]) \rceil + 1)|) | + |\mathcal{M}_2(i-j)| + (i-j) \cdot \lceil \log_2(max(L[j : i-1])) \rceil)$ as defined in the previous section).

To start the recurrence we set $E[1] = 0$, since it corresponds to the cost of encoding an empty list. Once we have solved Recurrence 2, the value of $E[n+1]$ tells us the cost of the optimal partition of L.

The above recurrence can be solved in $O(n \cdot \mathsf{maxK})$ by resorting to the classic algorithm for this type of recurrences [7] (see Algorithm 1). In this algorithm we start by setting $E[1] = 1$, then we compute entries of E from left to right (Steps 2–9 in Algorithm 1). At the generic step, we compute $E[i]$ by identifying an index $j^* < i$ among the ones having the minimum value of $E[j^*] + c(j^*, i)$. This index j^* is identified by simply trying all indexes j between i − maxK and $i - 1$ (Steps 4–9) with the only wariness of doing this from the largest index to the smallest one. In this way, we are able to compute the value b of sublist $L[j : i-1]$ knowing the value of b of sublist $L[j-1 : i-1]$ in constant time (Step 5). During the execution of the algorithm, we also keep track of above index j^* in the array P (Step 9), so that, at the end of the computation, we are able to reconstruct the vector of splits inducing the optimal partitioning by jumping back from $n+1$ through values of P (namely, $P[n+1]$, $P[P[n+1]]$, $P[P[P[n+1]]]$, and so on). In our tools we implemented this simple algorithm mainly due to the fact that experimental evidences show that good values of maxK are small constants between 16 and 64 for our choices of encoders $\mathcal{M}_1$ and $\mathcal{M}_2$. For completeness, we point out that faster algorithms are possible by adapting known solutions (see [10] and references therein). For example, by resorting to the result in [10], we can compute an $(1+\epsilon)$ approximate solution of Recurrence 2 in time $O(n \log_{1+\epsilon} n)$, where ϵ is an arbitrary positive value. Moreover, we are able to extend this result to compute the **exact** solution of Recurrence 2 in time $O(n \log^2 \mathsf{maxK})$ whenever the encoders $\mathcal{M}_1$ and $\mathcal{M}_2$ are chosen among most of integer encoders described in Section 3. As a final remark, we point out that in terms of encoding time VSEncoding is not less efficient than OPT-P4D [20]. OPT-P4D computes, for all the possible values of b, both space and time taken to encode/decode each list of integers. Therefore, encoding using OPT-P4D costs $O(bn)$ where b is 20, whereas VSEncoding costs $O(n \log^2 \mathsf{maxK})$, where in practical implementation maxK ranges between 16 and 64. Moreover, since OPT-P4D does two passes over data (one pass to encode and verify the occupied space, the second pass to verify the decoding speed), the two methods have comparable performances.

4.2 Experimented Instantiations

We can obtain a valid instantiation of our encoders by choosing any possible combination of integer encoders among the ones described in Section 3.1 or the myriad introduced in literature [13]. We tried many of them in our experimental investigation but we report here only the two most promising in terms of space achieve and decompression speed. It should not surprise that, since we particularly care about decompression speed, they are quite simple.

In the first instantiation (referred to as VSE in the experiments) we use two simple encoders. Given the list L to be encoded, we firstly compute the maximum value M of its elements, then $\mathcal{M}_1$ simply encodes possible values of b using fixed codewords of length $\lfloor \log_2 \lceil \log_2 M \rceil \rfloor + 1$ bits. As far as $\mathcal{M}_2$ is concerned, we still use a fixed representation which encodes values among $\{1, 2, 4, 6, 8, 12, 16, 32\}$ using 3 bits each. Any other value is considered non valid for the length of a block.

The second instantiation (referred as VSE-R in the experiments) uses similar encoders for $\mathcal{M}_1$ and $\mathcal{M}_2$ but performs a further, recursive, step. Firstly, from the original list L we produce a new list L' such that $L'[i] = \lfloor \log_2 L[i] \rfloor + 1$ (i.e., $L'[i]$ is equal to the number of bits needed to represent value $L[i]$). Then, we encode each value $L[i]$ by writing bin($L[i]$) without its most significant bit. Notice that if $L[i] = 1$, no bit is emitted. Finally, we apply a variant of VSE to encode the list L'[7]. Clearly, the value of $L[i]$ can be reconstructed once we know the value of $L'[i]$. VSE-R is designed to reduce the space wasted by encoding a sub-block of integers using a fixed amount of bits. In fact, of the kb bits used to encode k integers within b bits, a certain number of bits are left unused (in particular those wasted in encoding numbers smaller than 2^b).

For a running example consider again the list in the example above (i.e., $L = \langle 8, 1, 1, 8, 1, 1 \rangle$). The list L' is then $L' = \langle 4, 1, 1, 4, 1, 1 \rangle$. We use VSEncoding on L' using the same S as before obtaining the following[8]

1. $\gamma(b_1 + 1 = 3) = 101$, Unary$(k_1 = 2) = 10$, 11 00;
2. $\gamma(b_2 + 1 = 3) = 101$, Unary$(k_2 = 2) = 10$, 00 11;
3. $\gamma(b_3 + 1 = 1) = 1$, Unary$(k_3 = 2) = 10$;

[7]We change $\mathcal{M}_2$ so that it encodes using 3 bits only values among $\{1, 2, 4, 8, 12, 16, 32, 64\}$

[8]Here we use again γ and Unary

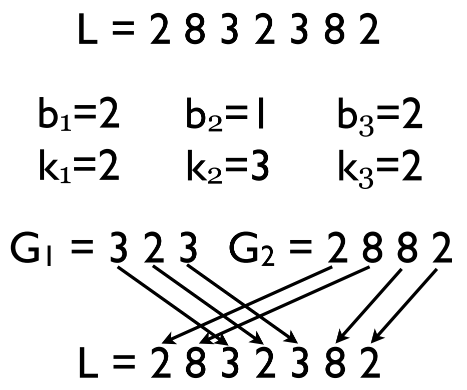


Figure 2: The Figure shows a running example of our layout. Assume that the list L has been partitioned in three blocks. First of all, we group integers of L into two groups G_1 and G_2 that contains, respectively, integers whose corresponding b are 1 and 2. Integers in the same group are written consecutively in the compress. The arrows show how to permute the integers in the groups to obtain back the original list L. Notice that this permutation can be easily derived since values of b and k are written in the correct order.

Finally, we emit bits corresponding to L's elements as 000 000, notice that 1s in L are not encoded at all in this final step. The final compress of the method on this list has size 26 bits.

As we shall see in Section 5, VSE is faster in decompression than VSE-R since it does not require the two steps of decoding while it is worse in compression. The better compression achieved by VSE-R is intuitively given by the fact that we encode a list of logarithmic values instead of plain values as in VSE. It is easy to show that in the case of highly skewed integer distributions, e.g. a power-law distributions with parameters $\alpha > 1$, with high probability the number of bits wasted by VSE-R is less than those wasted by VSE on the same vector of splits S. Roughly, it suffices to compute the number of bits wasted by the two methods with respect to the ideal case where each integer x requires $\lfloor \log_2(x) \rfloor + 1$ bits.

Finally, since we compute the *optimal* partitioning, we do not compare neither VSE nor VSE-R with any simpler *heuristics* for data splitting (e.g., taking simple fixed-length blocks of k elements at a time): either the simpler heuristics or the most sophisticated ones cannot outperform our optimal partitioning.

4.3 Compress Layout and Decompression Algorithm

In order to achieve a very fast decompression algorithm for VSE and VSE-R we have to carefully organize information on the compress file. A trivial layout in memory for VSE and VSE-R has been briefly described in Section 4.2: We encode each block separately by simply writing its values of b and k followed by the k integers encoded by using b bits each. In this way, the decompressor is very simple but, unfortunately, slower than the fastest known methods like Simple9, Simple16 and P4D. The reason is mainly given by the fact that block representations are not word aligned. This forces us to perform at least a conditional jump for every decompressed value[9]. It is well-known that conditional jumps are very expensive, and an efficient algorithm should avoid them as much as possible. For example, in Simple9 or Simple16 a single conditional jump followed by a call to an appropriate ad hoc function suffices to decode each encoded word. The parameter k in P4D is chosen so that the encoded representation of a block is word aligned. This implies that the k integers in a block can be decoded by resorting to very effective ad hoc functions that completely avoid conditional jumps. To be more precise, we have a function for each possible value of b that simply perform the correct operations required to decode k integers encoded with b bits each. For example, Figure 3 shows the function used in P4D to decode a block of $k = 32$ integers encoded by using $b = 8$ bits each. The decompression with these kind of functions is very fast. However, we recall that the decompression of blocks P4D has also to manage exceptions. This second step, in turn, significantly reduces its speed.

The layout we use for VSE and VSE-R is more involved with respect to the trivial one but allows a faster decompression algorithm. The idea is to organize the information so that the number of conditional jumps is considerably reduced. In the explanation we concentrate on VSE, since the layout for VSE-R is similar. Assume that the list we have to compress has been partitioned into l blocks by the partitioning step and that the obtained values of b and k are $b_1, b_2, \ldots, b_l$ and $k_1, k_2, \ldots, k_l$ respectively. Firstly, we group the integers of the list accordingly to the number of bits that we have to use to represent them. Then, we write separately the values in each group: first the values that have to be represented with 1 bit, then with 2 bits, and so on. If necessary, we pad the representation of each group so that it becomes word aligned. Finally, we write values of b and k in their order (i.e., $b_1k_1, b_2k_2, \ldots, b_lk_l$). Decompression is done in the following way. We decompress each group by resorting to the same fast functions of P4D (e.g., the one in Figure 3). This is possible since groups representations are word aligned. At this point we obtained groups of original integers that are out of order. In order to reconstruct the original list we appropriately permute these integers by exploiting the fact that values of b and k has been stored in the correct order. See Figure 2 for a simple example.

This algorithm, combined with the fact that we do not have to perform any conditional branch, allows for fast decompression speed as experiments in the next Section show.

5. EXPERIMENTS

In our experiments we use three collections to cover different possible sizes: gov2, wbr and wt10g. gov2 and wt10g are TREC test collections for use in the Terabyte Track. The former is a crawl of 25, 205, 170 .gov sites (as they were in early 2004) with documents truncated to 256 kb. wt10g is made up of 1, 692, 096 documents crawled in early 2000. wbr is made up of 5, 939, 061 web pages, representing a snapshot of the Brazilian web (domains .br) as spidered by the

[9]Notice that, in order to read values from a non-word aligned sequence of bits, we have to keep in memory a buffer of bits and check if it contains a sufficient number of bits before any read.

```
Decode8(decoded, encoded)
decoded[0] = *encoded >> 24 ;
decoded[1] = (*encoded >> 16) & 255;
decoded[2] = (*encoded >> 8) & 255;
decoded[3] = *encoded++ & 255;
decoded[4] = *encoded >> 24 ;
decoded[5] = (*encoded >> 16) & 255;
decoded[6] = (*encoded >> 8) & 255;
decoded[7] = *encoded++ & 255;
decoded[8] = *encoded >> 24 ;
decoded[9] = (*encoded >> 16) & 255;
decoded[10 = (*encoded >> 8) & 255;
decoded[11] = *encoded++ & 255;
decoded[12] = *encoded >> 24 ;
decoded[13] = (*encoded >> 16) & 255;
decoded[14] = (*encoded >> 8) & 255;
decoded[15] = *encoded++ & 255;
decoded[16] = *encoded >> 24 ;
decoded[17] = (*encoded >> 16) & 255;
decoded[18] = (*encoded >> 8) & 255;
decoded[19] = *encoded++ & 255;
decoded[20] = *encoded >> 24 ;
decoded[21] = (*encoded >> 16) & 255;
decoded[22] = (*encoded >> 8) & 255;
decoded[23] = *encoded++ & 255;
decoded[24] = *encoded >> 24 ;
decoded[25] = (*encoded >> 16) & 255;
decoded[26] = (*encoded >> 8) & 255;
decoded[27] = *encoded++ & 255;
decoded[28] = *encoded >> 24 ;
decoded[29] = (*encoded >> 16) & 255;
decoded[30] = (*encoded >> 8) & 255;
decoded[31] = *encoded++ & 255;
```

Figure 3: The ad-hoc C function used in P4D to decode $k = 32$ integers represented by using $b = 8$ bits each.

	gov2	wbr	wt10g
Size plain (Mbytes)	21,052.89	3,542.26	1,507.89
# Documents	25,205,170	5,939,061	1,692,096
# Terms	2,093,442	748,281	392,956
# Encoded Integers	5,413,133,900	915,962,369	371,589,409
Max list length	20,436,598	3,683,860	1,444,829
Avg list length	2,585.76	1,224.09	945.62

Table 1: The table reports some basic statistics on the collections we use in our experiments.

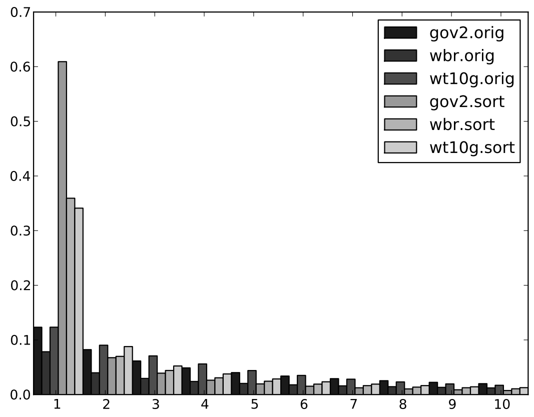


Figure 4: Distribution of the first 10 d-gaps for our collections before (.orig) and after (.sort) the reassigment of document Ids.

crawler of the TodoBR search engine in 1999. More information about these three collections are shown in Table 1 which reports basic statistics such as the size of plain collection in Mbytes, the number of documents, the number of terms (i.e., the number of lists), the number of encoded integers, the length of the longest list, and the average lists length.

We tested the different methods on a PC with an Intel Xeon Quad-Core Processor equipped with 8GBytes RAM and SATA hard disks. The operating system is a 64-bit version of Linux 2.6.31-20. All our code is written in C and is available at `http://hpc.isti.cnr.it/~integerencoding`.

In the experiments we restricted our attention on compressing lists larger than 16 elements. The reason of this choice is given by the fact that we want to limit the overhead of function calls when we measure the decompression speed of the different methods. We experimentally observed that this choice does not affect the comparison among the different methods with respect to achieved compression.

We also restrict our attention on lists in which document Ids are assigned by sorting the corresponding URI lexicographically. In this way we obtain lists that are much more compressible as well documented in many preceding works (see for example [17, 15, 20] and references therein). This phenomenon finds its explanation in the fact that documents in the same domain are likely to be similar (i.e., they contain almost the same set of terms). Thus, the reassignment above assigns close Ids to documents that belong to the same domain, so that it is likely to obtain very small d-gaps in the lists. By this reason, the resulting collections are much more compressible. Figure 4 shows the distribution of the first 10 smallest d-gaps in our collections after and before the above reassignment.

Table 2 shows the gain in compression achievable with Interpolative and δ on our datasets. The gain is impressive: the compress is from 21 to 110 more compact than the best performing method, i.e. P4D . Notice that the gain of the reordering largely compensates the negligible cost (few Mbs) of storing in an array the inverse assignments which may be necessary for some reason. Thus, the reassignment is a very profitable choice even when a different document Ids assignment is necessary. In the following, we restrict our attention to compress our datasets in which document Ids are sorted in this way.

Compression performance. In our experiment we tried different compressors as reported in Table 3. In particular, OPT-P4D refers to the OPT-PforDelta described in [20] with blocks of size 128 values. We choose this parameter after experimental evaluations. The smaller the block length, the better the achieved compression, but slower is the decompression. With blocks larger than 128 we obtain compression performance which are significantly worse while the decom-

Method	Interpolative	δ
gov2.orig	6.689	8.234
gov2.sorted	3.229	3.930
Gain	2.07	2.10
wbr.orig	8.761	10.948
wbr.sorted	6.342	6.973
Gain factor	1.38	1.570
wt10G.orig	6.798	8.115
wt10G.sorted	6.636	6.389
Gain factor	1.21	1.27

Table 2: The table compares the compression achieved on original and sorted version of our datasets with Interpolative and δ. The compression is expressed in bits per integer. The Gain factor tells the improvement in compression obtainable by reassigning document Ids.

pression is just slightly faster. With smaller blocks, i.e., 32 or 64, the decompression speed is up to four times slower. We remark that we tested our implementation of OPT-P4D whose performance has been validated against the original implementation kindly provided by the authors of [20].

We point out that only our methods, together with Interpolative, are able to beat the entropy of the lists on the datasets. This quasi-paradoxical effect is, indeed, present because entropy does not consider *context* information. Entropy, or to use a notation commonly used in text compression, zeroth-order entropy, does not take into account patterns (i.e. the context) that can be present in lists of blocks of integers. By grouping together blocks of integers, in fact, we are able to assign codewords to more than a single value at a time. Therefore, it appears obvious that we can beat the entropy in the case of VSE, VSE-R and Interpolative. Essentially, this is possible since we exploit regularities on the lists on these very skewed *d*-gaps lists (e.g., small values close to each other or quite long runs of 1s). We remark that beat the entropy is not possible with any prefix code (e.g., statistical compressors like Arithmetic and Huffman or integer encoders like γ, δ, ζ's, Golomb, and so on). Therefore, our methods is certainly better in compression than any of these kind of methods without the need of any comparison. Anyway, for the sake of completeness, we report those results as well in Table 3.

To resume, our experiments show incontrovertibly that our methods achieve compression performance comparable (and in the case of wbr better) to those achieved by the state-of-the-art (in terms of space) compression method, i.e. Interpolative. As we are going to show, decoding speed is an issue in the case of Interpolative while our methods are instead faster than the state-of-the-art P4D .

Decompression speed. Table 4 reports results on the decoding speed, in terms of millions of integers per second, of the different methods we tested. We report the performance computed over different postings lists and we indicate the average decoding speed along with its standard deviation. All the values have been rounded to the nearest ten.

As expected, Interpolative is the slowest as opposed to VSE which tops others with more than 800 millions of integers per second. Our methods, VSE and VSE-R, are among the fastest in decoding with a number of mis (millions of integers

Compression	gov2		wbr		wt10g	
Method	bpi	loss %	bpi	loss %	bpi	loss %
Interpolative	**3.227**	0.000	6.301	0.196	**5.630**	0.000
VSE-R	3.321	2.912	**6.289**	0.000	5.738	1.922
VSE	3.626	12.360	6.758	7.740	6.007	6.696
Entropy	3.768	16.764	6.578	4.593	6.048	7.418
OPT-P4D	4.232	31.143	7.373	17.220	6.314	12.149
δ	3.929	21.751	6.928	10.161	6.382	13.358
ζ_3	4.117	27.564	7.648	21.608	6.814	21.029
γ	4.820	49.360	7.013	11.517	6.449	14.550
Simple9	4.561	41.338	8.267	31.451	7.181	21.549
Simple16	4.441	37.620	7.923	25.981	6.839	21.474
VBytes	8.665	168.473	9.817	56.106	9.331	65.722

Table 3: Compression achieved by the various encoders on our datasets expressed in bits per integers (bpi). In bold we report the best compressor. For each compressor we also report its increase (in percentage) with respect to the best compressor.

Method	mis
Interpolative	75 ± 5
VSE-R	450 ± 20
VSE	835 ± 35
OPT-P4D	460 ± 20
δ	130 ± 10
ζ_3	140 ± 10
γ	120 ± 10
Simple9	630 ± 30
Simple16	630 ± 30
VBytes	260 ± 10

Table 4: Average decompression speed on the various compression methods on our datasets expressed in millions of integers per second (mis). The value after $\pm$ indicates how much the speed of various executions are different from the reported value.

per second) decoded ranging from 450 of VSE-R to 835 of VSE both of them measured using the gov2 collection. It is interesting to observe the better performance in terms of decoding speed of VSE with respect to others, and in particular with respect to OPT-P4D, Simple9 and Simple16 which are considered state-of-the-art as far as decompression speed is concerned.

We would like to point the attention on the quite good decompression performance of γ, δ and ζ_3. In our implementations their decoders have been particularly optimized for decoding speed using table lookups to quickly decode sequences of bits. We have measured the effect of such a table and we observe that, by only using $2^{16} = 65,536$ entries, a single table lookup suffices to decode the codeword for about 90% of the integers, so that only remaining integers are decoded with the classic and slow algorithm.

From the experiments, Interpolative, VSE, and VSE-R, as Figure 5 shows, dominate all the others we tested. In particular, what can be highlighted from the plot in Figure 5 is that our two methods optimize both decoding speed and compression space at the same time. Obviously, in environments like those typical of web search engines, where one should aim at being both fast and space efficient, our methods VSE and VSE-R result to be those of choice with a preference for VSE if one care more about speed than space.

Encoders statistics. We report in this paragraph some statistics on our encoders, VSE and VSE-R, that help in understanding the correlation between the skewness of a

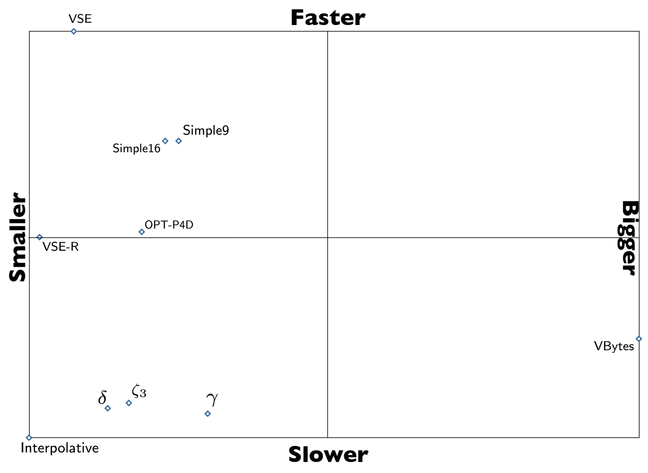


Figure 5: A graphical comparison of the different encoders showing the trade-offs in time and space. On the x-axis is represented the compress size (normalized between -1 and 1), on the y-axis is represented the decoding speed (normalized between -1 and 1, as well.)

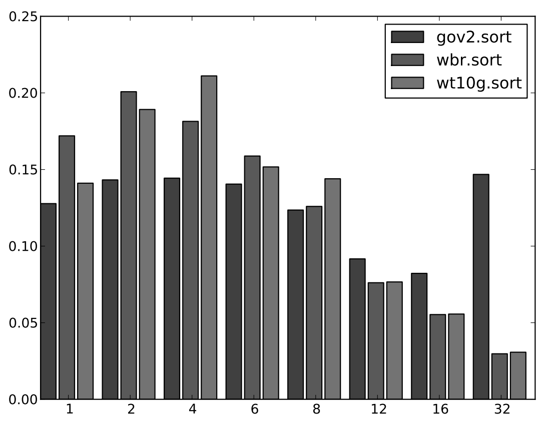
dataset and, the number of bits and length of blocks produced by the two methods.

As it can be observed in Figure 6 (above), using VSE we do not have a large variation in terms of block lengths. This means that VSE is able to adapt, correctly, to the underlying distribution of integers. In addition, another important aspect to point out is the large fraction of blocks encoding their members using 0 bits. This can be seen in Figure 6 (below), where it is shown the distribution of number of bits used to encode elements in each block using VSE. Interesting to notice that still a large fraction of elements needs more than 8 bits to be encoded, this is due, again, to the high skewness of our datasets characterized by long runs of 1s.

The two bar charts in Figure 7, instead show the empirical explanation for the reason why VSE-R appears to perform better, in practice, than VSE for skewed datasets. First of all, as in the previous case runs of '1's are frequent and from this we have a large fraction of blocks encoded using 0 bits. The main difference, though, is observed in the case of the number of blocks having a relatively large size. Blocks of length 8 and 16 are the most frequent (with a total frequency that is around the 40%). It is quite likely, then, that a large fraction of long blocks can be encoded using 0 bits. This is, again empirically, confirmed by the experiments shown above.

6. CONCLUSION AND FUTURE WORK

We have described VSEncoding, a class of encoders that through a dynamic programming algorithm are able to encode lists of integers beating the entropy of the gaps distribution. The assignment of codewords is done with the goal of optimizing both the space taken by the codewords themselves and the time needed to decode. We have shown, through extensive experiments, that our methods constantly outperform the others in terms of both space and time and, in our opinion, should be the methods of choice for data



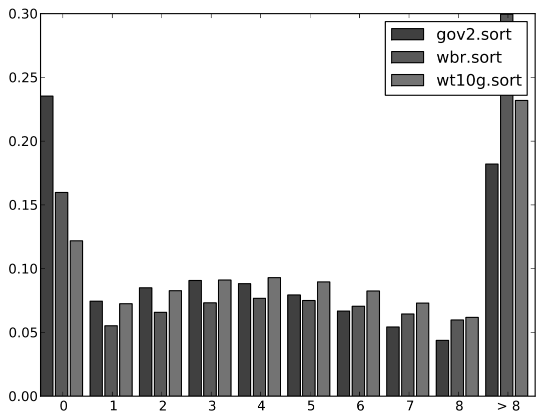


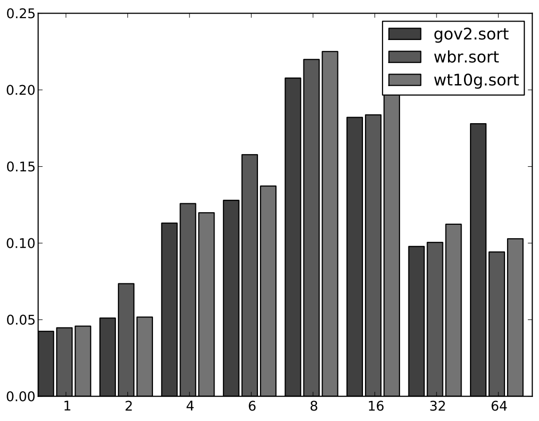
Figure 6: Distribution of lengths of the blocks (above) and the number of bits used to encode elements in the blocks (below) in VSE over our datasets.

management systems (e.g. web search engines) aiming at very high performance and low space consumption.

Even if our methods are already among the fastest state-of-the-art fast-encoders (e.g. those of the PForDelta family or Simple9 like), we would like to more extensively experiment other variations of our methods that could be obtained by varying encoders $\mathcal{M}_1$ and $\mathcal{M}_2$ in order to further improve either compression rate or decompression speed. Ideally, one would like to have a scheme that has decompression speed of VSE achieving compression rate of VSE-R.

We defer to a future work the study of the impact of list skipping [6] on the effectiveness of our method. Apart from the straightforward approach consisting in partitioning each list according to the strategy by Chierichetti *et al.* [6]. The challenge, anyway, is to find an optimal way of partitioning the lists of integers also in light of how skips are placed.

As it has been shown in the discussion of the data layout, the impact of the architecture is of fundamental importance to the efficiency of the decoding method. We are currently developing a very fast, and ad-hoc, VSE encoding like method for GPUs [8]. Preliminary experiments are



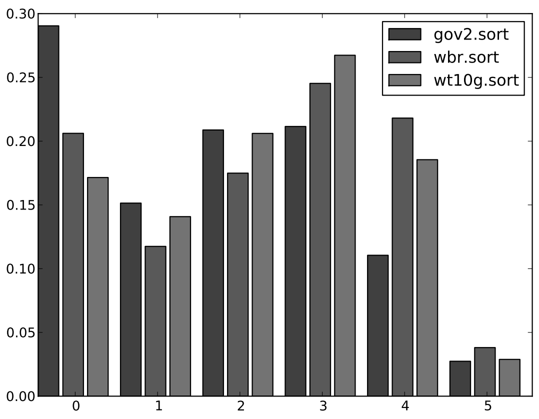


Figure 7: Distribution of lengths of the blocks (above) and the number of bits used to encode elements in the blocks (below) in VSE-R over our datasets.

very encouraging showing a sharp improvement in decoding speed.

Finally, we are aware that in Web Search Engines not all the lists are accessed with the same frequency. We are currently studying strategies for the optimal encoding of posting lists also considering access patterns. We are using information available from query logs [16] to extract lists access patterns.

7. REFERENCES

[1] B. Abali, H. Franke, X. Shen, D. E. Poff, and T. B. Smith. Performance of hardware compressed main memory. In *Procs. of HPCA*, page 73, Washington, DC, USA, 2001. IEEE Computer Society.

[2] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.

[3] V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Trans. on Knowl. and Data Eng.*, 18(6):857–861, 2006.

[4] P. Boldi and Sebastiano V. Codes for the world wide web. *Internet Mathematics*, 2(4), 2005.

[5] A. Bookstein, S. T. Klein, and T. Raita. Modeling word occurrences for the compression of concordances. *ACM Trans. Inf. Syst.*, 15(3):254–290, 1997.

[6] F. Chierichetti, S. Lattanzi, F. Mari, and A. Panconesi. On placing skips optimally in expectation. In *WSDM '08: Proceedings of the international conference on Web search and web data mining*, pages 15–24, 2008.

[7] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill Science/Eng/Math, 2006.

[8] F. Dehne and K. Yogaratnam. Exploring the limits of gpus with parallel graph algorithms. *CoRR*, abs/1002.4482, 2010.

[9] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, Mar 1975.

[10] P. Ferragina, I. Nitto, and R. Venturini. On optimally partitioning a text to improve its compression. In *Procs. of European Symposium on Algorithms (ESA)*, pages 420–431, 2009.

[11] S. Golomb. Run-length encodings. *Information Theory, IEEE Transactions on*, 12(3):399–401, 1966.

[12] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.

[13] D. Salomon. *Variable-length Codes for Data Compression*. Springer, 2007.

[14] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Commun. ACM*, 7(3):166–169, 1964.

[15] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. of ECIR*, pages 101–112, 2007.

[16] F. Silvestri. Mining query logs: Turning search usage data into knowledge. *Found. Trends Inf. Retr.*, 4(1—2):1–174, 2010.

[17] F. Silvestri, Salvatore Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Procs. of ACM SIGIR*, pages 305–312, 2004.

[18] A. Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, 2003.

[19] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[20] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Procs. of WWW*, pages 401–410, New York, NY, USA, 2009. ACM.

[21] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE '06: Procs. of ICDE*, page 59, 2006.