Variable-Byte Encoding Is Now Space-Efficient Too

Giulio Ermanno Pibiri
University of Pisa and ISTI-CNR
Pisa, Italy
giulio.pibiri@di.unipi.it

Rossano Venturini
University of Pisa and ISTI-CNR
Pisa, Italy
rossano.venturini@unipi.it

ABSTRACT

The ubiquitous *Variable-Byte* encoding is considered one of the fastest compressed representation for integer sequences. However, its compression ratio is usually not competitive with other more sophisticated encoders, especially when the integers to be compressed are small that is the typical case for inverted indexes. This paper shows that the compression ratio of Variable-Byte can be improved by 2× by adopting a partitioned representation of the inverted lists. This makes Variable-Byte surprisingly competitive in space with the best bit-aligned encoders, *hence* disproving the folklore belief that Variable-Byte is space-inefficient for inverted index compression.

Despite the significant space savings, we show that our optimization almost comes for free, given that: we introduce an optimal partitioning algorithm that, by running in linear time and with low constant factors, does not affect indexing time; we show that the query processing speed of Variable-Byte is preserved, with an extensive experimental analysis and comparison with several other state-of-the-art encoders.

1. INTRODUCTION

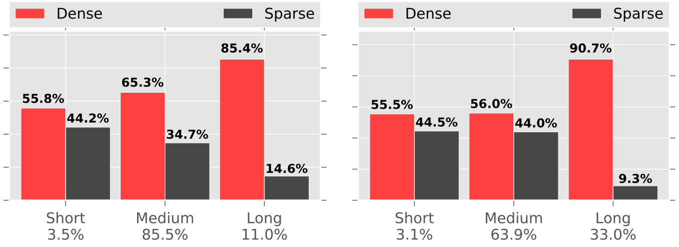
The *inverted index* is the core data structure at the basis of search engines, massive database architectures and social networks [47, 27, 12, 14, 11]. In its simplicity, the inverted index can be regarded as being a collection of sorted integer sequences, called inverted or posting lists.

When the index is used to support full-text search in databases, each list is associated to a vocabulary term and stores the sequence of integer identifiers of the documents that contain such term [27]. Then, identifying a set of documents containing all the terms in a user query reduces to the problem of intersecting the inverted lists associated to the terms in the query. Likewise, an inverted list can be associated to a user in a social network (e.g., Facebook) and stores the sequence of all the friend identifiers of the user [14]. Moreover, database systems based on SQL often precompute the list of row identifiers matching a specific frequent predicate over a huge table, in order to speed up the execution of a query involving the conjunction of, possibly, many predicates [24, 36]. Also, finding all occurrences of twig patterns in XML databases can be done efficiently by resorting on an inverted index [9]. In recent years, a vast number of key-value stores has emerged, e.g., Apache Ignite, Redis, InfinityDB, BerkeleyDB and many others. Common to all such architectures is the organization of data elements falling into the same bucket due to an hash collision: the list of all such elements is recorded, which is nothing but an inverted list [16].

Because of the huge quantity of data available and processed on a daily basis by the mentioned systems, *compressing* the inverted index is indispensable since it can introduce a two-fold advantage over a non-compressed representation: feed faster memory levels with more data and, *hence*, speed up the query processing algorithms. As a result, the design of algorithms that compress the index effectively while maintaining a noticeable decoding speed is an old problem in computer science, that dates back to more than 50 years ago, and still a very active field of research. Many representation for inverted lists are known, each exposing a different compression ratio vs. query processing speed trade-off. We point the reader to [34] and Section 2 of this paper for a concise overview of the different encoders that have been proposed through the years.

Among these, *Variable-Byte* [40, 44] (henceforth, VByte) is the most popular and used byte-aligned code. In particular, VByte owes its popularity to its sequential decoding speed and, indeed, it is the fastest representation up to date for integer sequences. For this reason, it is widely adopted by well-known companies as a key database design technology to enable fast search of records. We mention some noticeable examples. Google uses VByte extensively: for compressing the posting lists of inverted indexes [15] and as a binary wire format for its protocol buffers [2]. IBM DB2 employs VByte to store the differences between successive record identifiers [8]. Amazon patented an encoding scheme, based on VByte and called Varint-G8IU, which uses SIMD (Single Instruction Multiple Data) instructions to perform decoding faster [39]. Many other storage architectures rely on VByte to support fast full-text search, like Redis [3], UpscaleDB [4] and Dropbox [1].

We now quickly review how the VByte encoding works. It was first described by Thiel and Heaps [40]. The binary representation of a non-negative integer is divided into groups of 7 bits which are represented as a sequence of bytes. In



(a) Gov2 (b) ClueWeb09

Figure 1: Percentage of postings belonging to Dense and Sparse regions of the posting lists for the Gov2 and ClueWeb09 datasets. The posting lists have been clustered by size into three categories: Short (size $<$ 10K); Medium (10K $\leq$ size $<$ 7M); Long (size $\geq$ 7M). Below each category we also indicate the percentage of postings belonging to its posting lists.

particular, the 7 least significant bits of each byte are reserved for the data whereas the most significant (the 8-th), called the *continuation bit*, is equal to 1 to signal continuation of the byte sequence. The last byte of the sequence has its 8-th bit set to 0 to signal, instead, the termination of the byte sequence. As an example, 65 790 is represented as VByte(65 790) $=$ $\underline{1}0000100\underline{1}0000001\underline{0}1111110$, where we have underlined the control bits. Also notice the padding bits in the first byte starting from the left, inserted to align the binary representation of the number to a multiple of 8 bits. In particular, VByte uses $\lceil \frac{\lceil \log(x+1) \rceil}{7} \rceil \times 8$ bits to represent an integer x. Decoding is simple: we just need to read one byte at a time until we find a value smaller than 2^7. The format is also suitable for SIMD for speeding up sequential decoding, as we will review in Section 2.

The main drawback of VByte lies in its byte-aligned nature, which means that the number of bits needed to encode an integer cannot be less than 8. For this reason, VByte is only suitable for large numbers. However, the inverted lists are notably known to exhibit a *clustering effect*, i.e., these present regions of close identifiers that are far more compressible than highly scattered regions [30, 32, 34]. Such natural clusters are present because the indexed data itself tend to be very similar. As a simple example, consider all the Web pages belonging to the same site: these are likely to share a lot of terms. Also, the values stored in the columns of databases typically exhibit high locality: that is why column-oriented databases can achieve very good compression and high query throughput [5].

The key point is that efficient inverted index compression should exploit as much as possible the clustering effect of the inverted lists. VByte currently fails to do so and, as a consequence, it is believed to be space-inefficient for inverted indexes.

The motivating experiment. As an illustrative example, consider the following two sequences: $\langle 1, 2, 3, 4, 5 \rangle$ and $\langle 127, 254, 318, 408, 533 \rangle$. To reduce the values of the integers, VByte compresses the differences between successive values, known as *delta*-gaps or *d*-gaps, i.e., the sequences $\langle 1, 1, 1, 1, 1 \rangle$ and $\langle 127, 127, 64, 90, 125 \rangle$ respectively (the first integer is left as it is). Now, it is easy to see that VByte will use 5 bytes to encode *both* sequences, but the first one can be compressed much better, with just $\approx \log 5$ bits. To better highlight how this behavior can deeply affect compression effectiveness, we consider the statistic shown in Figure 1. This statistic reports the percentage of postings belonging to *dense* and *sparse* regions of the lists for the, widely used, two datasets Gov2 and ClueWeb09. More precisely, the plot originated from the following experiment: we divided each inverted list into chunks of 128 integers and we considered as *sparse* a chunk where VByte yielded the best space usage with respect to the *characteristic bit-vector* representation of the chunk (if u is the last element in the chunk, we have the i-th bit set in a bitmap of size u for all integers i belonging to the chunk), regarded to as the *dense* case. We also clustered the inverted lists by their sizes, in order to show where dense and sparse regions are most likely to be present.

The experiment clearly shows that *we have a majority of dense regions*, thus explaining why in this case VByte is not competitive with respect to bit-aligned encoders and, thus, motivating the need for introducing a better encoding strategy that adapts to such distribution without compromising the query processing speed of VByte. We can also conclude that such optimization is likely to pay off because the majority of integers, i.e., 85% for Gov2 and 64% for ClueWeb09, concentrate in the lists of medium size (thanks to the Zipfian distribution of words in text), where indeed *more than half* of them belong to dense chunks.

Our contributions. We list here our main contributions.

1. We disprove the folklore belief that VByte is too large to be considered space-efficient for compressing inverted indexes, by exhibiting an improved compression ratio of 2× on the standard datasets Gov2 and ClueWeb09.

   The result is achieved by partitioning the inverted lists into blocks and representing each block with the most suitable encoder, chosen among VByte and the characteristic bit-vector representation. Partitioning the lists has the potential of adapting to the distribution of the integers in the lists, such as the ones shown in Figure 1, by adopting VByte for the sparse regions where larger *d*-gaps are likely to be present.

2. Since we cannot expect the dense regions of the lists be always aligned with uniform boundaries, we consider the optimization problem of minimizing the space of representation of an inverted list of size n by representing it with variable-length partitions. To solve the problem efficiently, we introduce an algorithm that finds the *optimal* partitioning in $\Theta(n)$ time and $O(1)$ space.

   We remark that the state-of-the-art dynamic programming algorithm in [32] can be used as well to find an $(1+\epsilon)$-optimal solution in $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$ time and $O(n)$ space for any $\epsilon \in (0, 1)$, but it is noticeably slower than our approach and approximated, rather than exact.

   We also remark that, although we conducted the experiments using VByte, our optimal algorithm can be applied to *any point-wise encoder*, that is whenever the chosen encoder needs a number of bits to represent an integer that solely depends on the value of the integer and not on the universe and size of the chunk to which it belongs to.

3. We conduct an extensive experimental analysis to demonstrate the effectiveness of our approach on standard large datasets, such as Gov2 and ClueWeb09. More precisely, when compared to the un-partitioned VByte indexes, the optimally-partitioned counterparts are: (1) significantly smaller, by 2× on average; (2) marginally slower at computing boolean conjunctions and perform sequential decoding, by only 5%; (3) even faster to build on large datasets thanks to the introduced fast partitioning algorithm and improved compression ratio.

   We compare the performance of partitioned VByte indexes against several state-of-the-art encoders, such as: partitioned Elias-Fano (PEF) [32], Binary Interpolative coding (BIC) [30], the optimized PForDelta (OptPFD) [45], a recent proposal based on Asymmetric Numeral Systems (ANS) [29] and QMX [41]. The partitioned VByte representation reduces the gap between the space of VByte and the one of the best bit-aligned compressors, such as, for example, PEF and BIC, by passing from an average original gap of 138% to only 11% with respect to PEF; from 174% to only 22% with respect to BIC. Moreover, it also offers the fastest query processing speed in all cases, except against the QMX mechanism which is slightly faster, by $7 \div 14\%$, but up to 30% larger.

2. RELATED WORK

2.1 Compressors for inverted lists

In this subsection we overview the most important compressors devised for efficient inverted list representation. Additionally to the ones we review in the following, we remark that well-known compressors like Elias' γ and δ [19] and Golomb [23] are known to obtain inferior compression ratios for inverted index storage with respect to the state-of-the-art, thus we do not consider them. We point the reader to [34] for a recent and concise survey for a description of such mechanisms.

Other recent techniques, instead, consider the compression of the index as a whole, by trying to represent many inverted lists together, thus reducing the implicit redundancy present in the lists [33, 13, 46]. These results are orthogonal to the work we present in this paper.

Block-based. Blocks of contiguous integers can be encoded separately, to improve both compression ratio and retrieval efficiency. This line of work finds its origin in the so-called *frame-of-reference* (For) [22]. A simple example of this approach, called *binary packing*, encodes blocks of fixed length, e.g., 128 integers [7, 25]. To reduce the value of the integers, we can subtract from integer the previous one (the first integer is left as it is), making each block be formed by integers greater than zero known as *delta*-gaps (or just d-gaps). Scanning a block will need to re-compute the original integers by computing the prefix sums. In order to avoid the prefix sums, we can just encode the difference between the integers and the first element of the block (base+offset encoding) [32, 43]. Using more than one compressors to represent the blocks, rather than only one, can also introduce significant improvements in query time within the same space constraints [31].

Other binary packing strategies are Simple-9 [6], Simple-16 [7] and QMX [41], that combine relatively good compression ratio and high decompression speed. The key idea is to try to pack as many integers as possible in a memory register (32, 64 or 128 bits). Along with the data bits, a *selector* is used to indicate how many integers have been packed together in a single unit. In the QMX mechanism the selectors are run-length encoded.

PForDelta. The biggest limitation of block-based strategies is that these are inefficient whenever a block contains at least one large element, because this causes the compressor to use a number of bits per element proportional to the one needed to represent that large value. To overcome this limitation, PForDelta was proposed [48]. The main idea is to choose a proper value k for the universe of representation of the block, such that a large fraction, e.g., 90%, of its integers fall in the range $[b, b+2^k-1]$ and, thus, can be written with k bits each. This strategy is called *patching.* All integers that do not fit in k bits, are treated as *exceptions* and encoded separately using another compressor.

The optimized variant of the encoding [45], which selects for each block the values of b and k that minimize its space occupancy, has been demonstrated to be more space-efficient and only slightly slower than the original PForDelta [45, 25].

Elias-Fano. This strategy directly encodes a monotone integer sequence without a first delta encoding step. It was independently proposed by Elias [18] and Fano [20], hence its name. Given a sequence of size n and universe u, its Elias-Fano representation takes at most $n\lceil\log\frac{u}{n}\rceil + 2n$ bits, which can be shown to be less that half a bit away from optimality [18]. The encoding has been recently applied to the representation of inverted indexes [42] and social networks [14], thanks to its excellent space efficiency and powerful search capabilities, namely random access in $O(1)$ and successor queries in $O(1 + \log\frac{u}{n})$ time. The latter operation which, given an integer x of a sequence S returns the smallest integer $y \in S$ such that $y \geq x$, is the fundamental one when resolving boolean conjunctions over inverted lists (see [42, 32, 33] for details). As standard, we refer to this primitive with the name NextGEQ (Next Greater than or EQual to) in the whole paper.

The partitioned variant of Elias-Fano (PEF) [32], splits a sequence into variable-sized partitions and represents each partition with Elias-Fano. The partitioned representation sensibly improves the compression ratio of Elias-Fano by preserving its query processing speed. In particular, it currently embodies the best trade-off between index space and query processing speed.

Binary Interpolative Coding. Binary Interpolative Coding (BIC) [30] is another approach that, like Elias-Fano, directly compresses a monotonically increasing integer sequence. In short, BIC is a recursive algorithm that first encodes the middle element of the current range and then applies this encoding step to both halves. At each step of recursion, the algorithm knows the reduced ranges that will be used to write the middle elements in fewer bits during the next recursive calls.

Many papers in the literature experimentally proved that BIC is one of the most space-efficient method for storing highly clustered sequences, though among the slowest at performing decoding [45, 38, 32, 33].

Asymmetric Numeral Systems. Asymmetric Numeral Systems (ANS) is a family of *entropy* coders, originally developed by Jarek Duda [17]. ANS combines the excellent compression ratio of Arithmetic coding with a decompression speed comparable with the one of Huffman. It is now widely used in commercial applications, like Facebook ZSTD, Apple LZFSE and in the Linux kernel.

The basic idea of ANS is to represent a sequence of symbols with a natural number x. If each symbol s belongs to the binary alphabet $\Sigma = \{0, 1\}$, then appending s to the end of the binary string representing x will generate a new integer $x' = 2x + s$. This coding is optimal whenever $\mathbb{P}(0) = \mathbb{P}(1) = 1/2$. ANS generalizes this concept by adapting it to a general distribution of symbols $\{p_s\}_{s \in \Sigma}$. In particular, appending a symbol s to x increases the information content from $\log x$ bits to $\log x - \log p_s = \log(x/p_s)$ bits, thus the new generated natural number will be $x' \approx x/p_s$.

Recently, ANS-based techniques for inverted index compression have been proposed [28, 29]. Such investigation has lead to a higher compression ratio than the one of BIC and with faster query processing speed.

The Variable-Byte family. Various encoding formats for VByte have been proposed in the literature in order to improve its sequential decoding speed. By assuming that the largest represented integer fits into 4 bytes, two bits are sufficient to describe the proper number of bytes needed to represent an integer. In this way, groups of four integers require one control byte that has to be read once as a header information. This optimization was introduced in Google's Varint-GB [15] and reduces the probability of a branch misprediction which, in turn, leads to higher instruction throughput. Working with byte-aligned codes also opens the possibility of exploiting the parallelism of SIMD (Single Instruction Multiple Data) instructions of modern processors to further enhance the decoding speed. This is the line of research taken by the recent proposals that we overview below.

Varint-G8IU [39] uses a similar idea to the one of Varint-GB but it fixes the number of compressed bytes rather than the number of integers: one control byte is used to describe a variable number of integers in a data segment of exactly 8 bytes, therefore each group can contain between two and eight compressed integers.

Masked-VByte [35] directly works on the original VByte format. The decoder first gathers the most significant bits of consecutive bytes using a dedicated SIMD instruction. Then, using previously-built look-up tables and a shuffle instruction, the data bytes are permuted to obtain the original integers.

Stream-VByte [26], instead, separates the encoding of the control bytes from the data bytes, by writing them into separate streams. This organization permits to decode multiple control bytes simultaneously and, therefore, reduce branch mispredictions that can stop the CPU pipeline execution when decoding the data stream.

2.2 Partitioning algorithms

The simplest partitioning strategy is to fix the length b of every partition, e.g., $b = 128$ integers, and split the list into $\lceil n/b \rceil$ blocks, where n is the size of the list (the last partition could be potentially smaller than b integers). We call this partitioning strategy, *uniform*. The advantage of this representation is simplicity, since no expensive calculation is needed prior to encoding. However, we cannot expect this strategy to yield the most compact indexes because the highly clustered regions of inverted lists could be likely broken by such fix-sized partitions.

This is the main motivation for introducing optimization algorithms that try to find the best partitioning of the list, thus minimizing its space of representation. Previous work used dynamic programming extensively [10, 38, 21, 32]. More precisely, Silvestri and Venturini [38] obtained a $O(n \times h)$ construction time, where n is the length of the inverted list and h its longest partition. Ferragina *et al.* [21] improve the result in [10] by computing a partitioning whose cost is guaranteed to be at most $(1+\epsilon)$ times away from the optimal one, for any $\epsilon \in (0, 1)$, in $O(n \log_{1+\epsilon} n)$ time. Their approach can be applied to any encoder E whose cost in bits can be computed (or, at least, estimated) in constant-time, for any portion of the input.

Finally, Ottaviano and Venturini [32] resort to similar ideas to the ones presented in [21] to obtain a running time of $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$, and yet, preserving the same approximation guarantees. Note that the complexity is $\Theta(n)$ as soon as ϵ is constant. Since their dynamic programming technique is the state-of-the-art and our algorithm aims at improving such result, we will review the algorithm in Section 3.

3. OPTIMAL PARTITIONING IN LINEAR TIME

In this section we study the problem of partitioning a monotone integer sequence S of size n to improve its compression, by adopting a 2-level representation. This data structure stores S as a sequence of partitions $L_2[S_1, \ldots, S_k]$ that are concatenated in the second level L_2. The first level L_1 stores, instead, a fix amount of bits, say F, for each partition S_i, needed to describe its size n_i and largest element u_i. Clearly, F can be safely upper bounded by $O(\log u)$ bits. This representation has several important advantages over a shallow representation:

1. it permits to choose the most suitable encoder for each partition, given its size and upper bound, hence improving the overall index space;

2. each partition S_i can be represented in a smaller universe, i.e., $u_i - u_{i-1} - 1$, by subtracting to all its elements the base value $u_{i-1} + 1$, thus contributing to further reduction in space;

3. it allows a faster access to the individual elements of S, since we can first locate the partition to which an element belongs to and, then, conclude the search in that partition only.

Now, the natural problem is *how* to choose the lengths and encoders for each partition in order to minimize the space of S. As already noted, the problem is not trivial since we cannot expect dense regions of the lists being always aligned with fix-sized partitions. While a dynamic programming recurrence offers an optimal solution to this problem in $\Theta(n^2)$ time and $O(n)$ space by trivially considering the cost of all possible splittings, this approach is clearly unfeasible already for modest sizes of the input. Therefore, we need smarter methods such as the ones we describe in the following.

3.1 Dynamic programming: slow and approximated

The core idea of this approach is to not consider *all* possible splittings, but only the ones whose cost is able of amortizing the fix cost F [32]. More precisely, it finds a solution whose encoding cost is at most $(1+\epsilon)$ away from the optimal one in $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$ time and $O(n)$ space, for any $\epsilon \in (0,1)$. Note that the time complexity is linear as soon as ϵ is constant. We now quickly describe the technique and highlight its main drawback.

The problem of determining the partitioning of minimum cost can be modeled as the problem of determining the path of minimum cost (shortest) in a complete, weighted and directed acyclic graph (DAG) $\mathcal{G}$. This DAG has n vertices, one for each position of S, and it is complete, i.e., it has $\Theta(n^2)$ edges where the cost $C(i,j)$ of edge (i,j) represents the number of bits needed to represent $S[i,j]$. Since the DAG is complete, a simple shortest path algorithm will not suffice to compute an optimal solution efficiently. Thus, we proceed by *sparsification* of $\mathcal{G}$, as follows. We first consider a new DAG $\mathcal{G}_\epsilon$, which is obtained from $\mathcal{G}$ and has the following properties: (1) the number of edges is $O(n \log_{1+\epsilon} \frac{U}{F})$ for any given $\epsilon \in (0,1)$; (2) its shortest path distance is at most $(1+\epsilon)$ times the one of the original DAG $\mathcal{G}$, where U represents the encoding cost of S when no partitioning is performed. It can be proven that the shortest path algorithm on $\mathcal{G}_\epsilon$ finds a solution which is at most $(1+\epsilon)$ times larger than an optimal one, in time $O(n \log_{1+\epsilon} \frac{U}{F})$, because $\mathcal{G}_\epsilon$ has $O(n \log_{1+\epsilon} \frac{U}{F})$ edges [21]. To further reduce the complexity by preserving the same approximation guarantees, we define two approximation parameters: $\epsilon_1 \in [0,1)$ and $\epsilon_2 \in [0,1)$. We first retain from $\mathcal{G}$ all the edges whose cost is no more than $L = \frac{F}{\epsilon_1}$, then we apply the pruning strategy described above with ϵ_2 as approximation parameter. The obtained graph has now $O(n \log_{1+\epsilon_2} \frac{L}{F}) = O(n \log_{1+\epsilon_2} \frac{1}{\epsilon_1})$ edges, which is $\Theta(n)$ as soon as ϵ_1 and ϵ_2 are constant. Again, it can be proven that the shortest path distance is no more than $(1+\epsilon_1)(1+\epsilon_2) \leq (1+\epsilon)$ times the one in $\mathcal{G}$ by setting $\epsilon_1 = \epsilon_2 = \frac{\epsilon}{3}$ [32].

Despite the *theoretical* linear-time complexity for a constant ϵ, the main drawback of the algorithm lies in high constant factor. For example, even by setting $\epsilon = 0.03$ we obtain a hidden constant of $\log_{1+0.03} 33.33 \simeq 118.63$, which results in a noticeable cost in practice. Although enlarging ϵ can reduce the constant at the price of reducing the compression efficacy, this remains the bottleneck for the building step of large inverted indexes.

3.2 Our solution: fast and exact

The interesting research question is whether there exist an algorithm that finds an *exact* solution, rather than approximated, in linear time and with *low* constant factors. This section answers positively to this question by showing that if the cost function of the chosen encoder is *point-wise*, i.e., the number of bits need to represent a single posting solely depends on such posting and not on the universe and size of the partition it belongs to, the problem admits an *optimal* and fast solution in $\Theta(n)$ time and $O(1)$ space.

In the following, we first overview and discuss our solution by explaining the intuition that lies at its core, then we give the full technical details along with a proof of optimality and the relative pseudo-code.

3.2.1 Overview

We are interested in computing the partitioning of S whose encoding cost is minimum by using *two* different encoders that take into account the relation between the size and universe of each partition. We already motivated the potential of this strategy by explaining Figure 1, which shows the distribution of the integers in dense and sparse regions of the inverted lists. Let us consider the partition $S[i,j)$, $0 \leq i < j \leq n$, of relative universe $u = S[j-1] - S[i-1] - 1$ and size $b = j-i$. Intuitively, when b gets closer to u the partition becomes denser, viceversa, it becomes sparser whenever b diverges from u. Thus the encoding cost $C(S[i,j))$ is chosen to be the minimum between $\mathsf{B}(S[i,j)) = u$ bits (dense case) and $\mathsf{E}(S[i,j))$ bits (sparse case), where B is the characteristic bit-vector of $S[i,j)$ and E is the chosen point-wise encoder for sparse regions.

Examples of point-wise encoders are VByte [40], Elias' γ-δ [19] and Golomb [23]. Other encoders, such as Elias-Fano [20, 18], Binary Interpolative Coding [30] and PForDelta [45] are not point-wise, since a different number of bits could be needed to represent the *same* integer when belonging to partitions having different characteristics, namely different length and universe. To clarify what we mean, consider the following exemplar sequence:

$$S[0,10) = \langle 8, 9, 10, 11, 12, 36, 37, 38, 39, 40 \rangle.$$

Let us now compare the behavior of Elias-Fano (non point-wise) and VByte (point-wise). By performing no splitting, Elias-Fano will use $\lceil \log(40/10) \rceil + 2 = 4$ bits to represent each posting. By performing the splitting $[0,5)[5,10)$, the first five values will be represented with 4 bits each, but the next five values with $\lceil \log(40-12-1)/5 \rceil + 2 = 5$ bits each. Instead, by performing the splitting $[0,6)[6,10)$, the first six values will use 5 bits each, while the next four only 2 bits each. Thus, performing different splittings change the cost of representation of the *same* postings for a non point-wise encoder, such as Elias-Fano. Instead, it is immediate to see that VByte will encode each element with 8 bits, regardless any partitioning.

The above example gives us an intuitive explanation of why it is possible to design a light-weight approach for a point-wise encoder E: we can compute the number of bits needed to represent a partition of S with E by just scanning its elements and summing up their costs, *knowing that performing a splitting will not change their cost of representation* nor, therefore, the one of the partition. This means that as long as the cost $\mathsf{E}(S[0,j))$, for some $0 < j \leq n$, is less than $\mathsf{B}(S[0,j))$ we know that $S[0,j)$ will be represented optimally with E. Therefore, we can safely keep scanning the sequence until the difference in cost between $\mathsf{E}(S[0,j))$ and $\mathsf{B}(S[0,j))$ becomes more than F bits. At this point, it means that E is wasting more than F bits with respect to B, thus we should stop encoding with E the current partition because we can afford to pay the fix cost F and continue the encoding with B. Now, the crucial question we would like to answer is: at which position $k < j$ should we stop encoding with E and switch to B? The answer is simple: we should stop at the position $k < j$ at which we saw the *maximum* difference between the costs of E and B, because splitting in *any* other point will yield a larger encoding cost. In other words, k represents the position at which E *gains most* with respect to B, so we will be wasting bits by splitting before or after position k. Observe that we must also require such

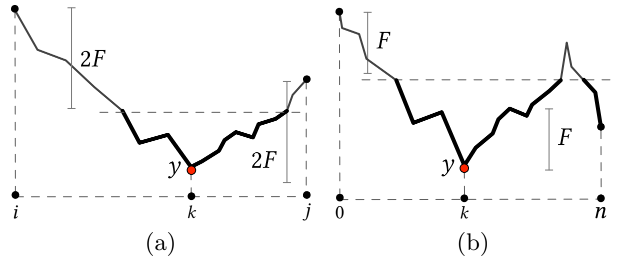


Figure 2: In case (a), we should split $S[i, j)$ in correspondence of position k because there the gain is the minimum among all points whose gain is below $2F$ from $S[i]$ *and* $S[j]$; in case (b) we should *not* split the sequence because, although an increase in gain of F bits follows, we do not have a sufficiently high gain up to $S[n-1]$ to amortize the cost of the splitting.

gain be more than F bits, otherwise switching encoder will actually cause a waste of bits. In other terms, we say that in such case the gain would not be sufficient to amortize the fix cost of the partition, meaning that we should *not* split the sequence yet.

In conclusion, we encode $S[0, k)$ with E and know that the elements $S[k, j)$ will be now best represented with B, rather than with E. Figure 2 offers a pictorial representation of how the difference between the encoding costs of E and B, referred to as the *gain* function, changes during the scan of S. When the function is decreasing, it means that E is winning over B, i.e., its encoding cost is less; conversely, when B is more efficient than E, the function is increasing.

After encoding the first partition $S[0, k)$, the process repeats: (1) we keep scanning S until B loses more than $2F$ bits with respect to E; at that point (2) we encode with B the elements in $S[k, k')$ if the maximum gain of B with respect to E, seen at position k', is greater than $2F$ bits. We keep alternating compressors until the end of the sequence.

Before sketching a compact pseudo-code of our algorithm, we first express some considerations. First of all note that, for all partitions except the first, we need to amortize twice the fix cost, because we could potentially merge the last formed partition with the current one, thus, in order to be beneficial, the difference in the cost of the two encoders must be larger than $2F$ bits. Again, refer to Figure 2a for an example. Also, for illustrative purposes, in the above discussion we have assumed that the first partition is best encoded with E: clearly, B could be better at the beginning but the algorithm will work in the very same way.

In the most general terms, call L the encoder used to represent the *last* encoded partition and C the *current* one. These will be either E or B. We also indicate with the same letters the costs in bits of their representation of the current partition. Finally, let g^* indicate the best gain of C with respect to L. At a high level, the skeleton of our algorithm looks as follows.

1. Encode the first partition.
2. If $|\mathsf{C} - \mathsf{L}|$ and g^* are greater than $2F$ bits, encode the current partition with C and swap the roles of C and L.
3. Repeat step (2) until the end of the sequence.
4. Encode the last partition.

In the above pseudo-code, the encoding of the first and last partitions its treated separately because these must amortize a fix cost of F bits instead of $2F$ bits, because we do not have any partition before and after, respectively (see Figure 2b).

It is immediate to see that the described approach can be implemented by using $O(1)$ space because we only need to keep the difference between the costs of E and B (plus some cursor variables), and that it runs in $\Theta(n)$ time because we calculate the cost in bits of each integer exactly once. We have, therefore, eliminated the linear-space complexity of *any* dynamic programming approach because we do not need to maintain the costs of the shortest path ending in each position of S. Moreover, the introduced algorithm has very low constant factors in the time complexity, since it just performs few comparisons and updates of some variables for each integer of S.

3.2.2 Technical discussion

Let $g : \mathbb{N} \cup \{0\} \to \mathbb{Z}$ be the *gain* function, defined as $g(S[j]) = \sum_{i=0}^{j-1}[\mathsf{E}(S[i]) - \mathsf{B}(S[i])]$, for $j = 0, \ldots, n-1$. In order to describe the properties of our solution, we first need the following definition.

Definition 1. Given $S[i, j)$, $0 \leq i < j \leq n$, the integer $y \in S[i, j)$ is the *point dominating* $S[i, j)$ for the encoder E, if

$$y = \underset{i<k\leq j}{\arg\min}\, g(S[k]) \text{ such that } g(S[i]) - g(y) > T, \qquad (1)$$

where $T = F$ if $i = 0$ or $2F$ otherwise, and $S[j]$ satisfies one of the following:

$$g(S[j]) - g(y) > 2F, \text{ or} \qquad (2)$$

$$g(z) - g(y) > F, \text{ for all } z \geq S[j]. \qquad (3)$$

Notice that the dominating point could not exist for any sub-sequence $S[i, j)$, but if it exists and $\mathsf{E}(x) \neq \mathsf{B}(x)$ for any $x \in S[i, j)$, it must be unique. Clearly, the definition of dominating point for encoder B is symmetric to Definition 1.

The above definition explains that, given $S[i, j)$, we can *always improve* its cost of representation by splitting the interval in correspondence of the dominating point y if it exists, otherwise we should *not* split $S[i, j)$. It is easy to see that the dominating point in $S[i, j)$ is the point in which the difference of the costs between the two compressors is maximized, thus it will be only beneficial to split in this point rather than any other point, as we explained in the previous paragraph. It is also easy to see why we should search the dominating point among the ones whose gain is at least T bits less than $g(S[i])$. The threshold T is set to the minimum amount of bits needed to amortize the cost of switching from one compressor to the other. Consider Figure 2a and suppose we are encoding with B before $S[i]$ and after y. If we compress with E the partition $S[i, k)$, we are switching encoder twice, thus the gain in y must be at least $2F$ bits less than $g(S[i])$ to be able of amortizing the cost for two switches. In Figure 2b, instead, we have no partition before $S[0]$, thus we strive to amortize the cost for a single switch.

Now, let $p(x) \in [0, n)$ be the position of integer x in $S[0, n)$, i.e., $S[p(x)] = x$. *Our strategy consists in splitting the sequence in correspondence of the dominating points*, as defined above. More precisely, the solution $\mathcal{P} = [p_1, \ldots, p_k]$

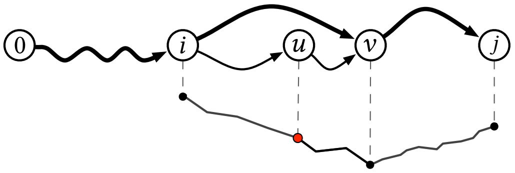


Figure 3: Path of minimum cost till position j (thick black line) and its representation in terms of gain function; u is the position of the point dominating $S[i, j)$.

output by this strategy can be described by the following recursive equation

$$p_i = p(y_i), (y_{i-1}, y_i, y_{i+1}), \text{ for } i = 1, \ldots, k, \qquad (4)$$

where $y_0 = S[0]$, $y_{k+1} = S[n-1]$ and notation $(S[i], y, S[j])$ means that y is the point dominating $S[i, j)$. In other words, any $p_i \in \mathcal{P}$, except for the first and the last, is the dominating point of the interval whose endpoints are dominating points as well.

Notice that, by definition, there cannot be two adjacent dominating points that are relative to the same encoder, but they must be relative to different encoders. In fact, suppose we have a dominating point y for E: it means that either we have seen an increase in gain of $2F$ after y and, thus, the dominating point after y (if found) will be relative to B, or the gain never goes under y and a dominating point after y if not found. This means that $\mathcal{P}$ alternates the choice of compressors, i.e., a partition encoded with E is delimited by two partitions encoded with B and viceversa (except for the first and last). We call such behavior, *alternating*.

In particular, our strategy will encode with compressor E all partitions ending with a dominating point for E (and starting with a dominating point for B, since $\mathcal{P}$ alternates the compressors). The same holds for B. As already pointed out, the only exception is made for the last partition $S[p_k, n)$, because $S[n-1]$ cannot be a dominating point by definition (no increase or decrease in gain is possible after the end of the sequence). In this case, the strategy selects the compressor that yields the minimum cost over $S[p_k, n)$.

Since a *feasible solution* to the problem is just either a singleton partition or consists in any sequence of strictly increasing positions, we argue that $\mathcal{P}$ is a feasible solution. This follows automatically by the definition of dominating point because such points are different one another and, therefore, their positions strictly increasing. If no dominating points exist, then $\mathcal{P}$ will be empty: it is a feasible solution too and indicates that S should not be cut (singleton partition). We now show the following lemma.

Lemma 1. $\mathcal{P}$ is *optimal.*

Proof. As already noted in Subsection 3.1, an optimal solution to the problem can be thought as a path of minimum cost in the DAG whose vertices are the positions of the integers of S and $C(i, j) = C(S[i, j])$ for any edge (i, j). Thus, suppose that $\mathcal{P}$ is not a shortest path and let $\mathcal{P}^* = [p^*_1, \ldots, p^*_m]$ be the shortest path sharing the longest common prefix with $\mathcal{P}$. Refer to Figure 3 for a graphical representation: i is the largest position shared by $\mathcal{P}^*$ and $\mathcal{P}$. We want to show that we can replace the edge (i, v), $v \in \mathcal{P}^*$, with the path $(i, u)(u, v)$, $u \in \mathcal{P}$, without changing the cost of $\mathcal{P}^*$, therefore extending the longest common prefix up to node $u < v$ (the case for $u > v$ is symmetric). We argue that this is only possible if $\mathcal{P}$ is optimal, otherwise it would mean that $\mathcal{P}^*$ is not a shortest path sharing a common longest prefix with $\mathcal{P}$, which is absurd by assumption.

First note that both edges (i, v) and (i, u) must be encoded with the same compressor. In fact, suppose that these are not, for example (i, v) is encoded with B and (i, u) with E. Since $v \in \mathcal{P}^*$, we know that it is optimal to encode with B until v. However, the fact that u is a dominating point for E implies that $\mathsf{B}(i, u) > \mathsf{E}(i, u)$, which is absurd because $u < v$ and B is optimal until v. Therefore, both edges must use the same encode. Assume that it is E (the case for B is symmetric).

The fact that v belongs to the optimal solution $\mathcal{P}^*$ means that if we split the edge into two (or more) pieces, we cannot decrease the cost, i.e., $\mathsf{E}(i, v) \leq \mathsf{E}(i, k) + \mathsf{B}(k, v) + F$, $\forall i \leq k \leq v$. Since E is point-wise, we have $\mathsf{E}(i, v) - \mathsf{E}(i, k) = \mathsf{E}(k, v)$ and thus, by imposing $k = u$, we obtain (1) $\mathsf{E}(u, v) \leq \mathsf{B}(u, v) + F$. Viceversa, the fact that u is a dominating point for E means that from u to v the cost is higher if we keep the same encoder, i.e., $\mathsf{E}(i, v) \geq \mathsf{E}(i, u) + \mathsf{B}(u, v) + F$. Again, by exploiting the fact that E is point-wise, we have (2) $\mathsf{E}(u, v) \geq \mathsf{B}(u, v) + F$. Conditions (1) and (2) together imply that it must be $\mathsf{E}(u, v) = \mathsf{B}(u, v) + F$, thus we have no change in the cost of $\mathcal{P}^*$ by performing the exchange, which contradicts our assumption that $\mathcal{P}$ was not optimal.

□

We are now left to present a detailed algorithm that computes $\mathcal{P}$, i.e., that iteratively finds all the dominating points of S according to Equation 4. The argue that the function `optimal_partitioning` coded in Algorithm 1 does the job.

Algorithm 1: The partitioning algorithm

```
Input: A monotone sequence $S[0, n)$
Output: The partitioning $\mathcal{P}$
Function optimal_partitioning
    $\mathcal{P} = \varnothing$
    $T = F$
    $i = j = g = 0$
    $min = max = 0$
    for $k = 0$; $k < n$; $k = k + 1$ do
        $g = g + \mathsf{E}(S[k]) - \mathsf{B}(S[k])$
        if $g$ is non-decreasing then
            if $g > max$ then
                $max = g$, $i = k + 1$
            if $min < -T$ and $min - g < -2F$ then
                update($min, max, j, i$)
        else
            if $g < min$ then
                $min = g$, $j = k + 1$
            if $max > T$ and $max - g > 2F$ then
                update($max, min, i, j$)
    close()
    return $\mathcal{P}$
```

Before proving that the algorithm is correct, let us explain the meaning of the variables used in the pseudo-code.

Algorithm 2: `update`(g_0, g_1, p_0, p_1)

```
append $p_0$ to $\mathcal{P}$
$T = 2F$, $p_1 = p + 1$
$g = g - g_0$
$g_0 = 0$, $g_1 = g$
```

Algorithm 3: `close()`

```
1 if $max > F$ and $max - g > F$ then
2     update($max, min, i, j$)
3 if $min < -F$ and $min - g < -F$ then
4     update($min, max, j, i$)
5 if $g > 0$ then
6     $i = n$, update($max, min, i, j$)
7 else
8     $j = n$, update($min, max, j, i$)
```

Call ℓ the last added position to $\mathcal{P}$. Variables i and j keep track of the positions of the points dominating the interval $S[\ell, k)$ for, respectively, B and E encoders. Likewise, *max* and *min* are the gains in correspondence of positions i and j; g indicates the gain at step k, for $k = 0, \ldots, n-1$.

Lemma 2. Algorithm 1 is *correct.*

Proof. We want to show that the array $\mathcal{P}$ returned by the function `optimal_partitioning` contains all the positions of the dominating points, as recursively described by Equation 4. We proceed by induction on the elements of $\mathcal{P}$.

The main loop in lines 6-17:

1. computes the gain g at step k (line 7);
2. updates the variables i, max (lines 9-10) and j, min (lines 14-15);
3. add new positions to $\mathcal{P}$ (lines 11-12 and 16-17).

Correctness of point 1. and 2. is immediate: the crucial point to explain is the third.

The `if` statements in lines 11 and 16 check whether positions i and j are the positions of a dominating point in $S[\ell, k)$, i.e., whether $S[i]$ and $S[j]$ satisfy Definition 1. Since the `if` statements are symmetric, we proved the correctness of the first one for non-decreasing values of g (line 11).

We first check whether the *min* gain, as seen so far, is sufficient to be the one of a dominating point for E as required by Equation 1. At the beginning of the algorithm, the current interval starts at $i = 0$ and $T = F$, therefore $g(S[0]) = 0$ in Equation 1 and the test $min < -T$ is correct. If $min < -T$ is true, then we also check if we have a sufficiently large increase in gain at the current step k with respect to the previously seen *min* gain according to Condition 2. Again, it is immediate to see that the test $min - g < -2F$ checks such condition and, therefore, it is correct. If both previous conditions are satisfied, then j is the position of the dominating point for E in the first interval $S[0, k)$ by Definition 1. If so, we can execute the `update` code, show in Algorithm 2, which adds j to $\mathcal{P}$ and sets T to $2F$ according to Definition 1. Moreover, it updates the gain g to maintain the invariant that its value is always relative to the current interval, which now begins at position j. In fact: since we have seen an increase of $2F$ bits, the *max* gain in $S[j, k)$ must be the current gain g, whereas the *min* gain is 0 because g is non-decreasing. Thus, the first point is computed correctly.

Now, assume that we have added m points to $\mathcal{P}$ and that the last added is for encoder E. We want to show that the next point will be dominating for encoder B. As explained before, whenever we add a dominating point for E to $\mathcal{P}$, it means that we have seen an increase of $2F$ bits with respect to the last added position, i.e., position $k+1$ is the one of a point that satisfies Equation 1 for encoder B. Therefore the $(m+1)$-th point added to $\mathcal{P}$ will be dominating for B.

To conclude, we have to explain what happens at the end of the algorithm. Refer to the `close` function, coded in Algorithm 3. Lines 1-2 (3-4) check Condition 3:

if successful, then *max* (*min*) is the next dominating point for B (E) and, since compressors must alternate each other, we close the encoding of the sequence with the other compressor in lines 5-8, that is E (B);

if both unsuccessful, i.e., no dominating point is found, then it means that the remaining part of the sequence should not be cut and, thus, encoded with a single compressor in lines 5-8.

□

In conclusion, since we consider each element of S once and use a constant number of variables, Lemma 1 and 2 imply the following result.

Theorem 1. A monotone integer sequence of size n can be partitioned *optimally* in $\Theta(n)$ time and $O(1)$ space, whenever its partitions are represented with a *point-wise* encoder and *characteristic bit-vectors.*

4. EXPERIMENTAL EVALUATION

Datasets. We performed our experiments on the following two standard datasets, whose statistics are summarized in Table 1.

- Gov2 is the TREC 2004 Terabyte Track test collection, consisting in roughly 25 million `.gov` sites crawled in early 2004. Documents are truncated to 256KB.
- ClueWeb09 is the ClueWeb 2009 TREC Category B test collection, consisting in roughly 50 million English web pages crawled between January and February 2009.

	Gov2	ClueWeb09
Documents	24 622 347	50 131 015
Terms	35 636 425	92 094 694
Postings	5 742 630 292	15 857 983 641

Table 1: Basic statistics for the tested collections.

Standard text pre-processing was performed before indexing the collections: terms were extracted using Apache Tika; lower-cased and stemmed using the `Porter2` stemmer; no stopwords were removed form the collections. Document identifiers (docIDs) were assigned by following the order of the URLs of the documents [37].

	Gov2						ClueWeb09					
	space GB	doc bpi	freq bpi	building minutes	decoding ns/int	AND query ms/query	space GB	doc bpi	freq bpi	building minutes	decoding ns/int	AND query ms/query
Varint-GB	15.06	11.15	9.77	10.60	2.35	0.88	42.23	11.43	9.80	46.50	2.45	5.32
Varint-G8IU	14.00	10.43	9.00	18.00	2.21	0.84	39.43	10.84	8.99	65.80	2.25	5.10
Masked-VByte	12.64	9.53	8.02	10.50	2.35	0.90	35.63	9.91	8.01	45.50	2.55	5.52
Stream-VByte	15.06	11.15	9.77	10.60	2.26	0.86	42.23	11.43	9.80	46.50	2.30	5.30

Table 2: The performance of the Variable-Byte family on Gov2 and ClueWeb09: space in giga bytes (GB); average number of bits (bpi) per document (doc) and frequency (freq); sequential decoding time in nanoseconds per integer (ns/int) and query processing time for TREC-05 AND queries in milliseconds per query (ms/query).

Experimental setting and methodology. All the experiments were run on a machine with 4 Intel i7-4790K CPUs (8 threads) clocked at 4.00GHz and with 32GB of RAM DDR3, running Linux 4.13.0 (Ubuntu 17.10), 64 bits. The implementation of our partitioned indexes is in standard C++14 and it is a flexible template library allowing *any* point-wise encoder to be used, provided that its interface exposes a method to compute the cost in bits of a single integer in constant time. The source code, that will be released upon publication of the paper to favor further research and reproducibility of results, was compiled with gcc 7.2.0 using the highest optimization setting.

To test the building time of the indexes we measure the time needed to perform the whole building process, that is: (1) fetch the posting lists from disk to main memory; (2) encode them in main memory; (3) save the whole index data structure back to a file on disk. Since the process is mostly I/O bound, we make sure to avoid disk caching effects by clearing the disk cache before building the indexes.

To test the query processing speed of the indexes, we memory map the index data structures on disk and compute boolean conjunctions over a set of random queries drawn from TREC-05 and TREC-06 Efficiency Track topics. We repeat each experiment three times to smooth fluctuations in the measurements and consider the mean value. The query algorithm runs on a single core and timings are reported in milliseconds. To test sequential decoding speed instead, we measure the time to touch each element of all the posting lists whose size is in between 0.5 and 5 million postings, for a total of 2033 lists for Gov2 (2.9 billion postings) and 3593 lists for ClueWeb09 (5.5 billion postings).

In all the experiments, we used the value $F = 64$ bits for partitioning the posting lists for both VByte and Elias-Fano (henceforth, EF).

Organization. The aim of this section is to measure the space improvement, indexing time and query processing speed of indexes that are optimally partitioned by the algorithm described in Section 3. Since we adopt VByte as exemplar point-wise encoder, the next subsection compares the performance of all the encoders in the VByte family in order to chose the most convenient for the subsequent experiments. Then, we measure the benefits of applying our optimization algorithm on the chosen VByte encoder, by comparing the corresponding partitioned index against the un-partitioned counterpart. Finally, we consider the comparison with many other compressors for inverted indexes at the state-of-the-art, as described in Subsection 2.1.

4.1 The performance of the Variable-Byte family

To help us in deciding which VByte encoder to chose for our subsequent analysis, we consider the Table 2. The data reported in the table illustrates how different VByte stream organizations actually impact index space. Since Varint-GB and Stream-VByte take exactly the same space, given that Stream-VByte stores the very same control and data bytes but concatenated in separate streams, in the following we refer to Varint-GB to mean both of them. As we can see, the original VByte format (referred to as Masked-VByte in Table 2 because it uses this algorithm to perform decoding) is the most space-efficient among the family. This is no surprise given the distribution plotted in Figure 1: it means that the majority of the encoded d-gaps falls in the interval $[0, 2^7)$, otherwise the compression ratio of VByte would have been worse than the one of Varint-GB and Varint-G8IU. As an example, consider the sequence of d-gaps $[132, 233, 246, 178]$. VByte uses 8 bytes to represent such sequence, whereas Varint-GB uses 1 control byte and 4 data bytes, thus 5 bytes in total. When all values are in $[0, 2^7)$, VByte uses 4 bytes instead of 5 as needed by Varint-GB. For this reason, the space usage of Varint-GB and Varint-G8IU is worse than the one of VByte: it is 16% and 15% on Gov2 and ClueWeb09 respectively for Varint-GB; more than 10% for Varint-G8IU. The control byte of Varint-G8IU stores a bit for every of the 8 data bytes: a 0 bit means that the corresponding byte completes a decoded integer, whereas a 1 bit means that the byte is part of a decoded integer or it is wasted. Thus, Varint-G8IU compress worse than plain VByte due to the wasted bytes. Finally notice that Varint-GB is slightly worse than Varint-G8IU because it uses 10 bits per integer instead of 9 for all integers in $[0, 2^8)$. In fact, the difference between these two encoders in less than 1 bit on both Gov2 and ClueWeb09.

The speed of the encoders is actually very similar for all alternatives, regarding both AND queries and sequential decoding: the spread between the fastest (Varint-G8IU) and the slowest alternative (Masked-VByte) is as little as $6 \div 10\%$. The same holds true for the building of the indexes where, as expected, the plain VByte is the fastest and Varint-G8IU is 40% slower on average.

In conclusion, for the reasons discussed above, i.e., better space occupancy, fastest index building time and competitive speed, we adopt the original VByte stream organization and the Masked-VByte algorithm by Plaisance, Kurz and Lemire [35] to perform sequential decoding.

	Gov2			ClueWeb09		
	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi
VByte	12.64 (+122.74%)	9.53 (+95.75%)	8.02 (+163.92%)	35.63 (+99.26%)	9.90 (+51.52%)	8.01 (+222.39%)
VByte uniform	6.26 (+10.22%)	5.41 (+11.05%)	3.31 (+8.92%)	19.95 (+11.58%)	7.37 (+12.73%)	2.69 (+8.54%)
VByte ϵ-optimal	5.73 (+0.93%)	4.93 (+1.21%)	3.05 (+0.49%)	18.15 (+1.53%)	6.66 (+1.84%)	2.50 (+0.68%)
VByte optimal	**5.68**	**4.87**	**3.04**	**17.88**	**6.54**	**2.48**

Table 3: Space in giga bytes (GB) and average number of bits (bpi) per document (doc) and frequency (freq).

4.2 Optimizing Variable-Byte: partitioned vs. un-partitioned indexes

In this subsection, we evaluate the impact of our solution by comparing the optimally-partitioned VByte indexes against the un-partitioned indexes and the ones obtained by using other partitioning strategies, like the ϵ-optimal based on dynamic programming (see Subsection 3.1) and the uniform one. The result of the comparison shows that our optimally-partitioned VByte indexes are 2× smaller than the original, un-partitioned, counterparts; can be built 2.6× faster than dynamic programming and offer the strongest guarantee, i.e., an exact solution rather than an ϵ-approximation; despite the significant space savings, they are as fast as the original VByte indexes.

Index space. Table 3 shows the results concerning the space of the indexes. Compared to the case of un-partitioned indexes, we observe gains ranging from 51% up to 222%, with a net factor of 2× improvement with respect to the original VByte format. For the uniform partitioning we used partitions of 128 integers, for both documents and frequencies. As we can see, this simple strategy already produces significant space savings: it is 43.3% and 25.6% better on doc sequences for Gov2 and ClueWeb09 respectively; 58.7% and 66.3% better on freq sequences. This is because most d-gaps are actually very small but *any* un-partitioned VByte encoder needs at least 8 bits per d-gap. In fact, notice how the average bits per integer on the doc sequences becomes less than 8.

We remark that the ϵ-optimal algorithm based on dynamic programming was proposed for Elias-Fano, whose cost in bits can be computed in $O(1)$: we adapt the dynamic programming recurrence in order to use it for VByte too. As approximation parameters we used the same as in the experiments of the original paper [32], i.e., we set $\epsilon_1 = 0.03$ and $\epsilon_2 = 0.3$. The computed approximation could be possibly large by enlarging such parameters, while our algorithm finds an exact solution. However, we notice that the ϵ-approximation is good and our optimal solution is slightly better: precisely by 1.2% and 1.84% on the doc sequences of Gov2 and ClueWeb09, respectively. Compared to uniform, the optimal partitioning pays off: indeed it produces a further saving of 11% on average, thus confirming the need for an optimization algorithm.

Index building time. Although the un-partitioned variant would be the fastest to build in internal memory because the posting lists are compressed in the same pass in which these are read from disk, the writing of the data structure to the disk imposes a considerable overhead because of the high memory footprint of the un-partitioned index. Notice how this factor becomes dramatic for the larger dataset ClueWeb09, resulting in a 50% overhead. This also causes the simple uniform strategy be not faster at all, being actually a little slower (by 5% on average). Despite the linear-time complexity as soon as ϵ is constant, the ϵ-optimal solution has a noticeable CPU cost due to the high constant factor, as we motivated in Section 3. The optimal solutions has instead low constant factors and, as a result, is faster than the dynamic programming approach by more than 2.6× on average on both Gov2 and ClueWeb09.

	Gov2	ClueWeb09
VByte	10.10 (−3.81%)	43.30 (+51.93%)
VByte uniform	11.30 (+7.62%)	29.30 (+2.81%)
VByte ϵ-optimal	26.70 (+154.29%)	72.30 (+153.68%)
VByte optimal	**10.50**	**28.50**

Table 4: Index building timings in minutes.

		Gov2	ClueWeb09
TREC 05	VByte	0.90 (+1.37%)	5.56 (−2.54%)
	VByte uniform	0.94 (+5.07%)	5.90 (+3.45%)
	VByte ϵ-optimal	0.92 (+2.70%)	5.89 (+3.34%)
	VByte optimal	**0.89**	**5.70**
TREC 06	VByte	2.12 (+0.02%)	8.35 (−6.90%)
	VByte uniform	2.22 (+4.98%)	9.02 (+0.60%)
	VByte ϵ-optimal	2.24 (+5.77%)	9.17 (+2.31%)
	VByte optimal	**2.12**	**8.96**

(a) AND queries (ms/query)

	Gov2	ClueWeb09
VByte	2.35 (−4.08%)	2.55 (−8.93%)
VByte uniform	2.75 (+12.24%)	2.90 (+3.57%)
VByte ϵ-optimal	2.60 (+6.12%)	2.80 (+0.00%)
VByte optimal	**2.45**	**2.80**

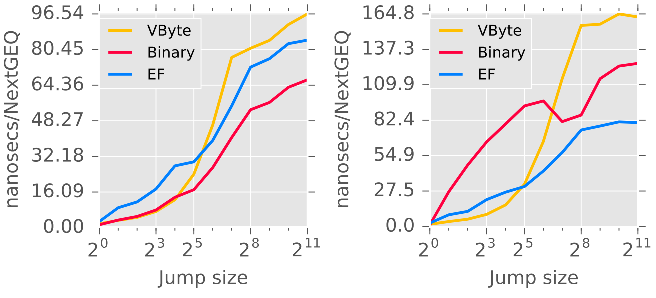
(b) decoding time (ns/int)

Table 5: Timings for AND queries in milliseconds (ms/query) and sequential decoding time in nanosecond per integer (ns/int).

Index speed: AND queries and decoding. Table 5 illustrates the results. The striking result of the experiment is that, despite the significant space reduction (2× improve-

	Gov2			ClueWeb09		
	space GB	doc bpi	freq bpi	space GB	doc bpi	freq bpi
PEF ϵ-optimal	4.65 (-18.06%)	4.10 (-15.69%)	2.38 (-21.82%)	15.94 (-10.84%)	5.85 (-10.57%)	2.20 (-11.56%)
OptPFD	4.96 (-12.56%)	4.48 (-7.97%)	2.38 (-21.76%)	17.15 (-4.11%)	6.18 (-5.43%)	2.41 (-2.86%)
BIC	4.30 (-24.17%)	3.80 (-22.00%)	2.14 (-29.49%)	14.01 (-21.63%)	5.15 (-21.28%)	1.87 (-24.81%)
ANS	4.17 (-26.53%)	3.96 (-18.73%)	1.85 (-39.01%)	14.47 (-19.09%)	5.36 (-18.02%)	1.94 (-21.91%)
QMX	6.77 $(+19.20\%)$	6.00 $(+23.27\%)$	3.37 $(+10.76\%)$	23.44 $(+31.12\%)$	8.01 $(+22.59\%)$	3.75 $(+51.19\%)$
VByte optimal	**5.68**	**4.87**	**3.04**	**17.88**	**6.54**	**2.48**

Table 6: Space in giga bytes (GB) and average number of bits (bpi) per document (doc) and frequency (freq).



(a) Dense

(b) Sparse

Figure 4: Nanoseconds per NextGEQ query for (a) Dense and (b) Sparse sequences of one million integers, having an average gap of 2.5 and 1850, respectively. These values mimic the ones for the Gov2 datasets, that are 2.125 and 1852. The ones for the ClueWeb09 dataset are 2.137 and 963, thus the plots have a similar shape.

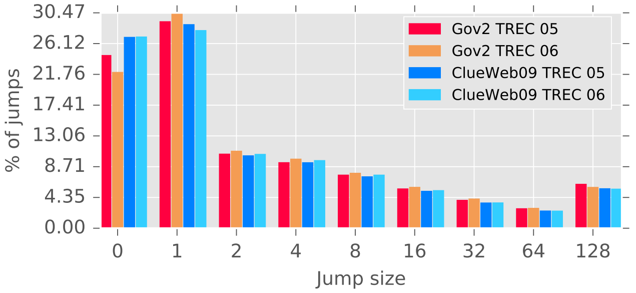


Figure 5: When the difference between two consecutive accessed positions is d, NextGEQ is said to make a jump of size d. The distribution of the jump sizes is divided into buckets of exponential size: all sizes between 2^{d-1} and 2^d belong to bucket d. The plot shows the jumps distribution, in percentage, for the query logs used in the experiments, when performing AND queries.

ment, see Table 3), the partitioned indexes are as fast as the un-partitioned ones on both the Gov2 and ClueWeb09 datasets, with only a minor slowdown of $4 \div 9\%$ in sequential decoding.

It is, therefore, important a careful explanation of such result. The answer is provided by understanding the plots in Figure 4, along with the ones in Figures 1 and 5. In particular, Figure 4 illustrates the average nanoseconds spent per NextGEQ query by VByte, the binary vector representation and Elias-Fano (EF) on a sequence of one million integers and with an average gap of: (a) 2.5, as a *dense* case and (b) 1850 as a *sparse* case. As the dense case illustrates, the binary vector representation is as fast as VByte for all jumps of entity less then or equal to 8, and becomes actually faster for longer jumps. Moreover, the distribution of the jump sizes plotted in Figure 5 indicates us that, whenever executing AND queries, the number of jumps of size less than 16 accounts for ≈90% of the jumps performed by NextGEQ. Furthermore, the distribution plotted in Figure 1 tells us that the majority of blocks are actually encoded with their characteristic bit-vector, thus explaining why the partitioned indexes exhibit no penalty against the un-partitioned counterparts.

However, VByte tends to be slower on longer jumps because of its block-wise organization: since a posting list is split into blocks of 128 postings that are encoded separately, a block must be completely decoded even for accessing a single integer, which is not uncommon for boolean conjunctions. Moreover, since d-gaps values are encoded, we need to access the elements by a linear scan of the block after decoding in order to compute their prefix sums. When the accessed elements per block are very few, even using SIMD instructions to perform decoding results in a slower query execution. Conversely and as expected, the binary vector representation is inefficient for the *sparse* regions since potentially many bits need to be scanned to perform a query, but still faster than VByte whenever the jump size becomes larger than 64 because it allows skipping over the bit stream by keeping samples of the bit set positions.

4.3 Overall comparison

In this section we compare the optimally-partitioned VByte indexes against several competitors: (1) the ϵ-optimal partitioned Elias-Fano (PEF) by Ottaviano and Venturini [32]; (2) the Binary Interpolative coding (BIC) by Moffat and Stuiver [30]; (3) the optimized PForDelta (OptPDF) by Yan *et al.* [45]; (4) the recent ANS-based technique by Moffat and Petri [29] (5) the QMX mechanism by Trotman [41].

For all competitors, we used the C++ code from the original authors, compiled with gcc 7.2.0 using the highest optimization setting as we did for our own code, to ensure a fair comparison.

It would also be interesting to compare against the MILC framework [43], although not tailored for AND queries but only for random access, if it were publicly available.

Index space and building time. Table 6 shows the results concerning the space of the indexes. Clearly, the space usage of the VByte optimal indexes is higher than the one of the bit-aligned encoders: this was expected since VByte is byte-aligned. However, the important result is that its space is not so high as it was used to be before: it is now 22% larger than BIC and only 11% larger than PEF on the larger ClueWeb09 (18% on Gov2). Comparing the results reported in Table 3 with the ones in Table 6, we see that, without partitioning, VByte was 172% larger than PEF and 194% larger than BIC on Gov2; 123% larger than PEF and 154% larger than BIC on ClueWeb09. Thus, we reduced the space gap between the original VByte index representation and the one of the best bit-aligned encoders by nearly 11×. In particular, notice that it is less than 11% larger than PEF on the docID sequences, while it is more inefficient on the frequency sequences. This is because the frequencies are made up of smaller d-gaps than the ones present in the docIDs. Very similar considerations hold for the other alternatives, such as PFD and ANS. In particular, we notice that on the ClueWeb09 dataset, the difference between VByte optimal and OptPDF is very small (only 4% overall); ANS is particularly better than the other methods on the freq sequences; the byte-aligned QMX is, instead, significantly larger (by 19% on Gov2 and by 31% on ClueWeb09).

We now consider the time needed to build the indexes. We do not show the corresponding table for space constraints. As already noted in the previous subsection, the dynamic programming approach used for PEF imposes a severe penalty with respect to VByte optimal of 4× on average. The penalty is due to not only the difference in speed between dynamic programming and the algorithm devised in Section 3, but also to the fact that Elias-Fano, being bit-aligned, is slower to encode with respect to VByte. Except for the ANS indexes which are slower to build, by 23% on average, because of the two-pass process of first collecting symbol occurrence counts and, then, encoding [29], the building timings for the other competitors are, instead, competitive: our optimization algorithm only takes a couple of minutes more overall the whole building process. Only BIC and QMX took considerably less indexing time (33% faster on average).

Index speed: AND queries and decoding. Table 7 shows the query processing speed of the indexes, along with the corresponding sequential decoding speed. Compared to PEF, the results are indeed very similar to the ones obtained by Ottaviano and Venturini [32], i.e., there is only a marginal gap between the speed of PEF and VByte when computing boolean conjunctions. The reason has to be found, again, in the plot illustrated in Figure 4b. As we can see, for all the jump sizes less than 32, VByte is 2× faster than Elias-Fano, while this advantage vanishes for the longer jumps thanks to the powerful skipping abilities of Elias-Fano [42, 32, 34]. However, we know that this advantage is shrunk because jumps larger than 32 are not very frequent on the tested query logs, as depicted by the distribution of Figure 5.

Compared to the other approaches, we can see significant gains with respect to OptPDF (by 40% on Gov2 and 21% on ClueWeb09), BIC and ANS (4× faster on average) and only a slight penalty with respect to QMX (by 7 ÷ 14% on ClueWeb09). The same conclusions apply to the sequential decoding speed of the indexes.

		Gov2		ClueWeb09	
TREC 05	PEF ϵ-optimal	0.98	(+9.51%)	5.87	(+3.04%)
	OptPFD	1.28	(+43.35%)	8.04	(+40.99%)
	BIC	4.14	(+364.16%)	25.42	(+345.90%)
	ANS	4.21	(+372.16%)	25.98	(+355.74%)
	QMX	0.88	(−0.96%)	5.30	(−7.01%)
	VByte optimal	**0.89**		**5.70**	
TREC 06	PEF ϵ-optimal	2.19	(+3.60%)	9.59	(+6.95%)
	OptPFD	3.00	(+41.58%)	11.95	(+33.33%)
	BIC	9.93	(+369.29%)	37.87	(+322.48%)
	ANS	9.48	(+347.86%)	38.07	(+324.68%)
	QMX	2.11	(−0.52%)	8.07	(−9.99%)
	VByte optimal	**2.12**		**8.96**	

(a) AND queries (ms/query)

	Gov2		ClueWeb09	
PEF ϵ-optimal	2.60	(+6.12%)	3.18	(+13.57%)
OptPFD	2.88	(+17.55%)	3.50	(+25.00%)
BIC	7.50	(+206.12%)	9.80	(+250.00%)
ANS	5.89	(+140.41%)	9.34	(+233.57%)
QMX	2.25	(−8.16%)	2.40	(−14.29%)
VByte optimal	**2.45**		**2.80**	

(b) decoding time (ns/int)

Table 7: Timings for AND queries in milliseconds (ms/query) and sequential decoding time in nanosecond per integer (ns/int).

5. CONCLUSIONS

We have presented an optimization algorithm for point-wise encoders that splits a posting list into variable-sized partitions to improve its compression and runs in linear time and constant space. We also proved that the algorithm is optimal, i.e., it finds the splitting that minimizes the space of the representation. This sensibly improves the previous approaches based on dynamic-programming on all aspects: time/space complexity and practical performance.

By applying our technique to the ubiquitous Variable-Byte encoding, we improved its compression ratio by 2× and build optimally-partitioned indexes 2.6× faster than the dynamic programming approach. Despite the significant space savings, the partitioned representation does not introduce penalties at query processing time, being actually the fastest when compared with many other state-of-the-art encoders.

As a last note, we mention the possibility of introducing another encoder for *representing the runs* of the posting lists. Clearly, a run of consecutive docIDs can be described with just the information stored in the first level of representation, i.e., the size of the run. Although our framework can be extended to include this case, the algorithm and its analysis become much more complicated. This additional complexity does not pay off, because space improved by only 4.5% on the tested datasets.

6. REFERENCES

[1] Dropbox Techblog. `https://blogs.dropbox.com/tech/2016/09/improving-the-performance-of-full-text-search/`. Accessed on 15-04-2018.

[2] Protocol Buffers - Google's data interchange format. `https://github.com/google/protobuf`. Accessed on 15-04-2018.

[3] RediSearch. `https://github.com/RedisLabsModules/RediSearch/blob/master/docs/DESIGN.md`. Accessed on 15-04-2018.

[4] UpscaleDB. `https://upscaledb.com/about01.html#compression`. Accessed on 15-04-2018.

[5] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013.

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

[7] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Software: Practice and Experience*, 40(2):131–147, 2010.

[8] B. Bhattacharjee, L. Lim, T. Malkemus, G. Mihaila, K. Ross, S. Lau, C. McArthur, Z. Toth, and R. Sherkat. Efficient index compression in DB2 LUW. *Proceedings of the VLDB Endowment*, 2(2):1462–1473, 2009.

[9] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321. ACM, 2002.

[10] A. Buchsbaum, G. Fowler, and R. Giancarlo. Improving table compression with combinatorial optimization. *Journal of the ACM*, 50(6):825–851, 2003.

[11] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at twitter. In *Proceedings of the 28th International Conference on Data Engineering (ICDE)*, pages 1360–1369. IEEE, 2012.

[12] S. Büttcher, C. Clarke, and G. Cormack. *Information retrieval: implementing and evaluating search engines.* MIT Press, 2010.

[13] F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Universal indexes for highly repetitive document collections. *Information Systems*, 61:1–23, 2016.

[14] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A system for searching the social graph. In *Proceedings of the Very Large Database Endowment (VLDB)*, volume 6, pages 1150–1161, 2013.

[15] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the 2nd International Conference on Web Search and Data Mining (WSDM)*, 2009.

[16] B. Debnath, S. Sengupta, and J. Li. Skimpystash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 25–36. ACM, 2011.

[17] J. Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, 2013.

[18] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

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

[20] R. M. Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT*, 1971.

[21] P. Ferragina, I. Nitto, and R. Venturini. On optimally partitioning a text to improve its compression. *Algorithmica*, 61(1):51–74, 2011.

[22] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, pages 370–379, 1998.

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

[24] V. Hristidis, Y. Papakonstantinou, and L. Gravano. Efficient IR-style keyword search over relational databases. In *Proceedings 2003 VLDB Conference*, pages 850–861. Elsevier, 2003.

[25] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2013.

[26] D. Lemire, N. Kurz, and C. Rupp. Stream-VByte: faster byte-oriented integer compression. *Information Processing Letters*, 130:1–6, 2018.

[27] C. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[28] A. Moffat and M. Petri. ANS-based index compression. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pages 677–686, 2017.

[29] A. Moffat and M. Petri. Index compression using byte-aligned ANS coding and two-dimensional contexts. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, WSDM 2018, Marina Del Rey, CA, USA, February 5-9, 2018*, pages 405–413, 2018.

[30] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval Journal*, 3(1):25–47, 2000.

[31] G. Ottaviano, N. Tonellotto, and R. Venturini. Optimal space-time tradeoffs for inverted indexes. In *Proceedings of the 8th Annual International ACM Conference on Web Search and Data Mining (WSDM)*, pages 47–56, 2015.

[32] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proceedings of the 37th International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 273–282, 2014.

[33] G. E. Pibiri and R. Venturini. Clustered Elias-Fano indexes. *ACM Transactions on Information Systems*, 36(1):1–33, 2017.

[34] G. E. Pibiri and R. Venturini. Inverted index compression. *Encyclopedia of Big Data Technologies*, pages 1–8, 2018.

[35] J. Plaisance, N. Kurz, and D. Lemire. Vectorized VByte decoding. In *International Symposium on Web Algorithms (iSWAG)*, 2015.

[36] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 773–784. ACM, 2007.

[37] F. Silvestri. Sorting out the document identifier assignment problem. In *Proceedings of the 29th European Conference on IR Research (ECIR)*, pages 101–112, 2007.

[38] F. Silvestri and R. Venturini. VSEncoding: Efficient coding and fast decoding of integer lists via dynamic programming. In *Proceedings of the 19th International Conference on Information and Knowledge Management (CIKM)*, pages 1219–1228, 2010.

[39] A. Stepanov, A. Gangolli, D. Rose, R. Ernst, and P. Oberoi. Simd-based decoding of posting lists. In *Proceedings of the 20th International Conference on Information and Knowledge Management (CIKM)*, pages 317–326, 2011.

[40] L. H. Thiel and H. Heaps. Program design for retrospective searches on large data bases. *Information Storage and Retrieval*, 8(1):1–20, 1972.

[41] A. Trotman. Compression, simd, and postings lists. In *Proceedings of the 2014 Australasian Document Computing Symposium*, page 50. ACM, 2014.

[42] S. Vigna. Quasi-succinct indices. In *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 83–92, 2013.

[43] J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson. MILC: inverted list compression in memory. *Proceedings of the VLDB Endowment*, 10(8):853–864, 2017.

[44] H. E. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[45] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web (WWW)*, pages 401–410, 2009.

[46] Z. Zhang, J. Tong, H. Huang, J. Liang, T. Li, R. J. Stones, G. Wang, and X. Liu. Leveraging context-free grammar for efficient inverted index compression. In *Proceedings of the 39th International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 275–284, 2016.

[47] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):1–56, 2006.

[48] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pages 59–70, 2006.