This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2911288, IEEE Transactions on Knowledge and Data Engineering

On Optimally Partitioning Variable-Byte Codes

Giulio Ermanno Pibiri and Rossano Venturini

Abstract—The ubiquitous *Variable-Byte* encoding is 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 $2\times$ 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 does not affect indexing time because of its linear-time complexity; 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.

Index Terms—Inverted Index Compression, Variable-Byte Encoding, Performance Evaluation

1 Introduction

The *inverted index* is the core data structure at the basis of large-scale search engines, database architectures and social networks [1], [2], [3], [4], [5]. In its simplicity, the inverted index can be regarded as being a collection of sorted integer sequences, called inverted or posting lists. For example, when the inverted 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 [2]. 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 [4]. 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 [6], [7]. Also, finding all occurrences of twig patterns in XML databases can be done efficiently by resorting on an inverted index [8]. 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 a hash collision: the list of all such elements is recorded that is, basically, an inverted list [9].

Because of the huge quantity of data 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 by also 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 space/time trade-off. We point the reader to [10] for an overview of the different encoders that have been proposed through the years.

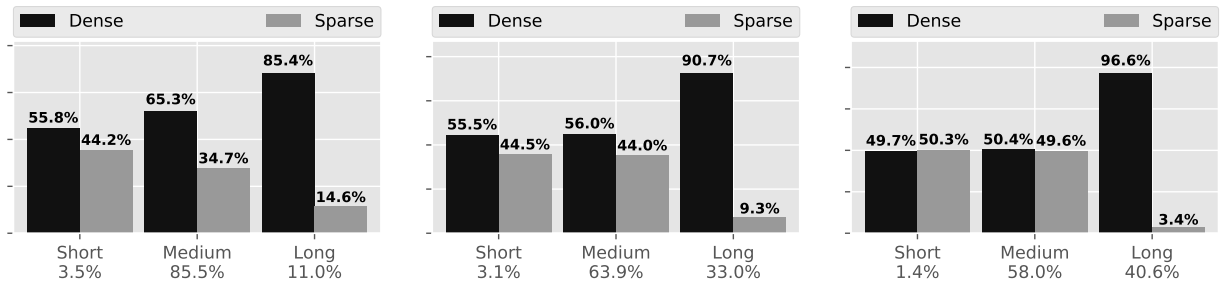
Among these, *Variable-Byte* [11], [12] (henceforth, VByte) is the most popular and used byte-aligned code. In particular, VByte owes its popularity to its *sequential* decoding speed and, indeed, it is one of the fastest representation up to date for integer sequences. For this reason, it is widely adopted by well-known companies as a key tool to enable fast search of records. We mention some noticeable examples. Google uses VByte extensively: for compressing the posting lists of inverted indexes [13] and as a binary wire format for its protocol buffers [14]. IBM DB2 employs VByte to store the differences between successive record identifiers [15]. Amazon patented an encoding scheme, based on VByte and called Varint-G8IU, which uses SIMD (Single Instruction Multiple Data) instructions to perform decoding faster [16]. Many other storage architectures rely on VByte, such as Redis [17], UpscaleDB [18] and Dropbox [19].

We now quickly review how the VByte encoding works. It was first described by Thiel and Heaps [11]. The binary representation of a non-negative integer is divided into groups of 7 bits which are represented as a sequence of bytes. In 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, the integer $65{,}790$ is represented as $\underline{1}0000100\ \underline{1}0000001\ \underline{0}1111110$ (with control bits underlined.) 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 $\lceil \frac{\lceil \log_2(x+1) \rceil}{7} \rceil \times 8$ bits to represent an integer $x \geq 0$. Decoding is simple: we just need to read one byte at a time until we find a value smaller than 2^7. As already mentioned, the format is also suitable for SIMD instructions for speeding up sequential decoding.

The main drawback of VByte lies in its byte-aligned na-

• *Affiliation: University of Pisa and ISTI-CNR, Italy.*
E-mails: giulio.pibiri@di.unipi.it, rossano.venturini@unipi.it.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2911288, IEEE Transactions on Knowledge and Data Engineering



(a) Gov2 (b) ClueWeb09 (c) CCNews

Fig. 1. Percentage of integers belonging to *dense* and *sparse* regions of the inverted lists for the tested datasets. The inverted 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 integers belonging to its posting lists.

ture, 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 contain regions of (very) close identifiers that are far more compressible than highly scattered regions [20], [21], [10]. 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 domain: these are likely to share a lot of terms. Also, the values stored in the columns of databases typically exhibit high locality and that is why column-oriented databases can achieve very good compression and high query throughput [22].

The key point is that effective 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, because its space occupancy can be up to $3\times$ larger than bit-aligned compressors [25], [21], [23], [24].

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 $\approx\log_2 5$ bits. To better highlight how this behavior can deeply affect compression effectiveness, we consider the statistic shown in Fig. 1. This statistic reports the percentage of postings belonging to *dense* and *sparse* regions of the lists for the datasets Gov2, ClueWeb09 and CCNews. More precisely, the plot originated from the following experiment: we divided each inverted list into blocks of 128 integers and we considered as *sparse* a block where VByte yielded a better space occupancy with respect to the *characteristic bit-vector* representation of the block (if u is the last element in the block, we have the i-th bit set in a bitmap of size u for all integers i belonging to the block), 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 almost the entirety of integers concentrate in the lists of medium (thanks to the Zipfian distribution of words in text) and long size, where indeed the majority 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 representing inverted indexes, by exhibiting an improved compression ratio of $2\times$ on standard datasets consisting in several billions of postings, such as Gov2, ClueWeb09 and CCNews.

The result is achieved by partitioning the inverted lists into blocks and representing each block with the most suitable encoder, chosen among VByte and its characteristic bit-vector representation. Partitioning the lists has the potential of adapting compression to the distribution of the integers in the lists, such as the one shown in Fig. 1, i.e. using 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 problem of minimizing the space of representation of an inverted list of size n by representing it with variable-length partitions. By exploiting the fact that VByte is a *point-wise* encoder, i.e., the number of bits to represent an integer solely depends on the *value* of the integer itself and not on the universe and size of the block to which it belongs to, we introduce an algorithm that finds an *optimal* partitioning in $\Theta(n)$ time and $O(1)$ space.

We remark that the dynamic programming algorithm presented in [21] can be used as well to find a $(1+\epsilon)$-optimal solution to the problem, by taking $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$ time and $O(n)$ space for any $\epsilon \in (0, 1)$. Apart from being approximated rather than exact, this solution is generally applicable to any encoder whose size in bits can be computed (or estimated) in constant time on a block and

does not rely on the fact that VByte is point-wise. As a consequence, it is also noticeably slower than the algorithm we introduce in this work. Lastly, we also remark that, although we use VByte in the experiments, our approach can be applied to *any* point-wise encoder.

3) We conduct an extensive experimental analysis to demonstrate the effectiveness of our approach on standard large datasets, such as Gov2, ClueWeb09 and CCNews. More precisely, when compared to the un-partitioned VByte indexes, the optimally-partitioned counterparts are: (1) *significantly smaller*, by $2\times$ on average; (2) *not* slower at computing boolean conjunctions; (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) [21], Binary Interpolative coding (BIC) [20], the optimized PForDelta (OptPFD) [26], an index organization based on Asymmetric Numeral Systems (ANS) [25] and the QMX mechanism [27]. The partitioned VByte representation reduces the gap between the space of VByte and the one of the best bit-aligned compressors, such as 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 does not introduce any query processing overhead: only QMX is slightly faster on ClueWeb09 by $1 \div 10\%$, but also 30% larger.

2 Problem Statement and Related Work on Partitioning Algorithms

In this paper 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 representation 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 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.

The problem. Now, the natural arising 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 computes 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.

2.1 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 likely be 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. Silvestri and Venturini [28] obtained a $O(n \times h)$ construction time, where h is the size of the longest allowed partition. Ferragina *et al.* [29] improve the result in [30] 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.

Dynamic programming: slow and approximated. Ottaviano and Venturini [21] resort to similar ideas to the ones presented in [29] 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. 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. We quickly review their approach.

The problem of determining the partitioning of minimum cost can be modeled as the problem of finding a 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 [29]. 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 $\epsilon_1 = \epsilon_2 = \frac{\epsilon}{3}$ [21].

Despite the *theoretical* linear-time complexity for a constant ϵ, the main drawback of the algorithm lies in the high constant factor. For example, even by setting $\epsilon = 0.03$ we obtain a hidden constant of $\log_{1+0.03} \frac{1}{0.03} \simeq 118.63$, 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 Optimal Partitioning in Linear Time: Fast and Exact

The interesting research question we now pose 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 an integer solely depends on the value of such integer and not on the universe and size of the partition it belongs to, the problem of determining and optimal partition admits an *exact* 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 pseudocode.

3.1 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 commenting on Fig. 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 representation of $S[i,j)$ and E is the chosen point-wise encoder for sparse regions.

Examples of point-wise encoders are VByte [11], Elias' γ-δ [31] and Golomb [32]. Other encoders, such as Elias-Fano [33], [34], Binary Interpolative coding [20] and PForDelta [26] are not point-wise, since a different number of bits could be used to represent the *same* integer when belonging to partitions having different characteristics, namely different size and universe. To clarify what we mean, consider the following example 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 $\lceil \log_2(40/10) \rceil + 2 = 4$ 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 $\lceil \log_2(40-12-1)/5 \rceil + 2 = 5$ 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 of any partitioning.

The intuition. 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 better represented with E rather than B. 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 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 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 now be better represented with B, rather than with E. Fig. 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 effective 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 pseudocode 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,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2911288, IEEE Transactions on Knowledge and Data Engineering

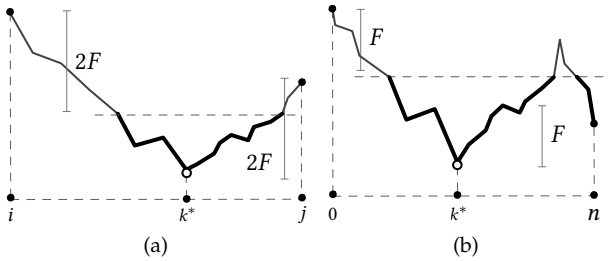


Fig. 2. In case (a), we should split $[i, j)$ in k^* because there the gain is the minimum among all points whose gain is below $2F$ from i *and* j; in case (b) we should *not* split the sequence because, although an increase in gain of F bits follows, we do not have a sufficiently high gain up to position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

in order to be beneficial, the difference in the cost of the two encoders must be larger than $2F$ bits. Again, refer to Fig. 2a for an example. Also, for illustrative purposes, in the above discussion we have assumed that the first partition is encoded with E: clearly, B could be better at the beginning but the algorithm will work in the very same way.

The algorithm. 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) Until the end of the sequence: 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) Encode the last partition.

In the above pseudocode, 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 Fig. 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 Technical discussion

Let S be a sorted integer sequence of size n. In order to describe the properties of our solution, we first need the following definitions.

Definition 1. Let $g : \mathbb{N} \cup \{0\} \to \mathbb{Z}$ be the *gain* function, defined as

$$g(i) = \begin{cases} 0 & i = 0 \\ \sum_{k=0}^{i-1} [\mathsf{E}(S[k]) - \mathsf{B}(S[k])] & 0 < i \leq n \end{cases}. \quad (1)$$

Definition 2. Given the interval $[i, j)$ with $0 \leq i < j \leq n$, the position k^* is *dominating* $[i, j)$ for the encoder E, if

$$k^* = \arg\min_{i<k\leq j} g(k) \text{ such that } g(i) - g(k^*) > T, \quad (2)$$

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

$$g(j) - g(k^*) > 2F, \text{ or} \quad (3)$$
$$g(z) - g(k^*) > F, \text{ for all } z \geq j. \quad (4)$$

Notice that the dominating point could not exist for any interval $[i, j)$, 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 2.

The definition of dominating point explains that we can *always improve* the cost of representation of $S[i, j)$ by splitting $[i, j)$ in the dominating point if it exists, otherwise we should *not* split $[i, j)$. It is easy to see that the point dominating $[i, j)$ 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 point dominating $[i, j)$ among the ones whose gain is at least T bits less than $g(i)$. 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 Fig. 2a and suppose we are encoding with B before position i and after k^*. If we compress with E the partition $S[i, k^*)$, we are switching encoder twice, thus the gain in k^* must be at least $2F$ bits less than $g(i)$ to be able of amortizing the cost for two switches. In Fig. 2b, instead, we have no partition before position 0, thus we strive to amortize the cost for a single switch.

Our strategy consists in splitting the sequence in the dominating points. More precisely, the solution $\mathcal{P} = [p_1, \ldots, p_m]$ output by this strategy can be described by the following recursive equation.

$$p_i = \begin{cases} 0 & i = 0 \\ n & i = m \\ \text{dominating } [p_{i-1}, p_{i+1}) & \text{otherwise} \end{cases} \quad (5)$$

In other words, any position 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 k^* for E. It means that we have seen an increase in gain of $2F$ after k^*, therefore: either the gain will then decrease sufficiently to find a dominating point for B, or the gain will never does so, thus, a dominating point after k^* 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2911288, IEEE Transactions on Knowledge and Data Engineering

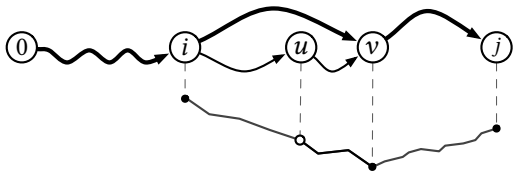


Fig. 3. Path of minimum cost till position j (thick black line) and its representation in terms of gain function; u is the point dominating $[i, j)$.

starting with a dominating point for B, since $\mathcal{P}$ alternates the compressors). Symmetrically, the same holds for B. As already pointed out, the only exception is made for the last partition, because the position n cannot be dominating 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 the last partition.

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 from each other and, therefore, strictly increasing. If no dominating points exist, then $\mathcal{P}$ will only contain position n (one-past the end): 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 2.1, 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 Fig. 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 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., $\mathsf{E}(i, v) \geq \mathsf{E}(i, u) + \mathsf{B}(u, v) + F$.

```
1  optimal_partitioning(S[0, n))
2      P = ∅
3      T = F
4      i = j = g = 0
5      min = max = 0
6      for k = 0; k < n; k = k + 1
7          g = g + E(S[k]) − B(S[k])
8          if g is non-decreasing
9              if g > max
10                 max = g, i = k + 1
11             if min < −T and min − g < −2F
12                 update(min, max, j, i)
13         else
14             if g < min
15                 min = g, j = k + 1
16             if max > T and max − g > 2F
17                 update(max, min, i, j)
18     close()
19     return P
```

Fig. 4. The **optimal_partitioning** algorithm.

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. □

```
1  update(g0, g1, p0, p1)
2      append p0 to P
3      T = 2F
4      p1 = k + 1
5      g = g − g0
6      g0 = 0
7      g1 = g
```

Fig. 5. The **update** algorithm.

```
1  close()
2      if max > F and max − g > F
3          update(max, min, i, j)
4      if min < −F and min − g < −F
5          update(min, max, j, i)
6      if g > 0
7          update(max, min, n, j)
8      else
9          update(min, max, n, i)
```

Fig. 6. The **close** algorithm.

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 5. We argue that the function optimal_partitioning coded in Algorithm 4 does the job.

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

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, $max = g(i)$ and $min = g(j)$ according to Definition 1, with g being the gain at step k.

Lemma 2. Algorithm 4 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 5. 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 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 dominating $[\ell, k)$, i.e., whether $S[i]$ and $S[j]$ satisfy Definition 2. 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 2. At the beginning of the algorithm, the current interval starts at $i = 0$ and $T = F$, therefore $g(0) = 0$ in Equation 2 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 3. 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 2. If so, we can execute the update code, show in Algorithm 5, which adds j to $\mathcal{P}$ and sets T to $2F$ according to Definition 2. 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 h 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 $k+1$ satisfies Equation 2 for encoder B. Therefore the $(h + 1)$-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 6. Lines 2-3 (4-5) check Condition 4: 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 6-9, 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 6-9. □

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*.

TABLE 1
Basic statistics for the tested collections.

	Gov2	ClueWeb09	CCNews
Documents	24,622,347	50,131,015	43,530,315
Terms	35,636,425	92,094,694	43,844,574
Postings	5,742,630,292	15,857,983,641	20,150,335,440

4 Experimental evaluation

The aim of this section is the one of measuring the space improvement, indexing time and query processing speed of indexes that are optimally-partitioned by the algorithm described in Section 3.

Datasets. We perform our experiments on the following 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.
- CCNews is an English subset of the freely available news from CommonCrawl[1], consisting of news articles in the period 09/01/16 to 30/03/18.

Identifiers were assigned to documents (docIDs) by following the lexicographic order of their URLs [35].

Experimental setting and methodology. All the experiments were run on a machine with 4 Intel i7-4790K CPUs (8 threads) clocked at 4.00 GHz and with 32 GB 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. We based our implementation on the popular ds2i[2] project. The source code, available at https://github.com/jermp/opt_vbyte to favour further research and reproducibility of results, was compiled with gcc 7.2.0 using the highest optimization setting (compilation flags `-march=native` and `-O3`).

1. http://commoncrawl.org/2016/10/news-dataset-available
2. https://github.com/ot/ds2i

TABLE 2
The performance of the Variable-Byte family.

	Gov2				ClueWeb09				CCNews			
	docs [bpi]	freqs [bpi]	building [min]	AND query [ms]	docs [bpi]	freqs [bpi]	building [min]	AND query [ms]	docs [bpi]	freqs [bpi]	building [min]	AND query [ms]
Varint-GB	11.15	9.77	10.60	0.88	11.43	9.80	46.50	5.32	11.12	10.01	58.40	7.38
Varint-G8IU	10.43	9.00	18.00	0.84	10.84	8.99	65.80	5.10	10.23	8.93	60.60	6.93
Masked-VByte	9.53	8.02	10.50	0.90	9.91	8.01	45.50	5.52	9.42	8.00	60.40	7.06
Stream-VByte	11.15	9.77	10.60	0.86	11.43	9.80	46.50	5.30	11.12	10.01	58.40	7.06

To test the building time of the indexes we measure the time needed to perform the whole end-to-end process, that is: (1) fetch the inverted 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.

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

Organization of the experiments. Since we adopt VByte as example point-wise encoder, the next section compares the performance of all the encoders in the VByte family in order to choose 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 compare our proposal with many other inverted index representations.

4.1 The Variable-Byte family

Several VByte variants have been proposed in the literature, each with a different stream organization. We now discuss them and inspect their performance.

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 [13] and reduces the probability of a branch misprediction which, in turn, leads to higher instruction throughput. As already mentioned in Section 1, 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 [16] 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 [36] 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 lookup tables and a shuffle instruction, the data bytes are permuted to obtain the original integers. Stream-VByte [37], 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.

The performance. To help us in deciding which VByte encoder to choose 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 both versions as Varint-GB. 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 Fig. 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 $\langle 132, 233, 246, 178\rangle$. 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 for Varint-GB and Varint-G8IU is larger than the one of VByte: it is $16 \div 18\%$ larger for Varint-GB; 10% larger 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 the tested datasets.

The speed of the encoders is actually very similar for

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2911288, IEEE Transactions on Knowledge and Data Engineering

TABLE 3
Space in average number of bits (bpi) per document (docs) and frequency (freqs).

	Gov2		ClueWeb09		CCNews	
	docs [bpi]	freqs [bpi]	docs [bpi]	freqs [bpi]	docs [bpi]	freqs [bpi]
VByte	9.53 (+95.7%)	8.02 (+163.9%)	9.90 (+51.5%)	8.01 (+222.4%)	9.42 (+37.4%)	8.00 (+234.8%)
VByte unif.	5.41 (+11.1%)	3.31 (+8.9%)	7.37 (+12.7%)	2.69 (+8.5%)	7.27 (+6.1%)	2.55 (+6.5%)
VByte ϵ-opt.	4.93 (+1.2%)	3.05 (+0.5%)	6.66 (+1.8%)	2.50 (+0.7%)	6.92 (+1.0%)	2.41 (+1.0%)
VByte opt.	4.87	3.04	6.54	2.48	6.85	2.39

all alternatives. We used the TREC-05 querylog to compute boolean conjunctions. 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 slower (by up to 40% on Gov2 and ClueWeb09).

In conclusion, for the reasons discussed above, i.e., better space occupancy, better index building time and competitive speed, we adopt the original VByte stream organization and the Masked-VByte algorithm by Plaisance, Kurz and Lemire [36] to perform sequential decoding.

4.2 Optimized Variable-Byte indexes

In this section, 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 uniform and the ϵ-optimal based on dynamic programming (see Subsection 2.1).

As a high-level overview, the result of the comparison shows that our optimally-partitioned VByte indexes are $2\times$ smaller than the original, un-partitioned, counterparts; can be built $2\times$ faster without resorting on dynamic programming and offer the strongest guarantee, i.e., an exact solution rather than an ϵ-approximation; despite the significant space savings, these 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 37% up to 235%, with a net factor of $2\times$ 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%, 26% and 23% better on the docs sequences for Gov2, ClueWeb09 and CCNews respectively; 59%, 66% and 70% better on the freqs 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 both docs and freqs becomes sensibly less than 8.

We remark that the ϵ-optimal algorithm based on dynamic programming was originally proposed for Elias-Fano [21], 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 values as used in the experiments of the original paper [21], 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 only slightly better (by $1 \div 1.8\%$), further confirming the analysis in [21]. Compared to uniform, the optimal partitioning pays off: indeed it produces a further saving of 10% on average, thus confirming the need for an optimization algorithm.

TABLE 4
Index building timings in minutes.

	Gov2	ClueWeb09	CCNews
VByte	10.1 (−4%)	43.3 (+52%)	60.4 (+70%)
VByte unif.	11.3 (+8%)	29.3 (+3%)	34.9 (−2%)
VByte ϵ-opt.	26.7 (+154%)	72.3 (+154%)	59.8 (+68%)
VByte opt.	10.5	28.5	35.5

TABLE 5
Timings for AND queries in milliseconds.

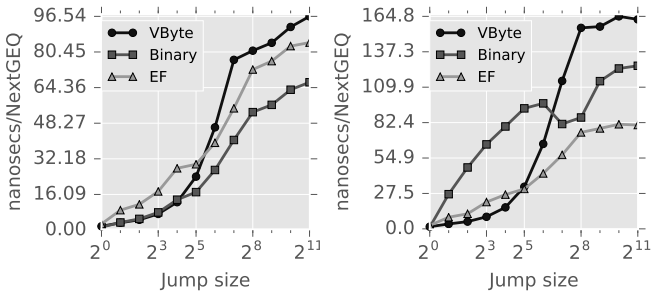
		Gov2	ClueWeb09	CCNews
TREC-05	VByte	0.90 (+1%)	5.56 (−3%)	7.06 (+10%)
	VByte unif.	0.94 (+5%)	5.90 (+3%)	7.20 (+13%)
	VByte ϵ-opt.	0.92 (+3%)	5.89 (+3%)	6.52 (+2%)
	VByte opt.	0.89	5.70	6.39
TREC-06	VByte	2.12 (+0%)	8.35 (−7%)	9.36 (+12%)
	VByte unif.	2.22 (+5%)	9.02 (+1%)	9.58 (+14%)
	VByte ϵ-opt.	2.24 (+6%)	9.17 (+2%)	8.56 (+2%)
	VByte opt.	2.12	8.96	8.38

Index building time. Although the un-partitioned variant would be the fastest to build in internal memory because the inverted 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 and CCNews, resulting in an end-to-end overhead of $50 \div 70\%$. Because of this, also observe that there is no appreciable difference between the indexing time of the simple uniform strategy and the optimal one. 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 Subsection 2.1. The optimal solutions has instead low constant factors and, as a result, is faster than the dynamic programming approach by more than $2.6\times$ on average on both Gov2 and ClueWeb09; by $1.7\times$ on CCNews.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2911288, IEEE Transactions on Knowledge and Data Engineering

TABLE 6
Space in average number of bits (bpi) per document (docs) and frequency (freqs).

	Gov2		ClueWeb09		CCNews	
	docs [bpi]	freqs [bpi]	docs [bpi]	freqs [bpi]	docs [bpi]	freqs [bpi]
PEF ϵ-opt.	4.10 (−15.7%)	2.38 (−21.8%)	5.85 (−10.6%)	2.20 (−11.6%)	5.84 (−14.8%)	2.18 (−8.9%)
OptPFD	4.48 (−8.0%)	2.38 (−21.8%)	6.18 (−5.4%)	2.41 (−2.9%)	6.41 (−6.5%)	2.53 (+5.9%)
BIC	3.80 (−22.0%)	2.14 (−29.5%)	5.15 (−21.3%)	1.87 (−24.8%)	5.37 (−21.7%)	1.98 (−17.3%)
ANS	3.96 (−18.7%)	1.85 (−39.0%)	5.36 (−18.0%)	1.94 (−21.9%)	5.76 (−16.0%)	2.01 (−15.8%)
QMX	6.00 (+23.3%)	3.37 (+10.8%)	8.01 (+22.6%)	3.75 (+51.2%)	7.31 (+6.6%)	3.72 (+55.5%)
VByte opt.	4.87	3.04	6.54	2.48	6.85	2.39



(a) Dense (b) Sparse

Fig. 7. Average nanoseconds spent per NextGEQ query for (a) Dense and (b) Sparse sequences of one million integers.

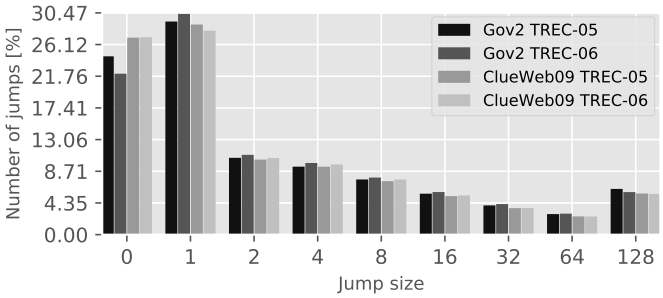


Fig. 8. When the difference between two consecutively accessed positions is d, NextGEQ is said to make a jump of size d. The distribution of the jump sizes is divided into buckets of exponential size: all sizes between 2^{d-1} and 2^d belong to bucket d. The plot shows the jumps distribution, in percentage, for the querylogs used in the experiments, when performing AND queries.

Index speed. Table 5 illustrates the results. The striking result of the experiment is that, despite the significant space reduction ($2\times$ improvement, see Table 3), the partitioned indexes are as fast as the un-partitioned ones on all datasets.

Therefore, it is important to provide a careful explanation of such result. The answer is provided by understanding the plots in Fig. 7, along with the ones in Fig. 1 and 8, that we generate for Gov2 and ClueWeb09 since we obtained even better timing results for CCNews (see Table 5). In particular, Fig. 7 illustrates the average nanoseconds spent per NextGEQ (Next Greater-than or EQual-to) query by VByte, the binary vector representation and Elias-Fano (EF). The NextGEQ$_t(x)$ query returns the *smallest* integer $z \geq x$ from the inverted list of the term t and it is the core operation needed to perform fast intersection (see [10] and references therein). The timings reported in Fig. 7 are relative to a sequence of one million integers and with an average gap between the integers of: (a) 2.5, as a *dense* case and (b) 1850 as a *sparse* case. These values mimic the ones for the Gov2 dataset, that are 2.13 and 1852 respectively. The ones for the ClueWeb09 dataset are 2.14 and 963, thus the plots have a similar shape.

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 Fig. 8 indicates that, whenever executing AND queries, the number of jumps of size less than 16 accounts for $\approx$90% of the jumps performed by NextGEQ. Furthermore, the distribution plotted in Fig. 1 tells us that the majority of blocks are actually encoded with their characteristic bitvector, 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:

- the ϵ-optimal partitioned Elias-Fano method (PEF) by Ottaviano and Venturini [21];
- the Binary Interpolative coding (BIC) by Moffat and Stuiver [20];
- the optimized PForDelta (OptPDF) by Yan *et al.* [26];
- the ANS-based index by Moffat and Petri [25], [38];
- the QMX mechanism by Trotman [27].

TABLE 7
Index building timings in minutes.

	Gov2	ClueWeb09	CCNews
PEF ϵ-opt.	41.3 (+293%)	125.5 (+340%)	85.2 (+140%)
OptPFD	8.2 (−22%)	25.8 (−9%)	36.7 (+3%)
BIC	7.0 (−33%)	20.5 (−28%)	28.3 (−20%)
ANS	12.6 (+20%)	35.7 (+25%)	55.1 (+55%)
QMX	7.0 (−33%)	18.0 (−37%)	31.1 (−12%)
VByte opt.	10.5	28.5	35.5

TABLE 8
Timings for AND queries in milliseconds.

		Gov2	ClueWeb09	CCNews
TREC-05	PEF ϵ-opt.	0.98 (+10%)	5.87 (+3%)	8.64 (+35%)
	OptPFD	1.28 (+43%)	8.04 (+41%)	8.92 (+40%)
	BIC	4.14 (+364%)	25.42 (+346%)	63.50 (+894%)
	ANS	4.21 (+372%)	25.98 (+356%)	27.49 (+330%)
	QMX	0.88 (−1%)	5.30 (−7%)	7.09 (+11%)
	VByte opt.	0.89	5.70	6.39
TREC-06	PEF ϵ-opt.	2.19 (+4%)	9.59 (+7%)	11.77 (+40%)
	OptPFD	3.00 (+42%)	11.95 (+33%)	11.73 (+40%)
	BIC	9.93 (+369%)	37.87 (+322%)	81.52 (+873%)
	ANS	9.48 (+348%)	38.07 (+325%)	35.48 (+323%)
	QMX	2.11 (−1%)	8.07 (−10%)	9.44 (+13%)
	VByte opt.	2.12	8.96	8.38

We point the reader again to the survey in [10] for an overview of such encoders. 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.

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 used to be before*. In fact, 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; 117% larger than PEF and 137% larger than BIC on CCNews. Now, thanks to our optimization strategy, this gap is reduced to 20% on average. In particular, notice that it is less than 11% larger than PEF on the docs sequences of ClueWeb09, while it is generally less effective on the freqs sequences. This is because the within-document frequencies are made up of integers smaller than docIDs. Very similar considerations hold for the other alternatives, such as OptPFD and ANS. In particular, we notice that on the ClueWeb09 dataset, the difference between VByte optimal and OptPDF is very small (only 4% overall); BIC is (as usual) generally better than other methods on the both docs and freqs; the byte-aligned QMX is, instead, significantly larger, by $19 \div 31\%$.

We now consider the time needed to build the indexes. Refer to Table 7. As already noted in the previous subsection, the dynamic programming approach used for PEF imposes a severe penalty with respect to VByte optimal of $4\times$ 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 33% on average, because of the two-pass process of first collecting symbol occurrence counts and, then, encoding [25], 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 less indexing time (33% faster on average on Gov2 and ClueWeb09, but only 16% more on the largest CCNews dataset).

Index speed. Table 8 shows the query processing speed of the indexes. Compared to PEF, the results are indeed very similar to the ones obtained by Ottaviano and Venturini [21], 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 Fig. 7b. As we can see, for all the jump sizes less than 32, VByte is $2\times$ faster than Elias-Fano, while this advantage vanishes for the longer jumps thanks to the powerful skipping abilities of Elias-Fano [10], [21], [40]. 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 Fig. 8.

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 ($4\times$ faster on average) and only a slight penalty with respect to QMX (by $7 \div 10\%$) on ClueWeb09. On the largest CCNews dataset, our proposal is consistently the fastest approach.

5 Conclusions

We have presented an optimization algorithm for point-wise encoders that splits a sorted integer sequence into variable-sized partitions to improve its compression and has a linear-time and constant-space complexity. We have also proved that the algorithm is *optimal*, i.e., it finds the partitioning that minimizes the space of the representation. For point-wise encoders, this is sensibly better than 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 have exhibited a $2\times$-better compression ratio and build optimally-partitioned indexes $2\times$ faster than the linear-time dynamic programming approach. Despite the significant space savings, the partitioned representation does not introduce penalties at query processing time compared to the un-partitioned case.

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 may not pay off, because space improved by only 4.5% on the tested datasets.

REFERENCES

[1] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys*, vol. 38, no. 2, pp. 1–56, 2006.

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

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

[4] 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)*, vol. 6, no. 11, 2013, pp. 1150–1161.

[5] 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)*. IEEE, 2012, pp. 1360–1369.

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

[7] 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*. ACM, 2007, pp. 773–784.

[8] 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*. ACM, 2002, pp. 310–321.

[9] 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*. ACM, 2011, pp. 25–36.

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

[11] L. H. Thiel and H. Heaps, "Program design for retrospective searches on large data bases," *Information Storage and Retrieval*, vol. 8, no. 1, pp. 1–20, 1972.

[12] H. E. Williams and J. Zobel, "Compressing integers for fast file access," *The Computer Journal*, vol. 42, no. 3, pp. 193–201, 1999.

[13] 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.

[14] "Protocol Buffers - Google's data interchange format," https://github.com/google/protobuf, accessed on 15-04-2018.

[15] 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*, vol. 2, no. 2, pp. 1462–1473, 2009.

[16] 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)*, 2011, pp. 317–326.

[17] "RediSearch," https://github.com/RedisLabsModules/RediSearch/blob/master/docs/DESIGN.md, accessed on 15-04-2018.

[18] "UpscaleDB," https://upscaledb.com/about01.html#compression, accessed on 15-04-2018.

[19] "Dropbox Techblog," https://blogs.dropbox.com/tech/2016/09/improving-the-performance-of-full-text-search/, accessed on 15-04-2018.

[20] A. Moffat and L. Stuiver, "Binary interpolative coding for effective index compression," *Information Retrieval Journal*, vol. 3, no. 1, pp. 25–47, 2000.

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

[22] 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*, vol. 5, no. 3, pp. 197–280, 2013.

[23] 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)*, 2015, pp. 47–56.

[24] G. E. Pibiri and R. Venturini, "Clustered Elias-Fano indexes," *ACM Transactions on Information Systems*, vol. 36, no. 1, pp. 1–33, 2017.

[25] 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*, 2018, pp. 405–413.

[26] 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)*, 2009, pp. 401–410.

[27] A. Trotman, "Compression, simd, and postings lists," in *Proceedings of the 2014 Australasian Document Computing Symposium*. ACM, 2014, p. 50.

[28] 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)*, 2010, pp. 1219–1228.

[29] P. Ferragina, I. Nitto, and R. Venturini, "On optimally partitioning a text to improve its compression," *Algorithmica*, vol. 61, no. 1, pp. 51–74, 2011.

[30] A. Buchsbaum, G. Fowler, and R. Giancarlo, "Improving table compression with combinatorial optimization," *Journal of the ACM*, vol. 50, no. 6, pp. 825–851, 2003.

[31] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 194–203, 1975.

[32] S. Golomb, "Run-length encodings," *IEEE Transactions on Information Theory*, vol. 12, no. 3, pp. 399–401, 1966.

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

[34] P. Elias, "Efficient storage and retrieval by content and address of static files," *Journal of the ACM*, vol. 21, no. 2, pp. 246–260, 1974.

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

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

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

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

[39] J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson, "MILC: inverted list compression in memory," *Proceedings of the VLDB Endowment*, vol. 10, no. 8, pp. 853–864, 2017.

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

Giulio Ermanno Pibiri (http://pages.di.unipi.it/pibiri) received his Bachelor Degree in Computer Engineering from the University of Florence in 2012; Master Degree and Ph.D. in Computer Science from the University of Pisa in 2015 and 2019 respectively. His research interests involve data compression algorithms for indexing massive datasets, data structures and information retrieval with focus on efficiency.

Rossano Venturini (http://pages.di.unipi.it/rossano) is Associate Professor of Computer Science at the University of Pisa. He received his Ph.D. from the University of Pisa in 2010. His research interests are mainly focused on the design and the analysis of algorithms and data structures with focus on indexing and searching large textual collections. He won two Best Paper Awards at ACM SIGIR in 2014 and 2015.