Inverted Index Compression

Giulio Ermanno Pibiri[1] and Rossano Venturini[1]

1 **Department of Computer Science, University of Pisa, Italy**
giulio.pibiri@di.unipi.it **and** rossano.venturini@unipi.it

Abstract

The data structure at the core of nowadays large-scale search engines, social networks and storage architectures is the *inverted index*, which can be regarded as being a collection of sorted integer sequences called *inverted lists*. Because of the many documents indexed by search engines and stringent performance requirements dictated by the heavy load of user queries, the inverted lists often store several million (even billion) of integers and must be searched efficiently.
In this scenario, *compressing* the inverted lists of the index appears as a *mandatory* design phase since it can introduce a twofold advantage over a non-compressed representation: feed faster memory levels with more data in order to speed up the query processing algorithms and reduce the number of storage machines needed to host the whole index.
The scope of the chapter is the one of surveying the most important encoding algorithms developed for efficient inverted index compression.

1 Introduction

The data structure at the core of nowadays large-scale search engines, social networks and storage architectures is the *inverted index*. Given a collection of documents , consider for each distinct term t appearing in the collection the integer sequence ℓ_t, listing in sorted order all the identifiers of the documents (docIDs in the following) in which the term appears. The sequence ℓ_t is called the *inverted list* or *posting list* of the term t. The inverted index is the collection of all such lists.

This simple, yet powerful, data structure owes its popularity to the efficient resolution of *queries*, expressed as a set of terms $\{t_1, \ldots, t_k\}$ combined with a query operator. The simplest operators are boolean AND and OR such that, for example, the query $q = \mathsf{AND}(t_i, t_j)$ means that the index has to report all the docIDs of the documents containing term t_i *and* term t_j. This operation ultimately boils down to *intersecting* the two inverted lists ℓ_{t_i} and ℓ_{t_j}. We can easily generalize the above example to an arbitrary number of query terms, as well as with other query operators.

Efficient list intersection relies on the operation $\mathsf{NextGEQ}_t(x)$, which returns the integer $z \in \ell_t$ such that $z \geq x$. This primitive is used because it permits to *skip* over the lists to be intersected. Suppose we have to compute $\mathsf{AND}(t_i, t_j) = \ell_{t_i} \cap \ell_{t_j}$, where ℓ_{t_i} is smaller than ℓ_{t_j}. We search for the first docID x of ℓ_{t_i} in ℓ_{t_j} by means of $\mathsf{NextGEQ}_{t_j}(x)$: if the docID z returned by the search is equal to x then it is a member of the intersection and we can just repeat this search step for the next docID of ℓ_{t_i}; otherwise z gives us the candidate docID to be searched next, indeed allowing to skip the searches for all docIDs in between x and z. In fact, since we have that $z \geq x$, $\mathsf{NextGEQ}_{t_j}(y)$ will be equal to z also for *all* $x < y < z$, thus none of such docIDs can be a member of the intersection.

Because of the many documents indexed by search engines and stringent performance requirements dictated by the heavy load of user queries, the inverted lists often store several million (even billion) of integers and must be searched efficiently.

In this scenario, *compressing* the inverted lists of the index appears as a *mandatory* design phase since it can introduce a twofold advantage over a non-compressed representation: feed faster memory levels with more data in order to speed up the query processing algorithms and reduce the number of storage machines needed to host the whole index. This has the potential

of letting a query algorithm work in internal memory, which is faster than the external memory system by several orders of magnitude.

For this reason, the scope of the chapter is the one of surveying the most important encoding algorithms developed for efficient inverted index compression. Not surprisingly, a plethora of different, practical, solutions have been proposed in the literature for representing in compressed space sorted integer sequences, each with a different time/space trade-off.

Chapter organization and notation. The content of the chapter is organized in three main sections dealing with encoding algorithms for representing, respectively: single integers (Section 2); a sorted list of integers (Section 3); the whole inverted index (Section 4).

We fix here a couple of notation conventions that we will use in the chapter.

All logarithms are binary, i.e., $\log x = \log_2 x$, $x > 0$. Let $B(x)$ represent the binary representation of the integer x, and $U(x)$ its *unary* representation, that is a run of x zeroes plus a final one: $\texttt{0}^x\texttt{1}$. Given a binary string B, let $|B|$ represent its length in bits. Given two binary strings B_1 and B_2, let B_1B_2 be their concatenation. We indicate with $S(n, u)$ a sequence of n integers drawn from a universe of size u and with $S[i, j]$ the sub-sequence starting at position i and including endpoint $S[j]$.

2 Integer Compressors

The compressors we consider in this section are used to represent a single integer. The most classical solution is to assign each integer a *self-delimiting* (or *uniquely-decodable*) variable-length code, so that the whole integer sequence is the result of the juxtaposition of the codes of all its integers. Clearly, the aim of such encoders is to assign the smallest codeword as possible in order to minimize the number of bits used to represent the sequence.

In particular, since we are dealing with inverted lists that are monotonically increasing by construction, we can subtract from each element the previous one (the first integer is left as it is), making the sequence be formed by integers greater than zero known as *delta*-gaps (or just *d*-gaps). This popular *delta encoding* strategy helps in reducing the number of bits for the codes. Most of the literature assumes this sequence form and, as a result, compressing such sequences of *d*-gaps is a fundamental problem that has been studied for decades.

2.1 Elias' Gamma and Delta

The two codes we now describe have been introduced by Elias [8] in the '60. Given an integer $x > 0$, $\gamma(x) = \texttt{0}^{|B(x)|-1}B(x)$, where $|B(x)| = \lceil \log(x + 1) \rceil$. Therefore, $|\gamma(x)| = 2\lceil \log(x + 1) \rceil - 1$ bits. For example, $\gamma(11) = \texttt{0001011}$ because the binary representation of 11, `1011`, has length 4 bits, therefore we prefix it by 3 zeroes. Decoding $\gamma(x)$ is simple: first count the number of zeroes up to the one, say there are n of these, then read the following $n + 1$ bits and interpret them as x.

The key inefficiency of γ lies in the use of the unary code for the representation of $|B(x)| - 1$, which may become very large for big integers. The δ code replaces the unary part $\texttt{0}^{|B(x)|-1}$ with $\gamma(|B(x)|)$, i.e., $\delta(x) = \gamma(|B(x)|)B(x)$. Notice that, since we are representing with γ the quantity $|B(x)| = \lceil \log(x + 1) \rceil$ which is guaranteed to be greater than zero, δ can represent value zero too. The number of bits required by $\delta(x)$ is $|\gamma(|B(x)|)| + |B(x)|$, which is $2\lceil \log(\lceil |B(x)| + 1 \rceil) \rceil + \lceil \log(x + 1) \rceil - 1$.

As an example, consider $\delta(113) = \texttt{00110111001}$. The last part, `111001`, is $B(113)$ whose length is 6. Therefore we prefix $B(113)$ by $\gamma(6) = \texttt{00110}$, which is the first part of the encoding.

The decoding of δ codes follows automatically from the one of γ codes.

2.2 Golomb-Rice

Rice [21] introduced a parameter code that can better compress the integers in a sequence if these are highly concentrated around some value. This code is actually a special case of the Golomb code [11], hence its name.

The Rice code of x with parameter k, $R_k(x)$, consists in two pieces: the *quotient* $q = \lfloor \frac{x-1}{2^k} \rfloor$ and the *remainder* $r = x - q \times 2^k - 1$. The quotient is encoded in unary, i.e., with $U(q)$, whereas the remainder is written in binary with k bits. Therefore, $|R_k(x)| = q + k + 1$ bits. Clearly, the closer 2^k is to the value of x the smaller the value of q: this implies a better compression and faster decoding speed. As an example, consider $R_5(113) = \texttt{000110000}$. Because $\lfloor \frac{112}{2^5} \rfloor = 3$, we first write $U(3) = \texttt{0}^3\texttt{1}$. Then we write in binary the remainder $113 - 3 \times 2^5 - 1 = 16$, using 5 bits. If we would have adopted $k = 6$, then $R_6(113) = \texttt{01110000}$. Notice that we are saving one bit with respect to $R_5(113)$ since $2^6 = 64$ is closer to 113 than $2^5 = 32$. In fact, the quotient in this case is 1 rather than 3.

Given the parameter k and the constant 2^k that is computed ahead, decoding Rice codes is simple too: count the number of zeroes up to the one, say there are q of these, then multiply 2^k by q and finally add the remainder, by reading the next k bits. Finally, add one to the computed result.

2.3 Variable-Byte

The codes described in the previous subsections are also called *bit-aligned* as they do not represent an integer using a multiple of a fixed number of bits, e.g., a byte. The decoding speed can be slowed down by the many operations needed to decode a single integer. This is a reason for preferring *byte-aligned* or even word-aligned codes when decoding speed is the main concern.

Variable-Byte (VByte) [22] is the most popular and simplest byte-aligned code: the binary representation of a non-negative integer is split 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. The main advantage of VByte codes is decoding speed: we just need to read one byte at a time until we found a value smaller than 128. Conversely, the number of bits to encode an integer cannot be less than 8, thus VByte is only suitable for large numbers and its compression ratio may not be competitive with the one of bit-aligned codes for small integers.

As an example, consider VByte$(65\,790) = \underline{1}0000100\underline{1}0000001\underline{0}1111110$, where we underline 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.

We now overview the various enhancements proposed in the literature for VByte to improve its (sequential) decoding speed.

In order to reduce the probability of a branch misprediction that leads to higher throughput and helps keeping the CPU pipeline fed with useful instructions, the control bits can be grouped together. For example, if we assume that the largest represented integer fits into four bytes (which is reasonable, given that we often represent d-gaps), we have to distinguish between only four different lengths, thus two bits are sufficient. In this way, groups of four integers require one control byte. This optimization was introduced in Google's Group-Varint [4], which is much faster than the original VByte format.

Working with byte-aligned codes also opens the possibility of exploiting the parallelism of SIMD instructions (single-instruction-multiple-data) of modern processors to further enhance decoding speed. This is the approach taken by the recent proposals VByte-G8UI [25], Masked-VByte [20] and Stream-VByte [14].

VByte-G8UI [25] uses a format similar to the one of Group-Varint: one control byte is used to describe a variable number of integers in a data segment of exactly eight bytes, therefore each group can contain between two and eight compressed integers. This data organization is suitable for SIMD instructions: an improvement in sequential decoding speed of $\times 3$ is observed with respect to scalar VByte.

Masked-VByte [20] works, instead, directly 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 decoded integers.

Stream-VByte [14] outperforms the previous SIMD-ized VByte-G8UI in decoding speed ($\times 2$ improvement). The idea is to separate the encoding of the control bits from the data bits by writing them into separate streams. This organization permits to decode multiple control bits simultaneously and, therefore, reduce branch mispredictions that can stop the CPU pipeline execution when decoding the data stream.

3 List Compressors

Differently from the compressors introduced in the previous section, the compressors we now consider encode a whole integer list, instead of representing each single integer separately. Such compressors often outperform the ones described before for sufficiently long inverted lists because they take advantage of the fact that inverted lists often contain *clusters* of close docIDs, e.g., runs of consecutive integers, that are far more compressible with respect to highly scattered sequences. The reason for the presence of such clusters is that the indexed documents themselves tend to be clustered, i.e., there are subsets of documents sharing the very same set of terms. As a meaningful example, consider all the Web pages belonging to a certain domain: since their topic is likely to be the same, they are also likely to share a lot of terms. Therefore, not surprisingly, list compressors greatly benefit from docID-reordering strategies that focus on re-assigning the docIDs in order to form larger clusters of docIDs.

An amazingly simple strategy, but very effective, for Web pages is to assign identifiers to documents according to the lexicographical order of their URLs [23]. A recent approach has instead adopted a recursive graph bisection algorithm to find a suitable re-ordering of docIDs [6]. In this model, the input graph is a bipartite graph in which one set of vertices represents the terms of the index and the other set represents the docIDs. A graph bisection identifies a permutation of the docIDs and, thus, the goal is the one of finding, at each step of recursion, the bisection of the graph which minimizes the size of the graph compressed using delta-encoding.

3.1 Block-based

A relatively simple approach to improve both compression ratio and decoding speed is to encode a *block* of contiguous integers. This line of work finds its origin in the so-called *frame-of-reference* [10].

Once the sequence has been partitioned into blocks (of fixed or variable length), then each block is encoded separately. An example of this approach is *binary packing* [2, 13], where blocks of fixed length are used, e.g., 128 integers. Given a block $S[i, j]$, we can simply represent its integers using a universe of size $\lceil \log(S[j] - S[i] + 1) \rceil$ bits by subtracting the lower bound $S[i]$ from their values. Plenty of variants of this simple approach has been proposed [24, 5, 13]. Recently, it has also been shown [18] that using more than one compressors to represent the blocks, rather than simply representing all blocks with the same compressor, can introduce significant improvements in query time within the same space constraints. In such hybrid approach, the main idea is to choose a more space-efficient compressor to represent rarely-accessed blocks and a more time-efficient compressor for very frequently-accessed blocks.

Among the simplest binary packing strategies, Simple-9 and Simple-16 [1, 28, 2] combine very good compression ratio and high decompression speed. The key idea is to try to pack as many integers as possible in a memory word (32 or 64 bits). As an example, Simple-9 uses 4 *header* bits and 28 *data* bits. The header bits provide information on how many elements are packed in the data segment using equally sized codewords. A header `0000` may correspond to 28 1-bit integers; `0001` to 14 2-bits integers; `0010` to 9 3-bits integers (1 bit unused), and so on. The four bits distinguish from 9 possible configurations. Similarly, Simple-16 has 16 possible configurations.

3.2 PForDelta

The biggest limitation of block-based strategies is their space-inefficiency whenever a block contains at least one large value, because this forces the compressor to use a universe of representation as large as this value. This has been the main motivation for the introduction of PForDelta (PFD) [31]. The 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 can be written in 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.

More precisely, two configurable parameters are chosen: a b value (base) and a universe of representation k, so that most of the values fall in the range $[b, b+2^k-1]$ and can be encoded with k bits each by shifting (delta-encoding) them in the range $[0, 2^k-1]$. To mark the presence of an exception, we also need a special *escape* symbol, thus we have $[0, 2^k-2]$ possible configurations for the integers in the block.

As an example, consider the sequence $[3, 4, 7, 21, 9, 12, 5, 16, 6, 2, 34]$. By using $b = 2$ and $k = 4$, we obtain the following transformed sequence:
$[1, 2, 5, *, 7, 10, 3, *, 4, 0, *][21, 16, 34]$, where we use the special symbol $*$ to represent an exception that is written in a separate sequence, reported to the right part of the example.

The variant OptPFD [28], 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 PFD [28, 13].

3.3 Elias-Fano

The encoding we are about to describe was independently proposed by Elias [7] and Fano [9], hence its name. Given a monotonically increasing sequence $S(n, u)$ of n positive integers drawn from a universe of size u (i.e., $S[i-1] \leq S[i]$, for any $1 \leq i < n$, with $S[n-1] < u$), we write each $S[i]$ in binary using $\lceil \log u \rceil$ bits. The binary representation of each integer is then split into two parts: a *low* part consisting in the right-most $\ell = \lceil \log \frac{u}{n} \rceil$ bits that we call *low bits* and a *high* part consisting in the remaining $\lceil \log u \rceil - \ell$ bits that we similarly call *high bits*. Let us call ℓ_i and h_i the values of low and high bits of $S[i]$ respectively. The Elias-Fano representation of S is given by the encoding of the high and low parts.

The array $L = [\ell_0, \ldots, \ell_{n-1}]$ is written explicitly in $n\lceil \log \frac{u}{n} \rceil$ bits and represents the encoding of the low parts. Concerning the high bits, we represent them in *negated unary*[1] using a bit vector of $n + 2^{\lfloor \log n \rfloor} \leq 2n$ bits as follows. We start from a `0`-valued bit vector H and set the bit in position h_i+i, for all $i \in [0, n)$. The effect is that now the k-th unary integer m of H indicates that m integers of S have high bits equal to k, $0 \leq k \leq \lfloor \log n \rfloor$. Finally the Elias-Fano representation of S is given by the concatenation of H and L and overall takes $\mathsf{EF}(S(n, u)) \leq n\lceil \log \frac{u}{n} \rceil + 2n$

[1] The negated unary representation of an integer x, is the bitwise NOT of its unary representation $U(x)$. An example: $U(5) =$ `000001` and $\mathsf{NOT}(U(5)) =$ `111110`.

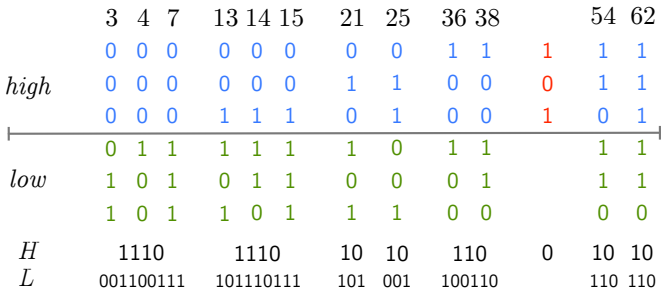

Figure 1 Elias-Fano encoding example for the sequence $[3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54, 62]$. Different colors mark the distinction between *high bits* (in blue), *low bits* (in green) and *missing high bits* (in red).

bits[2].

While we can opt for an arbitrary split into high and low parts, ranging from 0 to $\lceil \log u \rceil$, it can be shown that $\ell = \lceil \log \frac{u}{n} \rceil$ minimizes the overall space occupancy of the encoding [7]. Figure 1 shows an example of encoding for the sequence $S(12, 62) = [3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54, 62]$.

In Figure 1 the "missing high bits" embody a graphical representation of the fact that using $\lfloor \log n \rfloor$ bits to represent the high part of an integer, we have *at most* $2^{\lfloor \log n \rfloor}$ distinct high parts because not all of them could be present. In Figure 1, we have $\lfloor \log 12 \rfloor = 3$ and we can form up to 8 distinct high parts. Notice that, for example, no integer has high part equal to `101` which are, therefore, "missing" high bits.

As the information theoretic lower bound for a monotone sequence of n elements drawn from a universe of size u is $\lceil \log \binom{u+n}{n} \rceil \approx n \log \frac{u+n}{n} + n \log e$ bits, it can be shown that less than half a bit is wasted per element by the Elias-Fano space bound [7]. Since we set a bit for every $i \in [0, n)$ in H and each h_i is extracted in $O(1)$ time from $S[i]$, it follows that S gets encoded with Elias-Fano in $\Theta(n)$ time.

Despite its simplicity, it is possible to randomly access an integer from a sequence encoded with Elias-Fano *without* decompressing the whole sequence. We refer to this operation as $\mathsf{Access}(i)$, which returns $S[i]$. The operation is supported using an auxiliary data structure that is built on bit vector H, able to efficiently answer $\mathsf{Select}_1(i)$ queries, that return the position in H of the i-th 1 bit. This auxiliary data structure is *succinct* in the sense that it is negligibly small in asymptotic terms, compared to $\mathsf{EF}(S(n, u))$, requiring only $o(n)$ additional bits [16, 26].

Using the Select_1 primitive, it is possible to implement $\mathsf{Access}(i)$, which returns $S[i]$ for any $i \in [0, n)$, in $O(1)$. We basically have to re-link together the high and low bits of an integer, previously split up during the encoding phase. The low bits ℓ_i are trivial to retrieve as we need to read the range of bits $L[i\ell, (i+1)\ell)$. The retrieval of the high bits deserve, instead, a bit more care. Since we write in negated unary how many integers share the same high part, we have a bit set for every integer of S and a zero for every distinct high part. Therefore, to retrieve the high bits of the i-th integer, we need to know how many zeros are present in $H[0, \mathsf{Select}_1(i))$. This quantity is evaluated on H in $O(1)$ as $\mathsf{Rank}_0(\mathsf{Select}_1(i)) = \mathsf{Select}_1(i) - i$ (notice that the succinct rank/select data structure does not have to support Rank). Finally, linking the high and low bits is as simple as: $\mathsf{Access}(i) = ((\mathsf{Select}_1(i) - i) \ll \ell) \mid \ell_i$, where $\ll$ is the left shift operator and $\mid$ is the bitwise OR.

The query $\mathsf{NextGEQ}(x)$ is supported in $O(1 + \log \frac{u}{n})$ time[3], as follows. Let h_x be the high

[2] Using the same choice of Elias for ceilings and floors, we arrive at the slightly different bound of at most $n\lfloor \log \frac{u}{n} \rfloor + 3n$ bits.

[3] We report the bound as $O(1 + \log \frac{u}{n})$, instead of $O(\log \frac{u}{n})$, to cope with the case $n = u$.

bits of x. Then for $h_x > 0$, $i = \mathsf{Select}_0(h_x) - h_x + 1$ indicates that there are i integers in S whose high bits are less than h_x. On the other hand, $j = \mathsf{Select}_0(h_x + 1) - h_x$ gives us the position at which the elements having high bits greater than h_x start. The corner case $h_x = 0$ is handled by setting $i = 0$. These two preliminary operations take $O(1)$. Now we have to conclude our search in the range $S[i, j]$, having *skipped* a potentially large range of elements that, otherwise, would have required to be compared with x. We finally determine the successor of x by binary searching in this range which may contain up to u/n integers. The time bound follows.

As an example, consider $\mathsf{NextGEQ}(30)$. Since $h_{30} = 3$, we have $i = \mathsf{Select}_0(3) - 3 + 1 = 7$ and $j = \mathsf{Select}_0(4) - 3 = 8$. Therefore we conclude our search in the range $S[7, 8]$ by returning $S[8] = 36$.

3.3.1 Partitioned Elias-Fano

One of the most relevant characteristics of Elias-Fano is that it only depends on two parameters, i.e., the length and universe of the sequence that poorly describe the sequence itself. As inverted lists often present groups of close identifiers, Elias-Fano fails to exploit such natural clusters. Partitioning the sequence into chunks to better exploit such regions of close docIDs is the key idea of the two-level Elias-Fano representation devised in [17].

The core idea is as follows. The sequence $S(n, u)$ is partitioned into n/b chunks, each of b integers. First level L is made up of the last elements of each chunk, i.e., $L = [S[b - 1], S[2b - 1], \ldots, S[n - 1]]$. This level is encoded with Elias-Fano. The second level is represented by the chunks themselves, which can be again encoded with Elias-Fano. The main reason for introducing this two-level representation, is that now the elements of the i-th chunk are encoded with a smaller universe, i.e., $L[i] - L[i - 1] - 1$, $i > 0$. This is a uniform-partition strategy that may be suboptimal, since we cannot expect clusters of docIDs be aligned to such boundaries. As the problem of choosing the best partition is posed, an algorithm based on dynamic programming is presented which, in $O(n)$ time, yields a partition whose cost (i.e., the space taken by the partitioned encoded sequence) is at most $(1 + \epsilon)$ away from the optimal one, for any $0 < \epsilon < 1$. To support variable-size partitions, another sequence E is maintained in the first level of the representation, which encodes with Elias-Fano the lengths of the chunks in the second level.

This inverted list organization introduces a level of indirection when resolving $\mathsf{NextGEQ}$ queries. However, this indirection only causes a very small time overhead compared to plain Elias-Fano on most of the queries [17]. On the other hand, partitioning the sequence greatly improves the space-efficiency of its Elias-Fano representation.

3.4 Binary Interpolative

Binary Interpolative coding (BIC) [15] is another approach that, like Elias-Fano, directly compresses a monotonically increasing integer sequence without a first delta encoding step. 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.

More precisely, consider the range $S[i, j]$. The encoding step writes the quantity $S[m] - low - m + i$ using $\lceil \log(hi - low - j + i) \rceil$ bits, where: $S[m]$ is the range middle element, i.e., the integer at position $m = (i + j)/2$; low and hi are respectively the lower bound and upper bound of the range $S[i, j]$, i.e., two quantities such that $low \leq S[i]$ and $hi \geq S[j]$. The algorithm proceeds recursively, by applying the same encoding step to both halves: $[i, m]$ and $[m + 1, j]$ by setting $hi = S[m] - 1$ for the left half and $low = S[m] + 1$ for the right half. At the beginning, the encoding algorithm starts with $i = 0$, $j = n - 1$, $low = S[0]$ and $hi = S[n - 1]$. These quantities

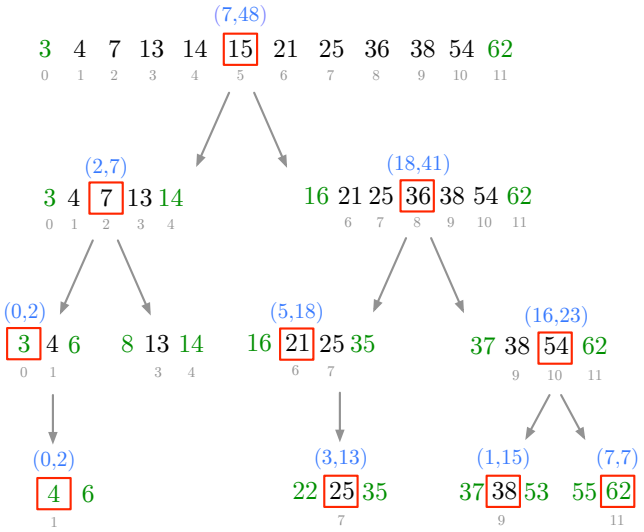

Figure 2 Binary Interpolative coding example for the sequence $[3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54, 62]$. In red boxes we highlight the middle element currently encoded: above each element we report a pair, in blue, where the first value indicates the element actually encoded and the second value the universe of representation. The green elements indicate, instead, current lower and upper bound of the ranges: notice that such elements can belong to the sequence itself.

must be also known at the beginning of the decoding phase. Apart from the initial lower and upper bound, all the other values of *low* and *hi* are computed on-the-fly by the algorithm.

Figure 2 shows an encoding example for the same sequence of Figure 1. We can interpret the encoding as a pre-order visit of the binary tree formed by the recursive calls the algorithm performs. The encoded elements in the example are, in order: $[7, 2, 0, 0, 18, 5, 3, 16, 1, 7]$. Moreover, notice that whenever the algorithm works on a range $S[i, j]$ of *consecutive integers*, such as the range $S[3, 4] = \{13, 14\}$ in the example, i.e., the ones for which the condition $S[j] - S[i] = j - i$ holds, it stops the recursion by emitting no bits at all. The condition is again detected at decoding time and the algorithm implicitly decodes the run $S[i], S[i] + 1, S[i] + 2, \ldots, S[j]$. This property makes BIC extremely space-efficient whenever the encoded sequences feature *clusters* of very close integers [27, 28, 24], which is the typical case for inverted lists. The key inefficiency of BIC is, however, the decoding speed which is highly affected by the recursive nature of the algorithm.

4 Index Compressors

This concluding section is devoted to recent approaches that encode many inverted lists together to obtain higher compression ratios. This is possible because the inverted index naturally presents some amount of redundancy, caused by the fact that many docIDs are shared between its lists. In fact, as already motivated at the beginning of Section 3, the identifiers of similar documents will be stored in the inverted lists of the terms they share.

Pibiri and Venturini [19] proposed a clustered index representation. The inverted lists are divided into clusters of similar lists, i.e., the ones sharing as many docIDs as possible. Then for each cluster, a *reference list* is synthesized with respect to which all lists in the cluster are encoded. In particular, the *intersection* between the cluster reference list and a cluster list is encoded more efficiently: all the docIDs are implicitly represented as the positions they occupy within the

reference list. This makes a big improvement for the cluster space, since each intersection can be re-written in a much smaller universe. Although *any* list compressor supporting NextGEQ can be used to represent the (rank-encoded) intersection and the residual segment of each list, the paper adopted partitioned Elias-Fano (see 3.3.1). With an extensive experimental analysis, the proposed clustered representation is shown to be superior to partitioned Elias-Fano and also Binary Interpolative coding for index space usage. By varying the size of the reference lists, different time/space trade-offs can be obtained.

Reducing the redundancy in highly repetitive collections has also been advocated in the work by Claude, Fariña, Martínez-Prieto, and Navarro [3]. The described approach first transforms the inverted lists into lists of d-gaps and then applies a general, i.e., *universal*, compression algorithm to the sequence formed by the concatenation of all the lists. The compressed representation is also enriched with pointers to mark where each individual list begins. The paper experiments with Re-Pair compression [12], VByte, LZMA (`http://www.7-zip.org/`) that is an improved version of the classic LZ77 [30] and Run-Length-Encoding. The experimental analysis reveals that significant reduction in space is possible on highly repetitive collections, e.g., versions of Wikipedia, with respect to encoders tailored for inverted indexes while only introducing moderate slowdowns.

An approach based on generating a *context-free grammar* from the inverted index has been proposed by Zhang, Tong, Huang, Liang, Li, Stones, Wang, and Liu [29]. The core idea is to identify common patterns, i.e., repeated sub-sequences of docIDs, and substitute them with symbols belonging to the generated grammar. Although the reorganized inverted lists and grammar can be suitable for different encoding schemes, the authors adopted OptPFD [28]. The experimental analysis indicates that good space reductions are possible compared to state-of-the-art encoding strategies with competitive query processing performance. By exploiting the fact that the identified common patterns can be placed directly in the final result set, the query processing performance can also be improved.

References

1 Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval Journal*, 8(1):151–166, 2005.

2 Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Software: Practice and Experience*, 40(2):131–147, 2010.

3 Francisco Claude, Antonio Fariña, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Universal indexes for highly repetitive document collections. *Information Systems*, 61:1–23, 2016.

4 Jeffrey 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.

5 Renaud Delbru, Stéphane Campinas, and Giovanni Tummarello. Searching web data: An entity retrieval and high-performance indexing model. *Journal of Web Semantics*, 10:33–58, 2012.

6 Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 1535–1544, 2016.

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

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

9 Robert Mario Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT*, 1971.

10 Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, pages 370–379, 1998.

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

12 N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Data Compression Conference (DCC)*, pages 296–305, 1999.

13 Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2013.

14 Daniel Lemire, Nathan Kurz, and Christoph Rupp. Stream-VByte: Faster byte-oriented integer compression. *Information Processing Letters*, 130:1–6, 2018.

15 Alistair Moffat and Lang Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval Journal*, 3(1):25–47, 2000.

16 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):1–79, 2007.

17 Giuseppe Ottaviano and Rossano Venturini. Partitioned elias-fano indexes. In *Proceedings of the 37th International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 273–282, 2014.

18 Giuseppe Ottaviano, Nicola Tonellotto, and Rossano 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.

19 Giulio Ermanno Pibiri and Rossano Venturini. Clustered Elias-Fano Indexes. *ACM Transactions on Information Systems*, 36(1):1–33, 2017. ISSN 1046-8188.

20 Jeff Plaisance, Nathan Kurz, and Daniel Lemire. Vectorized VByte Decoding. In *International Symposium on Web Algorithms (iSWAG)*, 2015.

21 Robert Rice and J. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communications*, 16(9):889–897, 1971.

22 David Salomon. *Variable-length Codes for Data Compression*. Springer, 2007.

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

24 Fabrizio Silvestri and Rossano 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.

25 Alexander Stepanov, Anil Gangolli, Daniel Rose, Ryan Ernst, and Paramjit Oberoi. Simd-based decoding of posting lists. In *Proceedings of the 20th International Conference on Information and Knowledge Management (CIKM)*, pages 317–326, 2011.

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

27 Ian Witten, Alistair Moffat, and Timothy Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 2nd edition, 1999.

28 Hao Yan, Shuai Ding, and Torsten 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.

29 Zhaohua Zhang, Jiancong Tong, Haibing Huang, Jin Liang, Tianlong Li, Rebecca J. Stones, Gang Wang, and Xiaoguang 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.

30 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

31 Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pages 59–70, 2006.