Inverted Index Compression

Giulio Ermanno Pibiri and Rossano Venturini
Department of Computer Science, University of Pisa, Pisa, Italy

Definition

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.

The scope of the entry is the one of surveying the most important encoding algorithms developed for efficient inverted index compression and fast retrieval.

Overview

The inverted index 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. For example, given an AND query, the index has to report *all* the docIDs of the documents containing the terms $\{t_1, \ldots, t_k\}$. This operation ultimately boils down to *intersecting* the inverted lists corresponding to the terms of the query set. 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.

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 millions (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.

Chapter Notation

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: $\mathtt{0}^x\mathtt{1}$. Given a binary string B, let $|B|$ represent its length in bits. Given two binary strings B_1 and B_2, let $B_1 B_2$ be their concatenation. We indicate

S. Sakr, A. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-63962-8_52-1

with $S(n,u)$ a sequence of n integers drawn from a universe of size u and with $S[i,j]$ the subsequence starting at position i and including endpoint $S[j]$.

Key Research Findings

Integer Compressors

The compressors we consider in this subsection 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 code word 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.

Elias' Gamma and Delta

The two codes we now describe have been introduced by Elias (1975) in the 1960s. Given an integer $x > 0$, $\gamma(x) = 0^{|B(x)|-1}B(x)$, where $|B(x)| = \lceil \log(x+1) \rceil$. Therefore, $|\gamma(x)| = 2\lceil \log(x+1) \rceil - 1$ bits. 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 $0^{|B(x)|-1}$ 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 $2\lceil \log(\lceil |B(x)| + 1 \rceil) \rceil + \lceil \log(x+1) \rceil - 1$.

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

Golomb-Rice

Rice and Plaunt (1971) 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 (Golomb 1966), 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.

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.

Variable-Byte

The codes described so far 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) (Salomon 2007) is the most popular and simplest byte-aligned code: the binary representation of a nonnegative 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 find 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.

Various enhancements have been proposed in the literature to improve the (sequential) decoding speed of VByte. This line of research focuses on finding a suitable format of control and data streams in order to reduce the probability of a branch misprediction that leads to higher instruction throughput and helps keeping the CPU pipeline fed with useful instructions (Dean 2009) and on exploiting the parallelism of SIMD instructions (single-instruction-multiple-data) of modern processors (Stepanov et al. 2011; Plaisance et al. 2015; Lemire et al. 2018).

List Compressors

Differently from the compressors introduced in the previous subsection, 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. Therefore, not surprisingly, list compressors greatly benefit from reordering strategies that focus on reassigning 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 (Silvestri 2007). A recent approach has instead adopted a recursive graph bisection algorithm to find a suitable reordering of docIDs (Dhulipala et al. 2016). 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. Since a graph bisection identifies a permutation of the docIDs, 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.

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 (FOR) (Goldstein et al. 1998).

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* (Anh and Moffat 2010; Lemire and Boytsov 2013), 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 (Silvestri and Venturini 2010; Delbru et al. 2012; Lemire and Boytsov 2013). Recently, it has also been shown (Ottaviano et al. 2015) that using more than one compressor 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.

Among the simplest binary packing strategies, Simple-9 and Simple-16 (Anh and Moffat 2005, 2010) 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 code words. The 4 header bits distinguish from 9 possible configurations. Similarly, Simple-16 has 16 possible configurations.

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 encode *all* values with the number of bits needed to represent this large value (universe of representation). This has been the main motivation for the introduction of PForDelta (PFD) (Zukowski et al. 2006). 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 base value b 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.

The variant OptPFD (Yan et al. 2009), 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 (Yan et al. 2009; Lemire and Boytsov 2013).

Elias-Fano

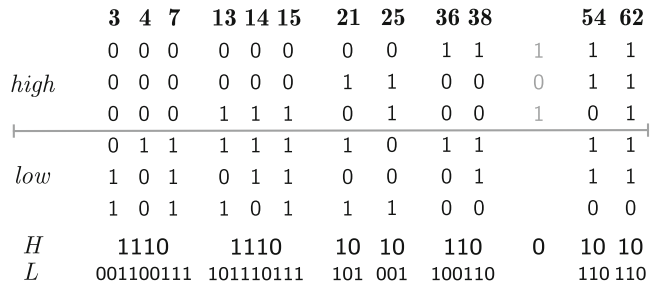
The encoding we are about to describe was independently proposed by Elias (1974) and Fano (1971), hence its name. Given a monotonically increasing sequence $S(n, u)$, i.e., $S[i-1] \leq S[i]$, for any $1 \leq i < n$, with $S[n-1] < u$, we write each $S[i]$ in binary using $\lceil \log u \rceil$ bits. The binary representation of each integer is then split into two parts: a *low* part consisting in the rightmost $\ell = \lceil \log \frac{u}{n} \rceil$ 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* 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 $i \in [0, n)$. Finally the Elias-Fano representation of S is given by the concatenation of H and L and overall takes $\mathsf{EF}(S(n, u)) \leq n\lceil \log \frac{u}{n} \rceil + 2n$ bits.

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 (Elias 1974). Figure 1 shows an example of encoding for the sequence $S(12, 62) = [3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54, 62]$.

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 (Navarro and Mäkinen 2007; Vigna 2013).

Using the Select_1 primitive, it is possible to implement $\mathsf{Access}(i)$ in $O(1)$ time. We basically have to relink 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 deserves, 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 $\mathsf{Rank}_0(\mathsf{Select}_1(i)) = \mathsf{Select}_1(i) - i$. 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



Inverted Index Compression, Fig. 1 An example of Elias-Fano encoding. We distinguish in light gray the *missing high bits*, i.e., the ones not belonging to the integers of the sequence among the possible $2^{\lfloor \log n \rfloor}$

$\ll$ is the left shift operator and | is the bitwise OR.

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

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. As inverted lists often present groups of close docIDs, Elias-Fano fails to exploit such natural clusters. Partitioning the sequence into chunks can better exploit such regions of close docIDs (Ottaviano and Venturini 2014).

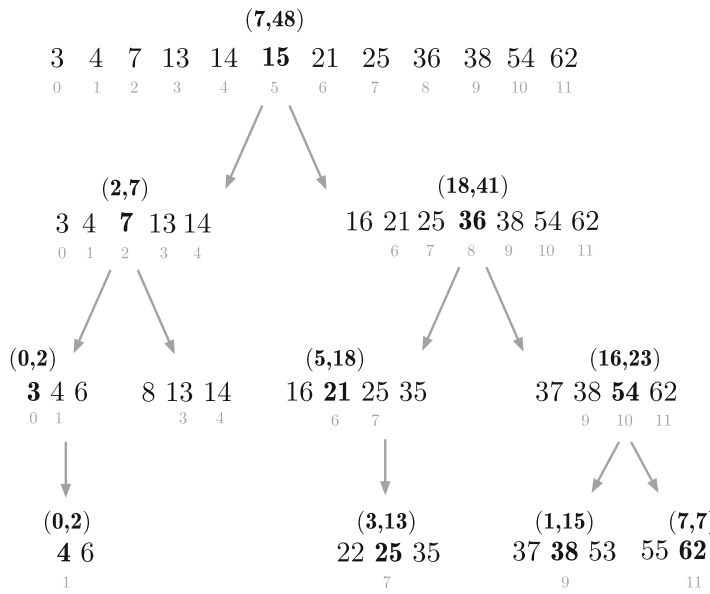
The core idea is as follows. The sequence $S(n, u)$ is partitioned into n/b chunks, each of b integers. First level L is made up of the last elements of each chunk, i.e., $L = [S[b-1], S[2b-1], \ldots, S[n-1]]$. This level is encoded with Elias-Fano. The second level is represented by the chunks themselves, 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., $L[i] - L[i-1] - 1, i > 0$. As the problem of choosing the best partition is posed, an algorithm based on dynamic programming can be used to find in $\mathrm{O}(\mathrm{n} \log_{1+\mathrm{eps}} 1/\mathrm{eps})$ time partition whose cost (i.e., the space taken by the partitioned encoded sequence) is at most $(1 + \varepsilon)$ away from the optimal one, for any $0 < \varepsilon < 1$ (Ottaviano and Venturini 2014).

This inverted list organization introduces a level of indirection when resolving NextGEQ queries. However, this indirection only causes a very small time overhead compared to plain Elias-Fano on most of the queries (Ottaviano and Venturini 2014). On the other hand, partitioning the sequence greatly improves the space efficiency of its Elias-Fano representation.

Binary Interpolative

Binary Interpolative coding (BIC) (Moffat and Stuiver 2000) 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 $m = (i + j)/2$, and low and hi are, respectively, the lower bound and upper bound of the range $S[i, j]$, i.e., two quantities



Inverted Index Compression, Fig. 2 An example of Binary Interpolative coding. We highlight in bold font the middle element currently encoded: above each element we report a pair where the first value indicates the element *actually* encoded and the second value the universe of representation

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 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 Fig. 1. We can interpret the encoding as a preorder 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 docIDs (Witten et al. 1999; Yan et al. 2009; Silvestri and Venturini 2010), 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. This is even more evident considering random access to individual integers that cannot be performed in constant time as it is the case for Elias-Fano (see 1).

Future Directions for Research

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 "List Compressors", the identifiers of similar documents will be stored in the inverted lists of the terms they share.

Pibiri and Venturini (2017) 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 rewritten 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 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 (2016). 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 (Larsson and Moffat 1999), VByte, and LZMA (http://www.7-zip.org/) that is an improved version of the classic LZ77 (Ziv and Lempel 1977) 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 (2016). The core idea is to identify common patterns, i.e., repeated subsequences 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 (Yan et al. 2009). 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.

Cross-References

- Compression and Indexing of Repetitive Textual Datasets
- Grammar-Based Compression
- (Web/Social) Graph Compression

References

Anh VN, Moffat A (2005) Inverted index compression using word-aligned binary codes. Inf Retr J 8(1):151–166

Anh VN, Moffat A (2010) Index compression using 64-bit words. Softw Pract Exp 40(2):131–147

Claude F, Fariña A, Martínez-Prieto MA, Navarro G (2016) Universal indexes for highly repetitive document collections. Inf Syst 61:1–23

Dean J (2009) Challenges in building large-scale information retrieval systems: invited talk. In: Proceedings of the 2nd international conference on web search and data mining (WSDM)

Delbru R, Campinas S, Tummarello G (2012) Searching web data: an entity retrieval and high-performance indexing model. J Web Semant 10:33–58

Dhulipala L, Kabiljo I, Karrer B, Ottaviano G, Pupyrev S, Shalita A (2016) Compressing graphs and indexes with recursive graph bisection. In: Proceedings of the 22nd international conference on knowledge discovery and data mining (SIGKDD), pp 1535–1544

Elias P (1974) Efficient storage and retrieval by content and address of static files. J ACM 21(2):246–260

Elias P (1975) Universal codeword sets and representations of the integers. IEEE Trans Inf Theory 21(2):194–203

Fano RM (1971) On the number of bits required to implement an associative memory. Memorandum 61. Computer Structures Group, MIT, Cambridge

Goldstein J, Ramakrishnan R, Shaft U (1998) Compressing relations and indexes. In: Proceedings of the 14th international conference on data engineering (ICDE), pp 370–379

Golomb S (1966) Run-length encodings. IEEE Trans Inf Theory 12(3):399–401

Larsson NJ, Moffat A (1999) Offline dictionary-based compression. In: Data compression conference (DCC), pp 296–305

I

Lemire D, Boytsov L (2013) Decoding billions of integers per second through vectorization. Softw Pract Exp 45(1):1–29

Lemire D, Kurz N, Rupp C (2018) Stream-VByte: faster byte-oriented integer compression. Inf Process Lett 130:1–6

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

Navarro G, Mäkinen V (2007) Compressed full-text indexes. ACM Comput Surv 39(1):1–79

Ottaviano G, Venturini R (2014) Partitioned elias-fano indexes. In: Proceedings of the 37th international conference on research and development in information retrieval (SIGIR), pp 273–282

Ottaviano G, Tonellotto N, Venturini R (2015) Optimal space-time tradeoffs for inverted indexes. In: Proceedings of the 8th annual international ACM conference on web search and data mining (WSDM), pp 47–56

Pibiri GE, Venturini R (2017) Clustered Elias-Fano indexes. ACM Trans Inf Syst 36(1):1–33. ISSN 1046-8188

Plaisance J, Kurz N, Lemire D (2015) Vectorized VByte decoding. In: International symposium on web algorithms (iSWAG)

Rice R, Plaunt J (1971) Adaptive variable-length coding for efficient compression of spacecraft television data. IEEE Trans Commun 16(9):889–897

Salomon D (2007) Variable-length codes for data compression. Springer, London

Silvestri F (2007) Sorting out the document identifier assignment problem. In: Proceedings of the 29th European conference on IR research (ECIR), pp 101–112

Silvestri F, Venturini R (2010) 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), pp 1219–1228

Stepanov A, Gangolli A, Rose D, Ernst R, Oberoi P (2011) Simd-based decoding of posting lists. In: Proceedings of the 20th international conference on information and knowledge management (CIKM), pp 317–326

Vigna S (2013) Quasi-succinct indices. In: Proceedings of the 6th ACM international conference on web search and data mining (WSDM), pp 83–92

Witten I, Moffat A, Bell T (1999) Managing gigabytes: compressing and indexing documents and images, 2nd edn. Morgan Kaufmann, San Francisco

Yan H, Ding S, Suel T (2009) Inverted index compression and query processing with optimized document ordering. In: Proceedings of the 18th international conference on world wide web (WWW), pp 401–410

Zhang Z, Tong J, Huang H, Liang J, Li T, Stones RJ, Wang G, Liu X (2016) Leveraging context-free grammar for efficient inverted index compression. In: Proceedings of the 39th international conference on research and development in information retrieval (SIGIR), pp 275–284

Ziv J, Lempel A (1977) A universal algorithm for sequential data compression. IEEE Trans Inf Theory 23(3):337–343

Zukowski M, Héman S, Nes N, Boncz P (2006) Super-scalar RAM-CPU cache compression. In: Proceedings of the 22nd international conference on data engineering (ICDE), pp 59–70