Fast Dictionary-Based Compression for Inverted Indexes

Giulio Ermanno Pibiri
The University of Pisa and ISTI-CNR
Pisa, Italy

Matthias Petri
The University of Melbourne
Melbourne, Australia

Alistair Moffat
The University of Melbourne
Melbourne, Australia

ABSTRACT

Dictionary-based compression schemes provide fast decoding operation, typically at the expense of reduced compression effectiveness compared to statistical or probability-based approaches. In this work, we apply dictionary-based techniques to the compression of inverted lists, showing that the high degree of regularity that these integer sequences exhibit is a good match for certain types of dictionary methods, and that an important new trade-off balance between compression effectiveness and compression efficiency can be achieved. Our observations are supported by experiments using the document-level inverted index data for two large text collections, and a wide range of other index compression implementations as reference points. Those experiments demonstrate that the gap between efficiency and effectiveness can be substantially narrowed.

KEYWORDS

Inverted index; efficiency; compression; decoding

ACM Reference Format:
Giulio Ermanno Pibiri, Matthias Petri, and Alistair Moffat. 2019. Fast Dictionary-Based Compression for Inverted Indexes. In *The Twelfth ACM International Conference on Web Search and Data Mining (WSDM '19), February 11–15, 2019, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3289600.3290962

1 INTRODUCTION

The compressed inverted file continues to be a critically important data structure that supports efficient keyword-based querying across large document collections. For each term t that appears in the collection a postings list is constructed, containing the sequences $\langle d_{t,i} \rangle$ and $\langle f_{t,i} \rangle$, where $d_{t,i}$ and $f_{t,i}$ are, respectively, the ordinal document number of the ith document containing t, and the number of times that t appears in that document. Such indexes support a wide range of querying modalities [44].

In this work, we revisit the question of representing the sequences $\langle d_{t,i} \rangle$ and $\langle f_{t,i} \rangle$. A wide range of compression techniques have been developed [30, 40], with recent work including the use of ANS-based compression [23, 24]; clustering of postings lists [29]; and the use of general-purpose compression libraries in conjunction with the well-known VByte approach [28]. We focus on the efficiency end of this spectrum, that is, how best to represent the sequences in compressed form if the primary goal is fast decompression. Competitors in this space include Trotman's QMX codec [36]; the VByte and Simple16 byte- and word-aligned mechanisms [2, 39]; and the PFOR approach of Zukowski et al. [45].

Our Contribution. We develop a new compression approach, DINT, based on a DIctionary of INTeger sequences. A key notion is that of *fixed-to-fixed* decoding, a marked contrast to the many *variable-to-fixed* and *fixed-to-variable* approaches that have been previously explored. The core idea is that each unit of decoding consumes one 16-bit or 8-bit integer codeword, and causes a fixed-length copying operation from the internal codebook – the *dictionary* – to the output buffer. The simplicity of this approach means that DINT decoding is fast. As well, DINT also provides remarkably good compression effectiveness. The improved combination of efficiency and effectiveness provides an important new reference point in the available spectrum of known trade-off options.

2 BACKGROUND

We assume that a sequence of integers $S = \langle s_i \rangle$ is to be stored, with $s_i \in \Sigma = \{1, \ldots |\Sigma|\}$ for $1 \leq i \leq |S|$. For inverted index compression, the sequences are composed of document identifiers $d_{t,i}$ (which we refer to as *docids*), and the frequencies $f_{t,i}$ associated with them (referred to as *freqs*), where each posting in the index has the form $\langle d_{t,i}, f_{t,i} \rangle$. The two components can be stored separately; fully interleaved; or in blocks of some size B that are themselves then interleaved at a coarser level. It is also usual to transform the docids within each postings list to a sequence of gaps, $\langle d_{t,i} - d_{t,i-1} \rangle$ (assuming $d_{t,0} = 0$), with the corresponding requirement on decoding to reconstitute the ascending sequence by computing a prefix sum. A key feature of inverted index data is that both of these two sequences are dominated by small values.

Byte- and Word-Aligned Codes. In byte-aligned compression methods [31, 35, 39] input integers are partitioned into 7-bit fragments, and the fragments are placed in bytes, leaving one bit spare per byte. That bit then serves as a flag to mark the last fragment of each integer, allowing the coded values to be reconstituted via byte-at-a-time shift and mask operations. Compared to earlier bit-aligned codes (see Witten et al. [40] for an overview), the elimination of bit-at-a-time decoding led to a substantial speed improvement, albeit with a corresponding loss of coding effectiveness. A range of byte-aligned coding variants have also been proposed [5, 6, 8, 9, 14, 28, 31]. All of these methods are fixed-to-variable, with one input token expressed as a variable number of output bits.

Word-aligned codes are also possible. For example, in the Simple16 representation [1] fixed-length 32-bit output words are formed, each consisting of a *selector* and a *payload* containing some number of same-length binary codewords. Variants include work by Zhang et al. [42], who add the flexibility to employ patterns of codewords not all the same length; and by Anh and Moffat [2], who consider

WSDM '19, February 11–15, 2019, Melbourne, VIC, Australia

ACM ISBN 978-1-4503-5940-5/19/02…$15.00
https://doi.org/10.1145/3289600.3290962

the use of 64-bit output units. The QMX mechanism of Trotman [36] also includes some elements of these approaches. Word-based codes can be though of as being variable-to-fixed, since the compression is achieved by varying the size of the input fragments rather than the lengths of the codewords assigned.

Packed and Patched Approaches. In these methods fixed-length input blocks containing B symbols are represented by variable-length compressed representations. The simplest option is to calculate the maximum binary magnitude across the symbols in the block, and code each value in the block in that many bits. Each output block starts with a selector that indicates the bit-width of each of its binary values. Lemire and Boytsov [17] explore such codes, including the use of SIMD instructions. Trotman [36] also makes use of SIMD operations to attain fast decoding.

A problem with Packed mechanisms is that unexpectedly-large values force long codewords for a whole block of symbols. Recognizing this issue, Zukowski et al. [45] introduced the *patched frame of reference* (PFOR) approach. A bit-width is chosen that covers most of the values in the block, but not necessarily all of them, and any values that require more than that many bits (referred to as *exceptions*) are represented using a secondary *patching* mechanism. A search over likely bit-widths can be performed, so that the most compact output representation, exceptions included, is achieved for each block; this is referred to as the Opt-PFOR approach [17, 41].

Packed approaches are typically fixed-to-variable arrangements.

Other Methods. Other recent work includes that of Ottaviano and Venturini [26], Ottaviano et al. [27], Wang et al. [38], Moffat and Petri [23, 24], and Pibiri and Venturini [29]. We have included several of these methods in our experimentation in Section 5.

Dictionary-Based Compression. Martinez et al. [22] introduce a dictionary-based approach that they call *plurally parsable*. Starting with a probability distribution over an alphabet of symbols, and an assumption of a memoryless source, they construct a set of strings with which to populate a dictionary of some target size, and then use a greedy parsing approach to render any input sequence into a stream of integer dictionary offsets. Their dictionary is allowed to contain sequences that are prefixes of other entries, and the entries are capped at some maximum length ℓ so that they can be stored in a rectangular two-dimensional array.

Table 1(a) gives an example of a plurally parsable dictionary, assuming an input alphabet of $\{a, b, c, d\}$, with symbol "a" dominant, and a dictionary of width $\ell = 4$ and of length $2^b = 8$. Each entry in the dictionary contains $\ell + 1$ entries, as many as ℓ of which are the corresponding string, and the last one of which is the number of symbols in that string. Using this dictionary, the example string $\langle$aaab aabc aaaa b$\rangle$ (with spaces introduced purely for visual separation) would be greedily parsed as $\langle$aaa, b, aa, b, c, aaaa, b$\rangle$, and coded as the sequence of $b = 3$-bit integers $\langle 5, 1, 4, 1, 2, 7, 1 \rangle$ using a total of 21 bits. Note how all three of the "b"s, and the "c" as well, are coded as sequences of length one. In the development below we refer to these instances as being *singletons*. The dictionary does not force the "b"s to be coded as singletons, but the left-to-right greedy parsing of the input has resulted in that happening. Singletons are relatively costly, because each of them requires a full codeword in the compressed stream.

Index	String				
0	a	-	-	-	1
1	b	-	-	-	1
2	c	-	-	-	1
3	d	-	-	-	1
4	a	a	-	-	2
5	a	a	a	-	3
6	a	b	-	-	2
7	a	a	a	a	4

(a)

Index	String				
0	-	-	-	-	1
1	a	-	-	-	1
2	b	-	-	-	1
3	a	a	-	-	2
4	a	b	-	-	2
5	b	a	-	-	2
6	a	a	a	a	4
7	a	a	a	b	4

(b)

Table 1: Two examples of plurally parsable dictionaries of width $\ell = 4$ over the alphabet $\{a, b, c, d\}$ where symbol "a" is highly probable, symbol "b" is moderately probable, and "-" entries indicate don't-care values. The last column provides the length of each string and is also stored as part of the dictionary. In (b), index zero is used as the code for rare symbol exceptions.

Martinez et al. [22] use the final $\ell + 1$ st column as a way of accelerating decoding. Rather than execute a loop that counts exactly the right number of symbols from a dictionary entry to the output buffer and in doing so tests a guard at every iteration, the decoding process always copies the full $\ell + 1$ symbols to the output buffer in a single fixed operation, and then increases the output pointer by the amount indicated by the $\ell + 1$ st copied value. Because the conditional in the innermost nested loop is eliminated, branch mis-predictions are reduced, and high decoding speeds can be achieved.

To build the dictionary Martinez et al. [22] describe a process that tentatively assigns strings to the dictionary based on their zero-order probability of occurrence as indicated by their corresponding symbol frequencies, and then iteratively refines those estimates, converging to a set of variable length strings that provides the best coverage. They build a suite of such dictionaries for different initial symbol distributions, and then use them to losslessly code 64×64-pixel blocks of grey-scale image data, with a matching dictionary selected for each block, and indicated to the decoder via a selector at the start of the block.

Hoobin et al. [13] and Liao et al. [20, 21] have also considered dictionary-based compression options, applying them to the text of large document collections; and Zhang et al. [43] have sought to apply the same Relative Lempel Ziv approach to index data.

3 BASE IMPLEMENTATION

We now describe our initial application of dictionary-based compression to inverted index data. Then, in Section 4, a range of refinements are introduced.

The Dictionary. There are two factors that make inverted index data highly distinctive. First, there are very long runs of "1"s (almost always the most frequent symbol in the alphabet) that create opportunities for the use of a *frequent symbol exception*, whereby long repetitive sequences are handled outside the normal regime. As is demonstrated in Section 5, upwards of a third of the docids and freqs in typical inverted index sequences are "1"s, and handling these economically is a key requirement. Second, the alphabet for docid gaps is very large, into the millions, and it is impossible to

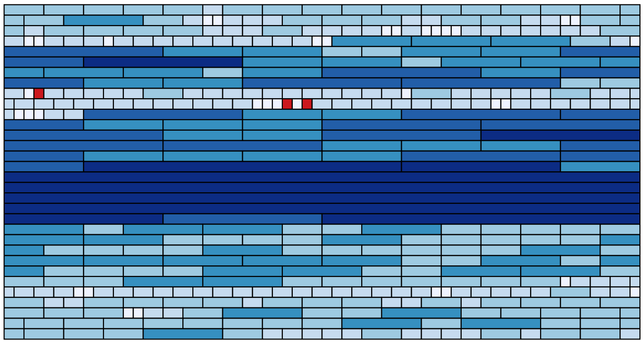

Figure 1: Analysis of a typical sequence of 2,048 docid gaps from the posting list of a single term in a large text collection, eight blocks of size $B = 256$, with each row spanning 64 docids. Long runs of "1"s are shown in the darkest blue color; other shades represent frequently-occurring subsequences of length 1, 2, 4, 8 and 16, and are coded as matches against the dictionary. The three red squares in the ninth and tenth rows are docid gaps that are relatively rare in the collection, and must be coded as exceptions because they do not appear in the dictionary.

consider providing a codeword for every symbol, even as a singleton. Instead, use must be made of *rare symbol exceptions*, a special code that indicates that the next symbol must be fetched from a secondary stream of uncompressed integers.

Figure 1 shows an example of repeated frequent subsequences occurring in a typical extract of 2,048 docid gaps. Each colored rectangle represents a sequence that occurs many times across the index, and hence can be represented as a codeword relative to a dictionary of 65,536 such sequences. Only three of the docid gaps in this typical fragment are sufficiently rare that they are coded as exceptions, rather than via the dictionary.

Rare Symbol Exceptions. To see the use of rare symbol exceptions, consider the dictionary shown in Table 1(b), in which only two singleton codes are provided. Using this table the same example string ⟨aaab aabc aaaa b⟩ would be parsed ⟨aaab, aa, b, –, c, aaaa, b⟩, and coded as the sequence of integers ⟨7, 3, 2, 0, c, 6, 2⟩ using a total of $6 \times 3 + 1 \times 2 = 20$ bits, where it is assumed that the rare symbol exception needed for "c" (following the escape codeword of "0") requires two bits over the alphabet of four symbols. In this small example an overall slight reduction in cost arises, primarily because of the presence of the string "aaab" in the dictionary. But the general principle is valid: the greater the number of long sequences that can be included in the dictionary, the better the compression rate that we can hope to achieve.

The large symbol alphabet used in index compression means that it makes sense to employ multiple exception options: code "0" to indicate that the corresponding patching symbol is a b-bit value between 1 and 2^b, where, as before, b is the width in bits of each of the codewords and the dictionary is 2^b entries; code "1" to indicate that the associated patching value is a $2b$-bit value in the range $2^b + 1$ to 2^{2b}; and so on. For example, when $b = 16$, two rare-exception codes cover the space of 32-bit integers; and four rare-symbol exception codewords are employed if $b = 16$ and the input is regarded as being the space of 64-bit integers.

Frequent Symbol Exceptions. To handle long runs of "1"s, further exception codes are added, covering sequences of length B, $B/2$, $B/4$, . . ., 2ℓ. The first of these covers an entire block that is all "1"s very economically; and short runs of (only) ℓ "1"s can be covered by a regular non-exception codeword if required. For a $b = \ell = 16$ configuration, there will thus be six dictionary slots reserved for exceptions – two rare symbol exception codes, and four frequent symbol exception codes – leaving 65,530 codewords for regular dictionary entries.

Frequency Estimation. The set of 2^b sequences making up the dictionary should be tailored to the data being compressed, so that the dictionary stores a selection of highly useful subsequences.

To count sub-sequence frequencies, an *interval sampling* approach is employed, examining the source sequence at uniform intervals of $L = 2^k \geq \ell$ and extracting samples of each length $\ell' \in \{1, 2, 4, \ldots, \ell\}$ at that point. The frequency of a sequence of length ℓ' is incremented by L/ℓ'. For example, if $\ell' = 2$ and $L = 8$, a two-symbol prefix is extracted every 8 symbols in the input sequence, and that two-symbol combination has its frequency incremented by four. To reduce the counting time L can be made relatively large, for example, $L = 1024$, and to reduce the space required by the data structure accumulating the counts, a reservoir-based approach can be employed [19, 37]. Both of these techniques produce estimates of the sequence frequencies and not exact counts. But having exact counts would not necessarily be any more useful, since any particular factor parsed from the source sequence might include part or all of other dictionary strings, affecting those counts. In the experiments reported in Table 2 and in Section 5 exhaustive sampling with $L = \ell'$ is used.

Dictionary Construction. In general, the problem of building a dictionary that minimizes the length of the message when coded relative to the dictionary is NP-hard [34]. Hence, rather than seek optimal solutions, we consider two heuristics for selecting the set of 2^b sequences with which to populate the dictionary. Both approaches suppose that each observed sequence S of length $|S|$ has been estimated to occur $freq[S]$ times.

The first approach – which we denote as *decreasing static volume*, or DSV – chooses the set of targets that provide the greatest coverage volume, where coverage volume is calculated as the product of frequency and length of the targets, with no consideration given to possible interactions between sequences. That is, each candidate sequence S is given a score of $|S| \times freq[S]$, and the set of sequences with the largest scores are used to form the dictionary.

A more nuanced analysis leads to the second heuristic we explore. Suppose that some sequence S is being considered to be placed in the dictionary. Given that S is now a candidate, it seems likely that both its first half, denoted S_1, of length $|S|/2$, and its second half, denoted S_2, also of length $|S|/2$, with $S = S_1 S_2$, will already be in the dictionary. This is because (assuming interval sampling) $freq[S_1] \geq freq[S]$ and (via symmetry, but not guaranteed) that $freq[S_2] \geq freq[S]$. And if S_1 and S_2 are already in the dictionary, then the saving generated by also adding S to the dictionary is only $freq[S]$, since one codeword will be used for each instance of S, rather than

```
1:  set output ← 0                                  ▷ initialize output counter
2:  while output < B do
3:      set codeword ← get_code()
4:      if codeword < 6 then
5:          if codeword < 2 then                    ▷ rare symbol exception
6:              use get_code() to access and copy
7:                  excep_lens[codeword] codes to output
8:              set output ← output + 1
9:          else                                    ▷ frequent symbol exception
10:             bulk copy excep_lens[codeword] "1"s to output
11:                 from an array containing 2^b "1"s
12:             set output ← output + excep_lens[codeword]
13:     else                                        ▷ normal codeword
14:         set index ← start[codeword]
15:         copy ℓ symbols from dictionary[index] to output
16:         set output ← output + length[codeword]
17: return
```

Figure 2: Decoding process for one block. Function *get_code*() returns the next b bits from the input sequence as an integer value; array *length*[] refers to the length ℓ' of the corresponding target sequence and might be stored as a component of the *start*[] array (see Figure 3); and the fixed array *excep_lens*[] contains values that match the choice of b and ℓ. For example, when $b = \ell = 16$ and $B = 256$, *excep_lens*$[] = \{1, 2, 256, 128, 64, 32\}$.

two. The same argument can be applied inductively, with the base case arising when singletons are being considered. The true cost of not including them in the dictionary is simply the difference in cost generated by the use of a rare symbol exception. Hence, the second heuristic we consider for populating the dictionary is that of *decreasing static frequency* or DSF, choosing the sequences with the highest *freq*$[S]$ estimates, regardless of length. As a secondary sort key, to break ties, we sort by decreasing length $|S|$. Note that this inductive argument is also why we focus on a restricted set of target lengths $\{1, 2, \ldots, \ell/2, \ell\}$, with $\ell = 2^k$ for some k.

We also explored adaptive selection heuristics, dynamically updating the *freq*$[S]$ count for sequences that were prefixes and suffixes of longer strings when they were committed to the dictionary, the idea being to maintain more precise frequency estimates. Small gains in compression effectiveness were observed in some test situations, but small losses in others; and overall we were unable to consistently outperform the DSF method. Other refined mechanisms for populating the dictionary will be a target for future research, noting that the problem we face here has parallels in the Re-Pair compression technique [16], which has also been used for index compression [7]; and that Apostolico and Lonardi [3] have also considered the question of identifying useful subsequences.

Decoding. The standard unit of access is a single block of B integers and we employ $B = 256$ throughout this investigation; that is, each postings list is partitioned into fixed-length blocks, with any remaining elements represented using a secondary mechanism. Fewer than 5% of the postings are coded in this manner. Each block of integers is represented as a set of one or more codewords, each of these being a b-bit binary code.

	Dictionary width		
	$\ell = 4$	$\ell = 8$	$\ell = 16$
$b = 8$	4.786	4.770	4.774
$b = 12$	4.893	4.486	4.396
$b = 16$	5.289	4.505	4.332

Table 2: Total index size in GiB for a complete document-level index (docids and freqs combined, including block-access overhead and dictionary space) for the Gov2 collection using a block size of $B = 256$ items and the DSV dictionary construction approach.

Quantity	Variable-length		Constant-length	
	docids	freqs	docids	freqs
instructions ($\times 10^9$)	53.63	35.02	41.72	28.35
instructions/cycle	1.16	1.13	1.28	1.24
cache-misses ($\times 10^7$)	10.77	9.06	8.21	7.60
branch-misses (%)	3.40	2.79	2.24	0.35
nanosec/integer	1.82	1.08	1.12	0.73

Table 3: Performance counts reported by the Linux `perf` tool, comparing variable-length copying and constant-length copying for $\ell = 16$ when decoding the index sequences of Gov2 using a rectangular dictionary.

The action of the decoding algorithm is described in Figure 2, where it is assumed that $b = \ell = 16$, with codewords "0" and "1" indicating rare symbol exceptions, and codewords "2"…"5" indicating frequent symbol exceptions. Small changes might be required if b or ℓ are varied, but note that the rare symbol exception codewords and frequent symbol exception codewords should always be grouped together in the code space, so that a single conditional is sufficient to reach the dominant case, that of a standard codeword referring to a symbol sequence in the dictionary (step 14).

Choosing Parameters. To establish likely parameter combinations for a full implementation, we carried out a preliminary exploration of the variables b (bits per codeword) and ℓ (maximum length of dictionary entries, in integers), using the Gov2 document collection (see Table 5), and the DSV dictionary construction heuristic. Table 2 lists the resultant total index sizes (in GiB); as can be seen, there are several combinations of b and ℓ that provide good compression effectiveness once suitable dictionaries have been identified, with the $b = 16$ and $\ell = 16$ combination slightly better than the other arrangements. Baseline compression rates for other methods on this dataset, and also on a second collection, are provided in Section 5, as well as decoding speed measurements. As a single preliminary reference point, a VByte index for the same data occupies 11.85 GiB, and requires more than twice the space.

Copying Fixed-Length Strings. To further motivate fixed-length dictionary-based compression, Table 3 shows statistics collected using the Linux `perf` utility when decoding the sequences of the same Gov2 dataset. In the left pair of columns, the copying process is executed via a loop controlled by a variable that copies the correct number of symbols from the dictionary to the output; in the right pair of columns, a constant ℓ symbols are always copied, with ℓ

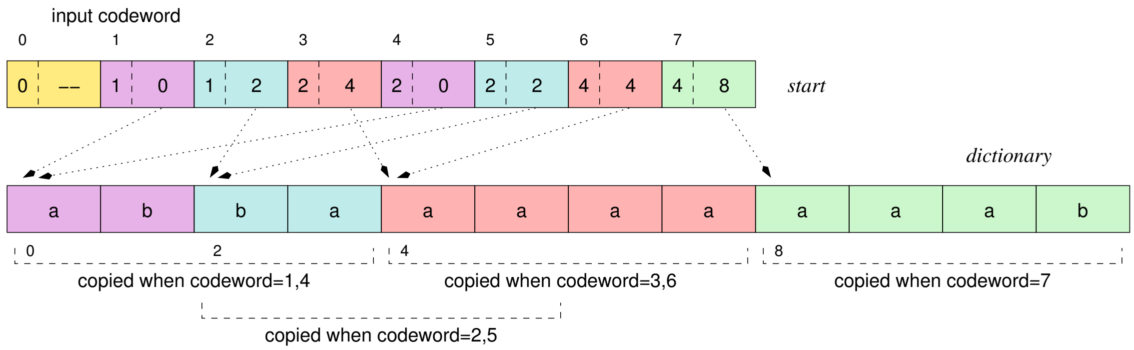

Figure 3: Packed layout for the dictionary shown in Table 1(b), with $\ell = 4$ and $b = 3$. The first number in each element in *start*[] is the sequence length. All trailing don't-care symbols have been trimmed, and dictionary sequences have been removed if they are a prefix of another longer sequence. As an example, when the input codeword is 3 the four integers in *dictionary*[4 . . . 7] are copied to the output, and then the output pointer is incremented by two. In this illustration no provision has been made for frequent symbol exceptions.

ℓ, c	docids			freqs		
	m	pred.	actual	m	pred.	actual
4, 2.495	4.25	0.59	0.82	5.02	0.50	0.50
8, 4.177	4.63	0.90	0.91	7.05	0.59	0.56
16, 5.342	4.69	1.14	1.12	7.87	0.68	0.73

Table 4: Average number of decoded integers per codeword, m, when decoding the sequences of Gov2, using DINT-DSF with a rectangular dictionary and $b = 16$; and predicted and actual decoding times measured in nanoseconds per integer, based on c, the measured decoding time per codeword.

fixed at compilation time. Copying a fixed number of bytes allows better exploitation of the instruction cache, and leads to a higher instruction throughput (instructions/cycle) with fewer cache- and branch-misses. Overall, the time taken to extract each decoded integer decreases to around two-thirds of what it would be if the copying process was controlled by a loop that copied fewer bytes, a substantial reduction.

Decoding Speed Analysis. Two parameters affect DINT's sequential decoding time: m, the average number of decoded integers per codeword, and c, the cost of the copy operation that is associated with each codeword. Decoding more values per codeword increases throughput; on the other hand, copying fewer words during a single decoding operation is faster. This means that using smaller (larger) values of ℓ decreases (increases) the cost of a single copy operation, but also that more (fewer) operations are needed to decode the sequence. To quantify this behavior, Table 4 reports measured values for c, and for m as a function of ℓ, using the docids and freqs sequences of Gov2. The predicted decoding time is calculated as c/m nanoseconds/integer, and provides a reasonable estimate of the measured decoding costs in the columns headed "actual".

4 FURTHER IMPROVEMENTS

This section describes several improvements to the initial scheme presented in Section 3.

Packed Dictionary Structure. The rectangular dictionary employed in Section 3 and shown in Table 1 is potentially expensive in terms of space, especially if there are relatively few targets of length ℓ, or if there is significant overlap between prefixes and suffixes of different targets. To this end we consider ways of reducing the space required by the dictionary, noting that the smaller the space required, the more likely it is to be retained primarily in cache.

To reduce the memory required by the dictionary the packed form shown in Figure 3 can be employed. Now the target lengths are separated from the sequences, and more than one target can indicate the same start position in the single consolidated dictionary string. Packing the dictionary both allows unused trailing symbols to be avoided, and also allows targets that are prefixes of each other to share space. The *length*[] component of each dictionary entry (see Figure 2) is stored as a one-byte field within each dictionary offset in the array *start*[], allowing sequences that have the same starting point to be distinguished, an arrangement that is valid provided that $b \leq 24$ and $\ell < 256$. The indirection via *start*[] means that one additional array dereferencing operation is required in each innermost loop in Figure 2, plus a mask/shift sequence to extract the two parts of *start*[*codeword*], but the net cost is moderate and might be warranted by the space savings. In rectangular form an $\ell = 16$ and $b = 16$ dictionary requires $4 \times 2^{16} \times (16 + 1) = 4.25$ MiB; in packed form that requirement can be reduced to around 1 MiB. Detailed results are presented in Section 5.

Exploiting String Overlap. Further savings are also possible, beyond those offered by prefix matches and trailing don't-cares. For example, with strategic reordering and overlapping, the twelve-symbol *dictionary*[] array in Figure 3 could be further condensed to just six symbols "baaaab", since every target listed in Table 1(b) appears within it as a subsequence. The problem of identifying a minimal-length super-sequence in which every one of an original set of supplied sequences occurs as a sub-sequence is NP-hard

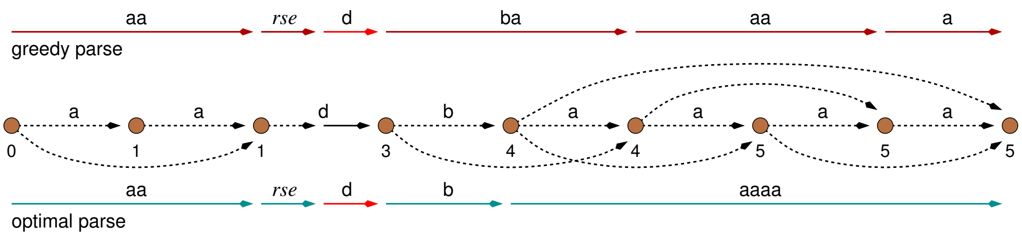

Figure 4: Example in which optimal parsing requires fewer codewords than greedy parsing. The sequence "aadbaaaa" is being represented relative to the dictionary shown in Table 1(b). The cost below each node is the length of the shortest path from the origin to that point. The parse shown at the bottom of the graph has a cost of 5; whereas the greedy parse shown above requires 6 codewords. In both cases it is assumed that "d" requires a rare symbol exception (*rse*) codeword, followed by a patch codeword that identifies the symbol required.

[12]. But simple greedy approximation algorithms can provide solutions that are within a constant factor of being optimal [4, 15]. The approach we employ here considers the initial set of sequences, determines the longest possible match between a prefix of one sequence and a suffix of another, and replaces the two sequences with their lapped concatenation. Each such step reduces the number of sequences in the set by one, and ensures that a single sequence emerges in which every one of the original sequences is embedded. For example, starting with the sequences in Table 1(b), the first cycle combines "aaaa" and "aaab" to form "aaaab".

Optimal Block Parsing. Dictionary-based compression implementations typically make use of greedy left-to-right parsing, with the longest matching dictionary entry employed at each coding step. But in the situation considered here it is straightforward to identify an optimal parse for each block, because each dictionary sequence or frequent symbol exception has a cost of one, and each rare symbol exception also has a fixed cost that depends only on its value. Figure 4 shows how the prefixes of the block correspond to nodes in an acyclic directed graph, and how the dictionary entries are edges that extend one prefix of the block to a longer one. Within this graph the shortest path from source to sink describes an optimal parse, and can be computed via a left-to-right iterative labeling process that assigns the source with cost zero, and then pushes tentative costs ahead from each node via the edges that emanate from it. The small number of edges that are possible at each node (a maximum of $\log_2 \ell + 1$ if target lengths are restricted to powers of two) makes this process only moderately slower than the more usual greedy approach, with a complexity of $O(n \log \ell)$ for a list of n integers, and hence a complexity of $O(n)$ if ℓ is regarded as being a constant.

Multiple Dictionaries. Following the example of Moffat and Petri [23], it is also possible for multiple dictionaries to be used. For example, if the input symbols are assumed to be integers between 1 and $2^{32} - 1$, then the use of 32 dictionaries allows each block to be handled within a *context* established by $\lfloor \log_2 max \rfloor$, where *max* is the largest value in the block. Stratifying the blocks according to their maximum value and coding each block against a dictionary specifically created for that maximum value offers clear benefits. For example, blocks in which $max < 4$ are likely to generate quite different dictionaries from those arising when (say) $max < 1024$, even though "1"s are likely to still be the most common symbol.

There is, however, a cost – each additional dictionary must be stored during decoding operations, and both adds to the memory cost, and also adds to the likelihood of cache misses. For this reason, other, less costly, categorizations might also be desirable. In Section 5 we make use of the mapping $context = \lceil \log_2 \log_2 max \rceil$ (taking $\log_2 0 = 0$ when $max = 1$), creating a set of six different contexts $(0 \ldots 5)$ with limiting values $2, 4, 16, 256, 65536$, and 2^{32}.

Once the suite of dictionaries has been created, the encoder either uses the same mapping to determine which context to use when encoding each block, or carries out an exhaustive search over all contexts to identify the one that minimizes the compressed size. Either way, each encoded block is prefixed by a *selector* indicating which dictionary to use. In the DINT implementation the selector is (slightly wastefully) stored as a one-byte integer.

There is a second way in which multiple dictionaries might be employed, and that is through the use of alternative combinations of b and ℓ. For example (see Table 2) it might be beneficial to consider both $b = 16$ and $b = 8$ dictionaries for each context, anticipating (say) that blocks in which *context* is small might be handled more compactly by a 256-element dictionary and the corresponding 8-bit codewords. Again, the selector is used to indicate which context is in use in any particular block. No extra memory space is required by this option, since the $b = 8$ dictionary for any context is an exact prefix of the $b = 16$ dictionary.

As already noted, when multiple contexts are in use memory consumption might become an issue. If so, rather than store each dictionary separately, a set of distinct *start*[] arrays can be used to index a single shared *dictionary*[] array (see Figure 3), with the complete set of contexts' sequences stored overlapped using the heuristic already described.

Finally in this section, note that Moffat and Petri [24] make use of the block median as a second factor in determining contexts, obtaining small compression gains when using a entropy-coder. This might be a possibility with dictionary coders too, but the codewords used here are far from being entropy based, and the dictionaries that are required are each an order of magnitude larger, likely eroding the savings that can be anticipated.

5 EXPERIMENTS

We now present the results of detailed experiments based on a range of public software and an implementation of the new DINT approach.

Collection	Lists	Postings	Documents
Gov2	39,180,840	5,880,709,591	25,205,179
CCNEWS	43,844,574	20,150,335,440	43,530,315

Table 5: Number of lists, postings and documents for the Gov2 and CCNEWS collections.

Datasets and Methodology. We use the standard Gov2 collection containing 426 GiB of text; and CCNEWS, an English subset of the freely available NEWS subset of the CommonCrawl[1], consisting of news articles in the period 09/01/16 to 30/03/18, following the methodology of Petri and Moffat [28]. Postings lists for both collections were extracted from the Indri search engine to ensure reproducibility, using a document ordering derived from the recursive graph bisection reordering technique of Dhulipala et al. [11] (rather than the more usual URL ordering). Each index was considered as two streams of integers, one containing gaps between document identifiers (docids) and one containing within-document frequencies (freqs), with both of those streams split into per-term postings list segments in the usual manner. Statistics for these two datasets are provided in Table 5.

All compression results are for complete indexes without stopping or other reduction mechanisms being applied, and cover all postings; with sizes reported in GiB and rates given in bits per integer. Where a blocksize is required, $B = 256$ is used, with trailing part-blocks represented using Interp [25]. All compression effectiveness results given for DINT include the overhead cost of the corresponding dictionaries.

Implementations and Hardware. All experimentation is based on the ds2i framework [26, 27], with methods implemented using C++14 and compiled with g++ 7.2.0 (using all optimizations) on a server equipped with 512 GiB RAM and an Intel Xeon 6144 processor employing 32 kiB of L1 cache, 1024 kiB of L2 cache, and 25344 kiB of L3 cache. The experimental framework and all code is available at https://github.com/jermp/dint.

Compression Effectiveness. Table 6 gives baseline compression effectiveness results for the four streams of integers that make up the indexes of these two collections, using a range of previous mechanisms, including: Varint-GB [10]; Varint-G8IU [33]; QMX [36]; Simple16 [1]; Opt-PFOR [41]; PEF [26]; Clust-EF [29]; and Interp [25]. The ANS version tested uses a set of 64 two-dimensional med-max contexts for each of docids and freqs [24].

Table 6 also includes the new DINT scheme, using both of the dictionary construction mechanisms discussed in Section 3, with greedy parsing, a single dictionary for each stream for each collection (that is, four dictionaries in total, one per stream), and $b = 16$ and $\ell = 16$ (see Table 2). As can be seen, with these standard settings DINT yields compression rates better than Simple16, and comparable to those attained by the Opt-PFOR approach. The two dictionary construction mechanisms give slightly different effectiveness, and the DSF approach has a small but consistent advantage over the DSV heuristic.

[1]http://commoncrawl.org/2016/10/news-dataset-available/

Method	Gov2			CCNEWS		
	GiB	docids	freqs	GiB	docids	freqs
Varint-GB	14.48	11.04	10.04	48.68	10.72	10.01
Varint-G8IU	12.77	9.90	8.69	43.87	9.75	8.93
VByte	11.85	9.22	8.02	39.65	8.88	8.00
QMX	5.59	4.99	3.11	19.20	5.12	3.04
Simple16	5.28	4.84	2.81	16.85	4.70	2.46
Opt-PFOR	4.55	4.33	2.26	15.50	4.49	2.10
DINT-DSV	4.33	4.25	2.00	15.25	4.52	1.95
DINT-DSF	4.29	4.22	1.98	15.09	4.48	1.92
PEF	4.16	3.85	2.23	13.75	3.89	1.97
Clust-EF	4.02	3.66	2.16	13.44	3.79	1.92
Interp	3.86	3.54	2.04	12.80	3.64	1.79
ANS, 2d	3.71	3.56	1.86	12.58	3.67	1.69

Table 6: Total index size (GiB) and compression rate (bits per integer) for docids and freqs for two test collections, using a range of compression techniques. The two DINT implementations both make use of $b = 16$ and $\ell = 16$, single dictionaries, and greedy parsing. The rows are ordered by decreasing total index size.

Method	Gov2		CCNEWS	
	docids	freqs	docids	freqs
Interp	7.84	7.56	8.59	7.91
PEF	3.13	3.78	3.05	2.68
Opt-PFOR	1.87	1.31	1.35	1.05
Simple16	1.46	1.15	1.45	1.06
VByte	1.24	0.85	1.07	0.79
QMX	1.13	1.06	1.48	1.38
DINT-DSF: $\ell = 16$	0.87	0.64	0.91	0.64
DINT-DSF: $\ell = 8$	0.80	0.55	0.79	0.54
Varint-GB	0.75	0.61	0.65	0.58
Varint-G8IU	0.66	0.61	0.57	0.52
Masked-VByte	0.66	0.59	0.59	0.49
Stream-VByte	0.58	0.57	0.57	0.54

Table 7: Sequential decoding throughput in nanoseconds per integer, measured over the complete index. Both DINT rows make use of $b = 16$ and a packed dictionary. The rows are ordered by increasing speed of docids decoding on Gov2.

Sequential Decoding Speed. Table 7 compares decoding speeds of DINT (using a packed dictionary constructed using the DSF process with $b = 16$ and two values of ℓ) and a range of other index compression approaches (including the SIMD-ized mechanisms of Masked-VByte [32] and Stream-VByte [18]), measured by decoding the entire index in a sequential manner. The net result of the experiment is that, compared to the methods that provide comparable or better effectiveness (Table 6), DINT is *faster* and, compared to methods of similar speed, DINT provides *better* compression effectiveness. Moffat and Petri [23, 24] report speeds for ANS decoding; based on their measured relativities, it would be second-from-top in Table 7; and based on the results reported by

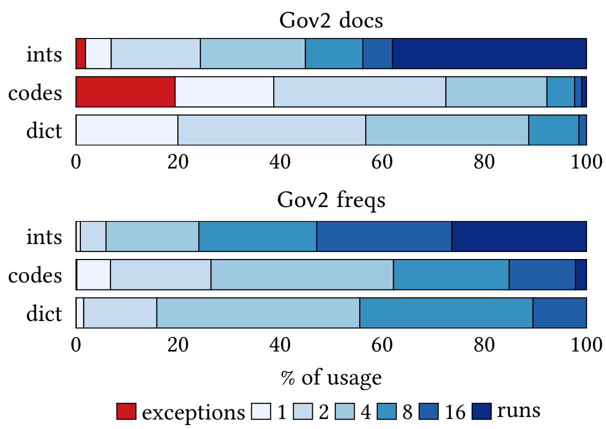

Figure 5: Percentage of integers, codewords, and dictionary entries associated with each target size for the docids and freqs of Gov2, using the DSF approach and parameters $b = 16$, $\ell = 16$.

Pibiri and Venturini [29], the Clust-EF mechanism can be expected to decode more slowly than the PEF approach, placing it also in the upper section of the table (neither of these two implementations was compatible with the sequential decoding test harness used to generate Table 7).

Note that once the sequence frequency estimates have been collected, DINT dictionary formation and encoding is very fast, and we do not report encoding times in this version of the work.

Dictionary Performance. Figure 5 provides a summary of the patterns already illustrated in Figure 1, and shows the distribution of target lengths in the raw index, in the compressed index, and in the dictionary respectively. For example, around 20% of the compressed codewords are rare symbol exceptions covering just 2% of the docids in the actual Gov2 index; whereas 38% of the docids can be handled by frequent symbol exceptions, consuming just 1% of the actual codewords. The freqs dictionary matches are longer than in the docids stream, leading to higher compression rates.

Optimal Parsing. The second row in Table 8 shows the additional gains that result from the use of optimal parsing. This gain comes at the expense of a small increase in encoding time, but has no effect on decoding time.

Multi-Context Operation. The third row of Table 8 shows the additional compression gains that result when a total of six dictionaries are used per stream, conditioned on the largest value *max* in each block via the mapping $context = \lceil \log_2 \log_2 max \rceil$. A one-byte selector is required per block, partially negating the gains, but even so, there is an overall benefit. In the fourth row, six further dictionaries are added per stream, allowing $b = 8$ operation (still with $\ell = 16$) in blocks where this provides an advantage. The choice between the $b = 8$ and $b = 16$ dictionary is made by test-compressing the block using the two options. There is again a consistent gain achieved. In the fifth row, an exhaustive test-compression search over all twelve available contexts is made on a per-block basis, slowing decoding time, but not affecting decoding throughput in any way. Further small compression gains emerge.

Method	Gov2			CCNEWS		
	GiB	docids	freqs	GiB	docids	freqs
DINT-DSF	4.29	4.22	1.98	15.09	4.48	1.92
+ opt. pars.	4.25	4.19	1.95	14.93	4.45	1.89
+ 6 contexts	4.22	4.09	1.94	14.64	4.31	1.88
+ $b = 8, 16$	4.10	4.02	1.83	14.21	4.23	1.78
+ exh. srch.	4.07	4.00	1.81	14.08	4.19	1.77
+ entropy	3.60	3.51	1.75	12.88	3.77	1.72

Table 8: Total index size (GiB) and compression rate (bits per integer) for docids and freqs using $b = 16$ and $\ell = 16$. The first row uses DINT-DSF with greedy parsing; four enhancements are then added, and in the penultimate row a total of 12 contexts are used with optimal parsing and an exhaustive search to identify the cheapest context for each block. In the last row, the dictionary indices are then assumed to be input to a set of 12 optimal entropy coders. Except for the last row (gray numbers), which contains values that are calculated rather than measured, these results can be directly compared with Table 6.

Dictionary	docids		freqs	
	MiB	ns/int	MiB	ns/int
rectangular, ×1	4.250	1.12	4.250	0.73
packed, ×1	1.045	0.87	1.750	0.64
overlapped, ×1	0.874	0.95	1.408	0.70
rectangular, ×6	21.796	1.66	21.343	1.04
packed, ×6	7.269	1.19	9.122	0.81
overlapped, ×6	6.234	1.37	7.736	0.90

Table 9: Dictionary space (MiB) for different schemes and corresponding decoding speeds, for Gov2. In the three "×1" rows, one dictionary is used for the docids and another for the freqs. In the "×6" rows, six dictionaries are used for each stream, with (in the last row) all six of them combined into a single *dictionary*[] array, and six *start*[] arrays maintained.

Table 9 shows the dictionary cost of the various combinations considered. Decoding using a packed dictionary is faster than decoding via a rectangular dictionary because of its more compact memory footprint, but overlapping the strings to further save space loses the alignment property of the packed arrangement, and increases decoding cost. Use of multiple contexts leads to slightly better compression, but slows decoding throughput because of the increased memory and greater number of cache misses.

6 CONCLUSIONS

Figure 6 summarizes the relative performance of the new DINT approach, combining Tables 6 and 7. When presented in this way it is clear that our dictionary-based technique represents an important new approach to inverted index compression, approaching the speed of the very fastest methods for decoding and, at the same time, approaching the compression effectiveness of the best methods in terms of space required. Verifying that the throughput gains (demonstrated in Section 5) translate to faster query processing in

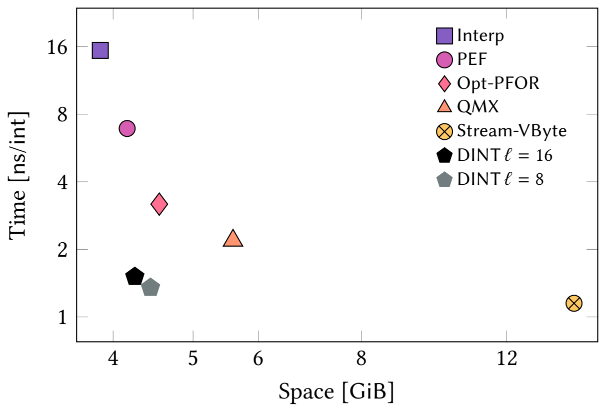

Figure 6: Final effectiveness-efficiency graph for Gov2. The vertical scale sums the per-posting docids and freqs times for each method; the horizontal scale shows total index size in GiB. The two DINT points both use a single packed dictionary per stream, $b = 16$, and optimal parsing. Both scales are logarithmic.

realistic settings is a clear area for future work. We also plan to explore less costly frequency estimation techniques, and quantify the extent to which accurate counts are needed for best compression.

Looking beyond the considerable gains we have already achieved, the last row of Table 8 shows that the streams of dictionary indices still possess a great deal of redundancy. The application of entropy codes should yield further important trade-off options for index compression, and we also plan to explore that possibility.

Acknowledgment. The first author was partially supported by the *Pegaso* Project, POR FSE 2014-2020; with additional resourcing provided by Shane Culpepper (RMIT University).

REFERENCES

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

[2] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Soft. Prac. & Exp.*, 40(2):131–147, 2010.

[3] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. *Proc. IEEE*, 88(11):1733–1744, 2000.

[4] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *J. ACM*, 41(4):630–647, 1994.

[5] N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller. (S, C)-Dense coding: An optimized compression code for natural language text databases. In *Proc. SPIRE*, pages 122–136, 2003.

[6] N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. SIGIR*, pages 139–146, 2008.

[7] F. Claude, A. Fariña, and G. Navarro. Re-Pair compression of inverted lists. *CoRR*, abs/0911.3318, 2009.

[8] J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *Proc. SPIRE*, pages 1–12, 2005.

[9] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Sys.*, 29(1):1.1–1.25, 2010.

[10] J. Dean. Challenges in building large-scale information retrieval systems: Invited talk. In *Proc. WSDM*, 2009.

[11] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proc. KDD*, pages 1535–1544, 2016.

[12] J. Gallant, D. Maier, and J. A. Storer. On finding minimal length superstrings. *J. Comp. Sys. Sc.*, 20(1):50–58, 1980.

[13] C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *PVLDB*, 5(3):265–273, 2011.

[14] A. Kane and F. W. Tompa. Skewed partial bitvectors for list intersection. In *Proc. SIGIR*, pages 263–272, 2014.

[15] H. Kaplan and N. Shafrir. The greedy algorithm for shortest superstrings. *Inf. Process. Lett.*, 93(1):13–17, 2005.

[16] N. J. Larsson and A. Moffat. Offline dictionary-based compression. *Proc. IEEE*, 88 (11):1722–1732, 2000.

[17] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Soft. Prac. & Exp.*, 45(1):1–29, 2015.

[18] D. Lemire, N. Kurz, and C. Rupp. Stream-VByte: Faster byte-oriented integer compression. *Inf. Process. Lett.*, 130:1–6, 2018.

[19] K.-H. Li. Reservoir-sampling algorithms of time complexity $O(n(1+\log(N/n)))$. *ACM Trans. Math. Soft.*, 20(4):481–493, 1994.

[20] K. Liao, M. Petri, A. Moffat, and A. Wirth. Effective construction of Relative Lempel-Ziv dictionaries. In *Proc. WWW*, pages 807–816, 2016.

[21] K. Liao, A. Moffat, M. Petri, and A. Wirth. A cost model for long-term compressed data retention. In *Proc. WSDM*, pages 241–249, 2017.

[22] M. Martinez, M. Haurilet, R. Stiefelhagen, and J. Serra-Sagristà. Marlin: A high throughput variable-to-fixed codec using plurally parsable dictionaries. In *Proc. DCC*, pages 161–170, 2017.

[23] A. Moffat and M. Petri. ANS-based index compression. In *Proc. CIKM*, pages 677–686, 2017.

[24] A. Moffat and M. Petri. Index compression using byte-aligned ANS coding and two-dimensional contexts. In *Proc. WSDM*, pages 405–413, 2018.

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

[26] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proc. SIGIR*, pages 273–282, 2014.

[27] G. Ottaviano, N. Tonellotto, and R. Venturini. Optimal space-time tradeoffs for inverted indexes. In *Proc. WSDM*, pages 47–56, 2015.

[28] M. Petri and A. Moffat. Compact inverted index storage using general-purpose compression libraries. *Soft. Prac. & Exp.*, 48(4):974–982, 2018.

[29] G. E. Pibiri and R. Venturini. Clustered Elias-Fano indexes. *ACM Trans. Inf. Sys.*, 36(1):2:1–2:33, 2017.

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

[31] G. E. Pibiri and R. Venturini. Variable-byte encoding is now space-efficient too. *CoRR*, abs/1804.10949, 2018.

[32] J. Plaisance, N. Kurz, and D. Lemire. Vectorized VByte decoding. *CoRR*, abs/1503.07387, 2015.

[33] A. Stepanov, A. Gangolli, D. Rose, R. Ernst, and P. Oberoi. SIMD-based decoding of posting lists. In *Proc. CIKM*, pages 317–326, 2011.

[34] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

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

[36] A. Trotman. Compression, SIMD, and postings lists. In *Proc. Aust. Doc. Comp. Symp.*, page 50, 2014.

[37] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Soft.*, 11(1): 37–57, 1985.

[38] J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson. MILC: Inverted list compression in memory. *PVLDB*, 10(8):853–864, 2017.

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

[40] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, 1999.

[41] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410, 2009.

[42] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, pages 387–396, 2008.

[43] Z. Zhang, J. Tong, H. Huang, J. Liang, T. Li, R. J. Stones, G. Wang, and X. Liu. Leveraging context-free grammar for efficient inverted index compression. In *Proc. SIGIR*, pages 275–284, 2016.

[44] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comp. Surv.*, 38(2):6.1–6.56, 2006.

[45] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, page 59, 2006.