# Index Compression Using Byte-Aligned ANS Coding and Two-Dimensional Contexts

Alistair Moffat
The University of Melbourne
Melbourne, Australia

Matthias Petri
The University of Melbourne
Melbourne, Australia

## ABSTRACT

We examine approaches used for block-based inverted index compression, such as the OptPFOR mechanism, in which fixed-length blocks of postings data are compressed independently of each other. Building on previous work in which asymmetric numeral systems (ANS) entropy coding is used to represent each block, we explore a number of enhancements: (i) the use of two-dimensional conditioning contexts, with two aggregate parameters used in each block to categorize the distribution of symbol values that underlies the ANS approach, rather than just one; (ii) the use of a byte-friendly strategic mapping from symbols to ANS codeword buckets; and (iii) the use of a context merging process to combine similar probability distributions. Collectively, these improvements yield superior compression for index data, outperforming the reference point set by the Interp mechanism, and hence representing a significant step forward. We describe experiments using the 426 GiB gov2 collection and a new large collection of publicly-available news articles to demonstrate that claim, and provide query evaluation throughput rates compared to other block-based mechanisms.

## KEYWORDS

Index compression; inverted index; postings list; entropy coder; asymmetric numeral systems

**ACM Reference Format:**
Alistair Moffat and Matthias Petri. 2018. Index Compression Using Byte-Aligned ANS Coding and Two-Dimensional Contexts. In *WSDM 2018: The Eleventh ACM International Conference on Web Search and Data Mining, February 5–9, 2018, Marina Del Rey, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3159652.3159663

## 1 INTRODUCTION

The inverted index remains the primary structure used to provide fast querying to large text collections. An inverted index consists of a set of postings lists, each describing the locations and occurrences in the collection of a single term. Proposals for compactly storing postings lists include byte-aligned codes [22, 25]; word-aligned codes [2, 3, 23, 28]; and binary-packed approaches [12, 30].

The use of entropy-based approaches such as Huffman codes for index compression have also been considered in the past [6, 11, 17, 26], but have had little attention since. However the last eight years have seen reawakened interest in entropy coding as a result of the *asymmetric numeral systems* (ANS) mechanism developed by Jarek Duda [8–10], and in recent work we combined whole-of-index two-pass ANS coding with the VByte, Simple, and Packed approaches, obtaining improved compression effectiveness in each case [14].

Here we continue that investigation, focusing on packed arrangements in which fixed-length blocks of symbols are coded relative to a probability distribution identified by a *selector*. In particular, in the Packed+ANS approach we described [14], the selector was taken to be the binary magnitude of the largest value in the block. But – as was also observed in connection with the *packed frame of reference* (PFOR) mechanism [30] – using other selector values, and being willing to handle some sequence values as *exceptions*, has benefits. The flexibility provided by entropy-coding (rather than binary coding) then creates scope for improved compression, with symbol probabilities (and hence codeword lengths) fitted more closely to the underlying distribution. Entropy coding also allows further refinements, and one that we explore in detail in this work is the use of two-dimensional selectors, with two parameters (rather than one) used to summarize each block's distribution of values. We also introduce a revised mapping from symbol identifiers to ANS values that requires less memory, and provides byte-friendly output for exception values. The third area we investigate is that of context merging, with the goal of reducing the memory space required during decoding operations.

Experiments with two large document collections demonstrate the substantial compression effectiveness gains that can be achieved – consistently beating the Interp mechanism, the best-performing approach for nearly twenty years now – and as well show that query throughput speeds are relatively unaffected.

In the interests of reproducibility, all of our code is publicly available.

## 2 BACKGROUND

### 2.1 Index Compression

**Inverted Indexing.** A *postings list* is a sorted sequence of document identifiers ("docids") and corresponding within-document term frequency counts. The postings list for a term $t$ of document frequency $f_t$ consists of two disjoint sequences: a list $\langle d_{t,i} \mid 1 \le i \le f_t \rangle$ in which $d_{t,i}$ is the ordinal document number containing the $i$ th occurrence of $t$; and a corresponding list $\langle f_{t,i} \mid 1 \le i \le f_t \rangle$ in which $f_{t,i}$ is the number of times $t$ appears in document $d_{t,i}$. It is usual to take *gaps* within the sequence of docids, with $d_{t,1}$ stored unchanged, and then $d_{t,i+1} - d_{t,i}$ stored thereafter. This yields a distribution in which small values are common and large values are rare. The set of $f_{t,i}$ values already has the same property, without any further transformation being required.

**Compression Approaches.** Stored as 32-bit integers, each posting requires 64 bits. But the strong bias in favor of small values means that much more compact arrangements are possible. Traditional techniques such as Golomb, Rice, and Elias codes (see Witten et al. [26] for details) operate on a bit-by-bit basis, and are relatively slow during the decoding process. Compromises that allow faster decoding, but with reduced compression effectiveness, include the byte-based VByte mechanism [22, 25] and variants thereof [4, 5, 7]; and the word-based Simple approaches [2, 3, 23, 28]. At the other end of the scale, the Interp mechanism of Moffat and Stuiver [15] provides very good compression effectiveness, but with even slower decoding than the bit-based Golomb and Elias codes. Standard byte-oriented compression libraries can also be employed, with improved effectiveness achieved if VByte is used as a preprocessing step [20].

**Packed Codes.** The use of *packed* codes over fixed-length postings blocks has been a relatively recent development, with the key idea being to exploit any localized consistencies that exist. A fixed-length block of (typically) $B = 128$ consecutive values is processed as a unit, and represented as $B$ same-width binary values, with the width indicated by a *selector* stored in the block header. In most arrangements a set of possible bit lengths is provided in advance via a set of options $S$. For example, the 16-element selector vector

$$S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 19, 22, 25\}, \qquad (1)$$

allows values up to $2^{25}$ to be represented, with (since $|S| = 16$) selectors represented in 4-bits. Each selector value $0 \le \ell < |S|$ allows values in the range $1 \ldots 2^{S[\ell]}$ to be represented in the corresponding *payload*; overall, a block of $B$ posting values coded via the selector $\ell$ requires $4 + S[\ell] \cdot B$ bits. Lemire and Boytsov [12] explore packed codes in detail and discuss implementation options. Trotman's [23] QMX approach combines the Simple and Packed approaches, using output blocks of either 128 bits or 256 bits.

**Patched Frame of Reference.** Zukowski et al. [30] noted that occasional larger-than-expected values are expensive with plain packed arrangements, and introduced the notion of *exceptions* – numbers bigger than $2^{S[\ell]}$ handled via a secondary mechanism. Their *patched frame of reference* (PFOR) approach chooses a bit-width $S[\ell]$ that covers most, but not all, of the $B$ values in the block. The minority of values needing more than $S[\ell]$ bits are represented via a secondary mechanism, and "patched" into place once the rest of the block has been regenerated using the $S[\ell]$-bit codes. To determine each block's $\ell$, in the Opt-PFOR variant an encode-time search is performed over likely values for $\ell$, to determine the selector that yields the most compact overall package [12, 27].

**Variable-Length Blocks.** Ottaviano et al. [18, 19] describe techniques for partitioning the input into blocks of variable length, seeking to amortize the selector storage cost over larger input units when homogeneous data appears, and to allow smaller blocks to be constructed when the data values are more locally volatile. They employ and extend Elias-Fano codes, a mechanism with a range of properties, including the ability to provide support for fast intersection of postings lists, to carry out conjunctive "and" operations, see also Anh and Moffat [1]. Ottaviano and Venturini make their experimental code available, and we have employed their platform in some of the experimentation described in Section 4.

**Other Approaches.** Zhang et al. [29] and Pibiri and Venturini [21] have also recently considered two-pass index compression techniques. Our work here is complementary to those proposals.

## 2.2 Asymmetric Numeral Systems

The "asymmetric numeral systems" (ANS) entropy coding technique developed by Jarek Duda [8–10] is a new way of performing entropy coding. It fulfills the same role as the previous Huffman and arithmetic methods, see Moffat and Turpin [16] for descriptions, in that it assigns low bit-costs to high-probability symbols, assigns high bit-costs to low-probability symbols, and transitions between the two extremes in accordance with the *information content* of the symbol in question. With all of Huffman, arithmetic, and ANS coding, a symbol $s$ with occurrence probability $Pr(s)$ should be assigned a codeword of length as close as is possible to $-\log_2 Pr(s)$.

**Example of ANS Coding.** Our initial exploration of ANS coding for index compression [14] provides a detailed example of ANS coding, pseudo-code descriptions, and an explanation that shows why ANS is an effective entropy coder. Duda provides extensive technical information about the processes he invented [8–10].

As an example, consider a three-element coding situation over an alphabet in which symbols P, Q, and R have (respectively) probabilities 1/2, 1/3, and 1/6. In a Huffman code, P gets a 1-bit codeword, and Q and R get 2-bit codewords. In an ANS-based coder for this arrangement, a *frame* of $M$ elements is constructed in which 1/2 of the entries are labeled P; 1/3 are labeled Q; and 1/6 are labeled R. A sequence of symbols over that alphabet is represented as a *state* value relative to the corresponding frame. One frame that captures the probabilities is [P][P][P][Q][Q][R], with $n(\mathrm{P}) = 3$, $n(\mathrm{Q}) = 2$, $n(\mathrm{R}) = 1$, and $M = 6$. Other frames might also be constructed, using other permutations of the symbols, and/or larger values of $M$.

Once the frame has been determined, it defines an encoding transition function $T(\cdot, \cdot)$, as shown in Table 1. The consecutive integer targets of $T$ are assigned in cycles defined by the frame, with 1/2 of them in row P; 1/3 of them in row Q; and 1/6 of them in row R. The columns of $T$ are indexed by a *state* variable, which takes on an initial value of 0. To encode one symbol $s \in \{\mathrm{P}, \mathrm{Q}, \mathrm{R}\}$, the assignment $state' \leftarrow T(s, state)$ is applied; and to encode a whole sequence each symbol in turn is encoded, with *state* becoming larger at each step. For example, given PPQP, *state* takes on the values $0 \rightarrow 1 \rightarrow 2 \rightarrow 10 \rightarrow 20$. The final value of *state* completely encodes the whole input string, and all that is necessary is to represent it

| | | Current *state* | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | ··· |
| P | 1/2 | 1 | 2 | 3 | **7** | **8** | **9** | 13 | 14 | 15 | 19 | 20 | 21 | 25 | 26 | 27 | ··· |
| Q | 1/3 | 4 | 5 | **10** | **11** | 16 | 17 | 22 | 23 | 28 | 29 | 34 | 35 | 40 | 41 | 46 | ··· |
| R | 1/6 | 6 | **12** | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60 | 66 | 72 | 78 | 84 | 90 | ··· |

**Table 1:** ANS encoding function $T(\cdot, \cdot)$ for alphabet {P, Q, R}, and probabilities 1/2, 1/3, and 1/6; and frame [P][P][P][Q][Q][R] of size $M = 6$. One cycle of the frame has been highlighted to show the regular structure within the table, covering next states 7 to 12.

as a binary number. In this case, five bits are sufficient. A Huffman code for the same input also happens to require five bits.

To decode, the inverse function $T^{-1}(\cdot)$ is used, with iterations of $(s, \mathit{state}') \leftarrow T^{-1}(\mathit{state})$ applied. For example, suppose the code 41 is received via a 6-bit binary value. It can only have arisen if the final encoding step was $\mathit{state}' \leftarrow T(\mathsf{Q}, 13)$, and hence the last symbol in the original sequence must have been Q. That previous *state* value of 13 must similarly have resulted from $\mathit{state}' \leftarrow T(\mathsf{P}, 6)$, meaning that a P must have preceded that final Q; by the same logic, an R must have come before the P. By stopping when a *state* of zero is reached, the initial sequence can always be uniquely recovered, in reverse order; in this case, it was RPQ. Note how a larger value for *state* might in fact correspond to a shorter input sequence; that difference is a direct consequence of the products of the probabilities of the symbols contained in the two sequences. In the case of PPQP, it has (among all four-symbol sequences) a probability of $1/(2{\times}2{\times}3{\times}2) \approx 0.0416$; whereas RPQ gives $1/(6 \times 2 \times 3) \approx 0.0277$.

As a more compelling example, the sequence QQQQQQQQQQ (ten Q's) gives rise to a *state* of 118,096, which requires 17 bits as a binary number; compared with a Huffman code of 20 bits. On the other hand, the sequence RRRRRRRRRR (ten R's) gives a *state* of 72,559,410, and needs 27 bits, ten bits more. This probability-based differential is why, over non-trivial sequences, or even short sequences in which the probabilities are highly skewed, ANS outperforms Huffman coding.

**ANS Compared.** Relative to Huffman and arithmetic coding, ANS offers compromises in a several dimensions, making it attractive for index compression. First, it shares with arithmetic coding the ability to closely match the ideal codeword lengths, and when amortized over a sequence of symbols, generates codes in which less than one output bit occurs for very high-probability symbols. In contrast, Huffman coding always requires at least one bit per symbol.

Second, ANS implementations are faster than arithmetic coding, and can approach the throughput rates attained by highly-engineered Huffman implementations. In each ANS decoding cycle, the integer *state* variable is modified by performing one shift, one bitwise AND, one integer multiplication and one table lookup in a relatively compact decoding table.

As a third area of compromise, ANS coding is a *static* mechanism, and requires a dedicated output stream for each *context*, where a context is one instance of a set of probabilities used to predict the symbol frequency distribution. This contrasts with adaptive arithmetic coding, which operates at throughput rates broadly comparable to static (with fixed probabilities) arithmetic coding, allowing (for example) a "q" to be transmitted using one inferred set of probabilities, and then the next symbol after it (in written English, usually a "u") to be transmitted in a fraction of a bit via a different context employing different inferred probabilities. That flexibility is not possible with ANS codes. Indeed, the ANS decoder generates output symbols in reverse order to that in which they were considered by the encoder, meaning that the symbols in each ANS message block must be regarded as being independently drawn from one context.

Fortunately, the ANS inability to be adaptive and to dynamically switch between contexts does not affect index compression.

**ANS in Practice.** Table 1 suggests that the mapping $T(\cdot, \cdot)$ is infinite, but that was simply to allow the general idea to be conveyed. The assignment of values based on a repeating $M$-element frame, means that $T(\cdot, \cdot)$ and $T^{-1}(\cdot)$ can be computed via integer arithmetic and table lookups against fixed-length arrays. In particular, fast and practical ANS encoding and decoding of sequences of symbols relative to an alphabet of $m$ symbols and using a frame size of $M$ requires two integer arrays of size $m$, and one of size $M$, where $M$ is typically $8m$ or more (that is, using three or more bits of precision for each of the symbol probability estimates making up the frame), and is also usually chosen to be a power of two.

## 2.3 ANS for Index Compression

In our previous investigation [14] we coupled ANS with three different index compression approaches. In particular, we paired ANS entropy coding with the VByte approach to compressing postings lists; with the Simple family of codes; and with the Packed family of index representations. We showed that in each case substantially more compact storage could be attained, with only moderate decreases in decoding throughput. In each scenario, two passes over the entire index are made during encoding: one to collect symbol occurrence counts so that ANS frames can be constructed, and then a second pass to generate output relative to those frames.

Our work here extends and improves on that first description of the "Packed+ANS" mechanism. In particular, we develop the notion of "two parameter" contexts; we describe a more compact implementation that requires substantially less memory; and we add a "context merging" process. To demonstrate the utility of these techniques, we also provide detailed experimental results for compression effectiveness and query throughput using two large text collections.

## 3 IMPROVED PACKED INDEX COMPRESSION

### 3.1 Packed+ANS

In standard Packed arrangements a context identifier, or selector, is used to condition the coding arrangement for each block. When the coding step makes use of binary codes, the selector is determined by finding the maximum value in the block, determining its binary magnitude $m$, and then searching a vector like the one shown in Equation 1 to find the first value $\ell$ such that $m \le S[l]$. The $\ell$th context is then used, which in traditional Packed arrangements gives rise to a payload containing $S[l]$-bit values.

In the Packed+ANS mechanism [14] these various elements are less tightly connected. The selector is again determined based on the maximum value in the block; but now identifies one of a pool of $|S|$ ANS frames. Each frame contains a different set of symbol probabilities, constructed during a first pass that processes the set of postings lists to accumulate statistics and collate the $|S|$ probability distributions. Symbol frequencies are accumulated in *binary-magnitude buckets*, and all symbols in each such bucket – for example, $5 \ldots 8$, $9 \ldots 16$, and $257 \ldots 512$ – are assumed to be equi-probable. That is, if $s_k$ is the $k$th symbol in the sequence $\sigma$ being coded as a set of blocks, and $\ell_k$ is the selector value computed from the largest binary magnitude across the symbols in the same block as $s_k$, then

$$L[\ell, b] = |\{s_k \in \sigma \mid \ell_k = \ell \text{ and } \lceil \log_2 s_k \rceil = b\}|$$

is a matrix of symbol magnitude counts within contexts. The probabilities used in the set of ANS frames – one for each context – are then computed from these occurrence counts. Working with equiprobable buckets avoids the need for the ANS frames to contain (potentially) $m \approx 30{,}000{,}000$ elements each.

Given this starting point, we now introduce a sequence of refinements that lead to improved compression and reduced space requirements, without compromising decoding speed.

## 3.2 Two-Dimensional Contexts

The first enhancement is to add further contexts. Doing so increases the cost of storing the selector, but allows more precisely-targeted probability distributions to be used. One way of adding contexts would be to employ a more fine-grained selector vector $S$, for example by using every power of two rather than a selected subset of them. Another option would be to alter the base of the logarithms, so that the buckets are smaller. But both of these options would continue to provide "one dimensional" conditional probabilities, in that the selector would still be completely determined by the maximum value in the block. A key benefit of using entropy coding rather than binary coding is that other options can also be considered.

In particular, we propose that "two dimensional" selectors be used, allowing two distinct observations about each block to influence the probability distribution used to code that block. The same mapping vector $S$ (perhaps the one described by Equation 1) is used for both observations, to form a compound context identifier. For example, suppose that the first dimension is the block maximum and the second the block minimum. Then a context "$max{:}min$" implies that all of the values in the block lie in the range $\lfloor 2^{S[min-1]} \rfloor + 1 \cdots 2^{S[max]}$. Trimming probability assignment to zero for smaller values is straightforward with an entropy coder such as ANS. Zukowski et al. [30] also describe the possibility of assigning a "lower" value to each block, and then exception-coding any values smaller than that value, in the same way that values larger than the upper value of the block are handled as exceptions. None of the current PFOR implementations actually employ this flexibility, primarily because the use of binary codes makes such range-adjustment relatively ineffective unless $max = min$.

Other features might also be used as the second selector component, including the block median (denoted "med"). In Section 4.2 we describe experiments using both max:min and max:med selectors, and also selectors derived from the 90th percentile value in the block, taking inspiration from the PFOR mechanism [30]. With ANS-based coding, any value $z$ can be coded relative to any selector $\ell$ provided $z$ has been assigned a non-zero count $n_\ell(z) > 0$ in the $\ell$ th ANS frame, and all that is being adjusted with these alternative approaches is the size of the set of possible contexts, and the blocks that get assigned to them for processing. That is, the fact that a first pass counts frequencies and constructs the matrix $L[\cdot,\cdot]$ means that there is no need for exceptions and patching in the sense in which they are used in the PFOR and OptPFOR approaches.

Table 2 illustrates the possibilities opened up by the switch from one-dimensional contexts to two-dimensional contexts, taking as test data the term frequencies associated with the postings lists of the gov2 collection (see Section 4.1). The distribution of block counts across the "Totals" row reflects the arrangement that would have arisen via a one-dimensional categorization in which a single-component selector was derived from the maximum value in each block. The percentage breakdown above each of those totals then shows how those blocks are further sub-categorized by the addition of a secondary selector based on the median value in the block. For example, 57% of the 5,720,796 blocks for which the maximum value lies between 17 and 32 (column 5) also have a median of 1 (row 0), and should thus correspond to coding distributions in which the probability of "1" is at least 0.5. Of the same set of blocks only 8% have medians of 3 or 4 (row 2). Across the full set of 16 columns and 16 rows, 94 of the available $|S|(|S|+1)/2 = 136$ contexts occur, and a 7-bit selector is required. In total there are 41,157,912 blocks in this data file, of which 8,697,157 contain all "1"s, and are handled in the context "0:0".

| *selector*(*med*) | *selector*(*max*) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 100 | 32 | 50 | 62 | 66 | 57 | 42 | 30 |
| 1 | | 68 | 9 | 9 | 17 | 27 | 33 | 33 |
| 2 | | | 42 | 4 | 5 | 8 | 14 | 20 |
| 3 | | | | 25 | 3 | 3 | 6 | 9 |
| 4 | | | | | 10 | 1 | 2 | 4 |
| 5 | | | | | | 4 | 1 | 2 |
| 6 | | | | | | | 2 | 1 |
| 7 | | | | | | | | 2 |
| Totals ('000) | 8697 | 3219 | 3319 | 4933 | 6011 | 5721 | 4336 | 2680 |

**Table 2:** Two dimensional contexts: percentage of blocks matching a selector for the maximum value in the block (columns), broken down by the selector for the block median (rows), for the filtered gov2.frqs file (see Section 4.1), with a blocksize of $B = 128$. Selectors are relative to the vector $S$ shown in Equation 1; the last row gives block counts in thousands for each column. Only the first eight columns and rows are shown.

## 3.3 Reducing The Frame Size

One of the implementation drawbacks of the Packed+ANS arrangement described in Section 3.1 is the cost of maintaining the frames in the three-table form employed for ANS decoding (see Section 2.2). With $m$ as large as $2^{25}$ in some frames, with $M \approx 8m$, and with the two-dimensional approach meaning that the number of contexts might be close to $|S|(|S|+1)/2$, execution-time memory space is a key factor that cannot be ignored. Caching effects mean that memory space also affects decoding speed. To reduce the space required, Moffat and Petri [14] describe an additional mechanism that uses condensed tables of size $\log_2 m$ elements, but additional computation is required during the indirection. A more subtle disadvantage of the Packed+ANS arrangement arises when two-dimensional contexts are introduced: it is no longer possible to always normalize the frame counts so that $M_\ell$ is a power of two.

One way of addressing these issues is to use ANS for the binary magnitudes of the symbols (and not their full numeric values), in the range (say) $0 \ldots 25$, and to associate a binary offset with each coded magnitude, stored in an auxiliary bit-stream. The decoder would process the ANS stream and the auxiliary bit-stream in parallel, decoding a magnitude from the first and then fetching the indicated

```
1:  function decode_packedANS_block(bytes[], B, ℓ):
2:  state ← final encoding state for this block
3:  b ← 0                                        ▷ offset within bytes
4:  for j ← 0 to B − 1 do
5:      syms[j] ← decode_ANS(bytes, b, ANSframe[ℓ], state)
6:  for j ← 0 to B − 1 do
7:      if syms[j] ≤ $2^8$ then
8:          continue                             ▷ syms[j] is now final
9:      syms[j] ← ((syms[j] − $2^8$) << 8) + bytes[b++]
10:     if syms[j] ≤ $2^{16}$ then
11:         continue                             ▷ syms[j] is now final
12:     syms[j] ← ((syms[j] − $2^{16}$) << 8) + bytes[b++]
13:     if syms[j] ≤ $2^{24}$ then
14:         continue                             ▷ syms[j] is now final
15:     syms[j] ← ((syms[j] − $2^{24}$) << 8) + bytes[b++]
16: return syms[0 . . . B − 1]
```

**Figure 1:** Decoding a block *bytes*[] of compressed data, relative to a selector $\ell$, to reconstruct a block of $B$ original values *syms*[], assuming that the encoder has processed the input block in reverse order. The function *decode_ANS*$(\cdot,\cdot,\cdot,\cdot)$ increments $b$ as the array *bytes* is consumed, returning decoded ANS values one by one, and altering the value of *state*. The completion bytes associated with values greater than 256 are then used to adjust the decoded values, inverting the mapping shown in Equation 2.

number of bits from the second, thereby reconstructing each value. This combination is somewhat reminiscent of the Elias $\gamma$ code, but with the unary "magnitude" part represented instead via an ANS code based on observed probabilities [11]. However, the auxiliary bit-stream introduces additional alignment overheads; and the bit-by-bit processing it implies is also costly.

A better arrangement is to employ an auxiliary stream that consists of whole bytes. To achieve that goal, we adapt the idea of *exceptions*, used in the PFOR and OptPFOR mechanisms, and define the following mapping between symbol values and ANS targets:

$$ANSmsb(s) = \begin{cases} s & \text{if } s \le 2^8 \\ \lfloor s/2^8 \rfloor + 256 & \text{if } 2^8 < s \le 2^{16} \\ \lfloor s/2^{16} \rfloor + 512 & \text{if } 2^{16} < s \le 2^{24} \\ \lfloor s/2^{24} \rfloor + 768 & \text{if } 2^{24} < s \le 2^{32} \end{cases} \quad (2)$$

Further stages are readily added if $s$ can be larger than $2^{32}$.

The key idea is to isolate the most significant byte of $s$, and map it to a unique number that captures both its value and also its position within the original word. A total of $m = 1024$ different options can be generated in this way from a 32-bit word. That is, ANS frames of at most 1024 different values are required (rather than 30,000,000), with all remaining bits provided via an auxiliary byte-stream that contains the zero, one, two, or three low-order bytes of each original value. Figure 1 describes the corresponding decoding and inverse mapping process.

Compared to the binary-magnitude ANS arrangement the space required is many orders of magnitude smaller; as well, more precise probability distributions are used across the entire range of values, potentially boosting compression effectiveness. For example, under

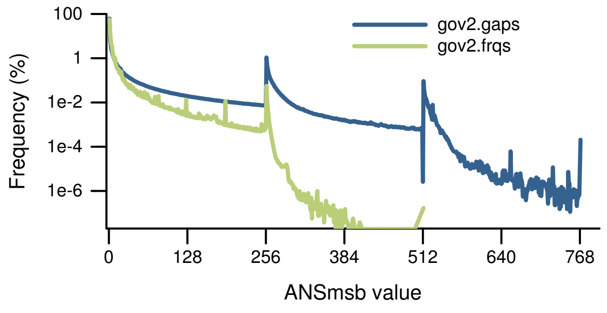

**Figure 2:** Probability distributions generated by *ANSmsb*$(\cdot)$ for the two sets of symbols $s$ making up the 5,268,212,736 postings in the filtered gov2 inverted index (see Section 4.1). For the purposes of this example, a single zero-dimensional context is assumed to be used to generate each of the two symbol sequences.

the binary magnitude arrangement described in Section 3.1, symbols 17 to 32 are assigned the same estimated probability. But with the mapping shown in Equation 2, they are treated as independent symbols and receive separate probability estimates during the first pass. Small ANS frames also lead to faster execution because of caching effects.

To illustrate the usefulness of the proposed mapping, Figure 2 plots the mapped distributions for the docid gaps and frequencies associated with the gov2 text collection (see Section 4.1), assuming that a zero-dimensional context is used, that is, one ANS frame for the docid gaps and another for the frequencies. The saw-tooth pattern is a direct consequence of the definition of *ANSmsb*$(\cdot)$, with mapped value 257 representing all symbols between 257 and 512 inclusive, and accounting for more than 150 times the probability mass of symbol 256 in the docid gaps file, and 90 times in the frequencies file. Results for measured compression effectiveness when contexts constructed using the *ANSmsb*$(\cdot)$ approach are applied in two-dimensions are given in Sections 4.2 and 4.3.

## 3.4 Context Merging

One risk associated with adding further contexts is that the selector requires additional bits. That is, having too many contexts might be as inefficient as not having enough. Given that the context selector is being coded into a binary value as part of the metadata associated with each block, it also makes sense to ensure that the number of contexts employed is a power of two such as 32 or 64.

These targets can be achieved by pairwise *merging* of selected contexts, replacing them by a single one. Suppose that symbol $s$ occurs $n_a(s)$ times in context $a$ which has a total occurrence count of $N_a = \sum_{s=1}^{m} n_a(s)$ across the $m$ different symbols being coded, and assume that a perfect entropy coder is available. Then the total cost of that context is

$$H_a = \sum_{s=1}^{m} \left( n_a(s) \log_2 \frac{N_a}{n_a(s)} \right) , \quad (3)$$

with $0 \cdot \log(N_a/0)$ taken to be zero. The net cost of fusing contexts $a$ and $b$ and replacing them by a context $v$ constructed via $n_v(s) = n_a(s) + n_b(s)$, for $1 \le s \le m$, is then given by $H_v - H_a - H_b$. This quantity can be computed for all possible pairs of $a$ and $b$, and the least-cost pairing chosen, reducing the number of contexts by one. The compute-select-fuse sequence can then be repeated in a greedy manner (including, perhaps, further mergings on contexts formed by previous fusings) until the desired number of contexts is reached. A small mapping table from previous selector value to new fused selector value must also be constructed.

## 4 EXPERIMENTS

### 4.1 Experimental Context

**Datasets.** Table 3 lists the four test files used in our experiments, derived from two large test collections. The first test pair, gov2.gaps and gov2.frqs, represent a document-level inverted index for the 426 GiB gov2 collection[1], with documents in URL-sorted order. A standard processing pipeline including Apache Tika and Apache Lucene is used to build the index, covering in excess of five billion postings. The second pair of test files are derived from publicly available web-sourced news articles[2], taking English language news sources (as identified by Apache Tika) from 01/09/2016 up until and including 28/02/2017, that is, a six month crawl period that contains 7,508,082 documents. Document identifiers were again assigned in URL-sorted order. We use this second file (rather than, for example, ClueWeb data) because of its unrestricted availability, allowing other researchers to readily reproduce our experiments.

The arithmetic mean of the symbols $s$ in each of the four files is shown in Table 3, as well as the geometric mean. The ratio between these two values (broadly) indicates the degree to which each distribution of symbols is biased in favor of small values.

**Hardware and Software.** All methods are implemented using c++11 and compiled with gcc 6.3.0 (using all optimizations) on a Linux server equipped with 148 GiB RAM and an Intel E5640 processor. In the interests of reproducibility, the experimental framework and all implementation details are available at http://github.com/mpetri/partitioned_ef_ans, including scripts for downloading and constructing the news collection; and we note with gratitude the previous software provided by Ottaviano and Venturini [18], which provides a foundation for our own experimental evaluation.

In terms of baselines, we have three: the OptPFOR mechanism from the FastPFor library [12]; the EF-opt approach of Ottaviano and Venturini [18]; and Interp. Zhang et al. [29] and Wang et al. [24] do not provide public implementations of their work; and while the QMX approach of Trotman [23] provides good compression effectiveness and very fast decoding, the quad-word alignment expectation of the version that we had available to us at the time we carried out our experiments was not well-suited to short sequences such as the blocks of $B = 128$ symbols considered here.

**Measurement and Methodology.** We measure compression effectiveness in three different ways. First, to obtain preliminary estimates of relative performance we employ Equation 3, summing over all contexts, and adding the cost of the necessary selectors. Section 4.2 gives computed effectiveness results based on this framework. Those results do not include any other block- or list-related overheads such as alignment costs, pointers, block maximum values, ANS frame storage costs, and coding inefficiency relative to entropy; that is, they are nothing more than estimates of costs relative to a set of probability distributions, and should only be compared against each other.

Having used estimation to explore the spectrum of improvements, in Section 4.3 we give confirmed compression results developed by full implementations of two Packed+ANS methods and two baseline mechanisms, all measured using the framework developed by Ottaviano and Venturini [18]. That is, in Section 4.3 the results are all-inclusive and reflect the complete cost of placing the compressed postings lists on permanent storage using demonstrably reversible compression software, including the ANS frames.

Finally, in Section 4.4, we measure the execution-time memory footprint required during querying, including all required auxiliary and metadata, term vocabulary search structures, pointers to postings lists, plus memory-mapped storage of the compressed postings lists themselves. (The measured costs do not include the block-max scores that support WAND pruning; these account for 0.19 GiB for gov2, and 0.13 GiB for news.)

Section 4.4 also reports query execution times. For these we employ the TREC 2005 Terabyte Track efficiency queries filtered such that all query terms occur in both test collections. In total, 44,368 queries (out of 50,000) were retained. Each of those queries was then executed using the BM25-based Ranked-AND and WAND top-$k$ retrieval methods to identify $k = 10$ top-scoring documents, as described by Ottaviano and Venturini [18].

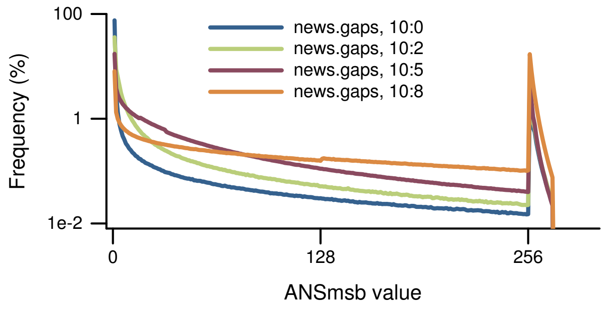

**Figure 3:** Probability distributions arising with the *ANSmsb* when two dimensional selectors are used on the filtered news.gaps. In each of the four plotted contexts the upper selector is 10; the lower selector is computed from the block median value. Blocks with different medians have different probability distributions.

### 4.2 Estimated Improvement

Figure 3 plots the probability distributions associated with four of the 122 contexts generated by the max:med selectors for the file news.gaps. All contexts have 10 as the max selector, that is, blocks in which the maximum is between 1025 and 4096, as described by Equation 1, with corresponding *ANSmsb*$(\cdot)$ values as large as 272. The four different medians that are plotted give rise to four different probability distributions, and illustrate the usefulness of allowing the dispersion of the block values, represented by the block median, to influence the choice of ANS frame.

[1]http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm.
[2]http://commoncrawl.org/2016/10/news-dataset-available/

| File | Unfiltered Index | | | | Filtered Index | | | |
|---|---|---|---|---|---|---|---|---|
| | Lists | Postings | Arith. | Geom. | Lists | Postings | Arith. | Geom. |
| gov2.gaps | 25,283,415 | 5,415,967,741 | 64,760.28 | 3.53 | 665,987 | 5,268,212,736 | 1452.78 | 2.90 |
| gov2.frqs | 25,283,415 | 5,415,967,741 | 4.23 | 1.85 | 665,987 | 5,268,212,736 | 4.31 | 1.87 |
| news.gaps | 26,240,031 | 4,457,492,131 | 24,678.69 | 3.14 | 273,978 | 4,375,559,040 | 356.50 | 2.74 |
| news.frqs | 26,240,031 | 4,457,492,131 | 2.17 | 1.48 | 273,978 | 4,375,559,040 | 2.19 | 1.49 |

**Table 3:** Test files used: number of postings lists, total number of postings $n$, and the arithmetic $(\sum_i s_i)/n$ and geometric $2^{(\sum_i \log_2 s_i)/n}$ means of the symbols stored in each file; for complete and filtered indexes. In the filtered indexes, postings blocks of fewer than $B = 128$ values are excluded. The filtered gov2 indexes contain 97.27% of the unfiltered postings; the filtered news indexes contain 98.16%.

| | gov2.gaps | gov2.frqs | news.gaps | news.frqs |
|---|---|---|---|---|
| Packed+ANS | 3.016 | 2.134 | 2.788 | 1.629 |
| 2d, max:min | 3.013 | 1.848 | 2.811 | 1.345 |
| 2d, 90%:med | 2.914 | 1.769 | 2.692 | 1.235 |
| 2d, max:med | 2.927 | 1.771 | 2.695 | 1.252 |
| *ANSmsb* | 2.909 | 1.727 | 2.680 | 1.224 |
| *contexts* $\leq 64$ | 2.903 | 1.719 | 2.673 | 1.216 |
| $\lvert S\rvert = 24$ | 2.902 | 1.719 | 2.673 | 1.216 |

**Table 4:** Estimated compression effectiveness derived from symbol frequency distributions (Equation 3), in bits per symbol, for four filtered test files, and (all but last row) starting with a selection vector $S$ of length 16. In each of the final three rows the modifications of previous rows are also included, adding to the "2d, max:med" row. A blocksize of $B = 128$ is used throughout. Except for the selector, no per-block overheads are included in these estimates.

Table 4 shows the evolution of computed entropies achieved for the four data sequences associated with the two filtered indexes. All values are expressed as entropy-based (Equation 3) bits per symbol estimates, and include the cost of the binary selector required in each block. The first row reflects (in estimated terms) the Packed+ANS mechanism (Section 3.1); each row thereafter adds another of the enhancements.

The use of two-dimensional max:min contexts brings a clear improvement in compression for the two frequencies files, and also brings modest gains in the two docid gaps files. Shifting to the 90%:med contexts yields further small gains in compression for all four files; and use of the max:med selector combination also improves on the max:min ones. The max:med selectors are a better operational choice than the 90%:med ones, and are used as the basis for the remainder of Table 4. Shifting from binary-magnitude ANS coding to the *ANSmsb* approach provides small gains in effectiveness (the fifth row) as well as the space savings that were the primary motivation; and then trimming the number of contexts to 64 (the sixth row) also brings small net gains in compression effectiveness. Finally, in the seventh row, the number of selectors is increased from 16 (shown in Equation 1) to 24, prior again to the greedy reduction to 64 contexts; the absence of consistent gain demonstrates that the vector in Equation 1 is a good initial choice.

Table 4 suggests that in combination the improvements described in Section 3 give rise to effectiveness gains of around 5% in the docids, and a remarkable 25% for the frequencies. The next section gives measured compression effectiveness results for the "packed, ANS, 2d, max:med, *ANSmsb*, merged contexts" method of the sixth row, and compares that preferred combination – denoted Packed+ANS2 – with baseline methods to validate the gains.

## 4.3 Measured Improvement

We now describe and measure a full implementation (including reversible decompression), using the same filtered postings files.

The ANS coder implemented as part of this work uses a 63-bit internal *state* value, and emits 32-bit binary words when *state* reaches overflow point; the encoder also reverses the ordering of the input block, so that decoded symbols are generated in left-to-right order, as is supposed by Figure 1. Hence, the ANS data stream consists of a sequence of 32-bit words, and a final *state* of between 1 and 8 bytes. To accommodate this, each block of compressed data Packed+ANS2 has the following structure:

- a 16-bit word that contains packed binary values for:
  - the selector to be used for this block (6 bits);
  - the number of bytes in the final value of *state* (3 bits);
  - the number of 32-bit words in the ANS component of the output block (7 bits);
- between 1 and 8 bytes recording the decoder's initial *state*;
- a sequence of 32-bit ANS words, stored in order of decoder consumption (by function *decode_ANS*$(\cdot,\cdot,\cdot,\cdot)$ in Figure 1);
- a sequence of completion bytes in regard to the *ANSmsb* process, stored in the order required by the decoder (Figure 1).

The 10 additional bits of control data (the 6 selector bits are already accounted for in the previous subsection) represent a non-trivial overhead. The rounding of each block's final ANS state to a byte boundary also introduces some penalty. As well, slight non-perfectness in the entropy coder adds to the measured bit cost.

Table 5 shows measured compression effectiveness on the four test files using two Packed and two reference approaches, and shows the strong gains in compression effectiveness that have been achieved. As expected, the Packed+ANS and Packed+ANS2 compression rates are around 0.1 bits per symbol higher than the computed entropy values in Table 4. But even so, the sequence of improvements made to Packed+ANS result in Packed+ANS2 having compression consistently better than that of Interp, a notable achievement.

| | gov2.gaps | gov2.frqs | news.gaps | news.frqs |
|---|---|---|---|---|
| VByte | 8.519 | 8.021 | 8.455 | 8.002 |
| OptPFOR | 3.621 | 2.391 | 3.292 | 1.757 |
| Packed+ANS | 3.131 | 2.248 | 2.867 | 1.717 |
| Interp | 3.047 | 2.245 | 2.918 | 1.684 |
| Packed+ANS2 | 2.979 | 1.830 | 2.734 | 1.323 |

**Table 5:** Measured compression effectiveness for four filtered test files, using blocks of $B = 128$ symbols. All compression rates are for reversible full-index coding, including per-block metadata and byte alignment overheads, and (for the two new methods) the cost of storing the ANS frame descriptions. The Packed+ANS2 mechanism employs 2d max:med contexts (Section 3.2), the *ANSmsb* approach (Section 3.3), and merging to 64 contexts (Section 3.4).

| | gov2 | | | news | | |
|---|---|---|---|---|---|---|
| | GiB | msec/q | msec/q | GiB | msec/q | msec/q |
| VByte | 11.053 | 5.9 | 11.4 | 9.050 | 4.4 | 8.4 |
| OptPFOR | 4.591 | 6.2 | 12.0 | 3.235 | 4.1 | 8.0 |
| Packed+ANS | 4.134 | 17.2 | 29.7 | 2.976 | 9.5 | 16.3 |
| EF-opt | 4.114 | 7.7 | 13.2 | 2.761 | 5.1 | 8.6 |
| Interp | 4.000 | 28.4 | 41.7 | 2.793 | 20.7 | 33.8 |
| Packed+ANS2 | 3.678 | 17.1 | 29.4 | 2.596 | 8.8 | 15.1 |

**Table 6:** Execution-time memory required and measured query throughput, using the unfiltered (all postings, see Table 3) version of each index. The first column in each group shows the total space required during querying operations, including all index data and storage overheads; the middle column shows the measured query rate for top-10 conjunctive ranked queries; and the third column in each group shows the measured query rate for disjunctive WAND queries, again identifying ten documents.

## 4.4 Retrieval Throughput

Table 6 shows retrieval throughput (average milliseconds per query) and the complete query-time in-memory index size (in GiB, unfiltered postings lists, excluding WAND data), as measured by the framework of Ottaviano and Venturini [18], for the test queries described earlier. The relativities for existing methods reflect those reported by Ottaviano and Venturini, and we can confirm that EF-opt has slightly larger space requirements than Interp, while providing much faster retrieval performance. The OptPFOR approach slightly outperforms EF-opt in terms of query throughput, but at the cost of more space, again matching the results reported by Ottaviano and Venturini. The well-known VByte mechanism [25] is fast, but not competitive in terms of memory space.

The initial Packed+ANS mechanism is better than other fixed-block approaches, but is outperformed by the variable-block EF-opt approach. The improved Packed+ANS2 reverses that situation in terms of memory space required, but not for query throughput. It does, however, outperform Interp in both dimensions. For example, in the case of gov2, the index is 8% smaller than Interp, and query performance is up to 40% faster.

The news collection behaves somewhat differently. Of the approaches tested, Packed+ANS2 again has the smallest memory footprint, with query performance closer to that of EF-opt and OptPFOR, caused by differences in compressibility between the two collections. For gov2, 13%/3.8% of all gaps/frequencies processed at query time are in uniform blocks that get decoded without ANS steps being required. For news, it is 33%/3.5% respectively. Of the remaining postings, 31%/28% are decoded using small models ($M \leq 256$) for gov2, against 28%/70.5% for news, meaning that a greater fraction of the ANS frames being used are retained in L1/L2 cache. At 2.94 MiB, the cumulative size of the decoding models for each of the collections is non-trivial, but individual ANS frames are very small if they only encode a few unique symbols.

Finally in this section, note that ANS encoding is fast: on our test hardware constructing the EF-opt index for gov2 requires 765 seconds using 16 threads, and OptPFOR requires 566 seconds using a single thread; while the two passes of Packed+ANS2 take 350 seconds using a single thread.

## 5 CONCLUSIONS AND FUTURE WORK

Starting with the Packed+ANS mechanism, we have improved it in three quite different ways: two-dimensional contexts, the most-significant byte mapping, and context merging. In doing so, we have shown that the two-decade-old benchmark set by Interp can be consistently beaten, a milestone outcome for index compression.

At present the Packed+ANS2 implementation is slower than OptPFOR and EF-opt in terms of query processing speed. However, there is considerable scope to improve the decoding throughput of the ANS-based approaches. We have already observed that most postings are encoded with compact models which span only a small number of unique symbols. For these, tailored ANS coders that employ expanded decoding tables and SIMD mechanisms to decode multiple states in parallel might increase decoding speed by a factor of four or more. Open-source entropy coders employing SIMD have already been shown to attain improvements of this magnitude[3].

We also observe that use of variable-length blocks formed via a strategy similar to that employed by EF-opt [18] and in other recent work [13] might lead to further compression gains when coupled with the new Packed+ANS2 mechanism.

As a third area for future work, we note the potential for different compression mechanisms to be used for different postings lists depending on querying statistics [19], and the possibility of relatively small compression losses to be exchanged for substantial query throughput gains.

**Postscript.** After submitting this work for review, and with the assistance of Giulio Pibiri and Rossano Venturini, we were able to also measure the performance of their recent clustered partitioned Elias-Fano approach [21] using our test files. In particular, we built a clustered partitioned Elias-Fano index for the gov2 collection using the standard parameters $\langle 24707817, 15540, 3, 5, 5, 8, 10 \rangle$ and a reference list size of 800,000, and generated an index of 2.82 GiB after 5449 seconds of computation. This index is 23% smaller than the Packed+ANS2 result shown in Table 6, somewhat muting our pleasure at having beaten the benchmark established by Interp.

[3]https://github.com/jkbonfield/rans_static

However, Pibiri and Venturini [21] show that compressing the same clustered postings list using Interp results in an index that is at most 5.6% larger than their method. Based on this result, we expect our ANS-based methods to be highly competitive when combined with the clustering approach, and we plan to examine that option next. Additionally, the complexity of the clustering process meant that construction time for the gov2 index was some 15 times greater than the Packed+ANS2 index, even though the former made use of 16 threads (rather than one in the case of Packed+ANS2).

We did not measure query performance for the clustered partitioned Elias-Fano index, but note that Pibiri and Venturini [21] report a 30–50% query performance penalty between the previous EF-opt mechanism and their clustered variant.

**Software.** The code related to the experiments described in this work is available at https://github.com/mpetri/partitioned_ef_ans and https://github.com/mpetri/TikaLuceneWarc. The software includes scripts to generate the news collection in ds2i format (see https://github.com/ot/ds2i). In the interests of reproducability we encourage other authors to similarly publish all of their experimental software.

**Acknowledgment.** This work was supported under the Australian Research Council's Discovery Projects funding scheme (project number DP140103256). We thank Giulio Pibiri and Rossano Venturini for assisting us with the use of their software.

## REFERENCES

[1] V. N. Anh and A. Moffat. Compressed inverted files with reduced decoding overheads. In *Proc. SIGIR*, pages 290–297, 1998.
[2] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.
[3] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Soft. Prac. & Exp.*, 40(2):131–147, 2010.
[4] 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.
[5] N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. SIGIR*, pages 139–146, 2008.
[6] Y. Choueka, A. S. Fraenkel, and S. T. Klein. Compression of concordances in full-text retrieval systems. In *Proc. SIGIR*, pages 597–612, 1988.
[7] J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *Proc. SPIRE*, pages 1–12, 2005.
[8] J. Duda. Asymmetric numeral systems. *CoRR*, abs/0902.0271, 2009.
[9] J. Duda. Asymmetric numeral systems: Entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *CoRR*, abs/1311.2540, 2013.
[10] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *Proc. Picture Coding Symp.*, pages 65–69, 2015.
[11] A. S. Fraenkel and S. T. Klein. Novel compression of sparse bit-strings: Preliminary report. In *Combinatorial Algorithms on Words, Volume 12*, NATO ASI Series F, pages 169–183. Springer, 1985.
[12] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Soft. Prac. & Exp.*, 45(1):1–29, 2015.
[13] A. Mallia, G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini. Faster BlockMax WAND with variable-sized blocks. In *Proc. SIGIR*, pages 625–634, 2017.
[14] A. Moffat and M. Petri. ANS-based index compression. In *Proc. CIKM*, pages 677–686, 2017.
[15] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.
[16] A. Moffat and A. Turpin. *Compression and Coding Algorithms.* Kluwer, Boston, 2002.
[17] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In *Proc. SIGIR*, pages 274–285, 1992.
[18] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proc. SIGIR*, pages 273–282, 2014.
[19] G. Ottaviano, N. Tonellotto, and R. Venturini. Optimal space-time tradeoffs for inverted indexes. In *Proc. WSDM*, pages 47–56, 2015.
[20] M. Petri and A. Moffat. Compact inverted index storage using general-purpose compression libraries. *Soft. Prac. & Exp.*, 2018. To appear.
[21] G. E. Pibiri and R. Venturini. Clustered Elias-Fano indexes. *ACM Trans. Inf. Sys.*, 36(1):2:1–2:33, 2017.
[22] A. Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, 2003.
[23] A. Trotman. Compression, SIMD, and postings lists. In *Proc. Aust. Doc. Comp. Symp.*, pages 50–57, 2014.
[24] 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.
[25] H. E. Williams and J. Zobel. Compressing integers for fast file access. *Comp. J.*, 42(3):193–201, 1999.
[26] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, 1999.
[27] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410, 2009.
[28] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, pages 387–396, 2008.
[29] 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.
[30] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, pages 59.1–59.12, 2006.