ANS-Based Index Compression

Alistair Moffat
The University of Melbourne
Melbourne, Australia
ammoffat@unimelb.edu.au

Matthias Petri
The University of Melbourne
Melbourne, Australia
matthias.petri@gmail.com

ABSTRACT

Techniques for effectively representing the postings lists associated with inverted indexes have been studied for many years. Here we combine the recently developed "asymmetric numeral systems" (ANS) approach to entropy coding and a range of previous index compression methods, including VByte, Simple, and Packed. The ANS mechanism allows each of them to provide markedly improved compression effectiveness, at the cost of slower decoding rates. Using the 426 GiB Gov2 collection, we show that the combination of blocking and ANS-based entropy-coding against a set of 16 magnitude-based probability models yields compression effectiveness superior to most previous mechanisms, while still providing reasonable decoding speed.

1 INTRODUCTION

The inverted index is an integral component of text search systems. In simplest form an inverted index consists of a set of *posting lists*, each of which contains a sequence $\langle d_{t,i}\rangle$, with $d_{t,i}$ the ordinal document number of the i th document containing term t. Each such posting might be accompanied by a corresponding $f_{t,i}$ value, the number of instances of t that appear in that document; and/or a list of locations within that document at which term t appears. One standard approach to storing postings lists – and other sequences of non-decreasing integers – is to transform them to *gaps*, and apply any integer compression regime that assigns short codewords to small values. A wide range of techniques have been used, including byte-aligned codes [19, 23]; word-aligned mechanisms [1, 2, 20, 26]; and binary-packed approaches [13, 28]. The same methods can also be applied directly to the $f_{t,i}$ values without any transformation being required, as they are also usually small integers.

There has been little application of entropy coders to index compression, despite a range of early proposals [5, 11, 16, 24]. This is partly a consequence of index data consisting primarily of small-valued integers with monotone-decreasing probability distributions; and partly a consequence of the complexity of and overheads associated with applying entropy-coders on a per-postings list basis.

In this paper we explore a recently-developed entropy-coding technique, *asymmetric numeral systems*, or ANS, and show that ANS can be usefully combined with several different index compression approaches, to yield improved compression effectiveness within reasonable additional resource costs. By joining ANS with each of byte-based codes, word-based codes, and packed codes, we establish new trade-offs for effectiveness and efficiency in index compression. Experiments on an inverted index for the 426 GiB Gov2 collection, covering five billion postings, support these claims.

Section 2 reviews index compression methods, and then provides an introduction to entropy-coding via asymmetric numeral systems. Section 3 then considers three index compression approaches, showing how ANS can be added to them to improve compression effectiveness. Section 4 describes the test environment, data, and methodology, and presents detailed compression and throughput results as a validation of our claims.

2 BACKGROUND

2.1 Index Compression Techniques

Definitions. We assume that a stream of integers $\sigma = \langle s_i\rangle$ is to be stored, with $s_i \in \Sigma = \{1, \ldots |\Sigma|\}$ for $1 \le i \le |\sigma|$. In the case of inverted index compression, the sequences are either the document identifiers ("*docids*") in the postings $\langle d_{t,i}, f_{t,i}\rangle$, or the frequencies associated with them. These two components are stored separately; with the docids also transformed to a sequence of gaps, $\langle d_{t,i} - d_{t,i-1}\rangle$, based on $d_{t,0} \equiv 0$. It is also assumed that σ is dominated by small values and that the distribution of symbol frequencies in σ is approximately such that $n(1) \ge n(2) \ge n(3)$ and so on, where $n(s)$ is an integer value that reflects the relative frequency of s, derived by counting occurrences in σ, or via a probability distribution supplied by an external estimation process.

Byte-Aligned Codes. In VByte compression [19, 23] each input integer s is partitioned into 7-bit fragments, and each fragment is placed into a byte with a flag bit that indicates whether there are further bytes yet to come. That is, if $s \le 2^7 = 128$ then a single byte suffices, with a flag bit of (say) 0. Values in the range 2^7+1 to 2^{14} are coded in two bytes, the first with a flag bit of (say) 1, and the second with a flag of 0. Decoding is performed by shift-or reassembly, continuing until a byte with a flag bit of 0 is reached, marking the end of this code. Standard 64-bit integer values might thus give rise to up to ten output bytes, but on average, for decreasing frequency symbol distribution, the average cost might be much less than that. No matter how biased the symbol distribution is towards low values, the average cost per symbol is at least eight bits. Other byte-aligned coding options have also been proposed [3, 4, 6].

Word-Aligned Codes. In the Simple representations fixed-length output words are formed, each consisting of a *selector* and a *payload*, with the selector in each word specifying the bit-packing arrangement in the corresponding K-bit payload. For example, Anh and Moffat [1] describe the Simple-9 approach in which words of 32

CIKM'17, November 6–10, 2017, Singapore.

ISBN 978-1-4503-4918-5/17/11…$15.00
DOI: https://doi.org/10.1145/3132847.3132888

bits are split into a 4-bit selector and a $K = 28$-bit payload. Nine different selector values allow for 1-, 2-, 3-, 4-, 5-, 7-, 9-, 14-, and 28-bit binary numbers to be placed into each payload; with one unused bit per word when 3 and 9 bit numbers are employed, and three unused bits when 5-bit numbers are packed.

A range of variants have been described. Zhang et al. [26] describe Simple-16, which improves on Simple-9 by adding additional non-uniform bit combinations that employ the remaining seven selector values, and by adjusting a subset of the packed binary lengths for 3-, 5-, and 9-bit codes, so that all bits in all of the packings are (at least potentially) used. Anh and Moffat [2] extend their initial method and describe the Simple-8b scheme, which employs 64-bit words, $K = 60$-bit payloads, and 14 different packings, determined via the integral quotients of 60. Anh and Moffat also note that long runs of ones often occur in index data, and allocate the two remaining Simple-8b selector options to extended runs of "1"s of two different lengths. A similar idea can also be added to the original Simple-9 mechanism, to allow a tenth selector value of 0, with the payload then a single binary integer indicating a run of up to 2^{28} ones. We will denote this enhanced approach as Simple-28-10, indicating the use of a 28-bit payload and 10 different selector values, including a runlength mode.

In all Simple methods the packing is determined by the magnitude of the largest value assigned to each payload. With the exception of the non-uniform arrangements of Simple-16, all of the other values are coded in that number of bits, regardless of whether or not they are of the same magnitude as the largest one. The usual approach when applying Simple is "greedy from the left" packing. Trotman et al. [21] observe that this is not optimal, but works well in practice.

Packed Codes. The Simple methods pack as many as possible values into a fixed-length payload. In the Packed approaches, fixed-length input blocks values are represented by variable-length payloads. The most elementary approach is to take fixed-length blocks of B input values, compute the maximum binary magnitude across the B values, and code each value in the block in that many bits.

More precisely, a vector $S = \{S[0], \ldots, S[|S|-1]\}$ of allowable binary magnitudes is employed, with each selector value $0 \le \ell < |S|$ allowing values in the range $1 \ldots 2^{S[\ell]}$ to be represented in the payload. 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, using a total of 16 different selectors. Note, as with the extended Simple codes, that the value $S[0] = 0$ indicates a repeat count (runlength-style) of "1"s in the input sequence, with, in this case, no corresponding payload – it is zero bits long. In the experiments described in Section 4 we refer to this approach using $|S| = 16$ different selectors as Packed-16.

Each output block stores a selector ℓ to indicate the bit-length $S[\ell]$ used for each binary value in the block; overall, the block requires $S[\ell] \cdot B$ bits. If B is a multiple of eight, each block's payload is thus a whole number of bytes long. In these arrangements selectors might also segregated, with (for example) eight four-bit selectors packed into a word, followed by eight payloads. Lemire and Boytsov [13] provide a detailed examination of such codes, including the use of SIMD instructions to accelerate decoding. Trotman's [20] QMX codec is a blend of Simple and Packed approaches, and also exploits SIMD operations during decoding.

Patched Frame of Reference. To reduce the disruption caused by isolated large values in Packed-like mechanisms, Zukowski et al. [28] introduced the *patched frame of reference* (PFOR) approach. A bit-width $S[\ell]$ is selected that covers most, but not all, of the values in the block, and is used to code those values. The values that require more than $S[\ell]$ bits are represented using a secondary mechanism, and "patched" after the main part of the block has been decoded. To determine each block's ℓ, a search over likely values is performed, a mechanism referred to as Opt-PFOR or OPF [13, 25].

Other Methods. Ottaviano and Venturini [17], Ottaviano et al. [18], Zhang et al. [27], and Wang et al. [22] have also recently considered index compression techniques in work that does not directly relate to the threads that we pursue here.

2.2 Asymmetric Numeral Systems

We now describe the "asymmetric numeral systems" (ANS) entropy coding technique developed recently by Jarek Duda [9, 10].

ANS Example. Figure 1 provides an example of ANS encoding against an alphabet of four symbols (to avoid confusion, denoted here as a, b, c, and d rather than 1, 2, 3, and 4) assumed to occur in a string σ with relative frequencies of $n(\mathrm{a}) = 7$ and $n(\mathrm{b}) = n(\mathrm{c}) = n(\mathrm{d}) = 1$. The sum of the relative symbol frequencies determines the *frame size* M; in the example, $M = 10$. A mapping table *symbol* converts each of the values in $1 \ldots M$ back to the corresponding symbol identifier, with $n(s)$ of the M elements in *symbol* containing s, for each of the input symbols s. The same number of instances of each symbol s is implicitly allocated into each of an infinite sequence of *frames*, always in the same order, as shown in the shaded box in the example. The mapping *base* indicates the offset in *symbol* of the first occurrence of each member of the alphabet Σ.

Using these structures, each string σ over the input alphabet is assigned a unique non-negative integer (its *state* value), commencing with 0 representing the empty string ϵ. To determine the state value associated with a string $\sigma \cdot s$, ANS first computes the state v_σ for the prefix string σ. The next state $v_{\sigma \cdot s}$ is then defined to be the ordinal position in the repeated frames (that is, in the shaded-background region in Figure 1) of the v_σ th occurrence of symbol s. For example, suppose that the string b has $v_{\mathrm{b}} = 8$. Then the string ba is mapped to the index of the 8 th occurrence of a across the frames, that is, $v_{\mathrm{ba}} = 12$, as listed in frame 1. Similarly, the string baa is mapped to the 12 th occurrence of an a across the frames, yielding $v_{\mathrm{baa}} = 16$. Finally, to confirm that v_{b} is indeed 8, as was initially claimed, note that the empty string ϵ is assigned $v_\epsilon = 0$ as a basis for the entire computation. It is the 8 th cell of the frames that stores the "0 th" occurrence of b, hence the statement earlier that $v_{\mathrm{b}} = 8$. Figure 2 extends this example, and also shows the state sequence associated with two other strings.

ANS Encoding. More generally, given an alphabet Σ with relative symbol frequencies given by $n(s)$ for $s \in \Sigma$, frames each of size $M = \sum_{s \in \Sigma} n(s)$ and each containing $n(s)$ instances of s in a regular pattern are formed; and the computation shown in the first part of

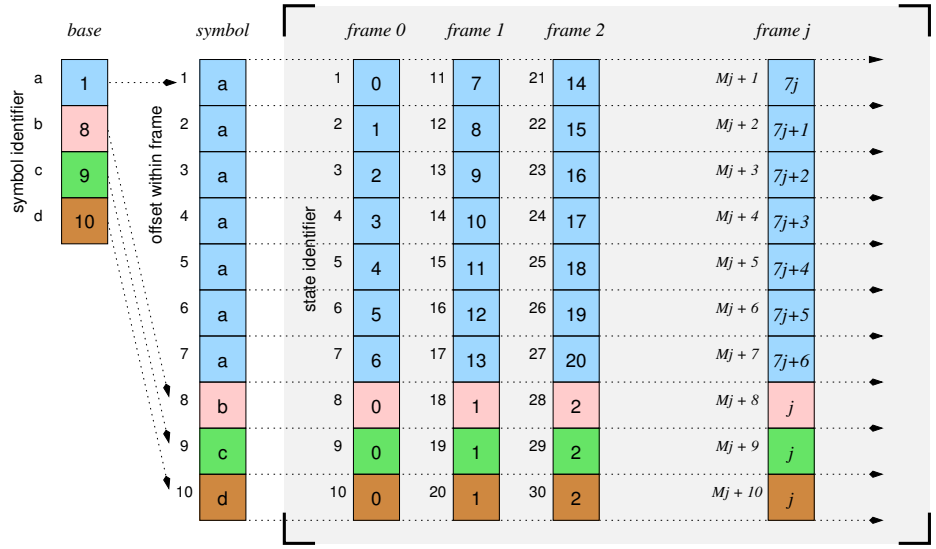

Figure 1: Example of ANS symbol allocation across frames. In this example the source alphabet contains four symbols, a, b, c, and d; with $n(\mathrm{a}) = 7$, and $n(\mathrm{b}) = n(\mathrm{c}) = n(\mathrm{d}) = 1$; and hence $M = 10$. In this "range ANS" approach the frames contain $|\Sigma|$ contiguous runs of symbols.

ϵ	$\rightarrow$	a	$\rightarrow$	aa	$\rightarrow$	aaa	$\rightarrow$	aaaa	$\rightarrow$	aaaad	$\rightarrow$	aaaada
0	$\rightarrow$	1	$\rightarrow$	2	$\rightarrow$	3	$\rightarrow$	4	$\rightarrow$	50	$\rightarrow$	72

ϵ	$\rightarrow$	b	$\rightarrow$	ba	$\rightarrow$	baa	$\rightarrow$	baaa	$\rightarrow$	baaac	$\rightarrow$	baaaca
0	$\rightarrow$	8	$\rightarrow$	12	$\rightarrow$	16	$\rightarrow$	23	$\rightarrow$	239	$\rightarrow$	342

ϵ	$\rightarrow$	b	$\rightarrow$	bd	$\rightarrow$	bdd	$\rightarrow$	bddc	$\rightarrow$	bddca	$\rightarrow$	bddcab
0	$\rightarrow$	8	$\rightarrow$	90	$\rightarrow$	910	$\rightarrow$	9109	$\rightarrow$	13013	$\rightarrow$	130138

Figure 2: Three examples of the ANS state values generated using the frames shown in Figure 1. The final value of *state* is larger for strings that contain low-$n(s)$ symbols.

Figure 3 is used to compute an integer *state* that uniquely characterizes any input string $\sigma \in \Sigma^*$. In the pseudo-code, j is the frame number in which the *state* th occurrence of symbol s must lie, with the exact address being based on the starting point for that frame (given by $j \cdot M + 1$), the remainder r, and the starting point $base(s)$ at which the instances of s appear in each frame.

Commencing the *symbol* mapping at an index of 1 (rather than 0 as in previous descriptions [9, 10]) provides an automatic incorporation of a runlength mode – when $|\Sigma| = 1$ and $n(1) = M$, the *state* value that is computed for an input σ that contains $|\sigma|$ "1"s is exactly the value $|\sigma|$.

ANS Effectiveness. To appreciate the effectiveness of the ANS approach, consider the final value of *state* for a string σ of length M that contains $n(s)$ instances of each symbol $s \in \Sigma$, that is, a string that matches the normalized symbol distribution. The arithmetic that takes place at steps 4 and 6 as each symbol s is processed in the first part of Figure 3 multiplies *state* by approximately $M/n(s)$:

$$state \approx \prod_{s_k \in \sigma} \frac{M}{n(s_k)},$$

1: // Encode the string σ
2: $state \leftarrow 0$
3: **for** each symbol $s \in \sigma$ **do**
4: $\quad j \leftarrow state$ **div** $n(s)$
5: $\quad r \leftarrow state$ **mod** $n(s)$
6: $\quad state \leftarrow j \cdot M + base(s) + r$
7: **return** *state*

1: // Decode the sequence in *state*
2: $\sigma \leftarrow \emptyset$
3: **while** $state > 0$ **do**
4: $\quad r \leftarrow 1 + (state - 1)$ **mod** M
5: $\quad j \leftarrow (state - r)$ **div** M
6: $\quad s \leftarrow symbol(r)$
7: $\quad state \leftarrow j \cdot n(s) - base(s) + r$
8: $\quad \sigma \leftarrow s \cdot \sigma$
9: **return** σ

Figure 3: Outline of the ANS encoding (top) and decoding (bottom) processes, where $n(s)$ is the number of instances of symbol s in each frame; $M = \sum_s n(s)$ is the size of each frame; $base(s)$ is the first instance of s in each frame; and $symbol(r)$ is the symbol associated with the r th position in each frame.

which, as a binary integer, requires at least $\lceil \log_2 state \rceil$ bits, that is,

$$\Big\lceil \sum_{s_k \in \sigma} \log_2 \frac{M}{n(s_k)} \Big\rceil \approx - \sum_{s \in \Sigma} n(s) \cdot \log_2 \frac{n(s)}{M}$$

bits, which is the zero-order entropy of σ. Looking at the three rows in Figure 2, and assuming that *state* can be coded in $\lceil \log_2 state \rceil$ bits without further overhead, then the first example requires 7 bits for the six symbols, the second example requires 9 bits, and the third

example requires 17 bits; with the difference a consequence of the relative frequencies of the symbols comprising the strings. Duda [9, 10] gives a more precise evaluation of the effectiveness of ANS; the overall summary of that analysis is that ANS yields compression rates closer to those attained by arithmetic coding than to those attained by Huffman coding (see Moffat and Turpin [15]).

ANS Decoding. The decoding process, also shown in Figure 3, starts with a value for *state*, and reverses the encoding computation, building up the output string σ from right to left. At each iteration the next symbol s is determined by computing the position r within the frame associated with *state*, and using it to index the *symbol* mapping. Once s has been isolated, it is used to compute the frame location from which that symbol would have generated the current *state*; it becomes the *state* considered at the next iteration. Note that in the decoder the div and mod operations are relative to the fixed value M. If M is chosen to be a power of two they can be implemented using mask and shift operations, rather than division.

In terms of space, the mapping $base(s)$ contains $|\Sigma|$ entries, and (in simplest form, as suggested by Figure 1) the mapping $symbol(r)$ contains M entries. When M is large relative to $|\Sigma|$ the $symbol()$ mapping can be replaced by a linear or binary search in the *base* array, slowing decoding throughput, but reducing the memory footprint. Values of $n(s)$ can be readily derived from $base(s)$.

Variations. To conclude this brief introduction to ANS encoding, note that for simplicity Figure 1 assumes that the assignment of symbols within each frame is contiguous, allowing a simple n-element *base* mapping. The pseudo-code shown in Figure 3 also makes this assumption, an approach that Duda refers to as "range ANS", or rANS. But if an additional mapping is introduced between *base* and *symbol*, the $n(s)$ instances of symbol s can be arranged in any desired manner over the M locations in each frame; in particular, dispersing the high-$n(s)$ instances throughout the table rather than clustering them at the beginning leads to slightly better compression effectiveness [9, 10]. It is also possible to fully pre-compute all of the transitions implied by the arithmetic shown in the second part of Figure 3 and create a "table ANS" (tANS) decoder [9, 10]. Finally, note that there is no upper bound on the magnitude of the final value of *state* for a sequence. However, to avoid arbitrary-precision arithmetic, a modest modification of the encoding and decoding processes allows *state* to be emitted, and likewise consumed, in byte-sized chunks. That is, the computations described in Figure 3 can be carried out for sequences of arbitrary length in finite-precision integer arithmetic, provided that M is comparatively small relative to the word-length [9, 10]. We omit description of that extension here, and refer the reader to other resources, including on-line commentary by an ANS implementor[1].

3 EMPLOYING ANS IN INDEX COMPRESSION

We now consider ways in which ANS coding might be used in conjunction with existing index compression techniques. In all cases we consider one postings list to be the "unit" of compression, so that index access during query processing operations can be carried out by fetching and decoding exactly the lists for the terms involved in the query. If required, internal structure within postings lists can be added to any of these methods to support skipping operations and processing regimes such as WAND and BlockMaxWAND [7, 8].

3.1 Vbyte + ANS

Several public implementations of ANS are available, including the one embedded in the ZStd library[2]. These operate on a byte-alphabet, using an input alphabet of size $\Sigma = 256$, and creating a compressed byte-stream that is entropy-coded using a probability distribution derived from the byte sequence being coded. The cost of representing that frame – the parameters $n(s)$, as described in Section 2.2, which we can regard as providing a *model* against which that byte stream or any other byte stream can be coded – is an overhead that must also be factored in to the final compressed cost. In the case of ANS coding, a model consists of a set of $|\Sigma|$ integers summing to M, and can itself be coded using one of a range of techniques in $|\Sigma|(\log_2(M/|\Sigma|) + 1.5)$ bits [15]. When $|\Sigma| = 256$ and σ is long, the overhead is modest.

However, in the case of postings lists, even after document gaps have been computed, the docids input stream consists of integers in an arbitrary range, with values potentially as large as the number of documents in the collection. The stream of document frequencies is similarly unbounded, though less likely to be as large. That is, $|\Sigma|$ is of the order of 10^7 or greater. Moreover, postings lists – the unit of compression and decompression – are typically of the order of 10^3 to 10^4 elements. It is thus inappropriate to seek to ANS-code each postings list using its own probability distribution, nor to retain the full range of possible docid gap and frequency values and give each distinct symbol s an independent frame allocation $n(s)$.

To bypass these issues, the first ANS-based mechanism we consider is the application of VByte to reduce each input sequence (a single postings list) to two streams of bytes, one for the docid gaps and one for the frequencies, and then to use ANS to represent those streams of bytes. That is, two "whole of index" ANS models are developed in a pre-processing pass that applies VByte to each value and accumulates aggregate byte statistics, one for the docid gaps, and for the frequencies. The two ANS models are then applied to the VByte-transformed components of each postings list.

3.2 Simple + ANS

The risk of the VByte+ANS arrangement is that global models built across the whole index cannot capture local variability. On the other hand the drawback of the Simple approach described in Section 2.1 is that it assumes that neighboring values are of a similar magnitude, meaning that isolated large values are disruptive, and cause other nearby values to be represented in more bits than are actually needed. For example, in the subsequence "1, 2, 2, 1, 8, 2, 1, 2, 1", the "8" causes 3-bit codes to be used, even though all of the remaining values can be covered by 1-bit codes. That is, once a selector value has been determined, all values in each Simple payload are treated identically. The nine values in the example will thus consume (as a minimum) 27 bits. Even worse would be if that "8" was "8000", which has a magnitude of 13. Because $5 \times 13 = 65$, the "8000" forces a payload to be flushed if $K < 65$. That is, the first payload would contain just four values, each of which nominally requires just one bit, but each of which (in the case of 64-bit payloads) would cost to

[1] https://fgiesen.wordpress.com/2014/02/02/rans-notes/

[2] http://www.zstd.net

16 output bits; more even than the 13 bits required by the "8000". These complex interactions mean that occasional large values can be very costly in Simple.

Suppose that fixed-width 4-bit selectors are retained, but instead of binary codes, each payload is used to convey an ANS state relative to one of a set of $2^4 = 16$ different models, each with its own frame allocations $n(s)$. This introduces considerable flexibility: not only can the 16 models be based on different alphabet sizes $|\Sigma|$, which is what happens with Simple; they can also be based on different distributions of normalized symbol frequencies $n(s)$ and different values of M. Given a set of 16 such models, the encoder chooses for each output word the option that spans the longest prefix of the sequence being coded, exactly as in Simple's binary-code case. Returning to the example, suppose that one of the models has frame allocations $n(1) = 5$, $n(2) = 3$, $n(3) = n(4) = 2$, and $n(5) = n(6) = n(7) = n(8) = 1$, with $M = 16$. If that distribution were available, an ANS state of 550,004 would be generated, which consumes 20 bits and represents a clear saving compared to nine 3-bit codes. It is also perfectly possible for a frame with the right $n(s)$ distribution to span "1, 2, 2, 1, 8000" in one 64-bit payload.

Model Formation. An obvious challenge is to determine a palette of 16 alternative models. We again manage them on a whole-of-index basis, so that the cost of storing their descriptions is amortized, and propose the following heuristic approach, with σ taken to be the complete concatenated sequence of input integers (docid gaps, or frequencies), all of which are one or greater.

(1) Compute b_k, the binary magnitude of the symbol σ_k occurring in the k th position of σ, that is, $b_k = \lceil \log_2 \sigma_k \rceil$.
(2) Then, for each position k in σ, compute

$$\ell_k = \mathit{select}(\mathit{estimator}(b_{k-w} \ldots b_{k+w}), S)\,,$$

where the operation *select*() identifies the first value in the ordered set of integers S that is greater than or equal to the supplied argument, and where a $(2w + 1)$-element window of magnitudes surrounding b_k is used in an *estimator*() function that seeks to predict the context that will be used to code σ_k during the second pass.
(3) Vector S determines the magnitude buckets used during both passes, and $S[|S| - 1]$ must be at least as large as the largest magnitude b_k. For example, to mimic the ten different Simple-28-10 payloads described in Section 2.1, S would be:

$$S = \{0, 1, 2, 3, 4, 5, 7, 9, 14, 28\}\,. \tag{2}$$

Alternatively, if it is known that $|\Sigma| \le 2^{25}$ – the case in the experiments described in Section 4 – we might instead choose to employ the vector S described by Equation 1, and use a finer-grained set of 16 contexts.
(4) Next, compute $L[\ell, b]$, the count over the positions k in σ of the combinations of ℓ_k and b_k that arise, that is,

$$L[\ell, b] = |\{k \mid 1 \le k \le |\sigma| \wedge \ell_k = \ell \wedge b_k = b\}|\,.$$

(5) For each of the 16 contexts $0 \le \ell < 16$ a set of frame parameters $n_\ell(z)$ is computed for $1 \le z \le 2^{S[\ell]}$, defined such that, if $\lceil \log_2 z \rceil = b$, then $n_\ell(z) \propto L[\ell, b]/2^{S[b-1]}$. That is, for a given context ℓ and each value of $1 \le b \le \ell$, all symbols z of magnitude b are considered to be equally likely, and to have a frame count $n_\ell(z)$ derived from their joint aggregate count $L[\ell, b]$.
(6) The ANS frame associated with the ℓ th model is then of size $M_\ell = \sum_{z=1}^{2^{S[\ell]}} n_\ell(z)$. It is also desirable that M_ℓ be a power of two, a further constraint on the normalization process.

Once the set of 16 models has been computed, the compression is accomplished in a second pass through σ. As with all Simple-based approaches, a "greedy from the left" process is followed. Assuming that $\sigma_{1\ldots k-1}$ has been processed and committed to the output stream, the suffix $\sigma_{k\ldots}$ is checked against each of the 16 available contexts, and the one that spans the greatest number of symbols is identified and applied. The stopping condition for each context is straightforward: a span of symbols can only be fitted into a K-bit payload if the *state* that is calculated (step 6 in the encoding process described in Figure 3) is less than 2^K. Note that the context used at position k need not be the ℓ_k th one that was nominally tailored for it, because the context estimates were constructed heuristically, without information about the eventual coding alignments. The degree of correlation between estimated ℓ_k and actual selector used for σ_k will in part determine the compression effectiveness attained.

The goal of the construction process is to allow the context $\ell = 3$ to contain – if that is what the magnitude-based analysis of σ generates – the normalized counts $n_3(1) = 5$, $n_3(2) = 3$, $n_3(3) = n_3(4) = 2$, and $n_3(5) = n_3(6) = n_3(7) = n_3(8) = 1$ that were assumed in the example with which this section commenced.

Estimators and Normalization. One key to the success or otherwise of this approach is the quality of the *estimator*() function, and the degree to which it accurately generates a likely selector ℓ_k from the window in σ of size $2w + 1$. Suppose that the focus is on σ_k, and that b_k is its magnitude. We consider three different types of groups that include σ_k: ones where σ_k is the last element in the group; ones where σ_k is the first element in the group; and ones where σ_k is centered within the group. Any such group is *feasible* with respect to a payload size K if the maximum magnitude within the group multiplied by the size of the group (its *span*) is less than or equal to K. For each of the three group types there is a feasible group that has the longest span, beyond which they become infeasible. To form the estimate ℓ_k, the largest magnitude within each maximal-span feasible group is computed, for each of the three different types of group; and then the median of those three magnitudes is chosen. That magnitude m is then checked against the vector S in the *selector*() function, to determine ℓ_k such that $S[\ell_k - 1] < m \le S[\ell_k]$. Note that since σ_k is always in every group, it is certain that $b_k \le m \le S[\ell_k]$.

Figure 4 describes another component of the processing pipeline that takes place in the Simple+ANS first phase. Given a set $L[0 \le \ell \le \mathit{maxb}]$ of integer frequencies counting across binary-power bands of symbols, we require each $n(z) \ge 1$ value to reflect the implied probability of symbol z, such that $\sum_{s\in\Sigma} n(s) = M = 2^k$ for some integer value k. To do this, the *excess* between the current M and the next largest power of two is computed, and distributed across the levels, starting at *maxb*, and finishing at zero – at which time all remaining unspent *excess* must be assigned. Knowing that on average when z is small this will result in *increment* amounts that boost $n(z)$ by approximately 50%, a constant multiplier C that would nominally map $L[\mathit{maxb}]$ to 0.5 is applied first, with integer rounding then lifting it and other high-z values of $n(z)$ to the minimum $n(z)$ of one. This approach also has the benefit of ensuring that

1: $C \leftarrow 0.5 \times bandsize(maxb)/L[maxb]$
2: **for** $b \leftarrow 0$ **to** $maxb$ **do**
3: $\quad n(b) \leftarrow \lceil L[b] \times C / bandsize(b) \rceil$
4: $M \leftarrow \sum_{b=0}^{maxb} n(b) \times bandsize(b)$
5: $M' \leftarrow M$
6: $excess \leftarrow 2^{\lceil \log_2 M \rceil} - M$
7: **for** $b \leftarrow maxb$ **downto** 0 **do**
8: $\quad$ **if** $b = 0$ **then**
9: $\quad\quad increment \leftarrow excess$
10: $\quad$ **else**
11: $\quad\quad increment \leftarrow \lfloor n(b) \times excess / M' \rfloor$
12: $\quad excess \leftarrow excess - bandsize(b) \times increment$
13: $\quad M' \leftarrow M' - bandsize(b) \times n(b)$
14: $\quad M \leftarrow M + bandsize(b) \times increment$
15: $\quad n(b) \leftarrow n(b) + increment$
16: // M is now a power of two, and *excess* is now zero
17: **return** $n(b)$, for $0 \le b \le maxb$

Figure 4: Normalization to ensure that M is a power of two. The input vector $L[b]$ is a set of $maxb + 1$ integer frequencies, with $L[b]$ the combined frequency of a group of $bandsize(b)$ items sharing the b th band. On output the value $n(b)$ is the frame allocation for each of symbols in the b th band, and over the whole model their sum is $M = 2^k$ for some integer k.

$n(2^{maxb}) = 1$ in each distribution, and hence that the ranges used in the subsequent ANS coding step are as compact as possible.

Table 1 presents an example of the scaling and normalization processes, showing a set of initial band counts for 16 symbols across $maxb = 4$ bands, together with their inferred probabilities; then the set of initial $n(b)$ estimates after scaling by $C = 0.5 \times 8/52 = 0.077$ at step 3 in Figure 4; and third, the $n(b)$ values after M is increased to a power of two, and the final probabilities. Note how the normalized probabilities for the low-b symbols are boosted back towards their original values as the *excess* is distributed.

Condensed Tables. One issue that arises in the process as described is the very large *base* and *symbol* tables that would appear to be necessary when (for example) $S[\ell] \ge 30$ and $M_\ell \ge 2^{31}$. But the fact that all of the $n_\ell(z)$ values are equal across large defined ranges of z eliminates the need for explicit symbol-by-symbol storage. In particular, context ℓ can be stored in a small table of $S[\ell]$ rows, out of which the mappings *base* and *symbol* are calculated rather than indexed. That is, the total space required by the 16 models remains small, albeit with additional calculations required during encoding and decoding. Structuring the larger tables in this way requires that symbol ranges within the frame be contiguous, and that is why we make use of the range ANS coder rather than the other variants. Note also that context $\ell = S[0] = 0$ provides natural run-length codes. Because it has only one symbol, $z = 1$, with (after normalization) $n_0(1) = 1$ and $M_0 = 1$, the ANS process will fit up to $2^K - 1$ consecutive symbols of value "1" into a single K-bit payload.

Variations. The choice of payload size and number of contexts are both dimensions that are open to exploration. For example,

b	Band	Orig., $L[b]$		Scaled, $n(b)$		Norm., $n(b)$	
		Freq.	Prob.	Freq.	Prob.	Freq.	Prob.
0	1	74	0.297	6	0.240	10	0.313
1	2	33	0.133	3	0.120	4	0.125
2	3–4	52	0.104	2	0.080	3	0.094
3	5–8	38	0.038	1	0.040	1	0.031
4	9–16	52	0.026	1	0.040	1	0.031

Table 1: Example of the normalization process applied to a vector $L[0 \ldots 4]$, covering sixteen symbols. The initial value of M is 249, the scaled value of M is 25, and the normalized value of M is 32.

payloads of $K = 60$ bits would have internal selectors stored in the same word, or payloads of $K = 64$ bits might be stored relative to a separate sequence of selectors. In the latter case, when the selectors are separated from the payloads, the limit $|S| = 16$ also becomes arbitrary. For example, a set of 20 or 25 models might be preferred. The selectors themselves might then be ANS coded into a word containing a variable number of selectors, potentially generating further small effectiveness gains.

3.3 Packed + ANS

Even with the ANS code employed, it is likely that each Simple payload contains unused bits, since it is improbable that the ANS state value will exactly fill each K-bit payload. Indeed, each payload – with the exception of context $\ell = 0$, which codes runlengths of "1"s – will on average have unused bits corresponding to roughly half the number of bits in the average value coded into that payload.

In the Packed+ANS variant we reduce the overhead caused by those wasted bits, fixing the ANS input unit rather than fixing the ANS output unit. Uniform-length blocks of values are taken from the input sequence, and the maximum magnitude $\hat{b_k}$ of any of the items in the block is mapped to a selector via $select(\hat{b_k})$. A first pass over the blocks builds 16 ANS models and their corresponding frame parameters, one model for each magnitude as determined by the set of selectors, accumulating the probability distribution across the subset of the blocks that share that selector value. Then, in a second pass, those 16 models are used in a deterministic manner to represent the values within the blocks, again via $select(\hat{b_k})$. Each block of compressed data – represented as an extended ANS-coded variable-length payload – is independently decodable, just as each payload in the Simple+ANS version is independently decodable.

Compared to the Simple+ANS scheme, the Packed+ANS approach has two potential advantages: a smaller proportion of wasted bits because of the greater span of elements covered in each block and the byte-based (rather than word-based) output unit; and more certainty in regard to the probability distribution associated with each selector. The latter is a consequence of the block structure – there is a single value for the largest magnitude $\hat{b_k}$ that appears in each block, constant between the first counting pass and the second coding pass. The first pass that establishes the probability distributions does not involve an estimation process, it simply counts occurrences. The other aspects of the ANS process already described in connection with Simple+ANS are carried over to Packed+ANS.

Component	VByte+ANS	Simple+ANS	Packed+ANS
Input unit	term list	variable/greedy	fixed size
Output unit	term list	one word	variable
Preprocessing with VByte	yes	no	no
Selectors	none	16	16
Search for best model	no	yes	no

Table 2: Overview of ANS-enhanced compression approaches.

	Terms	Postings
Initially	25,285,522	5,416,085,998
Removed	24,619,529	147,761,262
Remaining	665,993	5,268,324,736

Table 3: Index data used for experimentation, before and after filtering. Partial blocks with fewer than 128 postings were removed. All other results given here pertain to the two filtered index files.

3.4 Comparison of ANS-Based Methods

Table 2 summarizes the ANS-enhanced index compression options we have described, and their points of similarity and difference. Section 4 then measures the effectiveness and efficiency of those three approaches. Many variants are possible, including those that have already been mentioned, such as increasing the size of S in the Simple+ANS and Packed+ANS approaches; using $K = 128$-bit payloads with Simple+ANS; using non-binary buckets when computing Simple+ANS and Packed+ANS "magnitudes" (they could be base ϕ, for example, with S containing the Fibonacci numbers); and applying VByte preprocessing to the Simple+ANS and Packed+ANS approaches.

4 EXPERIMENTS

We now examine the performance of the various ANS-enhanced schemes described in Section 3. We first describe the dataset and methodology, and provide baseline results. Three different ANS mechanisms are then explored in detail. The final subsection compares baseline and ANS-enhanced systems in terms of compression effectiveness, and encoding and decoding throughput. To gain an overview of those results, the eager reader may wish to browse Tables 5 and 7 prior to continuing through the next few subsections.

4.1 Experimental Context

Dataset and Methodology. Our emphasis in this work is on index compression techniques, and we have selected an experimental design accordingly. To construct a set of postings lists, a full inverted index of the approximately 25,000,000 $< 2^{26}$ documents in the 426 GiB Gov2 collection[3] was formed, containing all occurrences of all terms, based on a URL-sorted document ordering. Lists of fewer than 128 postings were then removed, on the basis that they could be represented using a common "secondary" mechanism such as VByte, regardless of how the longer lists were stored. In order to provide consistency with BlockMaxWAND processing, we further restricted lists to a multiple of 128 postings in each list, assuming that the "leftover" elements in partial final blocks could (and would) also be stored using the same secondary mechanism. These filtering steps reduced the index in the manner shown in Table 3; the majority of the postings lists were dropped, but only 2.73% of the postings themselves were removed. Table 4 lists the cost of storing those removed postings, expressed as bits per posting relative to the filtered index. If the full cost of any index is required, the appropriate values from Table 4 should added, and then the sum multiplied by 0.9727, to give a whole-of-index rate in bpp.

[3] http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm.

Method	docid.gaps	frequencies
VByte	0.536	0.317
Interp	0.527	0.227

Table 4: Additional amortized cost (bits per posting relative to the number of postings in the filtered index, bpp) to encode all postings removed by the filtering process described in Table 3, for the two Gov2 test files and two secondary methods.

Meta-data associated with the postings lists – such as the pointers in the vocabulary used to access the postings for each term, and so on, was assumed to be constant across compression methods, and is not included in any of the quoted compression rates. That is, the compression rates given below are measured over files containing one *nlists* value and followed by that many *list* elements; with each *list* element consisting of one *listlen* integer followed by that many *item* elements, either *docid gap* values or *frequency* values. Compression rates are in terms of bits per posting (*bpp*), where the numerator is the total cost (including alignment overheads) of storing each set of *item* elements in a manner that has each *list* byte-aligned (or greater) and independently decodable; and the denominator is the total number of *item* elements in the input sequence. That denominator was thus 5,268,324,736 (see Table 3) for both the docid.gaps test file and for the frequencies test file constructed from the Gov2 index.

Hardware and Software. All methods are implemented using c++11 and compiled with gcc 5.4.0 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/ans-list-compression.

The FastPFor library [13] is employed to provide efficient implementations of VByte, OPF and Simple-16. We additionally use the QMX implementation described by Trotman [20], adapted to fit our experimental methodology. We had also hoped to include the Opt-EF method of Ottaviano and Venturini [17], but were unable to resolve incompatibilities between their software and our experimental framework. Zhang et al. [27] and Wang et al. [22] do not provide public implementations of their work.

Baseline Effectiveness. Table 5 provides baseline effectiveness results for the two Gov2-derived test sequences. A range of implementations were used to obtain these results, adapted where necessary to present a uniform experimental interface and to ensure that the output sizes measured were consistent with the methodology described above. The preponderance of "1"s in both test files (3,260,187,271, or 61.9% in docid.gaps; and 2,975,880,945, or 56.5% in frequencies) means that the Simple-28-10 approach outperforms

Method	docid.gaps	frequencies
VByte	8.523	8.021
QMX	4.679	3.198
Packed-26, $B = 16$	4.324	2.840
Simple-16	4.227	2.945
OPF	4.193	3.002
Simple-28-10	3.903	2.543
Packed-16, $B = 8$	3.879	2.480
Interp	2.988	2.005

Table 5: Baseline compression (bpp) for the filtered Gov2 index, including a variant of Simple-9 in which a tenth selector option is added to handle runs of "1"s (as per Equation 2, using a $K = 28$-bit payload); and two versions of Packed, one using all 26 binary magnitudes and 8-bit selectors with $B = 16$-item blocks; and one using 16 contexts (Equation 1), $B = 8$-item blocks, and 4-bit selectors.

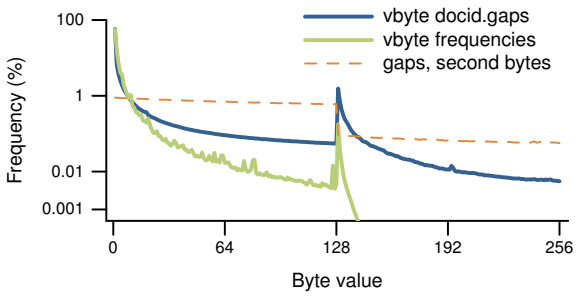

Figure 5: Frequency distributions of bytes emitted when the VByte transformation is applied to the docid.gaps and frequencies files for the filtered Gov2 index. The third line, denoted "gaps, second bytes", shows the distribution of byte values for the subset of non-first bytes in the transformed docid.gaps.

the Simple-16 implementation, and suggests that a blend might be better than both. The two Packed methods also make use of zero-selectors, and when they occur the compressed block contains no other information. At the top of the table, the domination of small values means that the VByte mechanism is relatively ineffective on this data; while at the other end the Interp mechanism of Moffat and Stuiver [14] sets the reference point for compression effectiveness.

4.2 ANS-Enhanced Compression Techniques

VByte + ANS. Figure 5 illustrates the application of the VByte transformation to the filtered Gov2 index, with the two solid lines the probability distributions over the bytes generated for the docid.gaps and frequencies test files. The process of generating byte sequences introduces a discontinuity at 128 into what are otherwise monotone decreasing symbol probabilities, but nevertheless, the distributions remains strongly skewed in favor of small values. Actual compression results based on applying ANS to the byte sequences for docid.gaps and frequencies are given in Table 7.

The dashed line in Figure 5 is the probability distribution across all "non-first" bytes in the VByte-transformed docid.gaps file. This category consists of the second and subsequent bytes for all original values that were greater than 128, and has a markedly different

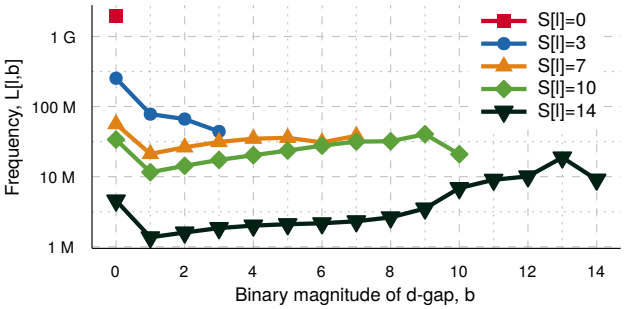

Figure 6: Counts $L[\ell, b]$, plotted for five different values of $S[\ell]$, for the filtered docid.gaps input file. A gap of "1" is the most likely scenario in every context.

distribution. That difference suggests that compression might be improved by separating the first bytes from the second and subsequent bytes, and coding the two byte streams using their own ANS probability distributions. However, the small fraction of bytes coded in the non-first distribution meant that in practice we were unable to exploit this effect, and the results given below are for a single VByte-based byte stream.

Simple + ANS. We now turn to the Simple+ANS hybrid. The combination used in these experiments makes use of 4-bit selectors choosing across 16 different ANS models described by Equation 1, and $K = 64$-bit payloads. That is, one 64-bit selector-only word is emitted prior to every 16 words of payload. Each payload word stores a single ANS state value in binary; with groups of input items parsed in a greedy-from-the-left manner. We use a window size of $w = 64$ for the estimation process.

Figure 6 shows values for $L[\ell, b]$ for the docid.gaps generated from the filtered Gov2 index, to quantify the spread of probabilities associated with a subset of five of the sixteen selectors. As ℓ increases, the accumulated frequencies both decrease in terms of absolute counts, and become more diffuse. But all of the distributions have "1" as the dominant element, and even in the $S[11] = 14$ distribution, it has a probability of 5.87%. It is this absolute prevalence of small values that suggests that the Simple+ANS mechanism can improve on Simple. Note that the increase in the counts towards the end of three of the plotted distributions is because the buckets double in size with each increase in b. In all of the models the normalized $n(z)$ frame allocation counts are non-increasing.

Table 6 shows the fraction of output words for docid.gaps coded using each of the 16 selectors, together with the fraction of postings they cover. Only 0.1% of the output words use selector $\ell = 0$, but it is a critically important one for compression overall, and nearly a third of all the docid gaps occur in long runs (averaging over 7000) of values that are all "1". Another 4.5% of the gaps are coded using selector $\ell = 1$. In that second model, a gap of "1" has a relative weight of 7, and gaps of "2" have a relative weight of 1; that is, the two possible symbols have estimated probabilities of 7/8 and 1/8 respectively. In the third model, associated with $\ell = 2$, a gap of "1" has a weight of 12 and an estimated probability of 12/16; a gap of "2" has a relative weight of 2 and an estimated probability of 2/16; and gaps of "3" and "4" have estimated probabilities of 1/16 each.

Selector value, ℓ	$S[\ell]$	Fraction of words	Avg. bits per sym.	Fraction of postings	Average unused bits
0	0	0.1%	0.01	31.9%	54.0
1	1	0.5%	0.34	4.5%	18.2
2	2	2.1%	1.12	6.1%	5.8
3	3	5.3%	2.10	8.2%	3.7
4	4	9.2%	3.17	9.6%	3.2
5	5	11.2%	4.15	8.9%	3.3
6	6	11.4%	5.06	7.4%	3.5
7	7	10.4%	5.87	5.8%	3.9
8	8	9.3%	6.94	4.4%	4.5
9	10	15.2%	7.12	7.0%	4.8
10	12	9.5%	10.77	2.9%	5.8
11	14	7.9%	13.80	1.9%	7.8
12	16	3.6%	15.11	0.8%	6.8
13	19	3.1%	19.02	0.5%	9.8
14	22	1.2%	21.79	0.2%	7.8
15	25	0.0%	21.42	0.0%	9.3
Overall	–	100.0%	3.28	100.0%	4.9

Table 6: Fraction of output words coded using each selector value for the filtered docid.gaps file with $K = 64$-bit payloads; the fraction of the postings encoded using each selector; and the average number of unused payload bits (that is, leading zero bits) after each *state* value was stored in binary. For the frequencies file the average span was 30.6, with 2.7 unused output bits per word on average.

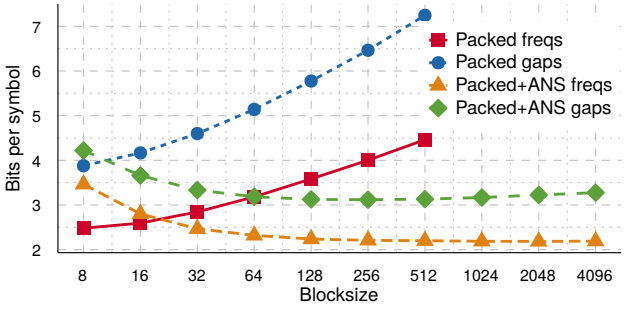

Figure 7: Compression effectiveness of Packed and Packed+ANS as a function of blocksize B. The ANS coding means that Packed+ANS provides good compression effectiveness over a range of block sizes.

We also measured the extent to which the predicted selector values ℓ_k, the basis of the 16 probability distributions, matched the actual selector values generated in the second pass by the greedy parsing. There was an overall strong (but by no means perfect) correlation between the estimated selectors and the actual selectors. For example, there were approximately 1.5×10^9 values in docid.gaps for which the estimation process indicated a likely selector ℓ_k of 0 and which were indeed coded using a selector of 0. In total some 51.20% of the ℓ_k s matched the actual selectors for docid.gaps, with a further large fraction in "off by one" categories. Even so, there were also many estimation failures, some by wide margins.

Packed + ANS. Figure 7 compares the compression effectiveness of the Packed and Packed+ANS approaches on the two filtered data

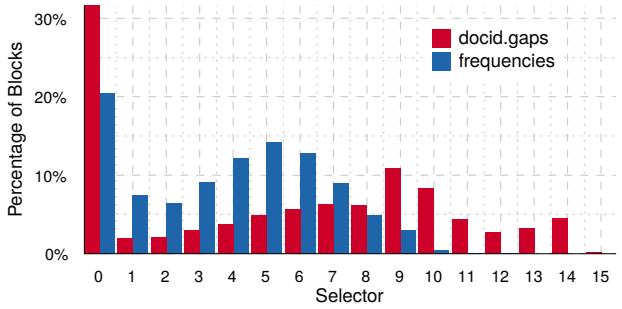

Figure 8: Selector utilization of Packed+ANS with $B = 256$ for both files. The docid.gaps file has a selector entropy of 3.41 bits per block; the frequencies file has a selector entropy of 3.18 bits per block.

Method	docid.gaps	frequencies
VByte+ANS	3.556	2.522
Simple+ANS, $K = 64$	3.282	2.235
Packed+ANS, $B = 256$	3.117	2.208

Table 7: Compression (bpp) for three ANS-enhanced techniques. Both Simple+ANS and Packed+ANS used 16 different contexts (Equation 1). These values can be compared with those shown in Table 5.

files, with 16 selectors employed (Equation 1) in all cases. When using Packed, small blocksizes give the best results, because any large elements force all of the codes to become longer. But the Packed+ANS approach is much less vulnerable to this problem, and good compression effectiveness is obtained over a wide range of block sizes B. In this regard, the use of ANS codes rather than binary codes (at least partially) obviates the need for OPF's exceptions – even when ℓ is relatively large, symbol "1" retains a high probability, and is represented compactly when it appears.

Figure 8 plots the distribution of Packed+ANS selectors for the docid.gaps and frequencies files when using the 16 selectors defined by Equation 1, and using $B = 256$-item blocks. The distribution for frequencies is more skewed towards low values than for docid.gaps, but there are more blocks in docid.gaps that consist entirely of "1"s than there are in frequencies.

4.3 Comparing Effectiveness and Efficiency

Table 7 gives compression effectiveness for the two test files, in a direct comparison with Table 5. Each of the "+ANS" mechanisms improves on the corresponding baseline, a very satisfying outcome. For example, the combined index cost of Packed+ANS of 3.12 + 2.21 = 5.33 bpp represents a 16% saving compared to the cost (Table 5) of 3.88 + 2.48 = 6.36 bpp for the previous Packed approach. On the other hand, none of the ANS-based methods outperform Interp (combined cost of 4.99 bpp), which suggests that further improvements in ANS-based index compression might still be possible – perhaps an Interp+ANS combination, for example.

Table 8 lists measured encoding and decoding speeds for a range of implementations, with the rows ordered by increasing decoding speed for docid.gaps, and the throughput rates computed for complete list-at-a-time memory-to-memory encoding of the entire

Method	docid.gaps		frequencies	
	Enc.	Dec.	Enc.	Dec.
Interp	55	59	55	59
Packed+ANS	60	82	61	148
Simple+ANS	3	89	3	160
VByte+ANS	54	92	60	105
VByte	582	492	719	613
OPF	14	524	21	650
Simple-16	197	547	258	647
QMX	87	1242	94	1176

Table 8: In-memory encoding and decoding speeds (Mibi-symbols per second) for the docid.gaps and frequencies test files.

filtered index, and then complete list-at-a-time memory-to-memory decoding of the entire filtered index. The encoding costs for all ANS variants includes extracting appropriate frequency counts over the postings lists and performing the necessary estimation and scaling processes; in the case of Simple+ANS, which has both an estimation process in the first encoding phase and a searching process in the second, encoding is very slow indeed. Encoding speeds for Packed+ANS and VByte+ANS are similar to Interp and OPF, and slower than the other methods. In terms of decompression speed, all of the ANS variants are slower than previous compression techniques, with the exception of Interp. That is, the three ANS-based techniques can be viewed as providing further effectiveness-efficiency tradeoffs, useful additions to the current spectrum that ranges from Interp through to QMX.

Our current implementations of ANS-based compression techniques omit several optimizations employed in industrial-strength ANS-based entropy coders[4]. For example, 64-bit word level state normalization and output compared to the byte-level operations currently used in our methods have been shown to increase decompression performance by around 25%. Similarly, interleaving decompression of multiple ANS states using SIMD instructions can increase decompression throughput by a factor of between 2 and 5 [12][5]. These algorithmic engineering techniques offer the potential of ANS-based sequence compression codecs with decoding throughput comparable with traditional integer compression approaches.

5 CONCLUSIONS

We have explored three different ways in which ANS-based entropy coding can be used in conjunction with existing index compression regimes. Our experiments show that substantial compression effectiveness gains can be achieved, and establish a platform on which other ANS-based combinations can now be considered. For example, we plan next to explore a VByte+Packed+ANS mechanism in which VByte is used to reduce the frame size range, rather than magnitude-based buckets; and will also consider the possibility of adding OPF-like exceptions to the current Packed+ANS version, to further focus the compression models. It may also be possible to combine ANS with Interp, and in doing so establish improved best-compression benchmarks. As noted in the previous section, there is also the engineering consideration of applying improved ANS techniques to our current implementation, to unlock efficiency and hence throughput improvements.

Software. In the interests of reproducibility our experiments can be rerun using the source code available at http://github.com/mpetri/ans-list-compression.

Acknowledgment. This work was supported under Australian Research Council's Discovery Projects funding scheme (project number DP140103256). Alexandru Tomescu (University of Helsinki) participated in a number of helpful discussions.

[4]See, for example, https://github.com/rygorous/ryg_rans.

[5]Advanced SIMD techniques are explored at https://github.com/jkbonfield/rans_static.

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] 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.
[4] N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. SIGIR*, pages 139–146, 2008.
[5] Y. Choueka, A. S. Fraenkel, and S. T. Klein. Compression of concordances in full-text retrieval systems. In *Proc. SIGIR*, pages 597–612, 1988.
[6] J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *Proc. SPIRE*, pages 1–12, 2005.
[7] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proc. WSDM*, pages 113–122, 2013.
[8] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. SIGIR*, pages 993–1002, 2011.
[9] J. Duda. Asymmetric numeral systems. *CoRR*, abs/0902.0271, 2009. URL http://arxiv.org/abs/0902.0271.
[10] J. Duda. Asymmetric numeral systems: Entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *CoRR*, abs/1311.2540, 2013. URL http://arxiv.org/abs/1311.2540.
[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] F. Giesen. Interleaved entropy coders. *CoRR*, abs/1402.3392, 2014.
[13] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Soft. Prac. & Exp.*, 45(1):1–29, 2015.
[14] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.
[15] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer, Boston, 2002.
[16] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In *Proc. SIGIR*, pages 274–285, 1992.
[17] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proc. SIGIR*, pages 273–282, 2014.
[18] G. Ottaviano, N. Tonellotto, and R. Venturini. Optimal space-time tradeoffs for inverted indexes. In *Proc. WSDM*, pages 47–56, 2015.
[19] A. Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, 2003.
[20] A. Trotman. Compression, SIMD, and postings lists. In *Proc. Aust. Doc. Comp. Symp.*, page 50, 2014.
[21] A. Trotman, M. Albert, and B. Burgess. Optimal packing in Simple-family codecs. In *Proc. ICTIR*, pages 337–340, 2015.
[22] 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.
[23] H. E. Williams and J. Zobel. Compressing integers for fast file access. *Comp. J.*, 42(3):193–201, 1999.
[24] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.
[25] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410, 2009.
[26] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, pages 387–396, 2008.
[27] 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.
[28] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, page 59, 2006.