Fast Dictionary-Based Compression for Inverted Indexes

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

Matthias Petri
The University of Melbourne
Melbourne, Australia

Alistair Moffat
The University of Melbourne
Melbourne, Australia

ABSTRACT

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

KEYWORDS

Inverted index; efficiency; compression; decoding

ACM Reference Format:

1 INTRODUCTION

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

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

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

2 BACKGROUND

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

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

Word-aligned codes are also possible. For example, in the Simple16 representation [1] fixed-length 32-bit output words are formed, each consisting of a selector and a payload containing some number of same-length binary codewords. Variants include work by Zhang et al. [42], who add the flexibility to employ patterns of codewords not all the same length; and by Anh and Moffat [2], who consider...
the use of 64-bit output units. The QMX mechanism of Trotman [36] also includes some elements of these approaches. Word-based codes can be thought of as being variable-to-fixed, since the compression is achieved by varying the size of the input fragments rather than the lengths of the codewords assigned.

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

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

Packed approaches are typically fixed-to-variable arrangements.

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

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

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

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

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

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

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

**3 BASE IMPLEMENTATION**

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

**The Dictionary.** There are two factors that make inverted index data highly distinctive. First, there are very long runs of “1”s (almost always the most frequent symbol in the alphabet) that create opportunities for the use of a *frequent symbol exception*, whereby long repetitive sequences are handled outside the normal regime. As is demonstrated in Section 5, upwards of a third of the docids and freqs in typical inverted index sequences are “1”s, and handling these economically is a key requirement. Second, the alphabet for docid gaps is very large, into the millions, and it is impossible to
consider providing a codeword for every symbol, even as a singleton. Instead, use must be made of rare symbol exceptions, a special code that indicates that the next symbol must be fetched from a secondary stream of uncompressed integers.

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

Rare Symbol Exceptions. To see the use of rare symbol exceptions, consider the dictionary shown in Table 1(b), in which only two singleton codes are provided. Using this table the same example an overall slight reduction in cost arises, primarily because any particular factor parsed from the source sequence might include part or all of other dictionary strings, affecting those counts. In the experiments reported in Table 2 and in Section 5 exhaustive sampling with $L = \ell'$ is used.

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

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

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

A more nuanced analysis leads to the second heuristic we explore. Suppose that some sequence $S$ is being considered to be placed in the dictionary. Given that $S$ is now a candidate, it seems likely that both its first half, denoted $S_1$, of length $|S|/2$, and its second half, denoted $S_2$, of length $|S|/2$, with $S = S_1S_2$, will already be in the dictionary. This is because (assuming interval sampling) $freq(S_1) \geq freq(S)$ and (via symmetry, but not guaranteed) that $freq(S_2) \geq freq(S)$. And if $S_1$ and $S_2$ are already in the dictionary, then the saving generated by also adding $S$ to the dictionary is only $freq(S)$, since one codeword will be used for each instance of $S$, rather than

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

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

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

In the experiments reported in Table 2 and in Section 5 exhaustive sampling with $L = \ell'$ is used.
The standard unit of access is a single block of \( B \) integers and we employ \( B = 256 \) throughout this investigation; that is, each postings list is partitioned into fixed-length blocks, with any remaining elements represented using a secondary mechanism. Fewer than 5\% of the postings are coded in this manner. Each block of integers is represented as a set of one or more codewords, each of these being a \( b \)-bit binary code.

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

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

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

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

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

**Exploiting String Overlap.** Further savings are also possible, beyond those offered by prefix matches and trailing don’t-cares. For example, with strategic reordering and overlapping, the twelve-symbol \( \text{dictionary}[] \) array in Figure 3 could be further condensed to just six symbols “baaab”, since every target listed in Table 1(b) appears within it as a subsequence. The problem of identifying a minimal-length super-sequence in which every one of an original set of supplied sequences occurs as a sub-sequence is NP-hard.
The approach we employ here considers the initial set of sequences, and hence a complexity of \((a \text{ maximum of } O(\log n \log \ell))\) if target lengths are restricted to powers of two) makes this process only moderately slower than the more usual greedy approach, with a complexity of \(O(n \log \ell)\) for a list of \(n\) integers, and hence a complexity of \(O(n)\) if \(\ell\) is regarded as being a constant.

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

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

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

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

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

## 5 Experiments

We now present the results of detailed experiments based on a range of public software and an implementation of the new DINT approach.
Datasets and Methodology. We use the standard Gov2 collection containing 426 GiB of text; and CCNEWS, an English subset of the freely available NEWS subset of the CommonCrawl\footnote{http://commoncrawl.org/2016/10/news-dataset-available/}, consisting of news articles in the period 09/01/16 to 30/03/18, following the methodology of Petri and Moffat\cite{petri2013reproducibility}. Postings lists for both collections were extracted from the Indri search engine to ensure reproducibility, using a document ordering derived from the recursive graph bisection reordering technique of Dhulipala et al. [11] (rather than the more usual URL ordering). Each index was considered as two streams of integers, one containing gaps between document identifiers (docids) and one containing within-document frequencies (freqs), with both of those streams split into per-term postings list segments in the usual manner. Statistics for these two datasets are provided in Table 5.

All compression results are for complete indexes without stopping or other reduction mechanisms being applied, and cover all postings; with sizes reported in GiB and rates given in bits per integer. Where a blocksize is required, $B = 256$ is used, with trailing part-blocks represented using Interp\cite{moffat2009}. All compression effectiveness results given for DINT include the overhead cost of the corresponding dictionaries.

Implementations and Hardware. All experimentation is based on the ds2i framework\cite{aspinall2015ds2i,aspinall2016ds2i}, with methods implemented using C++14 and compiled with g++ 7.2.0 (using all optimizations) on a server equipped with 512 GiB RAM and an Intel Xeon 6144 processor employing 32 kib of L1 cache, 1024 kib of L2 cache, and 25344 kib of L3 cache. The experimental framework and all code is available at https://github.com/jermp/dint.

Compression Effectiveness. Table 6 gives baseline compression effectiveness results for the four streams of integers that make up the indexes of these two collections, using a range of previous mechanisms, including: Varint-GB\cite{moffat2009}; Varint-G8IU\cite{moffat2013}; QMX\cite{ziv1996}; Simple16\cite{aspinall2015}; Opt-PFOR\cite{aspinall2016}; PEF\cite{aspinall2016}; Clust-EF\cite{moffat2014}; and Interp\cite{moffat2009}. The ANS version tested uses a set of 64 two-dimensional med-max contexts for each of docids and freqs\cite{moffat2017}.

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

<table>
<thead>
<tr>
<th>Collection</th>
<th>Lists</th>
<th>Postings</th>
<th>Documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gov2</td>
<td>39,180,840</td>
<td>5,880,709,591</td>
<td>25,205,179</td>
</tr>
<tr>
<td>CCNEWS</td>
<td>43,844,574</td>
<td>20,150,335,440</td>
<td>43,530,315</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Gov2 GiB</th>
<th>docids freqs</th>
<th>CCNEWS GiB</th>
<th>docids freqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Varint-GB</td>
<td>14.48</td>
<td>11.04</td>
<td>10.04</td>
<td>48.68</td>
</tr>
<tr>
<td>Varint-G8IU</td>
<td>12.77</td>
<td>9.90</td>
<td>8.69</td>
<td>43.87</td>
</tr>
<tr>
<td>VByte</td>
<td>11.85</td>
<td>9.22</td>
<td>8.02</td>
<td>39.65</td>
</tr>
<tr>
<td>QMX</td>
<td>5.59</td>
<td>4.99</td>
<td>3.11</td>
<td>19.20</td>
</tr>
<tr>
<td>Simple16</td>
<td>5.28</td>
<td>4.84</td>
<td>2.81</td>
<td>16.85</td>
</tr>
<tr>
<td>Opt-PFOR</td>
<td>4.55</td>
<td>4.33</td>
<td>2.26</td>
<td>15.50</td>
</tr>
<tr>
<td>DINT-DSV</td>
<td>4.33</td>
<td>4.25</td>
<td>2.00</td>
<td>15.25</td>
</tr>
<tr>
<td>DINT-DSF</td>
<td>4.29</td>
<td>4.22</td>
<td>1.98</td>
<td>15.09</td>
</tr>
<tr>
<td>PEF</td>
<td>4.16</td>
<td>3.85</td>
<td>2.23</td>
<td>13.75</td>
</tr>
<tr>
<td>Clust-EF</td>
<td>4.02</td>
<td>3.66</td>
<td>2.16</td>
<td>13.44</td>
</tr>
<tr>
<td>Interp</td>
<td>3.86</td>
<td>3.54</td>
<td>2.04</td>
<td>12.80</td>
</tr>
<tr>
<td>ANS, 2d</td>
<td>3.71</td>
<td>3.56</td>
<td>1.86</td>
<td>12.58</td>
</tr>
</tbody>
</table>

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

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

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

Sequential Decoding Speed. Table 7 compares decoding speeds of DINT (using a packed dictionary constructed using the DSF process with $b = 16$ and two values of $\ell$) and a range of other index compression approaches (including the SIMD-ized mechanisms of Masked-VByte\cite{aspinall2016} and Stream-VByte\cite{aspinall2016}), measured by decoding the entire index in a sequential manner. The net result of the experiment is that, compared to the methods that provide comparable or better effectiveness (Table 6), DINT is faster and, compared to methods of similar speed, DINT provides better compression effectiveness. Moffat and Petri\cite{moffat2013,moffat2017} report speeds for ANS decoding; based on their measured relativities, it would be second-from-top in Table 7; and based on the results reported by
Pibiri and Venturini [29], the Clust-EF mechanism can be expected to decode more slowly than the PEF approach, placing it also in the upper section of the table (neither of these two implementations was compatible with the sequential decoding test harness used to generate Table 7).

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

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

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

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Gov2</th>
<th>CCNEWS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT-DSF</td>
<td>4.29</td>
<td>15.09</td>
</tr>
<tr>
<td>+ opt. pars.</td>
<td>4.25</td>
<td>14.93</td>
</tr>
<tr>
<td>+ 6 contexts</td>
<td>4.22</td>
<td>14.64</td>
</tr>
<tr>
<td>+ (b = 8, 16)</td>
<td>4.10</td>
<td>14.21</td>
</tr>
<tr>
<td>+ exh. srch.</td>
<td>4.07</td>
<td>14.08</td>
</tr>
<tr>
<td>+ entropy</td>
<td>3.60</td>
<td>12.88</td>
</tr>
</tbody>
</table>

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

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

**6 CONCLUSIONS**

Figure 6 summarizes the relative performance of the new DINT approach, combining Tables 6 and 7. When presented in this way it is clear that our dictionary-based technique represents an important new approach to inverted index compression, approaching the speed of the very fastest methods for decoding and, at the same time, approaching the compression effectiveness of the best methods in terms of space required. Verifying that the throughput gains (demonstrated in Section 5) translate to faster query processing in
realistic settings is a clear area for future work. We also plan to explore less costly frequency estimation techniques, and quantify the extent to which accurate counts are needed for best compression.

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

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

**REFERENCES**


