Author/s:
Petri, M;Moffat, A

Title:
Compact inverted index storage using general-purpose compression libraries

Date:
2018-04-01

Citation:
Petri, M. & Moffat, A. (2018). Compact inverted index storage using general-purpose compression libraries. Software Practice and Experience, 48 (4), pp.974-982. https://doi.org/10.1002/spe.2556.

Persistent Link:
https://hdl.handle.net/11343/283747

SOFTWARE—PRACTICE AND EXPERIENCE
*Softw. Pract. Exper.* 2017; **00**:1–11
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

# Compact Inverted Index Storage Using General-Purpose Compression Libraries

Matthias Petri* and Alistair Moffat

*School of Computing and Information Systems, The University of Melboure, Australia 3010*

SUMMARY

Efficient storage of large inverted indexes is one of the key technologies that support current web search services. Here we re-examine mechanisms for representing document-level inverted indexes and within-document term frequencies, including comparing specialized methods developed for this task against recent fast implementations of general-purpose adaptive compression techniques. Experiments with the Gov2-URL collection and a large collection of crawled news stories show that standard compression libraries can provide compression effectiveness as good as or better than previous methods, with decoding rates only moderately slower than reference implementations of those tailored approaches. This surprising outcome means that high-performance index compression can be achieved without requiring the use of specialized implementations. Copyright © 2017 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: inverted index; index compression; web search

## 1. INTRODUCTION

Inverted indexes remain a key technology used in retrieval and search systems. Each possible query term $t$ that appears in any document in the collection is recorded in a vocabulary, and associated with a *postings list* containing one *posting* for each of the $f_t$ documents that contain $t$. Each posting consists of:

- a document identifier (or *docid*) $d_{t,i}$, the ordinal identifier of the $i$th of the $f_t$ documents that contains term $t$; plus (usually)
- a corresponding within-document frequency $f_{t,i}$ that records the number of times $t$ appears in $d_{t,i}$; plus (sometimes)
- a list of $f_{t,i}$ integers that record the word or byte locations at which $t$ appears in document $d_{t,i}$.

The first of these three components is used in Boolean querying; the second is required if ranked querying is to be supported; and the third allows phrase and proximity queries to be handled efficiently. Witten et al. [25] provide an overview of these structures and querying modes. Inverted indexes containing $d_{t,i}$ values alone, and conjunctive querying operations based on them, are also used in a range of other applications, including graph manipulation algorithms.

Regardless of whether an index is being maintained on disk or in main memory, compression can be used to reduce the amount of space required. Section 2 provides a brief summary of the main compression options that have emerged for postings data, all of which are premised on the



*Correspondence to: Matthias.Petri@unimelb.edu.au; School of Computing and Information Systems, The University of Melboure, Australia 3010

assumption that each postings list is to be compressed independently, so that term-based querying can be carried out by decoding only the lists associated with the words appearing in each query. We also provide a brief summary of general-purpose adaptive compression methods, including the recent fast ZStd library, a re-development and enhancement of the earlier widely-used ZLib package. These tools can also be used for postings list compression. Section 3 then examines the specific requirements associated with index compression, and measures compression effectiveness in several different modalities, including when a byte-based preprocessing step is introduced in conjunction with general-purpose adaptive compression tools. The preprocessing steps explored are attractive because each maps integer sequences into word-aligned or byte-aligned streams, thereby creating the possibility for general-purpose compression tools – which themselves expect byte-aligned input – to identify and exploit more complex relationships between consecutive values than might otherwise be possible. Section 4 measures corresponding decoding throughput rates, and demonstrates that general-purpose adaptive compression approaches allow high compression to be attained, with decoding throughput rates that are only moderately slower than the fastest of the tailored mechanisms.

The key purpose of this short paper is to demonstrate that general-purpose adaptive compression tools allow new points to be added to the Pareto frontier that represents interesting tradeoffs between compression effectiveness and decompression efficiency. Notable also is that one of the new combinations provides compression effectiveness comparable to the best of the previous tailored methods, including the Binary Interpolative Coding (Interp) approach [14] that has set the benchmark for more than two decades.

## 2. BACKGROUND AND RELATED WORK

We now survey mechanisms for inverted index compression, and also give a brief overview of methods for general-purpose compression. We assume throughout that the postings lists are sorted by document identifier $d_{t,i}$.

**Bit-Based Methods** Central to most representations of sorted postings lists is the notion of *docid gaps*, formed by taking differences between consecutive elements, $d_{t,i+1} - d_{t,i}$, with the first element $d_{t,1}$ in each list stored as itself. This simple process generates values that must of necessity be predominantly small for frequent terms, and can be consistently large only for rare terms. A range of integer codes that allocate short codewords to smaller values can be then applied to the gaps, including Golomb, Rice, and Elias codes, see Witten et al. [25] for details of these standard approaches. Further variants are explored by Fraenkel and Klein [10], Moffat and Zobel [15], and Bookstein et al. [5].

Rather than work with docid gaps, Moffat and Stuiver [14] use a recursive decomposition process to represent a monotonic sequence of docids in a known range, with the middle value in the list encoded first; then the left half of the list encoded recursively, in a sub-range constrained by the now-known middle value; and finally the right half encoded recursively, with a similar additional constraint. The benefit of the non-sequential ordering is that this Interp mechanism is sensitive to localized clustering of term appearances, as occurs when (for example) the documents in the underlying collection are URL-sorted, or sorted by date. This flexibility has meant that Interp has set the reference point for postings list compression effectiveness since its first description in 1996. However, like all bit-by-bit encoding approaches, Interp is relatively slow to decode, and it is not usually regarded as being useful in practice.

**Byte- and Word-Based Methods** A range of approaches have been developed in which compression effectiveness is traded against accelerated decoding throughput. Williams and Zobel [24] (see also Trotman [19]) experiment with VByte, a static code in which positive integers are mapped to byte sequences formed by splitting each input value in to 7-bit pieces, with the eighth bit in each byte used to signal the last byte of each codeword. In this approach, numbers between 1 and



DOI: 10.1002/spe

128 are all represented as 8-bit codes; numbers between 129 and $2^{14} = 16{,}384$ as 16-bit codes; and so on. Because no bit-at-a-time shift or mask operations are required, and only a small number of conditional branches are needed per generated value, VByte is much faster to decode than bit-based approaches.

In a different thread of development, Anh and Moffat [2] explore a range of methods that they name the *simple* codes, in which as many as possible docid gaps are stored as same-width binary values in a single output word, again with the objective of providing fast decoding, through the elimination of branching while decoding, and incorporation of loop unrolling. The Simple-16 variant of this approach uses 16 different bit-packing arrangements relative to a 28-bit *payload* [27], and decodes whole words at a time. Four bits in each words are used as a *selector*, and the remaining 28 bits carry the payload, ranging from a single 28-bit value through to 28 one-bit values. Another option is to use 64-bit output words, with 4 bits again used for the selector, and 60 bits for the payload [4]. It is also possible for one of the selector values to be reserved to indicate a run of "1"s in the stream of input values [4, 20]. If this option is incorporated, runs of as many as $2^{28}$ "1"s can be represented via a single 32-bit output word.

More complex packed mechanisms are also possible. For example, Trotman's QMX mechanism uses payloads of either 128 or 256 bits, with selectors separated from payloads, and multiple selectors combined into separate words [20]. The use of vectorized operations in the implementation allows very fast decoding rates to be achieved. 17

**Block-Based Methods** Zukowski et al. [29] describe a different packing approach. In their PFOR ("packed frame of reference") method blocks of 128 docid gaps are coded using fixed-width binary values of $b$ bits. Any over-sized values that require more than $b$ bits are coded as exceptions, and "patched" (repaired) after the 128 $b$-bit values have been decoded. This means that single instances of large values don't force the parameter $b$ to shift higher than is needed for the majority of values in the block. In the "optimized" OPF variant the value of $b$ used for each block is chosen so as to minimize the length of the block, and is included as meta-data in the block prelude. Yan et al. [26] and Lemire and Boytsov [12] add further enhancements to the underlying PFOR approach.

Elias-Fano (EF) codes provide another block-based mechanism that is suited to inverted file compression, working (like Interp) with the monotonic sequence of docids, rather than gaps. Given a monotonic sequence of $m$ docids drawn from a universe of size $n$, EF codes partition the universe into blocks of size $2^l$, where $l = \log_2(n/m)$. The lower $l$ bits of each docid are stored explicitly. The remaining (high) bits of each docid also form a monotone sequence and are stored as a sequence of unary coded gaps. In total, at most $2 + \log_2(n/m)$ bits are used per integer, while allowing efficient random access to the sequence [1, 9, 21].

The space usage of the EF representation depends only on $n$ and $m$, and is unaffected by any clustering present in the data. Ottaviano et al. [17] propose a two level structure building on top of EF codes which allow near-optimal partitioning of the sequence into variable-length blocks, such that different bucket parameters $l$ can be used within the blocks, and overall storage cost is minimized. We refer to this technique as EF-opt.

**Recent Approaches** Zhang et al. [28] describe a grammar-based approach to postings list compression, and report improved compression effectiveness in some situations. In recent work, Pibiri and Venturini [18] build clusters of similar postings lists, and compress the set of lists in each cluster using shared information common to each cluster. Other recent work by Wang et al. [22] and Wang et al. [23] has provided further insights into the effectiveness of tailored index compression approaches.

**Standard Compression Tools** A range of general-purpose byte-oriented compression tools can also be applied to data stored in binary files, including when the data is 32-bit integers, and including when those integers are binary docid gaps or within-document term frequencies $f_{t,i}$. For



DOI: 10.1002/spe

example, the well-known ZLib[†] library provides functions to represent sequences of bytes via an adaptive Lempel-Ziv technique, which achieves compression by using the already-encoded byte sequence as a dictionary, and wherever possible representing upcoming bytes as a back-pointer into that dictionary, thereby exploiting repeated and overlapping strings. Other standard options include BZip2[‡], which employs a byte-based Burrows-Wheeler Transform, and XZ[§], which also uses a Lempel-Ziv approach. Both of these tools are slower than ZLib for encoding and decoding, but usually provide superior compression effectiveness. Witten et al. [25, Chapter 2] describe all of these methods. The recent introduction of the ZStd[¶] library, which refines and extends the previous ZLib approach and obtains compression that is both faster and more effective than is provided by ZLib, adds a further possibility.

None of these general-purpose compression approaches is targeted specifically to docid gaps or to term frequencies (like the docid gaps, the great majority of the $f_{t,i}$ values are small), but their adaptivity means that they should be capable of learning – given a long enough input sequence, even when presented as four-byte binary integers – some of the structure of the input, and generating compact representations. While we presume that practitioners will have considered the possible use of standard compression tools for index data, comparative results have not been previously documented in the literature. We then add to that knowledge by coupling general-purpose adaptive methods with tailored mechanisms such as VByte, Simple-16, and OPF, in each case applying them to the integer values arising in typical postings lists. 18

## 3. INDEX COMPRESSION EFFECTIVENESS

In this section we measure the compression rate obtained for web-scale inverted index data using a range of mechanisms, including standard compression libraries, and three different input representations.

**Experimental Context** We make use of two large web-derived document collections. The first is the 426 GiB Gov2-URL collection[‖], which contains 24,751,688 documents, and gives rise to an inverted index of 25,285,523 terms and 5,416,085,998 postings when parsed using Apache Tika and Apache Lucene. To form the Gov2-URL index, postings lists were extracted from Lucene after docids had been reassigned in URL order, which is now standard practice when comparing index compression regimes.

We also develop (and make freely available) a new IR test collection based on the News sub-collection of the Common Crawl[**]. The News sub-collection provides daily crawls of news websites in many languages. We refer to this collection as CC-NEWS-URL. We provide all scripts to download the freely available source WARC files from Amazon AWS and process them using Apache Tika and Apache Lucene in a consistent manner. The resulting consistency enables researchers to perform experiments on exactly the collection in their experiments, and improves comparability of results between different rounds of experimentation. For example, the number of terms reported for the Gov2-URL collection ranges from 18 million up to 48 million, preventing fair and direct comparison between results reported in different papers.

The number of WARC files in CC-NEWS-URL increases each day, and hence we specify the collection using: (1) a date range; and (2) a language filter. For example, in this work, we utilize the CC-NEWS-20160901-2017028-en collection which uses all English language news sources (as identified by Apache Tika) from 01/09/2016 up until and including 28/02/2017, that is, a six month crawl period that contains 7,508,082 documents, 26,240,031 unique terms

[†] http://www.zlib.net/
[‡] http://www.bzip.org/
[§] http://tukaani.org/xz/
[¶] http://www.zstd.net
[‖] http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm.
[**] http://commoncrawl.org/2016/10/news-dataset-available/



DOI: 10.1002/spe

and 4,457,492,131 postings. Currently the CC-NEWS-URL collection grows by roughly 50,000 English documents per day. This exact parsing can be reproduced by the scripts provided at `https://github.com/mpetri/rlz-invidx` and `https://github.com/mpetri/TikaLuceneWarc`, with raw postings lists stored in the popular "ds2i" format[††]. Document identifiers are again reassigned in URL order. We also explored a date-ordered collection based on the same source data, and obtained – method-for-method – uniformly weaker compression outcomes than for URL-sorted, in part because many of the URLs contain dates encoded in them anyway.

While both of these two collections are orders of magnitude smaller than the whole web, they are comparable in size to the volume of web data that might sit on one processor in a distributed web search environment.

Three different test files were formed from each of the two indexes. In the two files denoted "docid.gaps", only the $d_{t,i}$ values are present, rendered into docid gaps; the two "frequencies" files only contain the corresponding $f_{t,i}$ values; and the two "interleaved" files contain fully interleaved $\langle d_{t,i}, f_{t,i} \rangle$ postings, with the docids again converted to gaps. As well, three different pre-processing regimes were considered: U32, in which case the data sequence was presented to the compression routines as a stream of unsigned 32-bit integers; VByte-preprocessed, in which case those 32-bit integers had been rendered into a byte-stream; and Simple-16-preprocessed, with the 32-bit integers converted into a corresponding stream of packed words. In total there are thus $2 \times 3 \times 3 = 18$ different data files considered, taking the product over collections, postings components, and preprocessing options. A subset of results across these 18 files are reported in the tables and graphs below. Note that some methods, including Interp and the EF-based approaches, require that both docid.gaps and frequencies be converted to prefix sums in order to be represented; we take this as given in the cases in which it is required.

All reported compression measurements are expressed in terms of bits per posting ("bpp") relative to the original inverted index files, and assume that the index meta-data is retained separately. That meta-data, which includes the vocabulary of the terms, and the pointers to secondary storage that allow parts of the index to be fetched independently, is non-trivial, but constant across compression methods. Note that in the case of the two interleaved files, each listed compression measurement covers two input integers, one docid gap and one frequency value.

**Whole-of-Index Compression** One lower-bound reference point for index compression is to completely disregard the needs of any querying operations, and require only that the index be retained as compactly as possible. This is likely to be the most favorable scenario for general-purpose adaptive compression tools, and represents an impractical, but "most optimistic" yardstick for compression effectiveness. Table I provides compression effectiveness results for the Gov2-URL and CC-NEWS-URL collections for this "whole-of-index" mode, with the three groups of columns showing the three different sequences derived from their inverted indexes. Within each of those three groups, compression rates can be compared, with the three columns in each group corresponding to the three preprocessing modalities: "U32", unsigned 32-bit integers (that is, no preprocessing); "VByte", the same files, but transformed by the standard byte-encoding process; and "Simple-16", also the same original files, but this time transformed by a word-packing process.

The general-purpose compression utilities listed in the rows of Table I handle whole-index data well, especially when VByte is applied as a pre-processing step. Applying Simple-16 first also improves compression relative to U32 format, but to a lesser extent. The compact outcomes from the three adaptive compression regimes are a consequence of the differencing process on docids that generates a preponderance of small values; of the use of the VByte transformation to pack those integers into a smaller number of bytes; and of the uncovering of further patterns in postings that repeat within and between individual postings lists, even after the gapping and byte-coding transformations. Note also that stand-alone VByte compression (in the row labeled "None", with that label indicating that no further compression tools are applied) is the only case in which interleaving the docids and $f_{t,i}$ values does not degrade compression effectiveness. In all of the other scenarios 19

[††] `https://github.com/ot/ds2i`



DOI: 10.1002/spe

Table I. Whole-of-index compression for the Gov2-URL and CC-NEWS-20160901-2017028-en collections, with no access possible to individual postings lists, using three different initial data representations, denoted U32, VByte, and Simple-16. All numbers are given as bits per posting. In the final group of columns, the $\langle d_{t,i}, f_{t,i} \rangle$ postings pairs are stored fully interleaved, and the listed compression rates cover the cost of both components.

| Method | docid.gaps | | | frequencies | | | interleaved | | |
|---|---|---|---|---|---|---|---|---|---|
| | U32 | VByte | Simple-16 | U32 | VByte | Simple-16 | U32 | VByte | Simple-16 |
| Gov2-URL | | | | | | | | | |
| None | 32.00 | 8.79 | 4.68 | 32.00 | 8.02 | 3.05 | 64.00 | 16.81 | 9.59 |
| BZip2 | 3.60 | 3.46 | 3.63 | 1.76 | 1.72 | 1.81 | 5.48 | 5.40 | 5.94 |
| ZStd | 4.45 | 3.25 | 3.58 | 2.45 | 1.75 | 1.85 | 7.25 | 5.35 | 6.29 |
| XZ –6 | 3.22 | 2.98 | 3.36 | 1.64 | 1.51 | 1.68 | 4.96 | 4.77 | 5.46 |
| CC-NEWS-20160901-2017028-en | | | | | | | | | |
| None | 32.00 | 8.65 | 4.22 | 32.00 | 8.00 | 2.38 | 64.00 | 16.65 | 8.57 |
| BZip2 | 3.38 | 3.29 | 3.42 | 1.27 | 1.23 | 1.30 | 4.78 | 4.75 | 5.26 |
| ZStd | 4.28 | 3.16 | 3.39 | 1.87 | 1.35 | 1.39 | 6.60 | 5.03 | 5.86 |
| XZ –6 | 3.13 | 2.90 | 3.22 | 1.27 | 1.13 | 1.21 | 4.54 | 4.40 | 4.93 |

it is more effective to store the components of the postings separately, rather than interleaved, confirming the generally accepted wisdom in this regard – see, for example, Anh and Moffat [3].

One interesting outcome of these experiments is that Simple-16 is consistently less effective than VByte is as a preprocessing tool, and in some cases is less effective than U32 as well. This may be because while Simple-16 generates word-aligned outputs, the byte boundaries within those words are only respected when 1-, 2-, 4-, and 8-bit codes are used.

Based on these whole-of-index "best possible" results, we now focus on the use of VByte as a pre-processing regime, and on the use of separated (that is, non-interleaved) postings.

**List-At-A-Time Compression** If Boolean and ranked query evaluation is to be efficient, access to a small number of postings lists is required, one per query term. Hence, the usual expectation of a compressed index is that the postings lists be independently decodable. In typical indexes the great majority of the terms occur a small number of times – indeed, it is not at all unusual for a sizeable fraction of the distinct terms to appear only once or twice in the collection. At the other extreme, a relatively small number of terms occur extremely frequently. This wide disparity means adaptive approaches are not normally considered to be applicable, because of the initially inferior compression performance that occurs while they are building their internal predictive model.

Table II gives compression effectiveness outcomes that include when standard adaptive tools are applied on a per-list basis, with VByte again used as an initial transformation. That is, the four rows of VByte-based results in Table II can be compared with the three VByte columns in Table I. To mitigate the risk of bad performance on short lists, a threshold value $\theta$ is employed: when $f_t < \theta$, that list is retained as VByte codes; and when $f_t \geq \theta$, the adaptive mechanism is applied to the whole of that postings list's VByte-processed values. Note that short postings lists on average have large docid gaps, reflecting the operating region in which VByte has the least relative inefficiency, meaning that the effectiveness loss from this strategy (compared to using some other more effective code such as Simple-16) is likely to be small. For the Gov2-URL collection, the fraction of postings handled in this way was less than 3%.

As anticipated, the use of independent postings lists degrades compression effectiveness compared to the "whole-of-index" approach (Table I), but applying standard compression tools to postings lists longer than length $\theta = 128$ is still worthwhile, and yields much better compression than when using VByte alone.



DOI: 10.1002/spe

Table II. List-at-a-time compression using a range of mechanisms. In the case of the three two-step "VByte then" methods, only lists with length $\theta = 128$ or longer are passed to the secondary approach; shorter lists are retained as VByte codes. In the case of the OPF approach, lists smaller than $b = 128$, and the trailing postings after the last 128-element block in each list, are also assumed to be coded using VByte. The rows are ordered by decreasing sum of the four listed values.

| Method | Gov2-URL | | CC-NEWS-URL | |
|---|---|---|---|---|
| | docid.gaps | frequencies | docid.gaps | frequencies |
| VByte | 8.79 | 8.02 | 8.88 | 8.33 |
| EF | 7.36 | 3.73 | 6.23 | 3.01 |
| EF* | 7.55 | 3.19 | 6.43 | 2.08 |
| QMX | 5.64 | 3.75 | 5.01 | 3.37 |
| OPF | 4.57 | 3.13 | 4.41 | 2.92 |
| EF-opt* | 4.18 | 2.35 | 3.58 | 1.74 |
| VByte then BZip2 | 3.88 | 1.90 | 3.63 | 1.70 |
| VByte then ZStd | 3.58 | 1.89 | 3.44 | 1.79 |
| VByte then XZ –6 | 3.39 | 1.73 | 3.25 | 1.62 |
| Interp | 3.37 | 1.97 | 2.96 | 1.37 |

The tailored methods described in Section 2 are usually deployed on a per-postings list basis. Table II also gives compression rates for a range of index compression approaches based on publicly available implementations. The OPF and EF-opt approaches both require additional meta-data – for example, the EF-opt method breaks each list in to blocks, and requires per-block information to be stored – and the cost of storing that additional information is included in the cases where it is required. Note also that the EF* and EF-opt* results are based on execution of the code of Ottaviano and Venturini [16], and those results are not directly comparable to the other four rows because of differences in the way that some secondary components are represented.

Notable in Table II is that on the Gov2-URL collection the VByte+XZ combination gives *better* compression effectiveness than the tailored Interp approach for the $f_{t,i}$ components, and all but matches Interp for the docids component. This relativity – overall compression for the docid and frequencies streams of the Gov2-URL index of $3.39 + 1.73 = 5.12$ bits per posting, 4% less than the $3.37 + 1.97 = 5.34$ cost associated with Interp – is a significant observation that has not previously been made. On the CC-NEWS-URL collection, the Interp approach retains its customary superiority.

**Block-At-A-Time Compression** The Block-Max Wand approach to query processing makes use of postings lists blocks of $b = 128$ (typically) non-interleaved $\langle d_{t,i}, f_{t,i} \rangle$ pairs [6, 7], and decoding is on a block-by-block basis rather than a whole-of-list basis. The main objective is to reduce the number of arithmetic and heap operations performed during query evaluation, with the expectation that there will be opportunities to bypass whole blocks. We do not explicitly consider that separate scenario in this short paper, but note that the use of such short blocks is in tension with the use of general-purpose compression tools, and that the "VByte+" approaches are unlikely to outperform methods such as OPF and Simple-16 for typical block sizes.

## 4. DECOMPRESSION SPEED

We now contrast the compression effectiveness results given in Section 3 with the CPU cost of regenerating lists from their compressed form.

**Hardware and Software** All of the methods were implemented using c++11 and compiled with gcc 5.4.0 running on a linux server equipped with 148 GiB RAM and an Intel E5640 processor. Much of the functionality we employ is built on top of the SDSL library [11]. We used the



DOI: 10.1002/spe

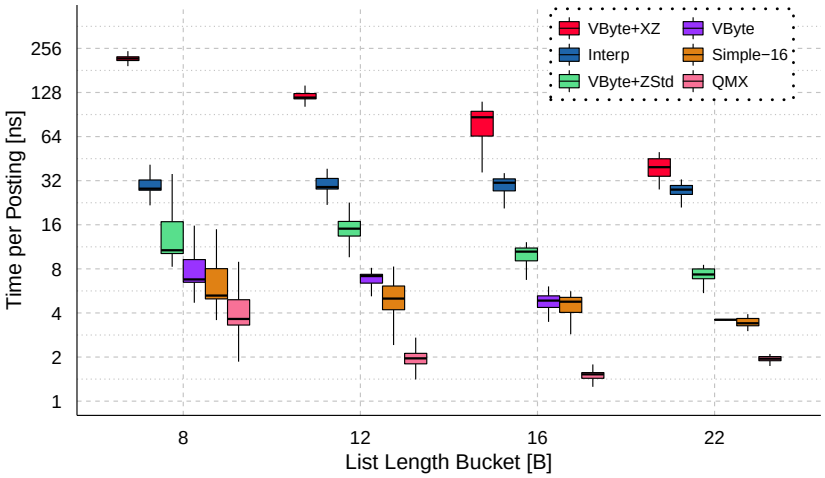


Figure 1. Access cost distribution in nanoseconds per posting for four selected subsets of 100,000 random terms, each occurring at least 128 times in Gov2-URL. Bucket $B$ contains lists of between $2^B$ and $2^{B+1} - 1$ postings. Note the logarithmic vertical scale.

FastPFor library [12] to provide efficient implementations of VByte, Simple-16, and OPF. Andrew Trotman's implementation of QMX [20] was also included as part of the test framework[‡‡]. Reference implementations of EF and EF-opt were taken from the work of Ottaviano and Venturini [16]; as was already noted in connection with Table II, we use these two guardedly, observing that some of the index meta-data is handled differently to our own implementations.

We were unable to execute the software made available to us by Zhang et al. [28], and as a result do not include results for their approach. We were also unable to access implementations for the approaches of Pibiri and Venturini [18] and Wang et al. [22], and do not include them in our experimentation. All of our own source code and scripts are made available, to encourage reproducibility and further development.

**Methodology** To measure posting list materialization cost, a total of 100,000 terms each with $f_t \geq 128$ were randomly selected from the Gov2-URL vocabulary. The time taken for each term to materialize the $f_t$ docids $d_{t,i}$ and frequency values $f_{t,i}$ into two arrays of 32-bit integers was recorded, and divided in each case by the corresponding $f_t$ to get a per-postings access cost, measured in nanoseconds. That set of 100,000 per-posting times, and the sizes of the compressed lists from which they were decoded, form the data discussed in this section. The restriction to terms occurring at least 128 times reflects that typical queries are unlikely to make use of terms that are very rare in the collection; and is not unreasonable, given that lists of length less than 128 are assumed to be coded directly using VByte by some of the methods.

Note that this measurement approach reflects processing modalities in which the entirety of each query term's posting list is decoded, and does not necessarily provide accurate guidance in regard to query processing times that might arise via more complex techniques, such as Block-Max Wand [7] and similar implementations. In particular, methods in which short fixed-length blocks of values are able to be independently located and decoded [16, 18, 20, 29] are able to respond to the needs of each query, and might decode only a minority of the postings associated with one or more of the terms, with a corresponding speed benefit possible when that occurs.

**Access Time** Figure 1 shows the distribution of per-posting decoding times for six different compression mechanisms, stratified by posting list length. The QMX approach is uniformly fastest; and VByte+XZ, then Interp, are the slowest options. Note also the significant disadvantage that

[‡‡] http://www.cs.otago.ac.nz/homepages/andrew/papers/QMX.zip.



DOI: 10.1002/spe

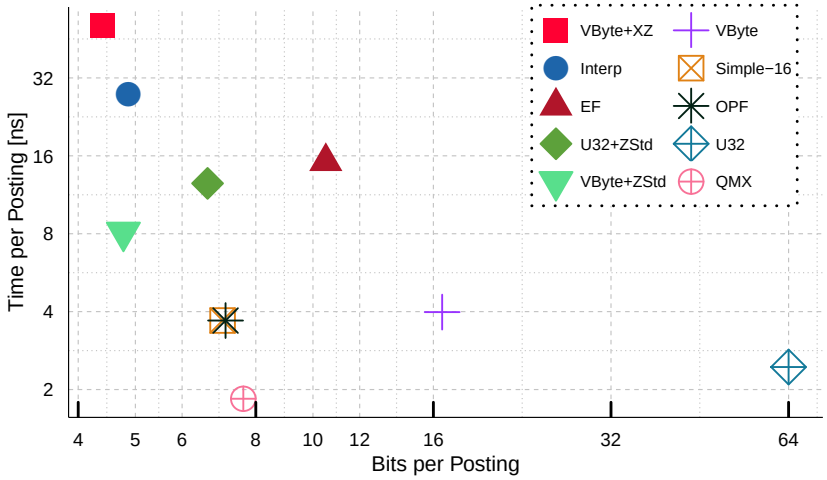


Figure 2. Time versus space tradeoff options, based on 100,000 randomly-selected terms each occurring at least 128 times in Gov 2-URL. The points for Simple-16 and OPF are coincident. Both scales are logarithmic.

XZ has on short postings lists, a consequence of the high start-up cost for each execution. The QMX implementation also provides faster decoding for longer lists. On the other hand, Interp is consistently slow, and gains no benefit from longer posting lists.

**Tradeoff Options** Figure 2 plots per-posting access time on the vertical axis (now averaged over all of the postings contained in the 100,000 selected postings lists) as a function of compression effectiveness (summed over docids and the $f_{t,i}$ components) on the horizontal axis. The "desirable" zone in this graph is at the lower left, where compression effectiveness is maximized and decoding time minimized. The Pareto frontier of interesting methods (for this document collection and this subset of postings lists) includes the VByte+XZ combination; includes the VByte+ZStd pairing, which outperforms the slower and less effective Interp, U32+ZStd, and EF methods; includes Simple-16 and/or OPF; and includes QMX. That is, if the input integers are first rendered to bytes via the application of VByte, general-purpose compression libraries provide a significant "boosting" effect, and allow enhanced compression effectiveness without (in the case of VByte+ZStd) greatly detracting from decompression throughput.

## 5. CONCLUSION AND FUTURE WORK

Our intention with this short paper was to explore the use of general-purpose compression tools for postings list compression. If the postings data – docid gaps $d_{t,i}$ and term frequencies $f_{t,i}$ – are retained as 32-bit integers, tools such as ZStd and XZ give a useful, but moderate, gain in performance. However, if the postings data is first transformed using VByte, much better compression rates can be achieved. This is a consequence of the fundamental byte-alignments that are assumed by these general-purpose compression libraries. The result is both compact inverted index storage requirements – in some cases even exceeding the twenty year-old benchmark established by the Interp approach – and also reasonable decoding access speed. That is, the new combinations provide additional tradeoff points in the space-versus-speed Pareto frontier for postings list representations, with important practical implications to web search engine implementation and other such applications.

To develop this work further, we plan to explore better options for handling short lists of fewer than $\theta = 128$ pointers – perhaps by grouping sets of them into larger units that are decoded in entirety if any parts of them are required. We will also explore term distribution effects based on query logs, to refine the measurements we have presented here, and to allow overall query processing



DOI: 10.1002/spe

times to be measured. The latter is an important area for further experimentation that is distinct from postings list materialization time, and will provide a more precise context for the tradeoffs that we have illustrated in Figure 2.

Finally, note that in recent work we have developed further "two-stage" index compression approaches in which the asymmetric numeral systems entropy coder of Duda [8] is combined with index compression techniques [13].

**Software** In the interests of reproducibility, the experimental software used to generate the results presented in this work is available at `https://github.com/mpetri/rlz-invidx` and `https://github.com/mpetri/TikaLuceneWarc`.

**Acknowledgments** This work was supported under the Australian Research Council's Discovery Projects scheme (project number DP140103256). We thank the referees for their detailed feedback. Au

## REFERENCES

[1] V. N. Anh and A. Moffat. Compressed inverted files with reduced decoding overheads. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 290–297, 1998.

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

[3] V. N. Anh and A. Moffat. Structured index organizations for high-throughput text querying. In *Proc. Symp. on String Processing and Information Retrieval (SPIRE)*, pages 304–315, 2006.

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

[5] A. Bookstein, S. T. Klein, and T. Raita. Modeling word occurrences for the compression of concordances. *ACM Trans. on Information Systems*, 15(3):254–290, 1997.

[6] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. Optimizing top-$k$ document retrieval strategies for block-max indexes. In *Proc. Conf. on Web Search and Data Mining (WSDM)*, pages 113–122, 2013.

[7] S. Ding and T. Suel. Faster top-$k$ document retrieval using block-max indexes. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 993–1002, 2011.

[8] 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`.

[9] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21 (2):241–260, 1974.

[10] 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-Verlag, 1985.

[11] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. Symp. on Experimental and Efficient Algorithms (SEA)*, pages 326–337, 2014.

[12] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software Practice & Experience*, 45(1):1–29, 2015.

[13] A. Moffat and M. Petri. ANS-based index compression. In *Proc. ACM International Conf. on Information and Knowledge Management (CIKM)*, pages 677–686, 2017.

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

[15] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 274–285, 1992.



DOI: 10.1002/spe

[16] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 273–282, 2014.

[17] G. Ottaviano, N. Tonellotto, and R. Venturini. Optimal space-time tradeoffs for inverted indexes. In *Proc. Conf. on Web Search and Data Mining (WSDM)*, pages 47–56, 2015.

[18] G. E. Pibiri and R. Venturini. Clustered Elias-Fano indexes. *ACM Trans. on Information Systems*, 36 (1):2:1–2:33, 2017.

[19] A. Trotman. Compressing inverted files. *Information Retrieval*, 6(1):5–19, 2003.

[20] A. Trotman. Compression, SIMD, and postings lists. In *Proc. Australasian Document Computing Symp. (ADCS)*, page 50, 2014.

[21] S. Vigna. Quasi-succinct indices. In *Proc. Conf. on Web Search and Data Mining (WSDM)*, pages 83–92, 2013.

[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] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. An experimental study of bitmap compression vs. inverted list compression. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 2017.

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

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

[26] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 147–154, 2009.

[27] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. Conf. on the World Wide Web (WWW)*, pages 387–396, 2008.

[28] 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. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 275–284, 2016.

[29] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. International Conf. on Data Engineering (ICDE)*, page 59, 2006.





DOI: 10.1002/spe