Lossy Compression Options for Dense Index Retention

Joel Mackenzie
The University of Queensland
Brisbane, Australia
joel.mackenzie@uq.edu.au

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

ABSTRACT

Dense indexes derived from whole-of-document neural models are now more effective at locating likely-relevant documents than are conventional term-based inverted indexes. That effectiveness comes at a cost, however: inverted indexes require less than a byte per posting to store, whereas dense indexes store a fixed-length vector of floating point coefficients (typically 768) for each document, making them potentially an order of magnitude larger. In this paper we consider compression of indexes employing dense vectors. Only limited space savings can be achieved via lossless compression techniques, but we demonstrate that dense indexes are responsive to lossy techniques that sacrifice controlled amounts of numeric resolution in order to gain compressibility. We describe suitable schemes, and, via experiments on three different collections, show that substantial space savings can be achieved with minimal loss of ranking fidelity. These techniques further boost the attractiveness of dense indexes for practical use.

CCS CONCEPTS

• **Information systems → Search engine architectures and scalability**; **Search index compression**.

KEYWORDS

Index compression; dense indexing; lossy compression

1 INTRODUCTION

As dense retrieval models continue to increase in popularity, managing their large storage footprints is a growing challenge. A body of prior work has focused on minimizing the size of dense indexes to accelerate training or retrieval speed [4, 21, 26], but the problem of long-term dense index retention has not yet been widely explored. This complementary aspect is an important direction for both researchers and practitioners, as it allows older versions of an

Table 1: Compressed size (fraction of uncompressed size) of general purpose compression tools with high compression settings (slower to code, but better compression) on the three dense indexes described in Section 4. Smaller values are better.

Codec	Con-Arg	DPR-Wiki	ANCE-MARCO
gzip −9	0.926	0.926	0.923
bzip2 −9	0.949	0.948	0.947
xz −9	0.911	0.904	0.901
zstd −15	0.922	0.925	0.922

index to be resurrected if and when the need arises, for purposes such as reproducibility, repeatability, recovery, and sharing.

One obvious approach for preserving indexes between uses is to compress them with a general purpose tool such as gz or xz. However, as Table 1 demonstrates, general purpose lossless (exact) compressors do not gain much traction on floating point data, with typical reductions of just 10% of the uncompressed index size. On the other hand, lossy techniques provide better compression, but risk eroding the quality of the index, and hence its utility.

We explore both lossless and lossy compression mechanisms for the storage of dense indexes. Our contributions are as follows:

- We cast the problem of index retention as a trade-off between storage consumption and result quality;
- We analyze the suitability of both lossless and lossy compression schemes for the task of dense index compression;
- We propose a family of simple lossy quantization methods that meet the particular requirements of dense indexes; and
- We demonstrate that our approach allows highly competitive fine-grained selection of storage/quality trade-offs for dense index retention, where quality is measured by ranking similarity.

2 COMPRESSING AND STORING INDEXES

This section summarizes dense indexing approaches, describes floating point number representations, and provides an overview of lossless and lossy compression techniques as they apply to text indexes. We assume a collection $\mathcal{D}$ containing r documents, with the i th of those documents denoted D_i.

Dense Retrieval. In a single-representation dense index, each document D_i is represented as a fixed-length vector of c floating point numbers, positive and negative, that are inferred by a large language model or neural network. Broadly speaking, each of those c values represents some kind of learned or inferred attribute, perhaps "animals", or "sport", or "business", or "Asia", or "youthful", with positive values in that column indicating degrees of support for that inferred concept in D_i; negative values indicating degrees

of anti-support; and values near zero representing ambivalence. The index for the r documents in $\mathcal{D}$ can thus be visualized as an $r \times c$ matrix $\mathcal{M}$ of floating point values. The topical affinities of the c columns are implicit and not identified in any way – there are no column labels or vocabulary – and the columns have no observable distinguishing properties.

Each incoming query Q is similarly mapped to a vector of c positive and negative values. The similarity between a document D_i and query Q is then computed via:

$$S(D_i, Q) = \sum_{j=0}^{c-1} \mathcal{M}[i, j] \cdot Q[j]. \qquad (1)$$

To generate a top-k answer set for a query Q, a retrieval system must either exhaustively compute all r scores $S(D_i, Q)$ via r inner-product computations, and then select the k largest values; or must pre-compute some sort of c-dimensional nearest neighbor index structure that allows the top k set to be identified more efficiently.

Our emphasis in this paper is on the permanent storage required by the matrix $\mathcal{M}[\cdot, \cdot]$, rather than any derived index structures to accelerate query processing.

Floating Point Formats. Most hardware platforms offer two standard floating point types: `float32` and double-precision `float64`. These consist of three components: one bit for a sign; 8 or 11 bits for a binary exponent; and 23 or 52 bits for a binary mantissa, not counting the leading mantissa bit which is always a 1 and does not require storage, in effect supplying a 24 th or 53 rd bit of mantissa precision. The relevant standard[1] also allows a `float16` format that contains a sign bit, a 5-bit binary exponent, and a 10-bit (plus one hidden bit) mantissa; a format that may only be available on certain language/compiler/hardware combinations. Some hardware platforms also offer a 128-bit `long double` format that has a 15-bit exponent and a 113-bit mantissa.

In each of these formats the three components can be thought of as three plain binary integers of S, E, and M bits each, where (S, E, M) is used to summarize the arrangement. If $0 \le s < 2^S$ is the sign, $0 \le e < 2^E$ is the exponent, and $0 \le m < 2^M$ is the mantissa, they combine to indicate the fractional value

$$(1 - 2s) \cdot 2^{e - 2^{E-1} + 1} \cdot (2^M + m)/2^M,$$

in which the division yields a real number.

Like all finite-precision representations, some numbers can be represented exactly, while others – the majority – are approximations of the original true values. For example, 0.5 and 0.75 both have exact representations in binary-based floating point representations, but 0.6 and 0.7 do not. The top section of Figure 1 shows the `float32` representation of $\pi \approx 3.1415$, with $s = 0$, $e = 128$, and $m = 4,788,187$ required relative to $(S, E, M) = (1, 8, 23)$. Our focus in this work is on numbers that have their origins in this single precision `float32` format. Their 24 mantissa bits are equivalent to about seven decimal digits of precision, a situation consistent with the top section of Figure 1, in which the magnitude of the error in the `float32` representation of π is a little under 10^{-7}. The second part of Figure 1 is discussed shortly.

[1]IEEE.754, see https://standards.ieee.org/ieee/754/6210/ and https://en.wikipedia.org/wiki/IEEE_754.
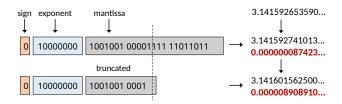

Figure 1: Floating point layout and the "normal" rounding error inherent when π is represented as a `float32`; then applying further mantissa truncation to form a `bfloat20`.

Index Compression. Conventional inverted indexes are highly compressible, and each occurrence of a distinct word in a document (a *posting*) can typically be represented in around one byte when compressed, see, for example, Zobel and Moffat [28] and Pibiri and Venturini [16]. That means a document containing 100 distinct words (and thus maybe a few hundred words in total, or around 1 kB of plain text) requires around one hundred bytes of index storage. In model-based dense representations, the same document might correspond to $c = 768$ floating point values, each of which requires 4 bytes – that is, a factor of 30 times as much space.

Inverted indexes are compressible because when they are considered as term-document frequency matrices (with the terms as the columns and the documents as the rows), the overwhelming majority of cells are zero; and because, of the small fraction of cells that are non-zero, the majority are small values. Those patterns do not hold for dense vector-based indexes. Now the matrix entries are all positive or negative floats, and almost none of them are zero. That is, while these dense indices can provide more *effective* similarity-based search over document collections, that gain has a cost in terms of storage space. Moreover, if a dense index is to be retained on disk and used in memory only periodically, disk storage might be a substantial fraction of the overall resource requirements.

Lossless Compression of Floats. Structural modeling, probability estimation, and entropy coding – the techniques behind any compression regime – are well understood [14]. It is not possible to directly model and entropy code a stream of 32-bit floats, because the fact that there are 2^{32} distinct values available means that unless the input matrix $\mathcal{M}[\cdot, \cdot]$ is huge, the cost of storing the entropy coder's parameters will swamp the modest savings that are available.

An alternative is to "deinterleave" the (S, E, M) components of each `float32` to make three parallel sequences of range-bounded integers. It is then possible to apply any suitably parameterized entropy coder to each stream of integers, and thereby compress the original stream of floating point values. For example, if there is a strong imbalance between positive and negative values, the sign bits might take less than one bit each to store; and if the exponents are dominated by a small set of the 2^E possible values then savings might be achieved in that stream too. On the other hand, floating point mantissas tend not to be compressible. Their low-order bits are, to an entropy coder, essentially random data, and so do not yield any significant space savings. In this case simply storing a stream of 23-bit integer values (packed so that 32 such values occupy 23 four-byte words) is a better option.

Table 2: Cost (bits per number) of losslessly coding the (S, E, M) components of a dense index of 6,661,632 `float32` numbers (the Con-Arg index). The rates shown in the first three rows would be achieved by an arithmetic coder (see Moffat and Turpin [14]) once the probability distributions were available (the fourth row).

Component	uncompressed	bits/value
Sign	$S = 1$	0.96
Exponent	$E = 8$	2.57
Mantissa	$M = 23$	21.96
Preludes		2.75
Total	32	28.24

Table 2 demonstrates these effects using the smallest of our three test indexes (see Section 4 for details). As can be seen, the main source of redundancy in this set of float variables is via their exponents, with an average of 5.43 bits per value able to be saved. There is also a small "on paper" saving available via the mantissas, as the distribution is slightly skewed in favor of small values. However that possible saving is completely eroded, plus more, if the cost of *describing* that distribution is factored in (shown in the "preludes" row, noting that the sign and exponent distributions have very low overhead costs because they involve only 2 and 256 parameters respectively). The best that a deinterleaving approach can do on this file is to save 5.43 bits by entropy coding the exponents while simply bit-packing the other two streams, thereby reducing the cost to 26.57 bits per float overall, that is, to 83.0% of the input file size, slightly better than is available using general-purpose compression tools such as gzip and xz (Table 1).

Lossy Compression. As already noted in connection with Figure 1, floats are almost always approximations, and there is nothing special about using $M = 23$ bits for the mantissa. The same is true for any data that has its origins in a signal that is continuous, such as sound or images. For such data the use of *lossy compression* might be a viable alternative. Fidelity relative to the initial set of quantized-to-float values will be lost, but the eventual use of the data may be unaffected. For example, lossy compression of audio signals in MP3 files still yields excellent sound quality, with the difference between MP3 and original uncompressed 16-bit signals (CD quality) only able to be detected by an audiophile; and the lossy compression of video data in MP4 files might not be noticeable to a person watching on a standard TV.

One simple way that the storage cost of general floats can be reduced is via *mantissa truncation*, shown in the lower section of Figure 1. In this example the original 23-byte mantissa is truncated by 12 bits, to be an 11-bit mantissa instead, a saving of 12 bits per float. The difference between the stored value and the original value increases, but a downstream task – for example, a laser-based tool cutting a circular steel plate – might not be affected by the loss of precision. The Google `bfloat16` is a truncated `float32` that has the pattern $(1, 8, 5)$, with 16-bit mantissa truncation.[2] We will use truncated floats – with the value shown in the lower section of

Figure 1 regarded as being a `bfloat20` – as a baseline for lossy representations in the experiments described in Section 4.

Other Index Reduction Methods. Another way to reduce the cost of a c-dimensional vector of real values, the type used in dense retrieval, is to simply reduce the total number of dimensions required, while noting that doing so will also probably reduce retrieval effectiveness. Prior work has explored the use of unsupervised methods to reduce dimensionality including random projections (selecting a random subset of dimensions) and principal component analysis (selecting dimensions with the highest variance) [13, 15, 29]. We explored the use of dimensionality reduction in preliminary experiments but found that it was not competitive with the lossy mechanisms explored in this work, and we omit them for brevity.

Product quantization (PQ) is another approach for reducing the storage requirements of dense indexes [7]. The main idea behind PQ is to partition the original c-dimension embedding into a sequence of c'/c dimension sub-vectors, each of which can then be quantized independently via k-means, and stored via a mapping to a dictionary. In our experiments, we explored PQ with dictionaries of 256, 4096, and 65,536 entries, and with $c'/c \in \{1/2, 1/4, 1/8, 1/16\}$.

Finally, there are a number of *supervised* approaches for reducing the size of dense indexes during training [3, 5, 12, 19, 24, 27], often with the dual aim of accelerating retrieval. We do not consider these methods in this work; our goal here is to compress and retain *deployed* indexes for future reference, rather than minimize their size at the time they are being accessed.

3 LOSSY DENSE INDEX REPRESENTATIONS

This section introduces methods for quantizing a set of `float32` values in ways that are tailored to the set of values needing representation, and in a way that also indirectly exploits the cost savings possible via the exponent part of floats (Table 2).

Domain- and Range-Quantization. Suppose that a set S containing n `float32` values is to be lossily represented using $B < 32$ bits per value. In the dense indexing application considered here we will have $n = r \times c$, but the techniques we consider can be applied to any stream of floating point values.

One obvious option is to design a (S, E, M) representation in which $S + E + M = B$, in the style of the IEEE `float16` and Google's `bfloat16` approaches. But as has been demonstrated in Table 2, this may be wasteful, with some combinations of the B bits likely to be rare or unused in any given input file.

An alternative is to note that 2^B different bit patterns are possible using B bits, and to assign those to a set of ordered *bins* that are created in some way that spans the range of values in S. The binning process should then assign the smallest value in S to bin 0, and the largest value to bin $2^B - 1$. Every other value in S would also map to a bin identifier between 0 and $2^B - 1$ inclusive. Each bin would have a *representative value* associated with it, to serve as a computational surrogate for all of the values in that bin. The set S would then be lossily represented by a sequence of n integer bin identifiers. That representation will be useful if the downstream task (in this case, computing inner-products via Equation 1) yields results that are unlikely to be "noticed as being different" by an average (or some other definition, such as "a 95% astute") user.
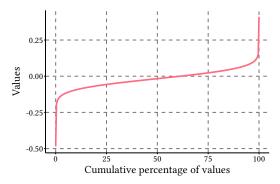
Figure 2: Cumulative distribution of floating point values in the Con-Arg index (6,661,632 `float32` values, see Section 4 for details).

This overall structure then leads to several design decisions: methods for deciding which ranges from $\mathcal{S}$ are placed in each bin; methods for determining the representative value to associate with each bin; methods for efficiently storing sequences of integer bin numbers; and methods for measuring the degradation of some downstream outcome when comparing it against the "full" `float32` outcome – again noting that the `float32` is also an approximation of some unknown "true" value on the continuous real number line.

Choice of Binning Strategy. Figure 2 shows the distribution of floating point values in one of the experimental indexes described in detail in Section 4. The `float32` values in this file span the range from approximately -0.5 to $+0.5$, with three somewhat linear segments visible: a steep climb at the beginning, indicating that there are a small number of strongly negative values; then a long slow-growth segment; and then a second steep section at the right-hand side, indicating a small number of strongly positive values. There is also a moderate asymmetry: in this index 61.2% of the stored values are negative, and only 38.8% of the stored values are positive. There are no index values that are zero.

One obvious binning strategy is to divide the horizontal scale in Figure 2 into regions of equal size. We call this the FD approach, *fixed domain* binning. If there are n values to be represented via B-bit bucket identifiers, each bucket will contain approximately $n/2^B$ values, and will have a representative value from within the corresponding range. The clear drawback of this options is that in a region of rapid change in values, fixed domain binning is likely to result in high approximation errors, and a consequent risk of degradation in the effectiveness of the downstream operations. Moreover, those high errors would occur at the two ends of the range, and be prevalent in high-magnitude values.

A second option is thus to form fixed-interval bins according to the vertical axis employed Figure 2, to create FR (*fixed range*) bins that each have a variable number of elements. The FR approach offers control of numeric errors, since the representative value used in regard to each bin will have a known upper bound difference between it and every value represented by that bin. In particular, if the numeric mid-point of each bin is taken as the representative value, and if x is a true value and $\hat{x}$ the corresponding representative value, then $|x - \hat{x}| \le (\max(\mathcal{S}) - \min(\mathcal{S}))/2^{B+1}$.

The drawback of the FR mechanism is that some bins might remain empty while others have a large number of data points in them. As a specific example, outlier values at either end of the range might give rise to dozens of empty bins, thereby reducing the effective resolution of any given parameter B.

As a blend between these two approaches we introduce two further heuristics, both of which ensure that every bin has at least one valid value in it, and at the same time result in high bin density at regions of rapid change in values. As can be seen in Figure 2, the beginning and end of the range are where a dense bin concentration is required if maximum error is to be bounded. Conveniently, it is also at the extremes of the range from which the greatest contributions to document scores might arise (see Equation 1), and hence where it might be supposed that numeric fidelity is most important. These observations suggest the first of the two additional strategies: a bin allocation function that consists of a geometric distribution and its symmetric reflection.

Consider the geometric sequence governed by a parameter θ and initial value v when taken to r terms and summed: $v + v\theta + v\theta^2 + v\theta^3 + \cdots + v\theta^r = v(\theta^r - 1)/(r - 1)$. If there are 2^B bins to be assigned, then half will be left of the mid-point, and half to the right, and the total of all of the bins left and right must be n. These relationships mean that θ needs to satisfy

$$v \cdot (\theta^r - 1)/(\theta - 1) - n/2 = 0\,, \tag{2}$$

where $r = 2^B/2$, which, given B, n, and v, is straightforward to solve using bisection. The simplest arrangement is when $v = 1$ and the leftmost and rightmost bins contain just a single item. The ith and $2^B - i - 1$th bins will similarly be assigned bin boundaries so that they span exactly θ^i of the n index values each, with a pair of middle bins being the largest. Those middle bins – and their neighbors near the middle of the range – will each hold a large number of values close to zero (but not necessarily "closest to" zero, noting again the asymmetric distribution in Figure 2, which is also visible in the FR line in Figure 3). An implementation needs to work with a real-valued θ but rounded-off integer bin sizes, and needs to also handle cases in which n is odd; the details are slightly tedious but not complex, and we omit discussion of them here. We refer to this approach as the GD method, for *geometric domain*.

The second heuristic we suggest, and the fourth binning strategy overall, is to use the FD approach for the central 2^{B-1} bins, and to store the smallest 2^{B-2} and largest 2^{B-2} values in single-element bins, that is, as *exceptions* that are represented exactly. For example, when $B = 8$, the smallest and largest 64 index values each occupy an assigned bin in isolation, and the remaining $n - 128$ values are assigned to the central 128 bins via a restricted application of the FD method. The fraction of the 2^B bins assigned to the central region introduces another parameter; for simplicity, we set that fraction at $1/2$ in all of the experiments reported here.

We call this the CFR approach, *central fixed range*. The average error in the central region will be greater than with the FR approach, but that might be compensated by having no loss for a small number of values at the extremes of the score range, and by greatly reducing the likelihood of empty bins occurring.

Figure 3 shows the result of applying these four approaches to the same index data as was used in Figure 2, plotting bin size as a function of bin number (expressed on the axis as a percentage
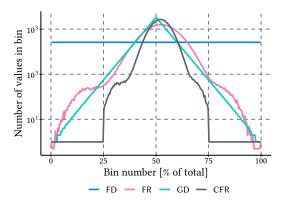
Figure 3: Bin sizes for $B = 8$ using four different strategies. Note the logarithmic vertical axis; the bins of size zero generated by the FR approached are plotted at a size of 0.5. Bin indices on the horizontal axis are expressed as a percentage of $2^B = 256$.

Figure 4: Bin maximum errors for $B = 8$ using four different strategies. Note that the FD curve has been clipped at the left and right extremities. Bin indices on the horizontal axis are expressed as a percentage of $2^B = 256$, as was also the case in Figure 3.

Table 3: Cost (average bits per approximated index value) of storing a binned index covering 6,661,632 floating point values using four binning methods and a range of bin counts.

Method	Number of bins employed			
	256	910	1050	1430
FD	8.00	9.83	10.04	10.48
FR	6.17	8.00	8.21	8.65
GD	5.77	7.77	8.00	8.49
CFR	5.40	7.32	7.54	8.00

of 256) for $B = 8$. The FR and CFR approaches give irregular bin sizes because they are defined via the range of stored values rather than the domain; the other two use bin sizes directly computed from B, n, and the bin number value. Note how the GD approach (which uses a computed parameter $\theta = 1.105$) approximates the distribution of the fixed range FD method, but without the risk of empty bins being created. For the data shown the FD mechanism generates 11 empty bins; whereas there are no empty bins in the other three approaches.

Choosing Bin Representatives. Given a set of numbers that are the members of a bin, we also need a mechanism for selecting a single value to represent them all. Several options are obvious: we could take the arithmetic mean of the values in the bin, or their median, or the midpoint between the largest and the smallest value in the bin. If the bin contains only one or two elements, all three of those approaches are identical. If the bin has a large number of elements and is not in a region of sharp curvature (see Figure 2), then the three will again likely result in similar outcomes. In the experiments reported in the next section we use the average of the bin's assigned values, choosing that approach because it minimizes the total error magnitude when summed across the set of values assigned to that bin.

Figure 4 illustrates the relationship between maximum bin error (that is, the biggest magnitude difference between bin mean value and the upper and lower extreme values) and the bin number (again, expressed as a percentage of 256) for the same data as was plotted in Figures 2 and 3. As expected, the FD approach has very large error magnitudes at the ends of the distribution (so large that they are clipped in the graph, the highest value is 0.235, nearly thirty times the axis that is shown); the FR approach has near-constant maximum error; and, by design, the CFR method has a higher maximum error in the central region, but zero error at the extremities.

Coding Bin Identifiers. If there are 2^B bins, then each `float32` value in the index can always be represented by a B-bit binary bin number. But if an entropy coder is applied to the sequence of bin identifiers, it may be possible to do better, perhaps substantially so.
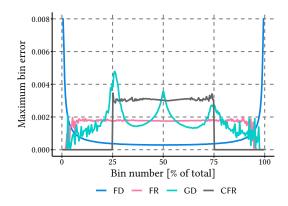
If n_b is the number of index values assigned to bin $0 \le b < 2^B$ by the partitioning regime, with $\sum_b n_b = n$, then an effective entropy coder can represent each instance of n_b using $\log_2(n/n_b)$ bits on average [14], and store the entire stream using $\sum_b n_b \log_2(n/n_b)$ bits. Moreover, coding in this way means that the number of bins need not be a power of two. In the experiments reported in Section 4 we employ arithmetic coding and allow the bin count to be an arbitrary integer.

In particular, looking again at Figure 3, it is clear that the distributions of bin occurrence counts in index data is highly skewed under the FR, GD, and CFR approaches. Table 3 shows the extent of the savings that can then result via arithmetic coding rather than binary coding. Down the table's diagonal, a suitable bin count has been identified for each of the four binning approaches so as to result in an average cost of 8.00 bits per stored index value. The four bin counts are listed across the top of the table, and vary from 256 for the GD approach to 1430 for the CFR method. The other table entries then show the per-value cost of arithmetic coding the same 6,661,632-value index using that combination of bin count and binning method. As can be seen, the CFR method provides the most compact index for any given bin count – in part because half of the bins are quite deliberately assigned only a single value.

Figure 5 revisits the representational errors that arise from the binning process, now plotting maximum errors by position in the sorted distribution of values shown in Figure 2 rather than by
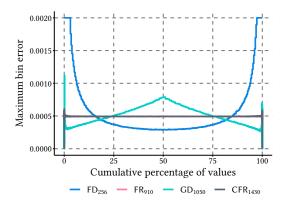
Figure 5: Bin maximum errors for a constant bin entropy of 8.00 bits per value (see Table 3), now plotted as a function of the underlying distribution of stored index values (also used in the horizontal axis in Figure 2). The FD curve has again been clipped at the left and right extremities; the FR curve is almost exactly coincident with the CFR one. Note the ×4 change in vertical scale relative to Figure 4.

bin number (with those maximums still computed on a per-bin basis). The four combinations of parameters in the diagonal of Table 3 are shown by the four plotted lines, so that quantization errors can be properly compared on an equal bit-cost basis. What is now clear is that the CFR and FR methods have almost the same error profiles except at the extreme endpoints; and that the other two methods offer two further trade-offs between fidelity in the center of the range of values and fidelity in the upper and lower sections of the set of index values. The next section undertakes a detailed exploration of these four lossy binning techniques, and of the effect their differences in representational accuracy have on the "downstream task" of similarity-based retrieval using Equation 1.

4 EXPERIMENTS

The upper half of Figure 6 shows the structure of the index retention capability envisaged in the previous section, with a float32 index analyzed by a binning component; the binning decisions then used to assign bin indices to all of the index coefficients; and with that sequence of bin indices arithmetically coded to form a lossy compressed quantized index. The binning information must also be retained, as it is required by the decoder, but is very small.

Based on that structure, this section describes the experimental context and datasets, and then reports our findings.

Hardware and Software. All experiments were conducted on a Linux workstation with a 32-core AMD Ryzen Threadripper Pro 5975WX operating at 3.6 GHz with 512 GiB of RAM.

The various approximation and quantization methods were implemented using C and C++, and compiled with GCC 11.3 and -O3 optimization. All other experimentation including product quantization and query processing was conducted using the highly optimized FAISS library [8]. Our query processing experiments used exhaustive search to ensure no confounding effects due to query-time approximation or tie-breaking ambiguities, and our custom software allows flat FAISS indexes to be directly saved, to ensure

a fair empirical comparison. We do not report query execution times, since that is not our focus here; but do note that once a binned-quantized index of the type discussed in Section 3 has been decompressed and placed in memory, from a query-processing-logic point of view it is identical in structure to the original index, and can thus be processed using any of the same techniques.

Datasets and Queries. We experiment with three indexes, each representing a unique system/collection pair:

- Contriever-ArguAna (Con-Arg): An index over the ArguAna *counterargument retrieval* collection [20], as featured in the heterogeneous BEIR benchmark [18]. The index is built using the Contriever model [6], represents $r = 8,674$ documents each mapped to $c = 768$ coefficients, and occupies around 25 MiB of storage.
- DPR-Wikipedia (DPR-Wiki): An index over a Wikipedia[3] crawl, built using the DPR model [9]. This index has $r = 21,015,324$ documents and $c = 128$ coefficients per document, constructed by product-quantization from an original 768 dimensions [13]. This index occupies about 10 GiB of storage as float32 values.
- ANCE-MSMARCO (ANCE-MARCO): This is an index for the MSMARCO-v1 passage collection [2] using the ANCE model [23], and represents $r = 8,841,823$ documents via a set of $c = 768$ coefficients per document, using just over 25 GiB of storage.

All three are publicly available as part of the Anserini/Pyserini open-source toolkit [11, 25].

A random sample of document vectors was taken from each collection and used as queries, 2,000 vectors in the case of the small Contriever-ArguAna index, and 10,000 vectors for each of the other two collections. This "self validation" approach means that the strongest match is always between a query and that same vector as a document in the collection; thereafter the ranked list generated for each query is a list of the other documents in decreasing order of similarity relative to the query document. The top $k = 1,000$ documents were retrieved for each query and compared to the list returned by a full-precision float32 index.

Measurement Objectives and Evaluation. Our key goal was to measure the relationship between the fidelity of index-based retrieval on the one hand, and index size when stored on secondary storage on the other. That is, we regarded long-term model retention cost as being an important resource, and sought to minimize that cost through the application of lossless or lossy compression techniques. To ensure that we always compared like with like, all indexes were assumed to be stored in compressed form. In particular, the quantized binned approaches introduced in Section 3 were all compressed using a zero-order arithmetic coder, with the cost of the auxiliary bin representative values included and requiring four bytes each. Similarly, all of the other indexes used as baselines are assumed to be compressed using xz −5 to reduce the space required to retain them on secondary storage, without any use of deinterleaving. The "−5" quality level was chosen as a compromise between reduced size and encoding speed. Table 1 provides some indicative results obtained from the more costly "−9" variants.[4]

---

[3]See https://huggingface.co/datasets/wiki_dpr, accessed May 17, 2023.
[4]In preliminary experiments, we found that xz −9 often gave only minuscule improvements over xz −5, and occasionally provided no or even reduced benefit.
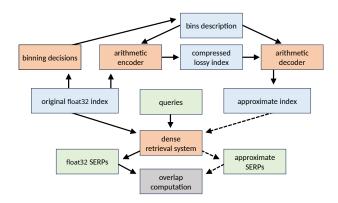
Figure 6: Structure of our experiments. The similarity between SERPs generated from the `float32` and an approximate index is computed via an overlap computation (grey box at bottom) and considered as a function of the compressed quantized index size (the combined cost of the two blue boxes at top).

Figure 7: Single bin perturbation of index coefficients. Each plotted point represents an index in which a band of 1/128 of the coefficients have a small random quantity added or subtracted, and then retrieval fidelity is measured. Sensitivity is greatest for bins at the ends of the range of coefficient values.

In all of the results presented below space is represented as "fraction of 32 bits per value", so that an index that is represented in (say) an average of 12 bits per value is regarded as taking "space" of $12.0/32.0 = 0.375$. Our objective is to achieve storage rates below 0.5 (that is, below 16 bits per index value) and still retain high-accuracy ranking; and to achieve storage rates below 0.3 with only minimal loss of retrieval effectiveness.

To quantify retrieval fidelity we make use of Rank-Biased Overlap (RBO) [22], always comparing the ranking of k documents generated by an approximate index with a ranking of k documents resulting from the "exact" (that is, `float32`) index. In the first round of retrieval experiments an RBO parameter of $\phi = 0.999$ was employed. The benefits of this RBO-based approach are twofold. Firstly, no annotations are required to determine quality, and the measurement compares rankings solely on a "without approximation" and "with approximation" basis. Secondly, quality is captured via very deep listwise comparisons, which provide a high degree of confidence in the quality of the approximated indexes. We also used an unweighted overlap at k as a secondary measure. We report both the median (P_{50}) and the 95 th percentile (P_{95}) RBO values.

In the second round of retrieval experiments, reported shortly, we also make use of RBO and $\phi = 0.95$, to simulate the effect of the approximation on top-20 retrieval. This depth is appropriate if the dense similarity computation (Equation 1) is being used in the final stage of a multi-stage ranking pipeline.

The lower half of Figure 6 shows the experimental pipeline we have assembled, with queries executed against the original `float32` index and each of the approximate indexes, and then the original SERPs compared with each of the corresponding approximate SERPs using one or more of the overlap computations.

Experiment One: Sensitivity to Error. To confirm our conjecture that the most negative and most positive coefficients are the ones with the greatest influence over the documents placed near the head of the eventual ranking, we carried out a sensitivity experiment. Figure 7 shows the outcome of that exploration. To construct the graph, the 6,661,632 coefficients in the Con-Arg collection were
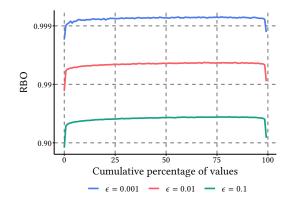
assigned to 128 equal-sized bins using the FD strategy. Taking one bin at a time as the focus, every value in that selected bin was then perturbed by a small random amount, thereby simulating the effect of the bounded error that might arise if the FR strategy was used. More precisely, each value x in the focus bin was replaced by a uniform-random value in the range $[x - \epsilon, x + \epsilon]$. All other values were held unchanged, and only one focus bin at a time had its values disrupted.

We then carried out a full retrieval run, and measured the resultant ranking degradation using 95 th percentile RBO and the parameter $\phi = 0.999$, with the curves in Figure 7 showing the extent to which each individual focus bin can affect overall retrieval fidelity. Moreover, because there is a constant number of elements in each of the focus bins, and a constant scale of disturbance applied to each of elements within them, the total volume of introduced error remains the same for each retrieval run, and fidelity can be fairly compared across the set of focus bins, that is, across the spectrum of coefficient values (the horizontal axis in the Figure 7). That entire process was repeated for three different values of ϵ. As is clear from the three plotted lines, the extremes of the coefficient range are where there is the greatest susceptibility to disruptions, and hence they are the regions in the range at which we can expect to need to have the smallest per-coefficient errors.

Experiment Two: First Stage Retrieval. Taken together, Figures 5 and 7 suggest that the FD method will not be an effective binning mechanism. Figure 8 confirms that outcome. Each pane shows one of the test collections, and each plotted point shows a possible trade-off that is available between stored index size on the horizontal axis, and ranked retrieval fidelity on the vertical axis. In this arrangement the top-left corner represents "perfection"; and each marked line corresponds to one method, with points at the top-right generated by high-precision approximations, and points trending to the lower-left as the lossiness grows via increasingly coarse approximations. In the case of the lossy quantized binning methods described in Section 3, the points on each curve from top-right leftward represent 4096, 2048, 1024, 512, 256, 128, 64, 32,
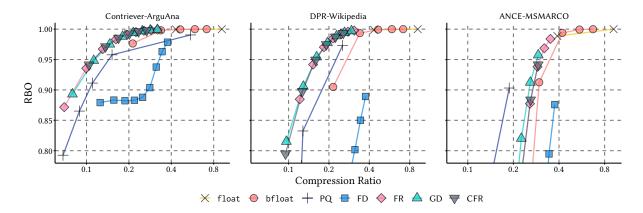
Figure 8: Trade-off curves showing the relationships possible between stored index size (as a fraction of 32 bits per index value, logarithmic scale) and answer fidelity when scored with RBO and $\phi = 0.999$ on lists of $k = 1,000$ answers. Three different collections are used and seven different representations for index coefficients. The RBO values plotted represent the 95 th percentile over the respective query sets.

and 16 bins respectively. The "X" at top-right of each plot indicates the location of the original `float32` index in this trade-off space. We reiterate that all indexes are assumed to be stored compressed, hence the "X" falling near 0.9 on the horizontal axis.

The FR, GD, and CFR approaches are absolutely superior for two of the collections, with very little to separate them in terms of "fidelity for size" performance; and are outperformed only by the complex PQ method in the low-fidelity part of the trade-off space for the third collection. As was anticipated above, the FD method is not competitive, and even the truncated $(S, E, M) = (1, 8, v - 9)$ `bfloat-v` floating point method offers a more attractive trade-off. Relative to the other three binned approaches the FD method suffers because it both allows high errors at the extremes of the coefficient range, and also gives rise to a uniform bin frequency distribution that does provide further space savings when the arithmetic coding stage is applied.

As already noted, the three graphs in Figure 8 reflect fidelity measurements recorded at the 95 th percentile (that is, 95% of the measured values across the query sets are higher than the points plotted). Very similar relationships at slightly higher RBO values arise if medians are used instead of 95 th percentiles, and we do not include those further graphs.

Experiment Three: Final Stage Retrieval. Figure 8 measures the effect of lossy approximations on deep retrieval, with RBO at $\phi = 0.999$ representing users who in average want their top-1,000 documents to be the "correct" set. This expectation might be associated with the fetching of a set of documents as part of a first-phase retrieval process, for example. Table 4 switches focus to shallower expectations, and repeats the experiment, but now using $\phi = 0.95$, simulating users who are on average intending to view the first 20 elements in the resultant rankings and want to know the extent of the overlap that will arise. This corresponds to final-stage ranking. Both median and 95 th percentile RBO scores are presented.

As can be seen from the table, the relationships between the various methods are very similar to what was already observed – indeed, when plotted as trade-off graphs, the overall pattern again

matches those illustrated in Figure 8. That is, for both shallow and deep retrieval, the FR, GD, and CFR approaches all allow very fine-grained adjustments to be made in a highly competitive region of the trade-off space between cost of permanently storing a dense index in lossily compressed form on the one hand, and retrieval fidelity on the other.

Time. While our primary focus in this investigation has been the relationship between long-term index storage space and retrieval accuracy, it is also interesting to consider the execution times for the various transformations that are involved, taking as a common starting point a `float32` dense index stored on SSD.

The binning process at the center of the approaches described in Section 3 requires that the complete index be sorted by value, and provided to a "quantizing" program that determines the bin boundaries and their representative values. Taking the largest index in all cases (ANCE-MSMARCO), the sorting cost was 937 seconds total CPU, executing in parallel taking 57 seconds elapsed time when allowed to use 32 threads. The quantizing decisions were then made in a further 40 seconds using a single thread. The third step, mapping index coefficients to bin numbers and then arithmetically coding those bin numbers, required between 300 seconds (64 bins) and 480 seconds (2048 bins) using a single thread, with the variation directly connected to the size of the compressed index being written. That step could also be parallelized with only a moderate amount of effort, further reducing the elapsed time requirement. In total, our current implementation builds a binned quantized index for ANCE-MSMARCO in around 10 minutes of elapsed time, with that time dominated by the single-threaded encoder.

This cost can be compared with the effort required to compute the PQ point present in the right-hand pane of Figure 8, which required 10 minutes on 32 threads, and approximately eight times the total amount of CPU work as our proposed approach.

5 CONCLUSION

We have described a representation for dense indexes that makes use of quantized index coefficients and arithmetic coding of integer bin numbers. Our goal was to discover new trade-off options for

Table 4: Detailed results for the same experiment as is depicted in Figure 8, except that fidelity is now measured using RBO with $\phi = 0.95$, reflecting overlap to an average depth of 20 in each answer ranking.

Type	E	M	Contriever-ArguAna			DPR-Wikipedia			ANCE-MSMARCO		
			Ratio	P_{50}	P_{95}	Ratio	P_{50}	P_{95}	Ratio	P_{50}	P_{95}
float32	8	23	0.915	1.000	1.000	0.919	1.000	1.000	0.917	1.000	1.000
float16	5	10	0.432	1.000	0.999	0.434	1.000	0.997	0.390	0.993	0.976
bfloat28	8	19	0.713	1.000	1.000	0.715	1.000	1.000	0.671	1.000	1.000
bfloat24	8	15	0.588	1.000	1.000	0.589	1.000	1.000	0.546	1.000	0.998
bfloat20	8	11	0.463	1.000	1.000	0.465	1.000	0.998	0.421	0.997	0.985
bfloat16	8	7	0.338	0.999	0.996	0.340	0.996	0.985	0.296	0.939	0.875
bfloat12	8	3	0.213	0.985	0.965	0.215	0.929	0.864	0.171	0.368	0.159
PQ-Best	-	-	0.549	0.994	0.981	0.253	0.981	0.946	0.188	0.924	0.856
FD4096	-	-	0.376	0.991	0.971	0.375	0.974	0.848	0.375	0.929	0.835
FD1024	-	-	0.313	0.970	0.932	0.313	0.933	0.761	0.313	0.798	0.616
FD256	-	-	0.250	0.944	0.889	0.250	0.853	0.658	0.250	0.552	0.285
FR4096	-	-	0.319	1.000	0.998	0.308	0.999	0.994	0.351	0.989	0.968
FR1024	-	-	0.256	0.999	0.995	0.246	0.994	0.983	0.289	0.956	0.906
FR256	-	-	0.193	0.995	0.984	0.183	0.977	0.948	0.226	0.842	0.701
GD4096	-	-	0.319	1.000	0.998	0.295	0.999	0.993	0.292	0.970	0.931
GD1024	-	-	0.249	0.999	0.994	0.228	0.992	0.977	0.226	0.881	0.768
GD256	-	-	0.180	0.993	0.980	0.162	0.964	0.928	0.160	0.576	0.329
CFR4096	-	-	0.366	1.000	0.998	0.366	0.999	0.989	0.366	0.990	0.970
CFR1024	-	-	0.304	0.998	0.992	0.304	0.990	0.974	0.304	0.920	0.840
CFR256	-	-	0.242	0.991	0.976	0.242	0.960	0.921	0.242	0.721	0.505

long-term storage of dense indexes, noting that even when stored in "full" as float32 values, there is approximation involved in the numeric representation, and in the underlying dot-product computations used for ranking documents. In experiments on three different dense indexes we demonstrated that three of the quantization methods we described offer an unequaled balance between stored index size and retrieval fidelity. The fourth quantization method is less interesting, and we have provided analysis that explains why.

The new methods offer a more fine-grained spectrum of trade-offs than do previous options based on truncated floating point representations; and are easier to compute than the product quantization method that allowed a small number of better trade-off options on one of the three test indexes. We are also aware that there has been other very recent work in this space [1], and developing an understanding of the relative merits of those techniques will be an important next step.

In future work we will explore compressed multi-representation dense indexes that store a c-dimensional embedding for each *term* in each document – such as the ColBERT family [10, 17] – for long-term retention. We will also explore methods for jointly coding sequences of versioned indexes to obtain further space savings, and consider multi-stage compression mechanisms, with the goal of allowing broad-to-fine hierarchical dense index representations.

Link to Software. In the interest of reproducibility, our implementations are available at https://github.com/JMMackenzie/lssy.

REFERENCES

[1] C. Aguerrebere, I. S. Bhati, M. Hildebrand, M. Tepper, and T. Willke. Similarity search in the blink of an eye with compressed indices. *Proc. VLDB Endow.*, 16(11): 3433–344, 2023.

[2] P. Bajaj, D. Campos, N. Craswell, L. Deng, J. Gao, X. Liu, R. Majumder, A. Mc-Namara, B. Mitra, T. Nguyen, M. Rosenberg, X. Song, A. Stoica, S. Tiwary, and T. Wang. MS MARCO: A human generated MAchine Reading COmprehension dataset. *arXiv:1611.09268v3*, 2018.

[3] T. Chen, L. Li, and Y. Sun. Differentiable product quantization for end-to-end embedding compression. In *Proc. ICML*, pages 1617–1626, 2020.

[4] H. De Silva, H. Tan, N.-M. Ho, J. L. Gustafson, and W.-F. Wong. Towards a better 16-bit number representation for training neural networks. In *Proc. Conf. Next Gen. Arithmetic*, pages 114–133, 2023.

[5] G. Izacard, F. Petroni, L. Hosseini, N. D. Cao, S. Riedel, and E. Grave. A memory efficient baseline for open domain question answering. *arXiv:2012.15156*, 2020.

[6] G. Izacard, M. Caron, L. Hosseini, S. Riedel, P. Bojanowski, A. Joulin, and E. Grave. Unsupervised dense information retrieval with contrastive learning. *Trans. Mach. Learn. Res.*, 2022.

[7] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Patt. Anal. Mach. Intell.*, 33(1):117–128, 2011.

[8] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *IEEE Trans. Big Data*, 7(3):535–547, 2021.

[9] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W. Yih. Dense passage retrieval for open-domain question answering. In *Proc. EMNLP*, pages 6769–6781, 2020.

[10] O. Khattab and M. Zaharia. ColBERT: Efficient and effective passage search via contextualized late interaction over BERT. In *Proc. SIGIR*, pages 39–48, 2020.

[11] J. Lin, X. Ma, S.-C. Lin, J.-H. Yang, R. Pradeep, and R. Nogueira. Pyserini: A Python toolkit for reproducible information retrieval research with sparse and dense representations. In *Proc. SIGIR*, pages 2356–2362, 2021.

[12] Y. Luan, J. Eisenstein, K. Toutanova, and M. Collins. Sparse, dense, and attentional representations for text retrieval. *Trans. Assoc. Comp. Ling.*, 9:329–345, 2021.

[13] X. Ma, M. Li, K. Sun, J. Xin, and J. Lin. Simple and effective unsupervised redundancy elimination to compress dense vectors for passage retrieval. In *Proc. EMNLP*, pages 2854–2859, 2021.

[14] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, Boston, MA, 2002.

[15] K. Pearson. LIII: On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[16] G. E. Pibiri and R. Venturini. Techniques for inverted index compression. *ACM Comp. Surv.*, 53(6):125.1–125.36, 2021.

[17] K. Santhanam, O. Khattab, J. Saad-Falcon, C. Potts, and M. Zaharia. ColBERTv2: Effective and efficient retrieval via lightweight late interaction. In *Proc. NAACL*, pages 3715–3734, 2022.

[18] N. Thakur, N. Reimers, A. Rücklé, A. Srivastava, and I. Gurevych. BEIR: Heterogenous benchmark for zero-shot evaluation of information retrieval models. In *Proc. NeurIPS (Datasets and Benchmarks)*, 2021.

[19] N. Thakur, N. Reimers, and J. Lin. Domain adaptation for memory-efficient dense retrieval. arXiv:2205.11498, 2022.

[20] H. Wachsmuth, S. Syed, and B. Stein. Retrieval of the best counterargument without prior topic knowledge. In *Proc. ACL*, pages 241–251, 2018.

[21] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Proc. NeurIPS*, pages 7686–7695, 2018.

[22] W. Webber, A. Moffat, and J. Zobel. A similarity measure for indefinite rankings. *ACM Trans. Inf. Sys.*, 28(4):20.1–20.38, 2010.

[23] L. Xiong, C. Xiong, Y. Li, K.-F. Tang, J. Liu, P. N. Bennett, J. Ahmed, and A. Overwijk. Approximate nearest neighbor negative contrastive learning for dense text retrieval. In *Proc. ICLR*, 2021.

[24] I. Yamada, A. Asai, and H. Hajishirzi. Efficient passage retrieval with hashing for open-domain question answering. In *Proc. ACL and IJNLP*, pages 979–986, 2021.

[25] P. Yang, H. Fang, and J. Lin. Anserini: Reproducible ranking baselines using lucene. *J. Data Inf. Qual.*, 10(4):1–20, 2018.

[26] J. Yun, B. Kang, and Z. Fu. The hidden power of pure 16-bit floating-point neural networks, 2023. arXiv:2301.12809.

[27] J. Zhan, J. Mao, Y. Liu, J. Guo, M. Zhang, and S. Ma. Jointly optimizing query encoder and product quantization to improve retrieval performance. In *Proc. CIKM*, pages 2487–2496, 2021.

[28] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comp. Surv.*, 38(2):6:1–6:56, 2006.

[29] V. Zouhar, M. Mosbach, M. Zhang, and D. Klakow. Knowledge base index compression via dimensionality and precision reduction. In *Proc. Wrksp. Spa-NLP*, pages 41–53, 2022.