



INSTITUT FÜR SOFTWARE & SYSTEMS ENGINEERING
Universitätsstraße 6a D-86135 Augsburg

Rethinking Vector Embeddings Search for Analytical Database Systems

Elena Krippner

Masterarbeit im Elitestudiengang Software Engineering





INSTITUT FÜR SOFTWARE & SYSTEMS ENGINEERING
Universitätsstraße 6a D-86135 Augsburg

Rethinking Vector Embeddings Search for Analytical Database Systems

Verfasserin: Elena Krippner, B. Sc.
Matrikelnummer: 1680792
Beginn der Arbeit: 15. April 2024
Abgabe der Arbeit: 09. Oktober 2024
Erstgutachter: Prof. Dr. Thomas Neumann
Betreuer: Prof. Dr. Peter Boncz
Leonardo Xavier Kuffo Rivero, M. Sc.



SOFTWARE ENGINEERING
Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 09. Oktober 2024

Elena Krippner

Abstract

Vector embeddings search (VES) is an important component of applications such as pattern recognition, recommendation systems, and retrieval-augmented generation. This search consists of performing a nearest neighbor search (NNS) on the numerical representation, called vector embeddings, of unstructured data like text and audio. Performing an exact NNS on big datasets is not feasible due to the computational demand that an exhaustive search poses. However, in many applications, having an error in the answers is acceptable. As such, research on VES has been focusing on computing approximate results. One of the most common ways to improve the search speed is to compress the vector embeddings in a lossy fashion.

State-of-the-art embedding compression algorithms are quantization-based algorithms that have either high encoding times or lead to aggressive changes in the values in the vectors. Other compression techniques, such as downcasting, maintain a high recall but have limited compression capabilities.

We propose Lossily Encoded floating-Points (LEP), a lossy variant of the ALP algorithm [2] that does not only reduce the size of vector embeddings substantially but also reaches low reconstruction errors and high recalls in an approximate NNS. Additionally, we propose new compression techniques and explore data formats reminiscent of the ones used in analytical databases, specially tailored for vector embeddings. In our experiments, LEP achieves compression ratios comparable to or better than quantization- and downcasting-based techniques when targeting the same recalls. Furthermore, LEP is able to compress vector embeddings without aggressively altering their individual values. As a result, the compressed representation of the data is more accurate in representing the original vectors than quantization-based techniques.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | Research Questions | 2 |
| 1.2 | Outline | 2 |
| 2 | Background | 3 |
| 2.1 | Nearest Neighbor Search | 3 |
| 2.1.1 | Exact Nearest Neighbor Search | 3 |
| 2.1.2 | Approximate Nearest Neighbor Search | 5 |
| 2.2 | Compression | 6 |
| 2.2.1 | Compression of Floating-Point Numbers | 7 |
| 2.2.2 | Compression of Vector Embeddings | 8 |
| 3 | Analysis of Vector Embedding Datasets | 15 |
| 3.1 | Overview of the Datasets | 15 |
| 3.2 | Compression of Embedding Datasets using ALP | 18 |
| 3.3 | Exceptions in the Frame Of Reference | 19 |
| 3.4 | Data Layouts for Storing the Embeddings | 20 |
| 3.5 | Frequently Occurring Values | 22 |
| 3.6 | Correlations between Dimensions | 22 |
| 4 | Lossily Encoded floating-Points | 25 |
| 4.1 | Compression of Exceptions | 27 |
| 4.2 | Bitmap-Compression of Frequent Values | 28 |
| 4.3 | Non-Decimal LEP for Fine-Tuning Compression Ratios and Recalls | 28 |
| 4.4 | Speedup of the Encoding Time | 29 |
| 5 | Evaluation | 31 |
| 5.1 | Assessment of Data Layouts | 31 |
| 5.2 | Comparison against Other Algorithms | 34 |
| 5.2.1 | Considered Algorithms | 34 |
| 5.2.2 | Compression Ratio and Mean Squared Error | 34 |
| 5.3 | Evaluation of the Encoding Speeds | 41 |
| 6 | Conclusion and Future Work | 42 |
| 6.1 | Answers to the Research Questions | 42 |
| 6.2 | Future Work | 43 |
| | Bibliography | 45 |

List of Figures

| | | |
|------|---|----|
| 2.1 | Storage of 3 embeddings in the (a) horizontal and (b) vertical layout | 4 |
| 2.2 | Pruning of distance calculations in the BOND algorithm (Figure borrowed from De Vries et al. work [14]) | 4 |
| 2.3 | An exemplary search in a graph built by the HNSW algorithm (Figure borrowed from Malkov and Yashunin work [47]) | 6 |
| 2.4 | Process of encoding doubles with the ALP algorithm | 8 |
| 2.5 | Overview of the different quantization approaches: (a) binary quantization, (b) scalar quantization, (c) product quantization, (d) additive product quantization, (e) additive quantization, and (f) vector quantization (Figures (c) and (e) are borrowed from Babenko and Lempitsky work [6]) | 9 |
| 2.6 | Mapping of a range of floating-point numbers to integers during a SQ encoding (Figure borrowed from Qdrant blog [76]) | 11 |
| 2.7 | Clustering of points (shown in black) in a 2-dimensional subspace into 4 clusters with centroids shown in orange | 11 |
| 2.8 | Encoding of an embedding (x) using two codebooks (C1 and C2) in the RVQ algorithm (Figure borrowed from Chen et al. work [13]) | 13 |
| 2.9 | The hierarchy of quantization algorithms (Figure borrowed from Douze et al. work [18]) | 14 |
| 2.10 | Bitwise representations of floating-point numbers: (a) IEEE single precision and (b) 16 bit | 14 |
| 3.1 | The recall of LEP for different LEP exponents and compression ratios on embedding datasets | 19 |
| 3.2 | Heatmaps of the floating-point number distribution within part of the datasets; every square represents one float in the datasets, and its color hue indicates its value (darker colors belong to higher values) | 21 |
| 3.3 | The share of LEP vectors with repeated values (occurring > 300 times in the LEP vector) for some embedding datasets | 22 |
| 4.1 | The process of encoding floats using LEP | 25 |
| 4.2 | An example of compressing a range of numbers containing compressible exceptions in the PFOR in LEP | 27 |
| 4.3 | Compressing a LEP vector using the bitmap compression of frequent values | 28 |
| 5.1 | The compression ratio of LEP on data preprocessed into the vertical and horizontal layout | 32 |
| 5.2 | The compression ratio of LEP on data preprocessed using clustering or no clustering | 33 |
| 5.3 | The compression ratio of LEP compared to other compression algorithms | 35 |
| 5.4 | The mean squared error of LEP compared to other compression algorithms | 36 |
| 5.5 | The compression ratio - MSE curve of LEP and SQ calculated on single dimensions (Dim); for SQ, parameters were both calculated per dimension and for the whole dataset | 38 |
| 5.6 | The compression ratio - MSE curve of LEP with and without a fixed bit width and SQ calculated on single dimensions (Dim) | 38 |
| 5.7 | The compression ratio - recall curve of LEP with the bitmap encoding on all the datasets where the bitmap encoding is applicable | 39 |

5.8 The compression ratio - recall curve of LEP using non-decimal LEP on some datasets 40

List of Tables

| | | |
|-----|---|----|
| 3.1 | Overview of the vector embedding datasets | 17 |
| 3.2 | The compression ratio of ALP on embedding datasets | 18 |
| 3.3 | The numbers of regular and compressible exceptions in the PFOR in LEP on some embedding datasets | 20 |
| 3.4 | The number of pairs of columns in embedding datasets that are correlated with a high Pearson coefficient | 23 |
| 3.5 | The compression ratio of LEP using correlated dimensions aggressively on GIST and Fashion-MNIST | 24 |
| 3.6 | The compression ratio of LEP using correlated dimensions losslessly on GIST and Fashion-MNIST | 24 |
| 5.1 | The compression ratio (CR) and encoding time (ET) of sampling vectors at encoding time in the LEP algorithm | 41 |

1 Introduction

In today’s digital age, there is an exponential growth of unstructured data like user-generated texts and images on social media platforms, sensor data from IoT devices, and songs stored in large databases. A typical operation on these data is finding similar items to a query item. For instance, songs that resemble those a user listened to can be searched for to generate recommendations. *Vectorization*¹ of unstructured data is an approach to making the items machine-processable by representing them as fixed-sized arrays of (usually) floating-point numbers. These representations are known as *vector embeddings*, or short, *embeddings*. Using these representations, similar items can be identified by performing a *nearest neighbor search* (NNS).

In order to vectorize a data item, features are identified and represented as a numerical vector with the same size for all items. One option to get such features is to define them manually. For example, a vector for a song could be represented by its overall loudness in dB, its tempo in BPM, its duration in seconds, and its danceability, which is calculated by another algorithm [7]. An alternative to the manual feature definition is using a machine learning model that learns features from the unstructured data, like the word2vec model that vectorizes words based on the context in which they appear [54]. On most features of this embedding representation, a human can no longer interpret what they represent. However, the model is trained so that similar items have similar vectorized representations when compared using a predefined similarity metric.

Typical applications of NNS include pattern recognition (e.g., recognizing numbers in images that represent hand-written digits [42]), recommendation systems (e.g., pointing readers of a digital newspaper to articles similar to ones they read before using the word2vec model [54]), and retrieval-augmented generation (e.g., by supplying large language models with factual information fitting to a prompt [70]). In addition to the typical applications mentioned earlier, it is possible to apply vectorization to data of any domain (e.g., molecule structures [50] and rentable apartments [30]) and perform NNS on it.

Computing the results of an exact NNS [14, 22, 37] requires a high number of floating-point computations. The latter makes it unfeasible on a large-scale because of the high query throughput required by modern applications. However, many applications can tolerate approximate answers, which led to an interest in developing algorithms for approximate NNS [21, 39, 47]. For example, in the aforementioned use-case of song recommendations, any song similar enough to the songs a user listened to can be of interest. This is because the vectorization of a song is an approximation by itself. An advantage of approximate NNS is that the query time can be sped up (sometimes even by orders of magnitudes [5, 72]), for example, by using approximate indices.

Reducing the size of the vectors has also been utilized to improve approximate NNS runtimes further. By applying the compression to the embeddings, less bytes need to be transferred in the memory hierarchy, and computations become cheaper (e.g., by fitting more values into a SIMD lane [52]). State-of-the-art compression approaches for vector embeddings are *quantization* [18, 26, 39] and *downcasting* [18]. Quantization consists of reducing the size of the vectors by mapping them onto another vector of integers. The most frequently used quantization technique is Product Quantization (PQ) [39]. PQ divides each vector into equally sized subvectors and maps each subvector to the index of its nearest neighbor in a list of subvectors computed by a clustering algorithm. PQ and most other quantization techniques can fail to achieve high recalls in some datasets because they aggressively change the values in the vectors and thus compromise their reconstruction error [25]. On the other hand, downcasting reduces the size of floating-point numbers

¹Here, vectorization denotes the creation of vector embeddings. In the context of analytical database systems, this term is usually used for the processing of multiple values at a time, which often leverages SIMD instructions. Thus, those terms are not related.

by using a smaller float representation that requires less bits. Despite having a smaller reconstruction error, downcasting is limited by the compression ratios it can get. A common datatype used in downcasting is a 16-bit float [18], which means that the compression ratio is only 2.00 (as embeddings are usually represented by 32-bit floats).

We introduce Lossily Encoded floating-Points (LEP), an algorithm to compress vector embeddings that minimizes the reconstruction error of vectors while simultaneously achieving high compression ratios. LEP is the lossy version of ALP [2]. ALP is a recently proposed algorithm to compress floating-point data that converts these numbers losslessly to integers and applies the integer compression algorithm Frame Of Reference on them.

Furthermore, we propose to rethink vector embeddings storage from an analytical database perspective. We leverage changing the data layout using a column-based storage to improve compression ratios. In addition to this, future research projects on NNS can exploit this layout to perform a faster search.

1.1 Research Questions

The topics of this research can be divided into four research questions that this thesis will answer.

Q1: Compressing Embeddings Lossily Based on the ALP Algorithm Which compression ratios can be reached when compressing embeddings losslessly using ALP? How is the recall of the nearest neighbor search affected by making ALP lossy?

Q2: Improving the Compression Ratio on LEP Can the compression ratio be improved by exploiting a data layout for vector embeddings that shares ideas with data layouts in analytical database systems? Which other characteristics do embedding datasets have that can be exploited to amplify the compression? Can we control the compression ratio reached by the lossy algorithm?

Q3: Using Correlated Dimensions Do the embedding datasets contain correlated dimensions? How can they be used to improve the compression ratio?

Q4: Comparing LEP to Other Embedding Compression Algorithms How does LEP perform concerning compression ratio and mean squared error compared to other embedding compression algorithms?

1.2 Outline

The rest of this thesis is organized as follows. In chapter 2, we give an overview of algorithms for nearest neighbor search and compression of floating-point numbers and vector embeddings. An in-depth analysis of often-used vector embedding datasets and their compressibility potential is given in chapter 3. Here, we uncover compression opportunities that have not been explored before on this type of data. In chapter 4, we present the Lossily Encoded floating-Points compression algorithm and a sampling scheme to speed up the encoding time of the algorithm. Results on the compression ratio and recall reached by LEP and the encoding speed when using the sampling scheme can be found in chapter 5. Finally, chapter 6 summarizes the work done in this thesis by answering the research questions formulated in section 1.1 and suggests improvements and future work.

2 Background

2.1 Nearest Neighbor Search

We define the NNS problem as follows. The set of embeddings searched to find the nearest neighbor of a query embedding is the *database*, and vector embeddings in it are called *database vectors*. A *query vector* or *query* is the embedding for which the algorithm must find its nearest neighbors in the database. A *nearest neighbor* is an embedding within the database whose distance to the query is not bigger than the distance of any other embedding to the query within the database. The distance between embeddings is measured using a *similarity metric*. Similarity metrics are, for example, Hamming distance, inner product, Euclidean distance, and cosine similarity. A variant of NNS is the k -nearest neighbor search. Here, the result consists of k embeddings closest to the query. Algorithms for NNS are classified into exact and approximate search. While the exact search does not allow false negative or positive results, it is computationally more expensive.

2.1.1 Exact Nearest Neighbor Search

The algorithmically easiest solution for getting exact results is computing the distance from the query to every database vector. Thus, the whole dataset needs to be scanned for each query. However, as distance computations are expensive, many optimization techniques have been proposed. These include tree-based methods that use tree data structures as indices that guide the search process. Examples are KD-Trees by Friedman et al. [22], R-Trees by Guttman [31], and Cover Trees by Beygelzimer et al. [9]. All of these require backtracking on the tree to guarantee correct results. While these data structures work well for low dimensional spaces (up to about 10 dimensions), they end up visiting almost all database vectors in high dimensional spaces (more than about 10 dimensions) [8, 57]. As modern vector embeddings usually have many dimensions, these techniques cannot find improvements over the linear scan.

Another approach is reducing the number of computations by computing bounds on the distance metric. During a search, algorithms can use such bounds to prune vectors after calculating the distance only on a few dimensions if the bound guarantees they cannot be in the result. Different bounds for the Euclidean distance were introduced by Jeong et al. [37], Hwang et al. [32] and Zhang et al. [73].

De Vries et al. [14] propose storing and processing vectors in a *vertical layout*. The idea behind this vertical layout and the difference from a *horizontal layout* is illustrated in figure 2.1. In the horizontal layout (figure 2.1 (a)), the vectors are stored sequentially in memory. Here, the horizontal layout stores the embeddings e_1 , e_2 , and e_3 one after the other. On the other hand, in the vertical layout (figure 2.1 (b)), the dimensions of each vector are stored together. In this example, the values of the first dimension of each embedding are placed after each other, followed by the values of the second dimension, and so on.

This vertical layout is used in the Branch-and-bound ON Decomposed data (BOND) algorithm by De Vries et al. [14]. The idea of BOND is to calculate the exact distance on only a few dimensions for all vectors. In the following steps, it computes bounds on the rest of the distance for all vectors, which enables BOND to exclude some vectors from further computations as they are guaranteed not to be in the NNS result. BOND then updates the distances of the vectors that are still candidates and repeats these steps until it has found the k results to the NNS query.

Figure 2.2 illustrates the idea behind the BOND search. The whole box represents the stored values. Here, the embeddings are stored using the vertical layout. First, the algorithm calculates the distance on the first

$$\mathbf{e}_1 = (7.2, 9.3, 8.6)$$

$$\mathbf{e}_2 = (1.6, 3.4, 5.5)$$

$$\mathbf{e}_3 = (9.0, 1.3, 4.3)$$



Figure 2.1: Storage of 3 embeddings in the (a) horizontal and (b) vertical layout

$m = 8$ dimensions of each embedding. Then, it calculates an upper and lower bound for the rest of the distance for each embedding using a formula less computationally expensive than the actual Euclidean distance. With these bounds, BOND can find the k -th best worst-case distance and use it as a threshold. All vectors whose best-case distance is greater than this threshold cannot be in the final result anymore, and the rest of the distance calculations on these vectors can be skipped. The light grey boxes in the figure contain the first dimensions of the embeddings that are still candidates after each step. The white boxes show the embeddings that can be pruned after each step. The dark grey embeddings are the ones that were pruned in the previous step. Therefore, the distance computations are avoided here. BOND repeats the steps of updating the distances and pruning computations until it reaches the last dimension, at which point it knows the exact distances and can obtain the best k vectors.

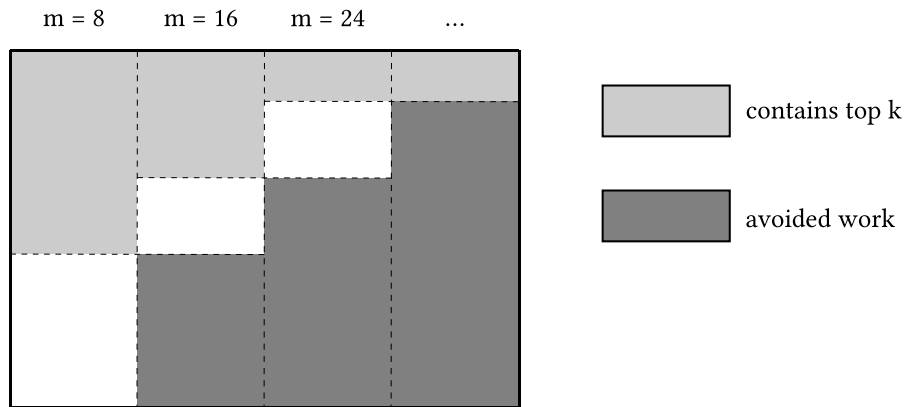


Figure 2.2: Pruning of distance calculations in the BOND algorithm (Figure borrowed from De Vries et al. work [14])

Using the vertical layout is required for this algorithm to be fast because it accesses the values per dimension sequentially when updating the distances. These values are stored sequentially in memory, and this leads to less random access. Parameters that affect the number of calculations pruned are the ordering of

the dimensions and the number of newly considered dimensions between two pruning phases (the parameter m).

2.1.2 Approximate Nearest Neighbor Search

Exact NNS is computationally expensive because it needs to guarantee that skipped computations do not change the result. However, some applications can tolerate approximate results. The latter opened opportunities to improve the speed of NNS by factors. An often-used metric to measure the quality of the answer to a query is " n -recall@ m ." To calculate this metric on a query, the search algorithm obtains the m nearest neighbors to the query. Furthermore, a ground truth containing the n nearest neighbors needs to be provided. The recall is then the share of the n ground truth values within the m search results. If $n = m$, this equals the set intersection of both result sets.

Approximate NNS algorithms are commonly categorized as table- [39, 35], tree- [56, 21], and graph-based [17, 23, 47] algorithms. All of these build index structures that aid the search.

Table-based algorithms divide the database vectors into buckets containing similar vectors. At search time, these algorithms determine if a bucket has a low probability of containing the results of a query. They exclude the vectors within these buckets from the distance evaluation. Ignoring buckets makes these algorithms approximate, as buckets they skip might contain embeddings that are true positive values. A commonly used algorithm to bucket database vectors is the Inverted File Index (IVF) [64]. On IVF, a clustering algorithm (usually k -means) partitions the vectors into buckets. Then, each vector is assigned to the bucket whose centroid is closest to it. Locality-Sensitive Hashing (LSH) is an alternative to using a clustering algorithm. LSH uses hash functions that map similar embeddings to the same bucket with a high probability [35].

Trees for approximate nearest neighbor search repeatedly split the remaining space that would have been searched. State-of-the-art algorithms for this include the Fast Library for ANN (FLANN) by Muja and Lowe [56] that chooses between different approximate tree-based algorithms based on the dataset or ANN Oh Yeah (ANNOY) by Bernhardsson that has been used at Spotify [21].

Graph-based indices represent each database vector as a node; for some, the distance between them is included as an edge. In k -nearest neighbor graphs, edges connect each node to the nodes representing its k -nearest or approximately nearest neighbors. Examples of k -nearest neighbor graphs are NN-Descent by Dong et al. [17] and EFANNA by Fu and Cai [23].

One of the most used graph-based algorithms in vector database systems is Hierarchical Navigable Smallest Worlds (HNSW) by Malkov and Yashunin [47]. HNSW has a logarithmic search time in best-case scenarios. Figure 2.3 illustrates the idea of the algorithm. HNSW builds a hierarchy of graphs. When searching, the algorithm performs a greedy search starting at a fixed node on the topmost layer. In the figure, the starting node is the orange one in layer 2. Its closest neighbor is the blue node. Once the greedy search cannot find a closer neighbor to the query in its current layer, it traverses to the next layer, where it performs the greedy search again until it finds the closest node in the bottom layer (layer 0). In the figure, this is the green node in the lowest layer.

Complementary to these index-based algorithms, Gao and Long propose ADSampling in [24]. They noticed that distance computations play a significant role in the execution time of most of these algorithms. Similar to the exact algorithms mentioned in section 2.1.1, they try only to calculate part of the Euclidean distance. Unlike these exact algorithms, however, ADSampling performs a hypothesis test every 32 dimensions to prune embeddings when it is unlikely that the embeddings can still be candidates. As such, it can only yield approximate results.

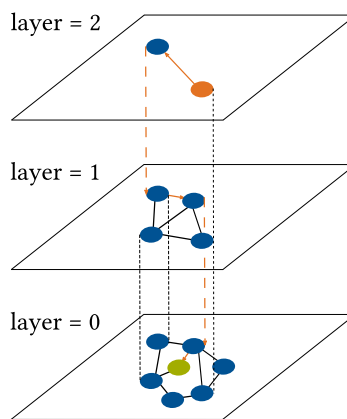


Figure 2.3: An exemplary search in a graph built by the HNSW algorithm (Figure borrowed from Malkov and Yashunin work [47])

2.2 Compression

In many applications like database queries [1] and approximate nearest neighbor search [39], compression is used to reduce storage usage and achieve faster data access. Compressed representations can fit more data higher in the memory hierarchy. As a result, answers to queries can be faster as less bits of data have to be fetched.

Compression methods are classified into *general-purpose compression* (GPC) and *lightweight compression* (LWC) methods. The first ones do not require information on the type of data they will compress. Instead, they analyze large chunks of bytes and leverage repeating patterns to achieve compression. For example, the GPC algorithm Zstandard [51], also called Zstd, finds recurring patterns to build a dictionary that maps these patterns into smaller codes. More frequent occurring patterns are assigned smaller codes using a variation of a Huffman coder. However, to find sequences leading to high compression ratios, these kinds of compressors need to work on large data blocks. Moreover, random access to compressed values is difficult since they must decompress the whole block in which the value is.

On the other hand, LWC algorithms are tailored to only work on specific datatypes. This enables them to exploit characteristics within the data domain that are not visible to GPC algorithms. Therefore, they are magnitudes of speed faster [2, 75] and allow decompressing single values faster than GPC algorithms. Among the most famous LWC compression techniques we have Frame Of Reference (FOR) by Goldstein et al. [29] and Tuple Differential Coding, also known as DELTA by Raman and Swart [63] for integers, Fast Static Symbol Table (FSST) by Boncz et al. [11] for strings and Adaptive Lossless floating-Point Compression (ALP) by Afroozeh et al. [2] and Chimp by Liakos et al. [44] for floating-point numbers.

While this thesis focuses on compressing floating-point numbers (as vector embeddings usually consist of floats), we explain the FOR algorithm in more detail as it is a building block of the ALP algorithm that is used for floating-point compression. FOR works well on data that can be stored within a smaller range than the integer datatype offers. By subtracting the minimum value of the original range, the compressed values are shifted to zero and can then be bitpacked. *Bitpacking* is the process of determining and using the minimum number of bits required to store a range of values. Take, for example, the binary values `0001` and `0010`. While the first one only requires one bit (`1`) to be stored, the second one needs two (`10`). For bitpacking, two is the bit-width with which both numbers will be stored. Then, the values are concatenated using this bit-width. In the example, `0110` would be the result of storing both `0001` and `0010` with a bit-width of 2.

A variant of FOR is the Patched Frame Of Reference (PFOR) from Zukowski et al. [75]. PFOR exploits

the fact that datasets usually contain outliers. Outliers are values that require a bigger bit-width than most of the values in the data range. PFOR tries to store the dataset using a bit-width that fits the values of the dataset excluding outliers. Consequently, the outliers in the data cannot be stored in the bitpacked array. To solve this, the authors propose treating them as *exceptions* kept in another array. To remember which values are exceptions, PFOR stores the index of the first exception separately. Within the bitpacked array, the bits at the position where there is an exception are used to remember the offset to the following exception. To patch the exceptions for decompression, the algorithm finds the exceptions by following the positions that form a linked list. To figure out which values should be outliers, PFOR calculates a FOR base that leads to a minimal number of exceptions for each bit-width. Out of those, it chooses the bit-width and FOR base leading to the highest compression ratio.

All the algorithms mentioned above, both for GPC and LWC, are lossless. This means that the decompressed values are identical to the values that the algorithm had to compress. In some applications, this is not required. For instance, sometimes scientific sensor data do not need to be stored using the full floating-point precision as they contain noise [71]. Similarly, in the context of NNS, exactness is not required, as returning "close-enough" neighbors can be acceptable. Therefore, a lossy compression algorithm can be used.

2.2.1 Compression of Floating-Point Numbers

Single-precision (32-bit) floating-point numbers (figure 2.10 (a)), which are also known as floats, are defined in the IEEE 754 standard [34]. Floats contain a sign bit, 8 exponent bits, and 23 mantissa bits. Let S be -1 if the sign bit is 0 and 1 otherwise. F are the mantissa bits, and E is the decimal number the exponent bits represent. The number can then be reconstructed as $S * 1.F * 2^{E-127}$ where $1.F$ is a binary number.

Lossless compression algorithms for floating-point numbers can be grouped into three main categories: predictive, XOR, and decimal-based schemes. Predictive schemes like Delta Predictive Coding (FSD) by Engelson et al. [20] and FPC by Burtscher and Ratanaworabhan [12] use a function that predicts a value close to the value to compress based on the data seen before. Then, the original value is compressed by using the deviation from the prediction.

On the other hand, XOR schemes use the XOR operator on two numbers that are stored close to each other. If the numbers are similar, the result of the XOR will contain many 0s, which these algorithms leverage to represent the value with less bits. Examples of XOR schemes include the algorithms Gorilla by Pelkonen et al. [60], Chimp by Liakos et al. [44], Patas developed by DuckDB Labs [19] or Elf by Li et al. [43].

On floating-point numbers, lightweight compression algorithms like FOR or DELTA cannot be used because they can introduce arithmetical errors on floats, which would lead to a lossy compression. However, decimal-based schemes represent floats as integers losslessly to enable the usage of integer compression schemes. Examples of these schemes include the algorithms BoUnded Fast Floats compression (BUFF) by Liu et al. [45], PseudoDecimals (PDE) by Kuschewski et al. [41] and ALP by Afroozeh et al. [2].

The idea behind the ALP algorithm is that, in real-world datasets, many floating-point numbers have a fixed precision. Therefore, it is possible to represent them as integers by performing a multiplication with a power of 10 and, in a second step, use integer lightweight compression algorithms on them. Figure 2.4 shows the process of encoding a vector $[n_0, n_1, n_2, \dots, n_{1023}]$ of 1024 values using ALP.

To represent the floating-point numbers as integers, ALP multiplies each value n_i by a power of 10, whose inverse does not introduce an error when decompressing. Because of floating-point arithmetic errors, this number might be bigger than the mathematically required value. Thus, the algorithm removes possible trailing zeros by doing another multiplication with a power of 10 with a negative exponent. In the figure, this is the step between the n_i and d_i values. ALP uses the same parameters per ALP vector of 1024 values. With this, it is possible that the chosen parameters do not work losslessly on all values. As a solution, the algorithm saves these values as exceptions by storing their positions and values in two arrays separated from the compressed data. The resulting integers are still of the same size as the doubles. To compress these integers, ALP uses the FOR to get the sequence $[d'_0, d'_1, d'_2, \dots, d'_{1023}]$.

For data that do not have a fixed precision, ALP uses ALP for Real Doubles (ALP_{rd}). Here, the algorithm leverages the fact that the first bits of the doubles in a dataset usually show little variance because data often have the same exponent and sign. Therefore, ALP_{rd} replaces this left part of the binary representation with a dictionary encoding while the rest is bitpacked.

Compared to the XOR-based compression algorithms mentioned above and PDE, ALP is superior in terms of compression and decompression speed. The authors report a speedup of factor 8x to 251x for compression and a factor of 7x to 215x for decompression on 30 real-world datasets. While ALP did not achieve the best compression factor in all of them, the average number of bits per value over all datasets was between 1.4 and 20.5 bits smaller compared to the other encodings.

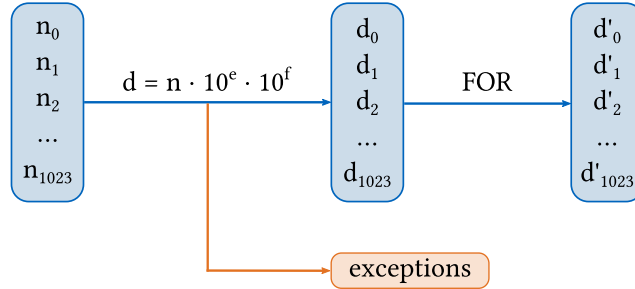


Figure 2.4: Process of encoding doubles with the ALP algorithm

Complementary to the lossless floating-point compression algorithms mentioned above, there has been research on lossy compression algorithms [16]. However, to the best of our knowledge, apart from quantization, they have not been used on compressing vectors. We attribute the latter to these algorithms being relatively slow in both compression and decompression for the throughput needed on NNS applications. However, the speed benchmarks reported by ALP [2] open the possibility of using it in an NNS context.

The algorithms Digit Rounding [15] and Bit Grooming by Zender [71] use the idea of pruning the precision of floating-point numbers after a fixed number of digits. They prune the precision on the bit representation. The algorithms calculate how many bits of the mantissa are required to keep the requested precision. The rest of the bits are then either set to 0 or 1. After pruning, the algorithms use a general-purpose compression algorithm.

2.2.2 Compression of Vector Embeddings

Vector embeddings are usually comprised of high-precision floating-point numbers. This makes them hard to compress losslessly. In the ALP paper [2], the authors compare Gorilla, Chimp, Patas, and ALP on high-precision machine learning data. While ALP reached a reduction of the size of floats to 28.1 bits per value (compression ratio 1.14), all other algorithms needed at least 33.4 bits per value (compression ratio 0.96) and thus had a negative compression. As such, lossy approaches like quantization are the de facto standard to reduce the size of the vectors. In the context of quantization, decompression is called reconstruction.

Depending on the chosen quantization algorithm and the chosen parameters for it, the values of the vectors can drastically change in order to reduce their size. However, to reach high recalls, the raw vectors must additionally be maintained in storage to be able to perform a re-ranking. As a result, the real compression ratio is negative, as both the raw and compressed values are kept.

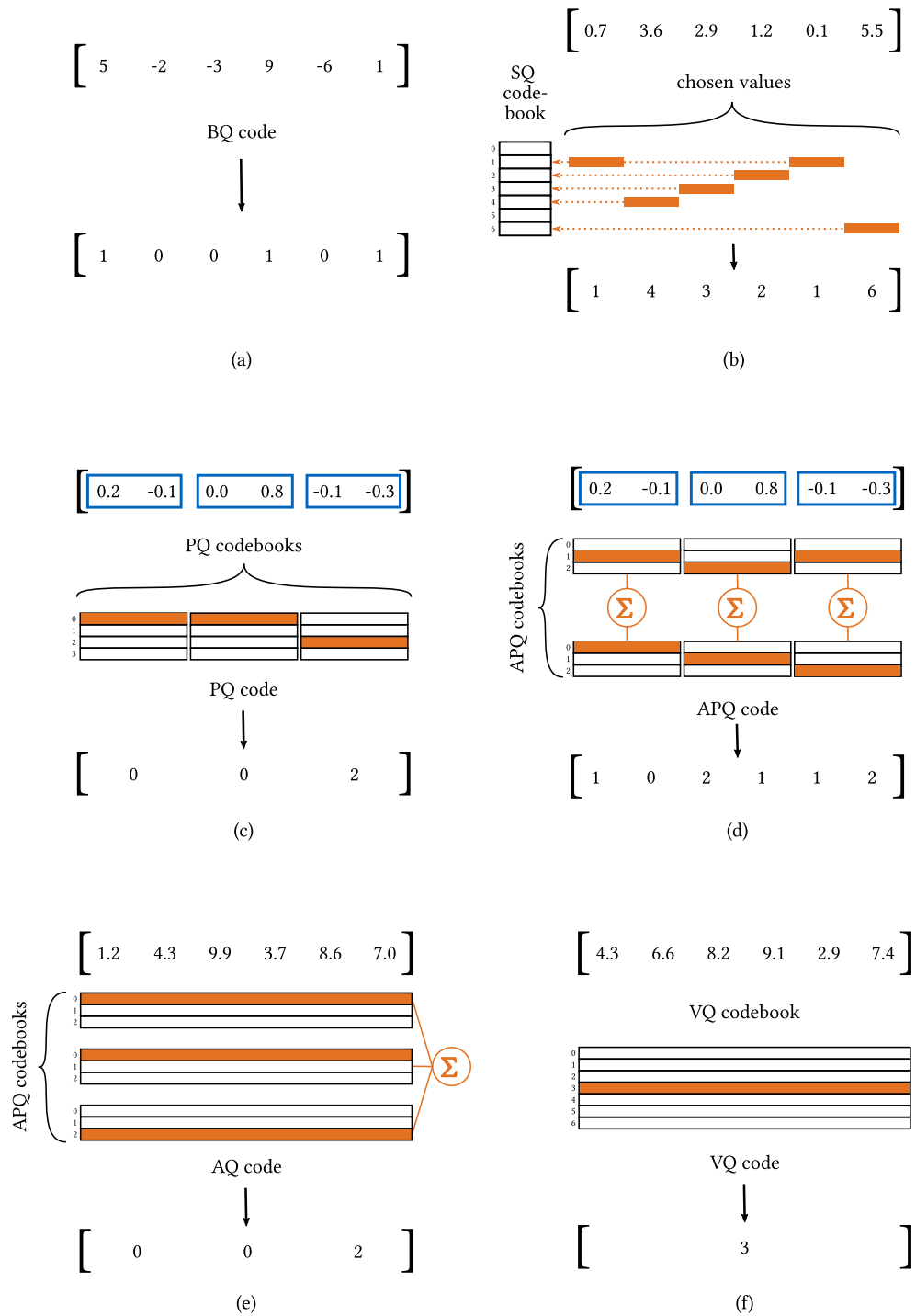


Figure 2.5: Overview of the different quantization approaches: (a) binary quantization, (b) scalar quantization, (c) product quantization, (d) additive product quantization, (e) additive quantization, and (f) vector quantization (Figures (c) and (e) are borrowed from Babenko and Lempitsky work [6])

Quantization

Quantization consists of mapping a sequence of floating-point numbers to a finite set of integers called codes with a usually smaller cardinality. A smaller cardinality is desired on this set of codes to be able to represent the original floating-point numbers with fewer bits to save space. As it might be impossible to map the floating-point numbers onto this smaller range of values, quantization is generally lossy. Quantization was first used in the context of compressing signals before transmission [27]. Jégou et al. proposed to use quantization in the context of ANNS [39]. Since then, it has been used as the de-facto technique to reduce the size of vector embeddings, and many vector database systems adopted it, including Pinecone [61], Milvus [55], and Weaviate [68].

Different types of quantization techniques exist that trade off higher query speeds, smaller data sizes, and higher recalls. Among the most popular ones, we have binary quantization, scalar quantization, product quantization, additive quantization, product additive quantization, and vector quantization.

Binary Quantization. Binary quantization (BQ), also called hashing, quantizes every value to one bit. Figure 2.5 (a) presents an example of BQ. The algorithm maps every single value in the vector to either 0 or 1, reducing the size of 32-bit floats by a factor of 32. The latter alleviates the storage requirements to maintain the vectors in-memory. Furthermore, the distance calculation, which is now the Hamming distance [67], can be sped up using the XOR operator on 64-bit values and the popcount instruction. However, this quantization technique greatly reduces the recall obtained during an approximate NNS. Therefore, it usually requires a re-ranking phase.

The RaBitQ algorithm by Gao and Long [25] reaches a sharp error bound using this representation, which means that it is not possible to reach an asymptotically better bound. However, it still needs re-ranking to achieve high recalls.

Scalar Quantization. In scalar quantization (SQ), each floating-point value is mapped to an integer called *code* defined within a *codebook* that defines the mapping. The algorithm’s simplest variant trains only one codebook shared by all dimensions. Figure 2.5 (b) shows an example of scalar quantization applied to a vector. There is exactly one codebook, out of which SQ chooses a code for each value.

One way to find the codebook is to first normalize the values using the whole dataset’s minimum and maximum and then divide them into equal-sized ranges gathered into one bucket. Figure 2.6 shows this process. The graph represents a data distribution within a float32 range of 32-bit floating-point numbers, which are then mapped to a uint8 range of unsigned 8-bit integers. For this, the values in the continuous float32 range have to be mapped into buckets that are addressed by a value between 0 and 255.

Ko et al. suggest a SQ algorithm that partially preserves the distance relationship between vectors [40]. The algorithm does so by leveraging the mean and variance of each dimension to construct a codebook. The authors also emphasize that because of the preservation of the distances, distance calculations on the integer values are possible and that they can be more efficient than calculations on floating-point numbers.

Aguerreberre et al. present another algorithm called Locally Adaptive Vector Quantization (LVQ) for SQ [3]. The algorithm leverages the mean of each dimension to compute parameters for the quantization of each embedding. In addition, Aguerreberre et al. propose a two-level quantization that uses the first part of the bits to quantize the values and the trailing bits (second part) to quantize the reconstruction error. By doing so, the algorithm uses the front bits containing the quantized values to obtain the most promising KNN candidates and the trailing bits containing the quantized reconstruction error to re-rank them. In both steps, the algorithm first converts the integers to floating-point values before doing the distance calculations. Therefore, using this quantization in queries has an overhead of having to decompress the values, and speedups of working on the smaller integer types are not possible since the algorithm transforms them back into floating-point numbers.

All these scalar quantization algorithms allow the user to choose the number of bits to which they will compress every value. Common sizes are 8, 6, and 4 bits, leading to compression ratios (without metadata) of 4, 5.33, and 8. While the quantization algorithms we will introduce next can reach higher compression ratios,

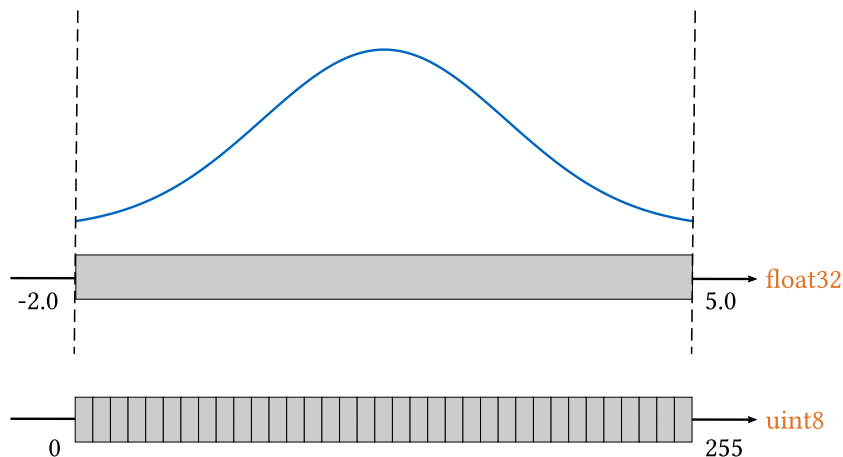


Figure 2.6: Mapping of a range of floating-point numbers to integers during a SQ encoding (Figure borrowed from Qdrant blog [76])

finding the codebooks can be cheaper in SQ. In the basic version, which normalizes the values using the minimum and maximum of the dataset, these values can be found by scanning the dataset once. Furthermore, if the search algorithm is adapted to be aware of this quantization, distance calculations can be done on the quantized domain as suggested by Ko et al. [40]. This not only saves the decompression time but also makes speedups possible by working on smaller data types.

Product Quantization. In product quantization (PQ) [39], the D -dimensional space is split into M $\frac{D}{M}$ -dimensional subspaces. For each subspace, the algorithm trains a codebook using the k-means clustering algorithm. Figure 2.7 shows how the clustering looks in a 2-dimensional subspace. The vectors, shown as 2-dimensional black points, are clustered into 4 sets. The orange points are the centroids of these sets.

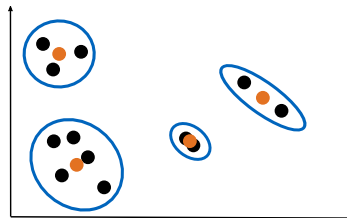


Figure 2.7: Clustering of points (shown in black) in a 2-dimensional subspace into 4 clusters with centroids shown in orange

When quantizing a vector, PQ splits it into M subvectors. For each subvector, the algorithm finds the nearest cluster center in the codebook for this subspace. The encoded vector then consists of the concatenation of the codes. PQ concatenates the centroids represented by the codes for reconstruction to get a D -dimensional vector again. In figure 2.5 (c), the vector with 6 elements is split into 3 subvectors with 2 elements each. For each of these, a codebook is trained.

Jégou et al. also introduce the concept of asymmetric distance computation (ADC). In ADC, the query vector does not need to be encoded, but the algorithm uses the distance between the raw query value and the reconstructed values of the database vectors. To speed up the distance calculations when using Euclidean distance, ADC precomputes the distances from each subvector in the query to each code in the codebook for the respective subvector. Then, to get the distance to one database vector, the algorithm looks up the

precomputed results to get the distances to the centroids in each subspace. As the Euclidean distance is the sum of the distances for each dimension, the result is the sum of the distances looked up per subspace.

The clustering to preprocess the embeddings is similar to the one used for IVF. Therefore, the algorithms can be combined without much overhead. For example, PQ with ADC is used within an IVF index in the algorithm Inverted File with Asymmetric Distance Calculation (IVFADC) [39].

Many optimizations on PQ have been proposed. Ge et al. [26] and Norouzi and Fleet [59] have researched a more optimized choice of a subspace to perform PQ. They propose rotating and permutating the dimensions using multiplication with a random orthogonal matrix. Ge et al. reported an improvement in the recall of their method called Optimized Product Quantization (OPQ) over PQ with randomly ordered dimensions on all datasets they tested. For example, when using the ADC on the GIST dataset, they improved the 1000-recall@1000 from approximately 0.35 to 0.70. Other research focuses on making the lookups SIMDizable [4, 10] or performing PQ using GPUs [38, 69].

Like in SQ, users can parameterize PQ to the compression ratio that it should reach. Therefore, the user must choose the number of subspaces and centroids per subspace. The compression ratio without metadata for one embedding is the number of subspaces times the number of bits required to represent the index of each centroid. The latter equals the number of bits required to store the count of centroids. Typical values for the number of centroids are 8, 12, or 16, while the number of subspaces should be chosen depending on the number of dimensions of the embeddings. For GIST, Jégou et al. report a compression ratio (without metadata) of 64 when having a recall over 90% [39]. However, by using OPQ, it was not possible to reach such high recalls on all datasets [25]. Moreover, storing the codebooks can noticeably affect the compression ratio when compressing small datasets. PQ can reach high compression ratios at the cost of the increased encoding time. This time is high because PQ needs to perform the k-means algorithm on every subspace.

Additive Quantization. Similar to PQ, in additive quantization, multiple codebooks are trained. However, each codebook value represents a D-dimensional vector. To reconstruct a value, AQ sums up codes from multiple codebooks. The latter helps to minimize reconstruction errors, thus achieving higher recalls. Figure 2.5 (e) shows an example of additive quantization that trains 3 codebooks. Finding an optimal quantization for each vector is NP-hard. Therefore, Babenko and Lempitsky suggest using beam search as an approximation [6]. Beam search is a graph search algorithm that continues only with the most promising nodes after each step. For finding a quantization, this means that the search finds the N candidate codes closest to the query in the first step and proceeds with them as code tuples with one element each. In every subsequent step, the beam search finds N new codes from the remaining codebooks per candidate code tuple. Of the resulting N^2 tuples, it chooses the best N for continuing. For N , the authors chose 64 in their experiments. They show that additive quantization reaches higher recalls than both PQ and OPQ on all datasets they tested. For example, when compressing SIFT to 32 bits per vector, the probability of finding the nearest neighbor within 5 candidates is approximately 65% using AQ and approximately 40% on PQ and OPQ, where the value is slightly higher in OPQ.

The authors do not report encoding times; however, they mention that doing the beam search is more expensive than finding the codes in PQ. Another problem of the algorithm is that distance computations need more table lookups than in PQ. Martinez et al. [48, 49] suggest improvements using iterated local search. This algorithm starts with a candidate solution consisting of the required number of codes. The algorithm alternately updates them using a local search to find codes that minimize the distance to the vector to be quantized and perturbs to escape local minima. Martinez et al. report that their implementation is 30x - 50x faster than the beam search implementation.

Chen et al. introduce Residual Vector Quantization (RVQ) in [13]. RVQ quantizes the quantization error repeatedly using a new codebook. The reconstructed value is again the sum of the reconstruction values of the codes. For a vector x , the algorithm greedily finds a code leading to a reconstructed value \tilde{x}_1 in the first codebook. As this reconstruction is lossy, there is an error $\epsilon_1 = x - \tilde{x}_1$. RVQ quantizes ϵ_1 to \tilde{x}_2 with error ϵ_2 . Figure 2.8 shows an example of this quantization technique using two steps. This process can

be repeated more often to minimize the reconstruction error. The authors show that RVQ reaches higher compression ratios than PQ (e.g., 0.65% versus 0.60% for finding the nearest neighbor within 10 candidates) when repeating this process 8 times on the SIFT dataset.

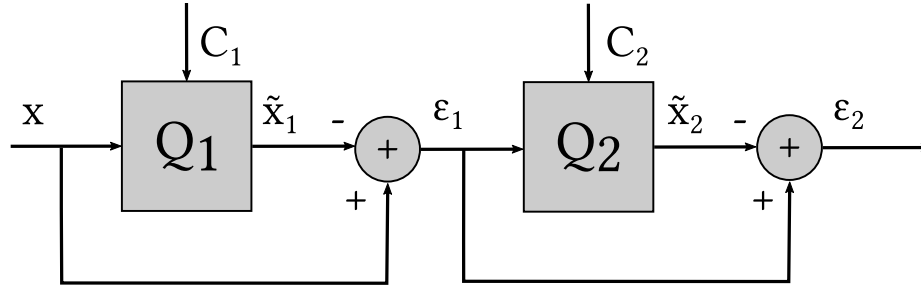


Figure 2.8: Encoding of an embedding (x) using two codebooks (C_1 and C_2) in the RVQ algorithm (Figure borrowed from Chen et al. work [13])

As in PQ, the user has to choose the codebook sizes and the number of codebooks. The mentioned AQ algorithms can all reach higher compression ratios than those of PQ when having the same recall. The problems with these approaches are the higher encoding and distance computation times, which is why vector databases use PQ more often.

Product Additive Quantization. Product additive quantization (PAQ) combines the product and additive quantization algorithms. The first PAQ algorithm was proposed by Babenko and Lempitsky in [6]. As in product quantization, PAQ splits the vectors into subspaces. However, it trains multiple codebooks that use additive quantization for each subspace instead of one. In the example in figure 2.5 (d), two codebooks were trained for each subspace, and each subvector is then reconstructed as a sum of 2 values of codes. As the complexity of the beam search for finding quantizations grows cubically with the number of codebooks, the encoding speed can be improved by having less codebooks per subspace. Furthermore, experiments of Babenko et al. show that the recall of PAQ is slightly higher than that of AQ. More recent research by Niu et al. [58] also combines the ideas of product and additive quantization. Their algorithm uses residual vector quantization for the codebooks per subspace. PAQ can be used to find a better balance between the higher accuracy of AQ over PQ and the higher encoding speed of the latter.

Vector Quantization. Vector quantization maps every vector to exactly one code representing a D -dimensional value. See figure 2.5 (f): Here, only one value that maps to a single vector in the codebook is required to represent the embedding. However, this approach is not feasible, as explained in [39]. If a 128-dimensional vector would be encoded using 64-bit codes, one would get 2^{64} centroids, which would require $128 \cdot 2^{64}$ floating-point numbers to store the codebooks. Furthermore, clustering the whole dataset requires more time than the other quantization algorithms.

The quantization techniques presented before can be represented as a hierarchy (figure 2.9). A binary quantizer is a scalar quantizer with one bit per value. A scalar quantizer is a product quantizer with one dimension per subvector. A product quantizer is an additive product quantizer with one codebook per addition. An additive product quantizer is an additive quantizer where all other dimensions are set to zero. Finally, an additive quantizer can be represented by a vector quantizer by just enumerating all values the AQ stores. Every possible AQ code represents exactly one vector. VQ would store all of them separately with only one code for each.

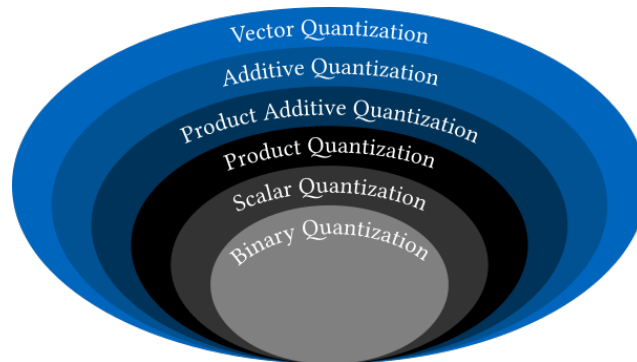


Figure 2.9: The hierarchy of quantization algorithms (Figure borrowed from Douze et al. work [18])

Downcasting

Vector embeddings usually are represented using IEEE 754 single-precision floating-point values that consist of 32 bits [34]. In downcasting, the floats are represented using 16 or fewer bits. Figure 2.10 (b) shows a half-precision float with 16 bits. The exponent and mantissa only have 5 and 10 bits, respectively. When calculating the value, 15 is subtracted from the exponent instead of 127. Because of this and the smaller number of bits used for the exponent, the range of representable exponents is -14 to 15 instead of -126 to 127 . Thus, it is possible that an exponent of a single precision float cannot be stored in the exponent bits of a 16-bit float. In those cases, special values for positive and negative infinity are used. Downcasting cuts the less significant bits of the mantissa and takes the sign bit over. Both cutting the mantissa and reducing the exponent range result in a loss of precision.

Some processors support the usage of this smaller float datatype [36]; calculations on these types can be faster than on the bigger data type. Furthermore, there are SIMD instructions supporting these datatypes, which can further speed up NNS [36]. For example, this compression approach is used in the FAISS library [53], or the USearch search engine [66].

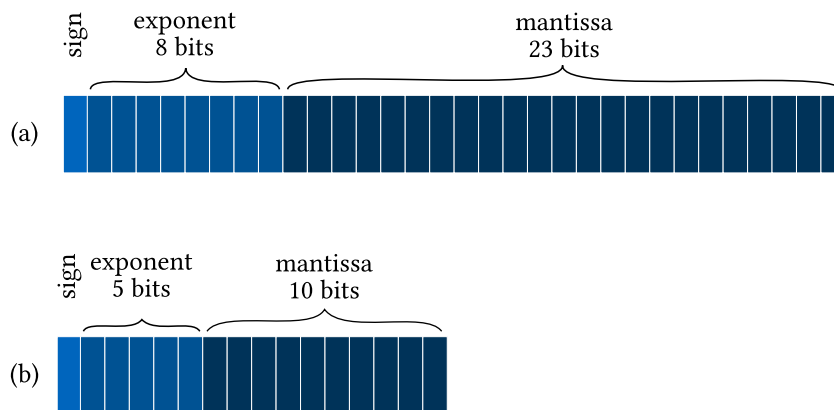


Figure 2.10: Bitwise representations of floating-point numbers: (a) IEEE single precision and (b) 16 bit

3 Analysis of Vector Embedding Datasets

3.1 Overview of the Datasets

In the previous section, we gave an overview of algorithms to compress vector embeddings in a lossy fashion. In all of these, we identified problems that explain the need for other compression algorithms that minimize the reconstruction error and avoid aggressive changes to the raw vector values while reaching high compression ratios and low encoding and decoding times. BQ needs re-ranking for high compression ratios; therefore, it has negative compression ratios (as the raw vectors are still needed). SQ can have low encoding and decoding times. However, it cannot reach compression ratios as high as, for instance, PQ. Furthermore, SQ significantly changes the vectors by replacing them with codes. The other quantization approaches, PQ, PAQ, and VQ, have a high overhead for encoding by performing a clustering algorithm like k-means several times. Downcasting is a compression technique that keeps the vectors mostly intact. However, when downcasting to the commonly used size of 16 bits, the compression ratio is only 2.00. Therefore, we analyzed datasets often used in the literature on NNS to develop a lossy compression algorithm that keeps the values mostly unchanged, has low encoding and decoding times, and reaches high compression ratios.

Table 3.1 presents the datasets we used. The first 6 sets are comprised of vectors, which are the output of machine learning models used to learn a representation of words or images. The embeddings of the following 4 datasets are the pixels of images flattened into a one-dimensional vector; they can either be from grayscale images or one channel of a color image. The following 5 datasets contain embeddings with manually chosen features from different domains. These can be, e.g., brightness for images, loudness for audio, and sensor output for motion. Finally, the last two entries are datasets with randomly generated data. For this, we created embeddings whose values follow a Gaussian distribution.

The datasets DEEP-1B, GloVe25, GloVe50, NYTimes256, Fashion-MNIST, MNIST, GIST, SIFT, Random20 and Random100 are used in the well-known ann-benchmarks repository. This repository contains a framework for comparing implementations of ANNS algorithms [5]. The other datasets are used in the literature. We chose Contriever-1M that is used in the FAISS paper [18]. Trevi and STL are used in the context of ANNS in [33]. In [25], MSong is reported to be a difficult dataset for product quantization, which makes it interesting for our approach, which is not quantization-based. Moreover, we chose HAR and Arcene to test the algorithm on datasets from two other domains (motion and medical data). [74] and [73], respectively, use them in the context of ANN. Arcene is additionally interesting because, with 10,000 dimensions, it is much higher dimensional than the others.

Most datasets' vectors are represented by 32-bit floating-point numbers or (in the case of SIFT) integers saved as floats. The image data (in Fashion-MNIST, MNIST, STL and Trevi) are saved as 8-bit integers; HAR and Arcene are integers stored as whitespace-separated text. We cast all these values to float32 because search algorithms often only support this datatype as input.

The last column of table 3.1 shows kernel density estimation plots of the dataset distributions. Each line belongs to one dimension. It shows how often each value occurs in the dataset. The plot additionally smooths the observations with a Gaussian kernel to have a continuous line. Most datasets containing learned features have a normal distribution, which is similar in all dimensions. Contriever-1M is the only dataset of learned features with dimensions depicting different distribution centers. For the image pixel and manually chosen feature datasets, we have different distributions. Fashion-MNIST, MNIST, Arcene, and MSong have one value often occurring while the rest is distributed approximately equally. STL and SIFT have multimodal

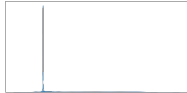

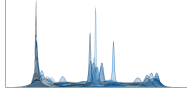

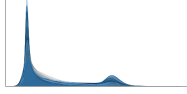
3 Analysis of Vector Embedding Datasets

distributions with two peaks each. The distribution of Trevi is normal, and the one of GIST is skewed. Moreover, on HAR and the random datasets, every dimension has its own distribution.

| Name | Dimensions | Train Embeddings | Kind of Data | Distribution |
|----------------------------|------------|------------------|---|---|
| | | Test Embeddings | Datatype | |
| Learned Features | | | | |
| Contriever-1M ([18]) | 768 | 990,000 | word embeddings |  |
| | | 10,000 | float32 | |
| DEEP-1B ¹ | 96 | 9,990,000 | deep image descriptors |  |
| | | 10,000 | float32 | |
| GloVe25 ² | 25 | 1,183,514 | word embeddings |  |
| | | 10,000 | float32 | |
| GloVe50 ² | 50 | 1,183,514 | word embeddings |  |
| | | 10,000 | float32 | |
| NYTimes256 ([5]) | 256 | 290,000 | normalized and randomly projected words |  |
| | | 10,000 | float32 | |
| Image Pixels | | | | |
| Fashion-MNIST ³ | 784 | 60,000 | grayscale images |  |
| | | 10,000 | int8 | |
| MNIST ⁴ | 784 | 60,000 | grayscale images |  |
| | | 10,000 | int8 | |
| STL ⁵ | 9,216 | 90,000 | red dimension of color images |  |
| | | 10,000 | int8 | |
| Trevi ⁶ | 4,096 | 91,120 | grayscale images |  |
| | | 10,000 | int8 | |

3 Analysis of Vector Embedding Datasets

Manually Chosen Features

| | | | | |
|---------------------|--------|-----------|-----------------------------|---|
| Arcene ⁷ | 10,000 | 800 | mass-spectrometric data |  |
| | | 100 | text | |
| GIST ⁸ | 960 | 1,000,000 | gist features of images |  |
| | | 1,000 | float32 | |
| HAR ⁹ | 561 | 9,299 | sensor data of human motion |  |
| | | 1,000 | text | |
| MSong ¹⁰ | 420 | 984,185 | features of songs |  |
| | | 10,000 | float32 | |
| SIFT ⁸ | 128 | 1,000,000 | sift features of images |  |
| | | 10,000 | float32 | |

Random Data

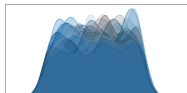
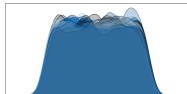
| | | | | |
|-----------|-----|--------|---------|---|
| Random20 | 20 | 9,000 | random |  |
| | | 1,000 | float32 | |
| Random100 | 100 | 90,000 | random |  |
| | | 10,000 | float32 | |

Table 3.1: Overview of the vector embedding datasets

¹<https://sites.skoltech.ru/compvision/noimi/>

²<http://nlp.stanford.edu/data/glove.twitter.27B.zip>

³<https://github.com/zalandoresearch/fashion-mnist>

⁴<https://yann.lecun.com/exdb/mnist/>; in August 2024, the files in the original source are no longer available; therefore, those from <https://www.kaggle.com/datasets/hojjatk/mnist-dataset> were used

⁵<https://cs.stanford.edu/~acoates/stl10/>

⁶<https://phototour.cs.washington.edu/patches/default.htm>

⁷<https://archive.ics.uci.edu/dataset/167/arcene>

⁸<http://corpus-texmex.irisa.fr/>

⁹<https://archive.ics.uci.edu/dataset/240/human+activity+recognition+using+smartphones>

¹⁰<https://www.cse.cuhk.edu.hk/systems/hash/gqr/datasets.html>

3.2 Compression of Embedding Datasets using ALP

The ALP algorithm described in section 2.2.1 needs the floating-point numbers to have a fixed precision to reach high compression ratios. Otherwise, it has to use the ALP for Real Floats (ALP_{rf}) option. Therefore, this algorithm is, by design, not optimal for compressing embeddings. Table 3.2 shows the compression ratios of ALP on some of the embedding datasets. On Contriever-1M, DEEP-1B, and GloVe25, ALP can only reach a compression ratio of at most 1.25. This is because of the nature of these floating-point numbers having a high precision. On Fashion-MNIST and SIFT, ALP reaches compression ratios of about 4. This is because these datasets consist of numbers that can be represented as 8-bit integers. In the original version of Fashion-MNIST, these numbers were stored as integers, and we cast them to floats so that we could use the ALP algorithm on them. On the other hand, in SIFT, these numbers were saved as floats in the published version of the dataset. Finally, on GIST, ALP can reach a higher compression than on the other float datasets because the values in the dataset have a fixed precision of 4 digits. Because of this, ALP does not fall back to ALP_{rf}, and a compression ratio of 2.032 can be achieved.

| Dataset | Datatype | Compression Ratio | Uses ALP for Real Floats |
|----------------|-----------------|--------------------------|---------------------------------|
| Contriever-1M | float32 | 1.149 | ✓ |
| DEEP-1B | float32 | 1.160 | ✓ |
| Fashion-MNIST | uint8 | 3.915 | ✗ |
| GIST | float32 | 2.032 | ✗ |
| GloVe25 | float32 | 1.234 | ✓ |
| SIFT | float32 | 4.028 | ✗ |

Table 3.2: The compression ratio of ALP on embedding datasets

However, as some NNS applications tolerate approximate answers, we changed the algorithm of ALP to be lossy by using smaller LEP exponents than the lossless algorithm would require. Therefore, we allow some reconstruction error. By converting the floating-point numbers to integers, they are rounded to the fixed number of decimals determined by the chosen LEP exponent. As such, the chosen LEP exponents control the recall, as bigger LEP exponents can store the embeddings with a higher precision.

Figure 3.1 (a) shows the approximate nearest neighbor search recall using different LEP exponents in our lossy version of ALP. All recalls reported in this chapter use a "10-recall@10" as explained in section 2.1.2. For Fashion-MNIST and SIFT, the recall is always 1 because these are integer datasets. For the other datasets, different LEP exponents are required to reach the same recall; however, it is possible to reach high recalls of more than 90% on all datasets with a LEP exponent of at most 4. Mind that a recall of 100% on datasets that are not actually integers does not necessarily mean that every query would return a correct result. This is because recall is a metric averaged over many queries (for these results, we averaged the recall over 1000 queries).

Using these small LEP exponents means that the second multiplication step of ALP, which cuts trailing zeros, can be skipped in both compression and decompression.

With figure 3.1 (b), we show how using small LEP exponents actually leads to high compression ratios. For the recalls above 90%, we reach compression ratios between 4 and 7 on all datasets. In this plot, the curves are mirrored to the ones comparing LEP exponents to recalls because smaller exponents lead to higher compression ratios that we plotted on the x-axis.

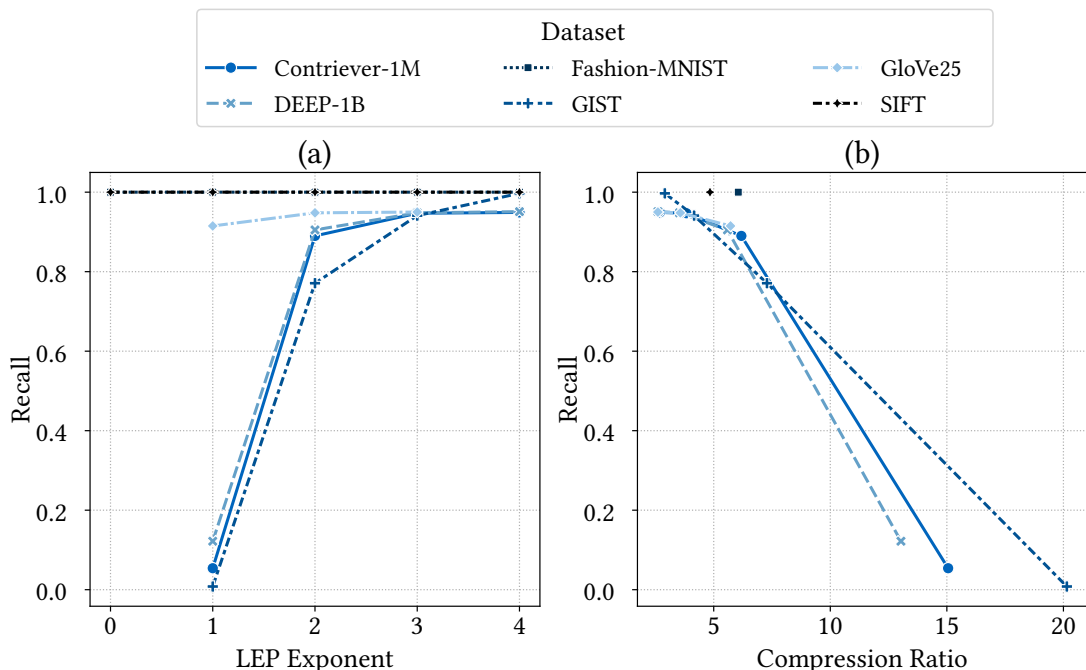


Figure 3.1: The recall of LEP for different LEP exponents and compression ratios on embedding datasets

3.3 Exceptions in the Frame Of Reference

In LWC approaches like FOR, exceptions can be used as suggested by Zukowski et al. [75]. Exceptions are values that cannot be reconstructed losslessly with the parameters chosen by the algorithm. In fact, ALP also uses exceptions after the step of converting the floating-point numbers to their integer representation. As it uses one ALP exponent and factor for every 1024 values, it might happen that some of those values cannot be losslessly reconstructed. Hence, they are stored as exceptions. However, there are no such exceptions in LEP due to its lossy nature. Despite this, we did experiments to achieve higher compression ratios by incorporating exceptions on the FOR component of our lossy version of ALP. We classified the exceptions into two types: Regular exceptions and ones that require only one bit more than the chosen bit-width. Take, for example, the positive values $(0000\ 0011)_2$, $(0000\ 0010)_2$, $(0000\ 0100)_2$ and $(0000\ 1000)_2$ and let us assume that the FOR base was already subtracted of them and that we want to use 2 as the bit-width. Then, $(0000\ 0011)_2$ and $(0000\ 0010)_2$ can be bitpacked to $(11)_2$ and $(10)_2$. $(0000\ 0100)_2$ can only be bitpacked to $(100)_2$ and therefore requires 1 bit too much while $(0000\ 1000)_2$ requires 4 bits when being bitpacked to $(1000)_2$. For values that are negative after subtracting the FOR base, by "requiring one bit more," we mean that we could represent its absolute value using the chosen bit-width.

When looking at the distributions of the datasets shown in table 3.1, one can see that there are many datasets with a normal distribution. For example, this is the case for all datasets with learned features. When we bitpack the data using PFOR, most of the exceptions are still close to the range of representable values due to the nature of their distribution and, therefore, are the type of exceptions that require only one bit more to be encoded. Table 3.3 shows for each dataset the number of exceptions of values that would need one and more than one bit more than the rest of the values after subtracting the FOR base and bitpacking them. We counted the number of exceptions per LEP vector, which contains 1024 values. Fashion-MNIST and SIFT are the only datasets containing regular exceptions (which need more than 1 bit more), and the

number of these exceptions is still low. Table 3.3 also shows the improvement in the compression ratio that we can reach if we exploit that some exceptions only need one bit more. Details on the algorithm used to compress the exceptions can be found in section 4.1. The compression ratio improvement by compressing exceptions is between 0.034 and 0.220, and the new compression ratio is between 1% and 5% higher. Even on Fashion-MNIST and SIFT, where not all exceptions could be compressed, the compression ratio was increased by 0.101 and 0.220, respectively.

| Dataset (LEP Exponent) | Number of Exceptions Requiring More Than One Bit More | Number of Exceptions Requiring Exactly One Bit More | Compression Ratio by Incorporating Uncompressed Exceptions in the FOR | Compression Ratio by Further Compressing Exceptions in the FOR |
|------------------------|---|---|---|--|
| Contriever-1M (3) | 0 | 3 | 3.670 | 3.831 |
| DEEP-1B (3) | 0 | 12 | 3.496 | 3.530 |
| Fashion-MNIST (0) | 6 | 8 | 5.953 | 6.054 |
| GIST (3) | 0 | 21 | 4.012 | 4.158 |
| GloVe25 (2) | 0 | 25 | 3.456 | 3.567 |
| SIFT (0) | 2 | 20 | 4.620 | 4.840 |

Table 3.3: The numbers of regular and compressible exceptions in the PFOR in LEP on some embedding datasets

3.4 Data Layouts for Storing the Embeddings

Historically, vector embeddings have been stored using the horizontal layout as described in section 2.1.1. Many modern data formats for analytical database systems like the storage in DuckDB [62] or the file formats Parquet [65] and FastLanes [1] store data in a columnar fashion. The latter not only increases analytical processing speed [2] but also allows for opportunities for higher compression by applying LWC techniques to data in the same column that tend to have similar values [1, 41].

An ongoing project at the Centrum Wiskunde & Informatica in Amsterdam is exploring opportunities to use the vertical layout for nearest neighbor search. This research has been inspired by the work of De Vries et al. [14] that uses the vertical layout in the BOND algorithm. Therefore, we believe that using this data ordering could improve compression ratios while at the same time benefitting the search itself.

Figure 3.2 shows heatmaps of the first 50 dimensions and the first 100,000 embeddings on some datasets. Each row of squares per plot corresponds to the values in one embedding; each square is one individual value, and the colors indicate which value the dataset contains here. A darker color indicates a higher absolute numeric value on the embedding. If one row in a plot has a similar color hue in almost all of the cells, the values are easy to compress because they are close to each other in the numerical range. Consequently, a small bit-width can be chosen for bitpacking. Similarly, columns with a similar color hue would lead to a high compression ratio when using the vertical layout. Then, the values in that column would lay next to each other in storage and would be compressed in the same LEP vector.

The plots show the desired similar color hues per column in some of the columns in Fashion-MNIST, GloVe25, and SIFT. On these datasets, using the vertical layout could lead to a higher compression ratio than the horizontal layout. On the other hand, on GIST, there are also rows that are easier to compress in the horizontal layout. Moreover, less apparent color hue patterns are observable on Contriever-1M and DEEP-1B.

Another possibility for changing the way in which the embeddings are stored is the order in which they are saved. The latter is possible because the result of the exact search does not change if a search algorithm processes the embeddings in a different order. One way of reordering the embeddings is to cluster them using an algorithm like k-means. This preprocessing is similar to the one required when using an IVF index. Therefore, a compression algorithm can benefit from this layout if an IVF index is already created. Clustering

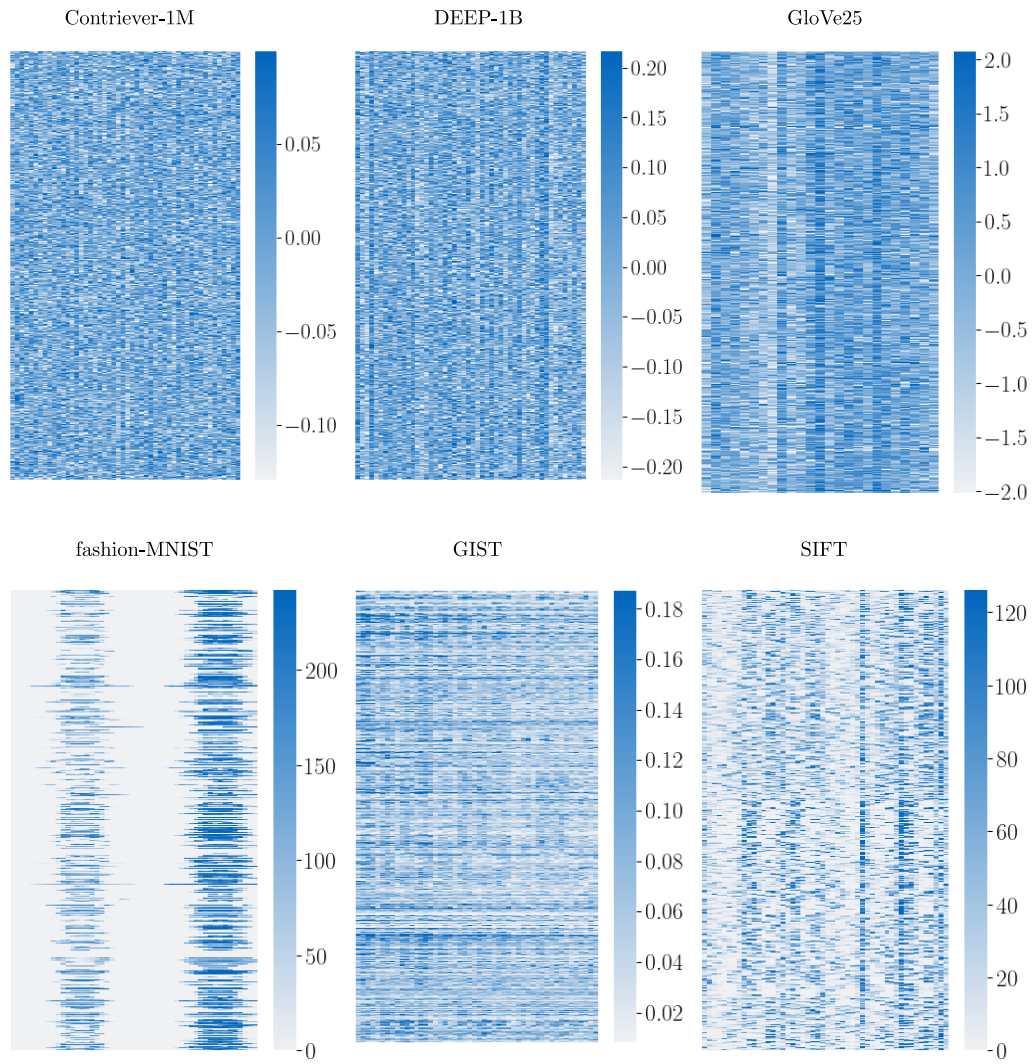


Figure 3.2: Heatmaps of the floating-point number distribution within part of the datasets; every square represents one float in the datasets, and its color hue indicates its value (darker colors belong to higher values)

algorithms build buckets based on a similarity metric over all dimensions of the embeddings. Values close to each other might also have similar values per dimension. Therefore, this might open opportunities to achieve higher compression ratios if embeddings are also stored using the vertical layout.

3.5 Frequently Occurring Values

Within the datasets we analyzed to design LEP, there are some datasets where one value occurs very often (see the data distributions column of table 3.1). This is the case, for example, for the value 0 in Arcene or SIFT. Figure 3.3 shows all datasets in which we found LEP vectors containing one often repeated value. For each of these datasets, the figure presents the share of LEP vectors with this characteristic. Especially on Arcene and MNIST, this share is close to 1, which means that higher compression ratios can be reached using an encoding that leverages repeated values.

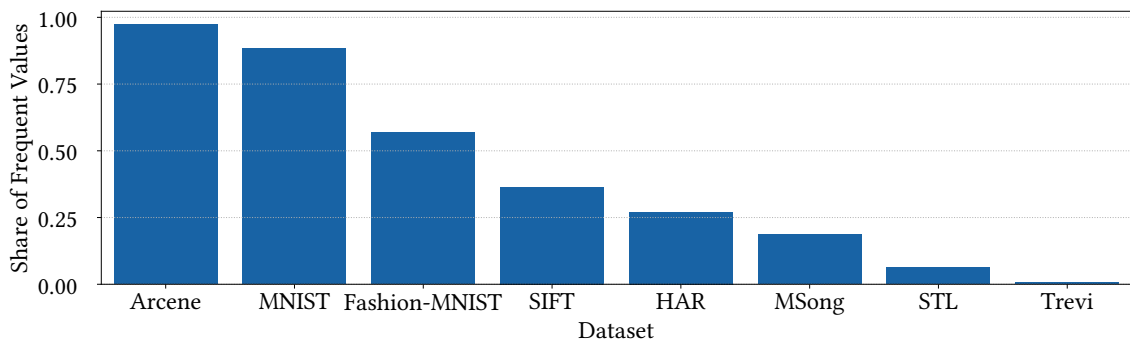


Figure 3.3: The share of LEP vectors with repeated values (occurring > 300 times in the LEP vector) for some embedding datasets

3.6 Correlations between Dimensions

Recently, there has been interest in increasing compression ratios in analytical database systems by leveraging correlations between dimensions [28, 46]. For vector embeddings, two dimensions, which are stored as two columns in the horizontal layout, can be correlated if they represent similar features. While this should not be the case in vector embeddings, as that would imply that features are redundant, it could still be a possibility that we could exploit.

One metric for measuring linear correlations between two columns is the Pearson coefficient. The Pearson coefficient has values between -1 and 1 . A higher absolute value indicates a higher correlation. Let's assume we have two columns $X = [x_1, x_2, \dots, x_n]$ and $Y = [y_1, y_2, \dots, y_n]$ with n elements each. Let us define the following values:

$$\text{mean of } X : \bar{X} = \frac{1}{n} \sum_{i=1}^n x_i$$

$$\text{mean of } Y : \bar{Y} = \frac{1}{n} \sum_{i=1}^n y_i$$

$$\text{Pearson numerator} : P_n = \sum_{i=1}^n (x_i - \bar{X})(y_i - \bar{Y})$$

$$\text{Pearson denominator of } X : P_{dx} = \sum_{i=1}^n (x_i - \bar{X})^2$$

$$\text{Pearson denominator of } Y : P_{dy} = \sum_{i=1}^n (y_i - \bar{Y})^2$$

We then define the Pearson coefficient as

$$P_c = \frac{P_n}{\sqrt{P_{dx} \cdot P_{dy}}}$$

Table 3.4 shows how many pairs of columns with a high Pearson coefficient exist in some datasets. We tested this on all datasets with less than 1000 dimensions, as the number of pairs to test grows quadratically with the number of dimensions. Apart from Fashion-MNIST and MNIST, all the datasets showing correlated columns contain manually chosen features. On the other hand, datasets with learned features or random data did not exhibit correlations between dimensions.

| Dataset (LEP Exponent) | Number of Column Pairs With a Pearson Coefficient Higher Than... | | |
|---------------------------|---|--------|-------|
| | 0.7 | 0.8 | 0.9 |
| Fashion-MNIST (0) | 5,959 | 2,005 | 246 |
| MNIST (0) | 902 | 289 | 70 |
| GIST (3) | 1,440 | 238 | 0 |
| HAR (3) | 37,431 | 23,044 | 8,226 |
| MSong (2) | 791 | 340 | 233 |
| SIFT (2) | 47 | 10 | 0 |

Table 3.4: The number of pairs of columns in embedding datasets that are correlated with a high Pearson coefficient

If the Pearson coefficient between two columns is -1 or 1 , one column can be reconstructed from another one by applying a linear function to the second one.

$$b_1 = P_n / P_{dx}$$

$$b_0 = \bar{Y} - b_1 \cdot \bar{X}$$

$$\forall i \in \{1, \dots, n\} : y_i = b_0 + b_1 \cdot x_i$$

The values in column Y can, therefore, be obtained by just knowing the values of column X , b_0 , and b_1 .

We tested leveraging columns with a Pearson coefficient higher than 0.8 on GIST and Fashion-MNIST. We first tried reconstructing target columns using only the formula above and called this an aggressive correlated-columns compression. The results of using this compression technique are compared with the LEP algorithm that does not leverage correlated columns in table 3.5. We applied the LEP algorithm to the data in the vertical layout clustered with an IVF index and compressed the exceptions. The aggressive correlated-columns compression reaches higher compression ratios on both datasets. However, it also influences the recall negatively and even reaches a recall of only 0.531 on Fashion-MNIST. Therefore, we switched to a less aggressive compression approach, which stores the errors between the result of the linear function and the original values. Contrary to the aggressive correlated-columns compression, this approach is lossless.

| Dataset (LEP Exponent) | Aggressive Correlated-Columns Compression | | PFOR | |
|------------------------|---|--------|-------------------|--------|
| | Compression Ratio | Recall | Compression Ratio | Recall |
| GIST (3) | 4.280 | 0.870 | 4.153 | 0.941 |
| Fashion-MNIST (0) | 7.062 | 0.531 | 6.054 | 1.000 |

Table 3.5: The compression ratio of LEP using correlated dimensions aggressively on GIST and Fashion-MNIST

For this lossless correlation approach, we first checked for each pair of dimensions if the Pearson coefficient between them is high enough and if the column chosen as a source is not already being compressed using another column. Then, we calculated b_0 and b_1 from the equations above. From those, we calculated the value that would be reconstructed by using the linear function. As this is not the same as the value we want to compress, we calculate the error between these reconstructed and original values. We finally saved these errors compressed with PFOR. For the other columns, we used the normal LEP compression. To keep this experiment simple, we did not try to further compress exceptions on the reconstruction errors as described in section 3.3.

The results of this approach are shown in table 3.6. As one can see, there is always an improvement in the compression ratios on GIST. The improvement is bigger if 0.7 is chosen for the Pearson coefficient threshold because we found more correlated columns. Nevertheless, on Fashion-MNIST, the compression ratio is lower for any Pearson coefficient threshold, and it gets higher for higher Pearson coefficient thresholds. This is because we always compressed one column using another one when the Pearson coefficient was high enough, independent of whether the compression ratio on the reconstruction errors was actually higher than on the raw values per LEP vector. Therefore, improvements on the compression ratio are possible if both compression ratios are calculated per LEP vector and the best is chosen.

| Min. Pearson Coefficient | Compression Ratio Without Correlations | Compression Ratio With Correlations |
|---------------------------------------|--|-------------------------------------|
| GIST (LEP Exponent 3) | | |
| 0.7 | 3.840 | 3.905 |
| 0.8 | 3.840 | 3.872 |
| Fashion-MNIST (LEP Exponent 0) | | |
| 0.7 | 4.450 | 4.210 |
| 0.8 | 4.450 | 4.244 |
| 0.9 | 4.450 | 4.378 |

Table 3.6: The compression ratio of LEP using correlated dimensions losslessly on GIST and Fashion-MNIST

Despite the slight improvement in compression ratios in GIST, the complexity of finding correlations on embedding datasets is high, as every pair of columns must be evaluated. Moreover, the decompression becomes more expensive, and two columns need to be loaded into memory to decompress the correlated columns. Ultimately, we believe that despite vector embedding features being correlated in some datasets, algorithms that leverage linear correlations are not effective on the majority of embedding datasets. Nonetheless, we only focused on linear correlations. Hence, it might be possible to achieve higher improvements using other types of functions for reconstruction, like polynomials.

4 Lossily Encoded floating-Points

Based on our experiments and observations presented before, we present LEP: Lossily Encoded floating-Points, a new compression algorithm for vector embeddings.

Figure 4.1 shows the process of compressing a dataset using the algorithm on a high level. First, the algorithm reorders the data in such a way that similar vectors are placed close-by in storage. For this, it applies clustering using the Inverted File Index implemented in the FAISS library [18] with 100 clusters, independent of the dataset size. The clustering is represented in the figure by the 2-dimensional coordinate system, where the orange points are the cluster centers. Once the algorithm reordered the embeddings so that all embeddings from one IVF cluster are laying after each other, LEP transposes them to the vertical layout as described in section 3.4. In the figure, the values are shown as written to consecutive memory, where each box represents one cluster, and each row in it is one dimension. As in figure 2.1, we use different colors to indicate that the values within one row come from different embeddings.

Next, LEP processes each 1024 consecutive values $n_0, n_1, \dots, n_{1023}$ together in one LEP vector. It multiplies each value with a power of 10. We take the LEP exponent for multiplication as an input parameter because it influences both compression ratio and recall. Therefore, the user needs to decide about its value depending on the desired result. The algorithm then converts the values to integers $d_0, d_1, \dots, d_{1023}$. Like the ALP algorithm, we use the same fast rounding trick for this¹.

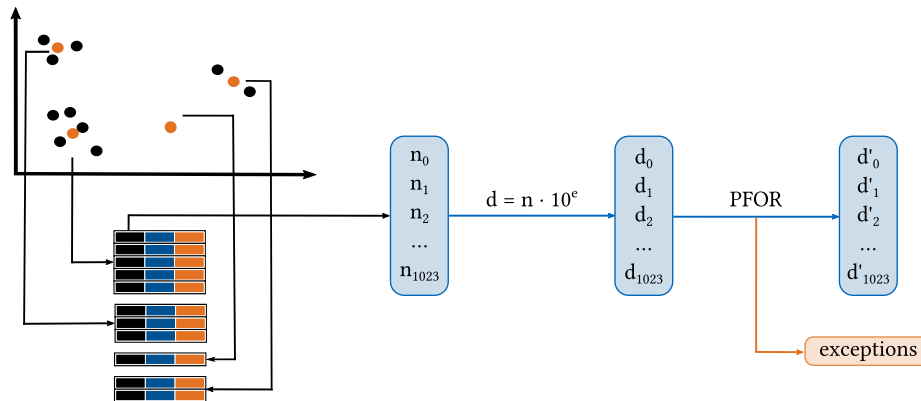


Figure 4.1: The process of encoding floats using LEP

LEP compresses these integers using the Patched Frame Of Reference. We use the algorithm described in [75] on every LEP vector to find the bit-width and FOR base. Algorithm 1 shows how the parameters bit-width and FOR base are calculated for the PFOR. Lines 7 to 16 are taken from the PFOR algorithm. This algorithm works on a sorted LEP vector (line 5). The PFOR algorithm uses two pointers, lo and hi , to define a candidate range of values that are no exceptions. min and len save a range of values that was chosen before as an intermediate result. If the range of values between lo and hi requires a bigger bit-width than the one to

¹Due to the IEEE float representation, they cannot store exact integers of more than 22 bits. In the range between 2^{22} and 2^{23} , floats do not have a decimal part. Thus, by adding and subtracting $2^{22} + 2^{23}$ to a float, it is automatically rounded to the float representing the next integer. The number can then be cast to a 32-bit integer.

test, then the values between them, including lo and excluding hi , are the maximal number of elements that fit into the chosen bit-width and start from lo . If the number of these candidates is bigger than the last len , taking them is a better choice (as it leads to a higher compression ratio), and the algorithm updates len and min . With this, min gets the value that guarantees a maximal number of elements that are no exceptions for a given bit-width.

We use the part from the PFOR algorithm in the context of embedding compression in the following way. The bit-width to start testing with is 1 smaller than the one required for having no exceptions (lines 2 - 4). It is unnecessary to try bigger bit-widths, as all of them will have no exceptions and, therefore, result in higher compression ratios. Then, the FOR base is found for each bit-width, and the compression size is calculated using the FOR base and bit-width (line 17). After that, the parameters $bestCompressionSize$, $bestBitWidth$, and $bestForBase$ are updated, if required (lines 17 to 21).

Algorithm 1: Finding the bit-width and FOR base to use for the PFOR

```

1 Function FIND_PARAMETERS ( vector ):
2    $bestBitWidth, bestForBase \leftarrow$  FIND_PARAMETERS_WO_EXC ( vector );
3    $bestCompressionSize \leftarrow bestBitWidth \cdot 1024$ ;
4    $bitWidthToTest \leftarrow bestBitWidth - 1$ ;
5   SORT ( vector );
6   while  $bitWidthToTest > 0$  do
7      $len \leftarrow 0$ ;
8      $min \leftarrow MAX\_INT$ ;
9      $range \leftarrow (1 \ll bitWidthToTest) - 1$ ;
10    for  $lo \leftarrow 0, hi \leftarrow 0; hi < 1024; hi++$  do
11      if  $ABS ( vector[hi] - vector[lo] ) > range$  then
12        if  $hi - lo > len$  then
13           $min \leftarrow vector[lo]$ ;
14           $len \leftarrow hi - lo$ ;
15          while  $ABS ( vector[hi] - vector[lo] ) > range$  do
16             $lo++$ ;
17       $compressionSize \leftarrow GET\_COMPRESSION\_SIZE ( vector, bitWidthToTest, min )$ ;
18      if  $compressionSize < bestCompressionSize$  then
19         $bestCompressionSize \leftarrow compressionSize$ ;
20         $bestBitWidth \leftarrow bitWidthToTest$ ;
21         $bestForBase \leftarrow min$ ;
22       $bitWidthToTest--$ ;
23  return  $bestBitWidth, bestForBase$ ;

```

As the LEP algorithm compresses the PFOR exceptions after they have been found, algorithm 1 will only lead to the optimal compression ratio in some cases. If different FOR bases that lead to the same number of exceptions can be chosen, they may have a different number of compressible exceptions and, therefore, different compression ratios. It can also happen that choosing more exceptions leads to a higher compression ratio if those additional exceptions are compressible and replace less regular exceptions that cannot be compressed. However, we chose not to adapt the algorithm for finding the bit-width and FOR base to take compressible exceptions into account as this encoding is already computationally expensive, and we do not expect high improvements in the compression ratio by this change.

The differences between our lossy to the lossless version of ALP are: (1) preprocessing the data by clustering the embeddings and reordering them to the vertical layout; (2) doing only one multiplication for getting the integers representing the values; (3) having no exceptions when converting the floats to integers; (4) adding two types of exceptions in the Frame Of Reference.

4.1 Compression of Exceptions

For each exception, we test if it needs only one bit more than the other values after subtracting the FOR base. See figure 4.2 for an example. The upper part is the distribution of the values, and below is an array of 8 integers in binary representation to be encoded. Let us assume that the bit-width to use is 1 and that the FOR base has already been subtracted. Then, we say that $(1111)_b$ and $(0010)_b$ only need one bit more than the bit-width to use. $(0010)_b$ can be represented with 2 bits as $(10)_b$ and $(1111)_b$ is the two complement representation of -1 ; 1 could be represented with 1 bit in binary, but we need more bits to signal that it is negative.

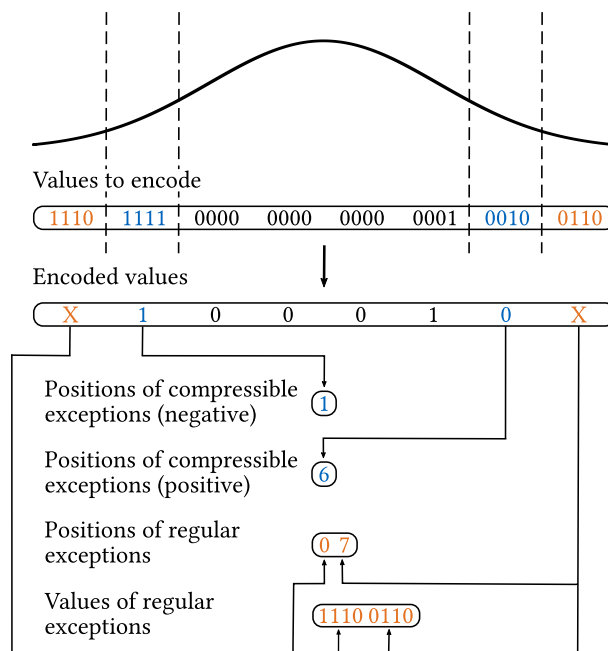


Figure 4.2: An example of compressing a range of numbers containing compressible exceptions in the PFOR in LEP

For $(0010)_b$, we know that we need precisely 2 bits to represent it. Therefore, the most significant bit needs to be 1. So we can save the rest of the bits $(10)_b$, which is $(0)_b$, in the array of compressed values and mark the position as a positive compressed exception.

For $(1111)_b$, we can multiply it with -1 and get $(0001)_b$, which also fits into the array of compressed values. Therefore, we save $(1)_b$ and mark the position as a negative compressed exception.

We mark positions by storing the indices in two separate lists using 16 bits for the index as the index is between 0 and 1023.

The regular exceptions that cannot be compressed in this way, which are $(1110)_b$ and $(0110)_b$ in the example, are saved by storing their positions with 16 bits each in a list and their values as regular exceptions

in another one with 32 bits each.

Therefore, the compression size for line 17 in the algorithm is calculated with the following formula:

$$\begin{aligned} \text{compression_size} &= \text{bit_width_to_test} \\ &+ \text{number_regular_exceptions} \cdot (16 + 32) \\ &+ \text{number_compressible_exceptions} \cdot 16 \end{aligned}$$

4.2 Bitmap-Compression of Frequent Values

As described in section 3.5, there are some datasets where a single value occurs often. If a value is frequent in a LEP vector, we propose not storing any of the numbers with this value at all. A bitmap of 1024 values, which correspond to the indices in the LEP vector, marks which positions within the LEP vector contain the frequently occurring value.

See the values in figure 4.3 as an example for this compression scheme. Here, 3 is the frequently occurring value. A 1 in the bitmap means that the value at that index is 3. For the other values, an array only containing those in the order they were in the original array is used. These values are then further compressed using the PFOR and compressed exceptions. However, here, the number of values that share a bit-width and FOR base is less than 1024.

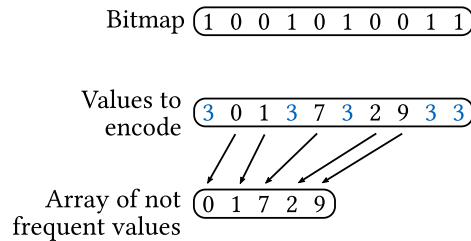


Figure 4.3: Compressing a LEP vector using the bitmap compression of frequent values

Because the bitmap requires 1024 bits of additional space, the number of frequent values needs to be high enough for using a bitmap to be beneficial. The number of bits that can be saved with this depends both on how often the frequent value occurred and the bit-width that each value is compressed to. For recalls above 90%, LEP reach a compression ratio of 5.124 in average, which corresponds to 6.245 bits per value. We want to have a threshold independent of the dataset and reach a noticeable effect on the compression ratio. Therefore we chose 300 as a threshold, which means that at least 3.413 bits per frequent value need to be saved (the 1024 additional bits are amortized by 300 frequent values). We want a noticeable effect because this scheme increases the encoding time as the frequent value needs to be found. Furthermore, LEP vectors have a size of 1024 values to enable auto-vectorization of the code for the FOR. Despite not measuring the decoding time of the algorithm, working with vectors of arbitrary size might increase decoding time in case the FOR algorithm can not be auto-vectorized efficiently.

4.3 Non-Decimal LEP for Fine-Tuning Compression Ratios and Recalls

LEP can only reach a limited number of different compression ratios and recalls. This is because the only parameter influencing these metrics is the LEP exponent used to obtain the integer representations. In our experiments, we were also only using LEP exponents smaller than 5 as they were already reaching

recalls close to 1. However, this leads to large gaps between the reachable recalls, as figure 3.1 (a) shows. For example, for Contriever-1M, the recall improves from less than 0.1 to about 0.9 when choosing a LEP exponent of 2 instead of 1. Likewise, there are also big gaps between the compression ratios, for example, in the case of Contriever-1M. Here, the compression ratio drops from 15 to 7 when using LEP exponents 1 and 2 (see figure 3.1 (b)).

Lossless ALP exploits the fact that floating-point values often have a fixed precision in real-world datasets. While this works well on the datasets the algorithm was designed for, this is often not the case for embeddings (as we showed in section 3.2). In fact, within our embedding datasets, only the integer datasets, SIFT, and GIST had this characteristic. Therefore, our lossy version of ALP does not require multiplying the floating-point numbers by a power of 10. We can use any floating-point number f . The algorithm only changes at the step $d = n \cdot 10^e$ in figure 4.1 to $d = n \cdot f$. We can use the integers that are the output of this for the Patched Frame Of Reference as before. By doing so, the compression ratio and recall can be tuned to any value, and the large gaps in the plots are reduced.

4.4 Speedup of the Encoding Time

In the PFOR work [75], the authors suggest sampling some values over the whole dataset and calculating their optimal parameters. However, using the same parameters for all dimensions might negatively affect the compression ratio as the dimensions can have different distributions (as can be seen, for example, in the distribution plot of Contriever-1M in table 3.1).

On the other hand, the lossless version of ALP samples some values every 100 vectors to determine a set of candidate parameters (ALP exponent, ALP factor) that would achieve high compression performance on the taken samples. At compression time, the algorithm performs a second sampling by taking samples for each LEP vector that will be compressed, and it probes every candidate parameter on the samples. Finally, ALP chooses the most promising parameters for compression. For the PFOR, parameters (bit-width, FOR base) can be tested by scanning each LEP vector once and checking which of these values would be exceptions.

We propose to also use a sampling in LEP to speed up the encoding time. Algorithm 2 presents how we can use sampling to find parameters for the PFOR algorithm. Here, we first sample 5 vectors of 1024 values equally distributed every 100 vectors (lines 2 - 8). We then find the bit-width and FOR base for each of these 5 vectors using the procedure presented in algorithm 1 (here referred to as FIND_PARAMETERS in line 9). If we obtain the same bit-width several times, we take the mean of the FOR bases as the candidate FOR base for this bit-width (lines 11 - 12).

At compression time, we evaluate the compression ratio obtained using each candidate parameter pair on the vector to compress. Here, one can iterate over the vector to check which values become which kind of exception.

Algorithm 2: Find parameters using sampling

```
1 Function FIND_COMBINATIONS (values, valuesCount):  
2   stepSize  $\leftarrow$  valuesCount / 5;  
3   for i  $\leftarrow$  0 to 5 - 1 do  
4     offsets[i]  $\leftarrow$  i · stepSize;  
5   stepSize  $\leftarrow$  stepSize / 1024;  
6   for offset in offsets do  
7     for j  $\leftarrow$  0 to 1024 - 1 do  
8       inputVector[j]  $\leftarrow$  values[offset + stepSize · j];  
9       bestBitWidth, bestForBase  $\leftarrow$  FIND_PARAMETERS (inputVector);  
10      combinations[bestBitWidth].PUSH_BACK (bestForBase);  
11   for bitWidth in combinations.KEYS () do  
12     combinations[bitWidth]  $\leftarrow$  MEAN (combinations[bitWidth]);  
13   return combinations;
```

5 Evaluation

After presenting the LEP algorithm in chapter 4, we now focus on measuring its performance using two metrics: Its compression ratio and the mean squared error of the values after reconstruction. We measure these at different recall levels using the data layouts we presented in the previous chapter. Furthermore, we compare LEP with other compression algorithms (SQ, LVQ, PQ, and downcasting). Finally, we show the impact of sampling values to increase the encoding speed on the compression ratio. All recalls reported in this chapter use the metric "10-recall@10" introduced in section 2.1.2 and are the average over 1000 queries.

5.1 Assessment of Data Layouts

In this section, we present the results of using the LEP algorithm to compress embeddings stored using the different data layouts described in section 3.4.

Figure 5.1 shows the compression ratio for different LEP exponents when using the vertical and horizontal layout. Here, the embeddings were reordered in clusters given by the IVF index. As one can see, the compression ratio on the vertical layout (solid line) is higher on almost all datasets than on the horizontal layout (dashed line). This effect is more prominent for LEP exponents that reach higher compression ratios. For example, on DEEP-1B for LEP exponent 1, the compression ratio improves from 10.257 on the horizontal layout to 13.031 on the vertical layout. We used bar plots to showcase the results on the integer datasets to make the difference between the compression ratios more visible. Thus, both bars belong to the same LEP exponent (in this case, 0, as the data are integers). Only on STL and Trevi the compression ratio on the horizontal layout is slightly higher (0.026 and 0.250, respectively).

Based on these results, using the vertical layout is better if one has no prior knowledge of the dataset. It is important to note that this vertical layout not only improves compression ratios but also works well on search algorithms that prune vectors by bounds as multiple vectors can be processed at a time. This is one of the aspects in which our compression algorithm contributes to rethinking vector embeddings search in analytical database systems.

We also explored the effect of reordering the embeddings in terms of their clusters assigned by an IVF index. The plots in figure 5.2 show the compression ratio with and without reordering the embeddings regarding their IVF clustering. Here, we used the IVF clustering from the FAISS library [53] with 100 clusters as a fixed parameter. Similar to using the vertical layout, reordering the embeddings using the IVF clusters improves the compression ratio on almost all datasets. Here, the improvement is also higher for smaller LEP exponents. Only on Arcene the compression ratio is negatively affected. The latter is due to Arcene having less than 1024 embeddings. In this case, a LEP vector (which always contains 1024 values) needs to include values from the following dimension. Moreover, as we were clustering the 800 embeddings of Arcene to 100 clusters, every cluster only contains a few values.

Furthermore, the effect on the compression ratio could be increasable by using different parameters in the IVF algorithm, depending on, for example, the size of the dataset. Maintaining metadata about the clusters in the stored data can also be exploited by search algorithms. However, one has to keep in mind that this preprocessing prolongs the encoding time.

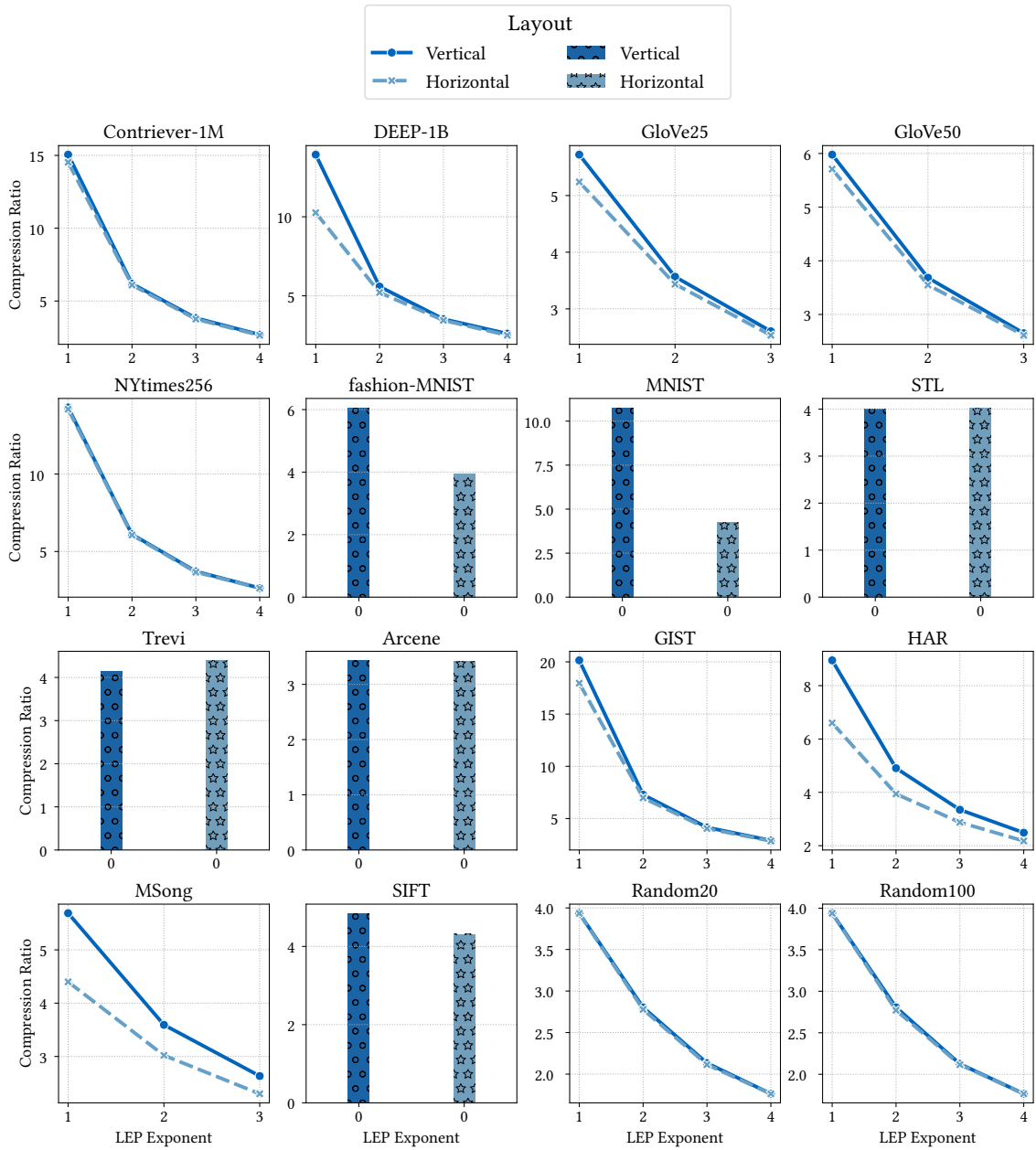


Figure 5.1: The compression ratio of LEP on data preprocessed into the vertical and horizontal layout

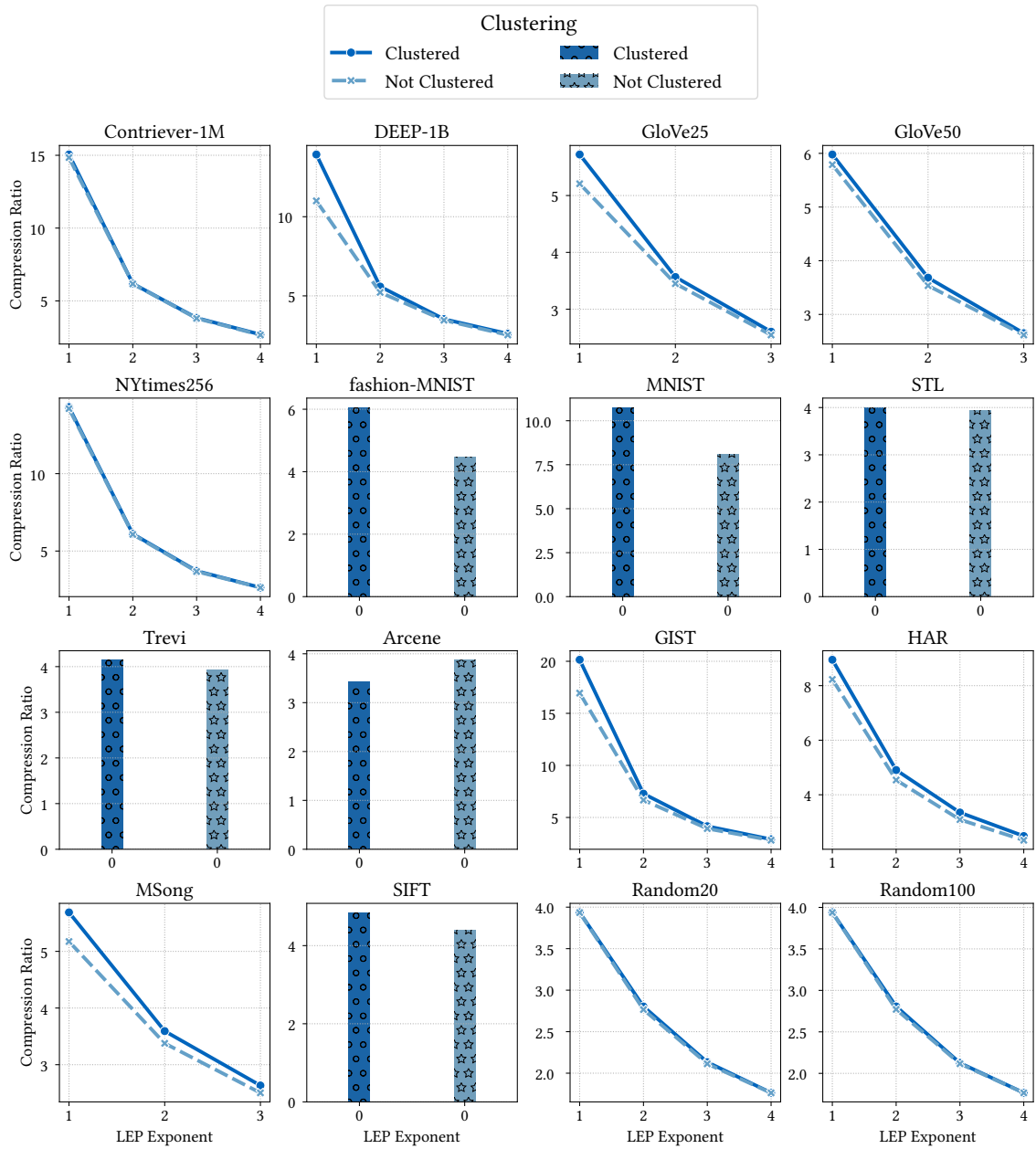


Figure 5.2: The compression ratio of LEP on data preprocessed using clustering or no clustering

5.2 Comparison against Other Algorithms

5.2.1 Considered Algorithms

We compared LEP to 5 other algorithms: ALP, downcasting, standard SQ, LVQ, and PQ. For LEP, we used data preprocessed in the vertical layout and reordered in terms of the IVF clustering with 100 clusters. To see the improvement in the compression ratio when using a lossy lightweight compression algorithm, we compared it to the lossless version of ALP. Based on the dataset, the algorithm chose to use ALP or ALP for Real Floats.

For downcasting, we chose the downcasting to 16 bits implemented in the FAISS library [53]. It behaves as the downcasting described in section 2.2.2. As our algorithm compresses single values, it can best be compared to scalar quantization. As implementations for this, we chose the implementation of scalar quantization in the FAISS library and LVQ [3]. The FAISS implementation divides the data range into uniform buckets using the minimum and maximum of the dataset as parameters. As sizes of the compressed values, we used all values offered by the implementation, which are 8, 6, and 4 bits. LVQ works as described in section 2.2.2. This algorithm lets one choose two parameters, the sum of which is the number of bits needed to represent a single value. The authors note this as axb , where a is the number of bits in the first phase of the algorithm and b is the one in the second. We chose to compare to 8 bits as $8x0$ and $4x4$, 6 bits as $4x2$ and 4 bits as $4x0$.

Because PQ is widely used, we also compared LEP to the implementation of PQ in the FAISS library. We tried to use parameters that lead to comparable compression ratios as LEP. Therefore, per dataset, we trained two quantizers where each subvector consists of one dimension, one of them where each value is represented with 8 bits and the other one with 4 bits. To take advantage of the fact that PQ is intended to be used on larger subvector sizes, we also trained one quantizer with a subvector size of 2 and 8 bits per value. For this to work, the number of dimensions needs to be dividable by 2. This was not the case for GloVe25 with 25 dimensions and HAR with 561 dimensions. For GloVe25, the only possibility of splitting the dimensions in less than 25 and more than 1 subspaces is having 5 elements per subvector. For HAR, we divided the dimensions by 3.

5.2.2 Compression Ratio and Mean Squared Error

Figure 5.3 shows the compression ratio - recall curve of LEP compared to the compression algorithms mentioned above. Values to the upper right of the plot have a higher compression ratio and recall and are, therefore, better. The plots only show one data point for the lossless ALP algorithm and the downcasting to float16 because there are no parameters for tuning the compression ratio. This is also the case for the integer datasets for LEP.

Overall, there is no dataset in which LEP has a recall of less than 90% on all compression ratios comparable with the compression ratios of the other algorithms. This is not the case for all of the other compression algorithms. For instance, on NYTimes256, all of the other algorithms had compression ratios below 75%, and on MSong, only product quantization was comparable to LEP.

On the integer datasets, LEP can always reach a recall of 100%. It also reaches the same or even higher compression ratios compared to the other algorithms on Fashion-MNIST, MNIST, and SIFT. In STL and Arcene, LEP is beaten by the other algorithms. However, in these datasets, the bitmap compression technique described in section 4.2 increases the compression ratio on all of them. On the other datasets (mainly containing learned features), LEP achieves compression ratios comparable to those of the competitors. For some datasets (e.g., Contriever-1M), the plots also show compression ratios with low recalls of less than 0.2. This is because we have a high difference in both the compression ratio and the recall between this value and the last one, and these plots are limited to integer exponents. With non-decimal LEP, better compromises between the compression ratio and the recall can be reached.

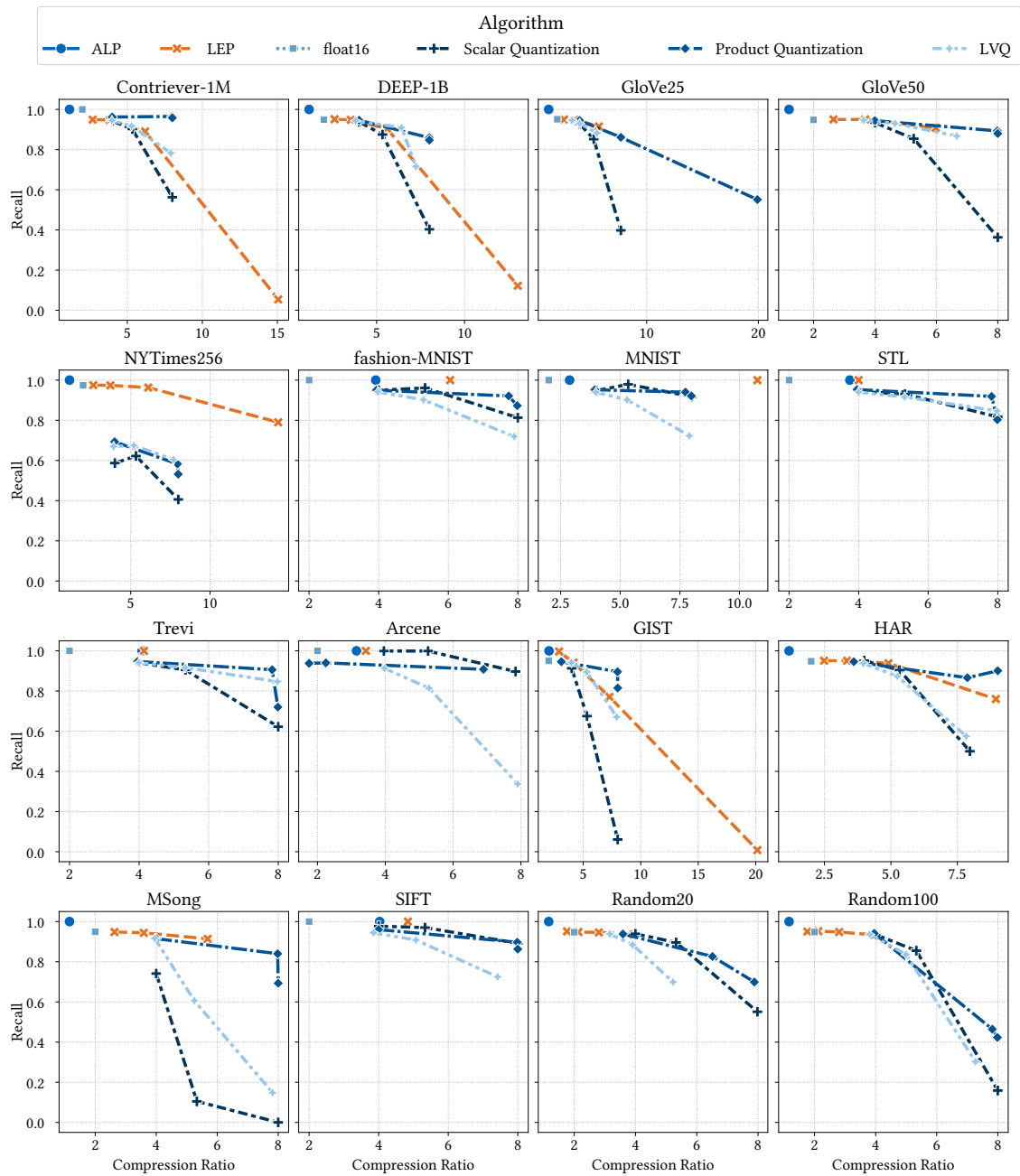


Figure 5.3: The compression ratio of LEP compared to other compression algorithms

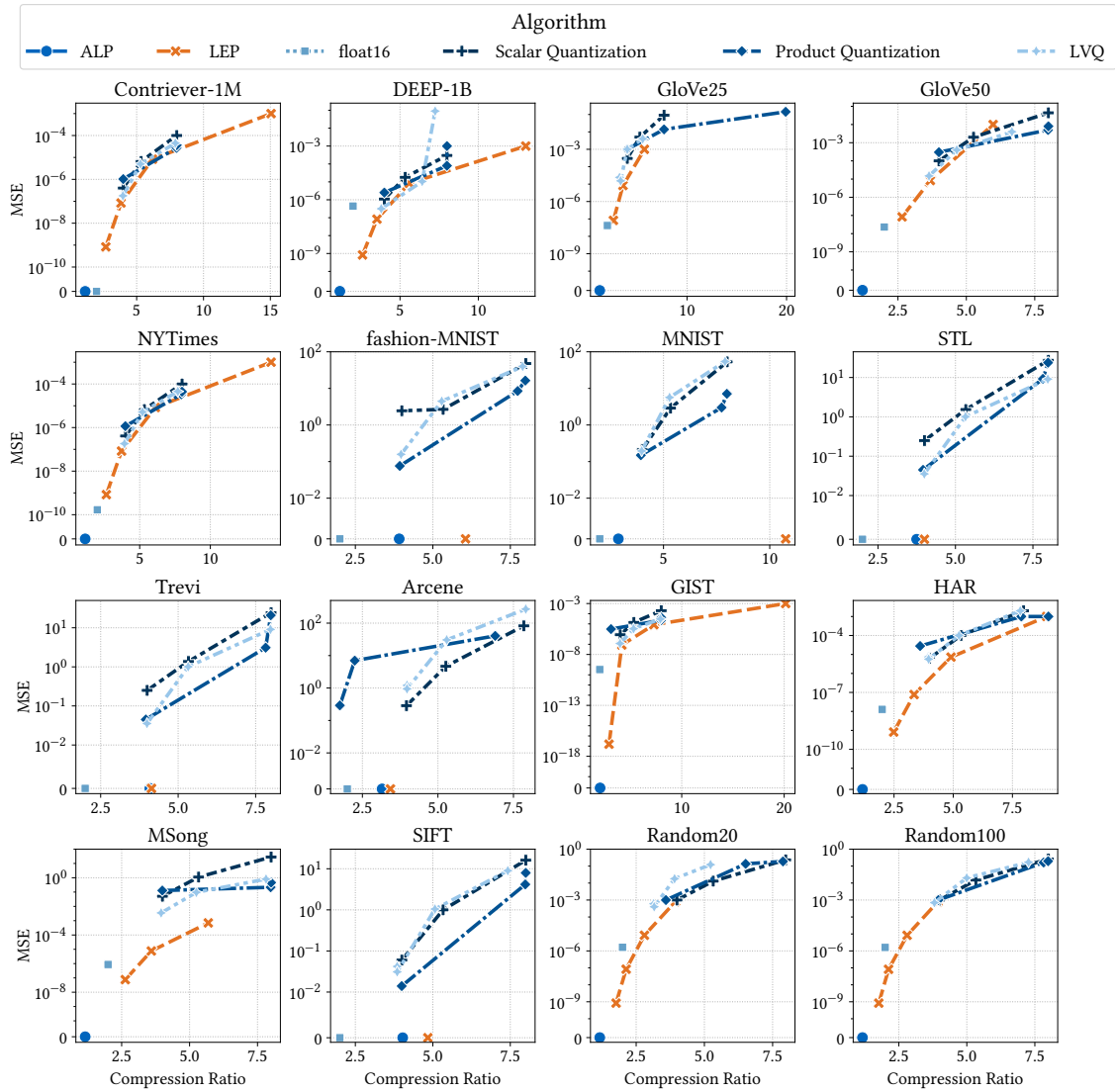


Figure 5.4: The mean squared error of LEP compared to other compression algorithms

We also measured the mean squared error (MSE) of every encoding algorithm. The MSE is a metric that shows how much information is lost between the original values of the vectors and their reconstruction values. A lower MSE means that the reconstructed values are closer to the original ones. The MSE between a vector of raw values $u = [u_1, u_2, \dots, u_n]$ and a vector of decompressed values $d = [d_1, d_2, \dots, d_n]$ that were obtained after compressing u and decompressing, is defined as follows:

$$MSE(u, d) = \frac{1}{n} \cdot \sum_{i=1}^n (u_i - d_i)^2$$

Figure 5.4 shows the MSE for different compression ratios on all datasets. In these plots, values to the lower right are better as they have a higher compression ratio and smaller MSE.

In this metric, LEP lies below the other curves for almost all computed compression ratios. The curves clearly demonstrate that LEP is superior in, e.g., Fashion-MNIST, MNIST, and MSong. Additionally, we can provide fixed error bounds for a chosen level of compression thanks to the nature of our algorithm. For a chosen LEP exponent, we know at which digit pruning starts, and, therefore, LEP has an upper bound for the distance between the reconstructed and original values.

Differences to Scalar Quantization

The transformation of the data performed by LEP is similar to that of SQ. Both algorithms map intervals of floating-point numbers onto buckets. In the case of LEP, the values that are the same after the multiplication with the exponent or factor and casting it to an integer are mapped to the same values representing one bucket. For instance, if exponent 1 is used, then 12.10 and 12.11 are both transformed to 121. On the other hand, SQ calculates buckets of equal size based on the minimum and maximum of the dataset. Data next to each other are then mapped to the same bucket or two adjacent buckets. This similarity between the algorithms can also be seen in figure 5.4. The line for SQ is usually close to the one for LEP.

However, on most datasets, the MSE of LEP is lower than that of SQ. The major differences between the two algorithms are that LEP compresses clustered data in the vertical layout, chooses the bit-width depending on the data, and uses compressed exceptions in the FOR. The parameters bit-width and FOR base are calculated on data stored in consecutive memory, which usually come from one cluster and one dimension. To justify that these differences are the reason why LEP usually has a lower MSE than SQ, we performed the following experiment. We calculated the compression ratio and MSE of LEP and SQ only on one dimension from one cluster at a time. Because of this, SQ chose its parameters only for that one dimension and cluster. Furthermore, we compressed whole datasets using SQ and calculated the MSE for each dimension separately. This enables us to compare SQ using parameters per dimension to SQ using parameters calculated for the whole dataset.

Figure 5.5 shows the results of this experiment on the first clusters of GloVe25 and HAR. In all plots, the results of SQ with parameters calculated for the chosen dimension are better than when using parameters calculated for the whole dataset. The effect is especially prominent on HAR, as in this dataset every dimension has a different distribution (see the data distribution plot in table 3.1). Moreover, while the dimensions in GloVe25 are more similar to each other, they are not identical, so having different parameters per dimension can better use the bits provided per value. In almost all plots, LEP is better than both SQ variants. We attribute this to the fact that LEP has compressed exceptions in the FOR and can choose the bit-width depending on each LEP vector. In the last plot, the MSE of SQ with parameters calculated per dimension is slightly better than that of LEP. However, LEP is still better than SQ with parameters calculated for the whole dataset.

To show the positive impact of letting LEP choose the bit-width adaptively per LEP vector, we compared LEP and SQ with parameters chosen per dimension to LEP with a fixed bit-width. To control the different MSEs we got for LEP with a fixed bit-width, we chose different factors as in non-decimal LEP. The results on

the first clusters of GloVe25 and HAR are shown in figure 5.6. For each fixed bit-width, the best compression ratio LEP can reach is slightly smaller than the one of SQ. This is because LEP has more metadata. In the cases when the factor is already so small that there are no exceptions, choosing a smaller factor leads to a higher MSE because then more of the precision of the floating-point numbers is pruned. However, the MSE of LEP is lower than the one of SQ as LEP has compressed exceptions. When choosing bigger factors, both the compression ratio and MSE of LEP get smaller. We also compared LEP with a fixed bit-width to LEP that chooses the bit-width depending on the data. Here, one can see that for some factors, the algorithm is able to choose a bit-width that leads to a much higher compression ratio as this parameter is chosen locally per LEP vector. Therefore, letting LEP choose this parameter depending on the data also leads to lower MSEs for compression ratios that are the same as SQ.

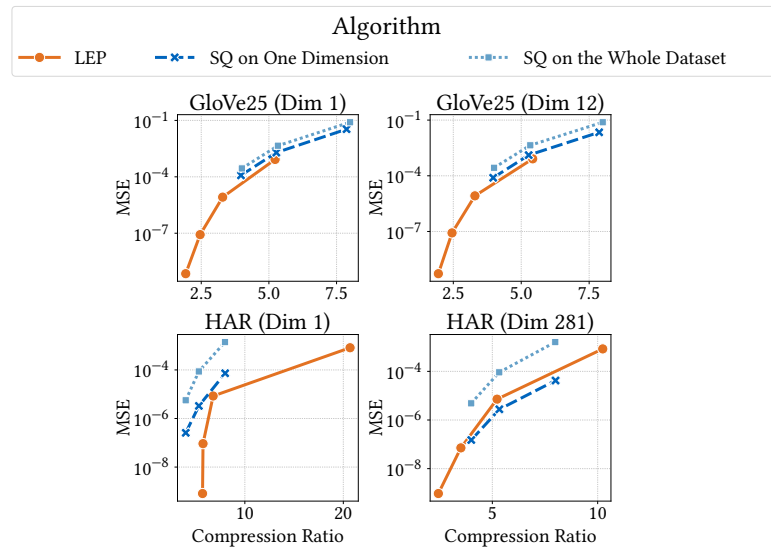


Figure 5.5: The compression ratio - MSE curve of LEP and SQ calculated on single dimensions (Dim); for SQ, parameters were both calculated per dimension and for the whole dataset

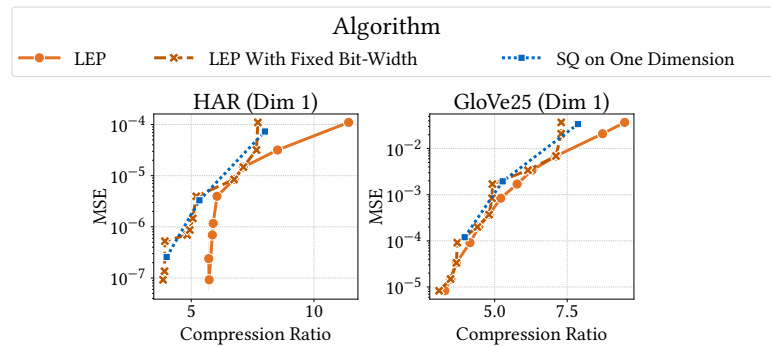


Figure 5.6: The compression ratio - MSE curve of LEP with and without a fixed bit width and SQ calculated on single dimensions (Dim)

Bitmap-Compression of Frequent Values

In section 3.5, we identified datasets that contain a single value so frequently that a bitmap encoding would increase the compression ratio. Figure 5.7 shows the compression ratio - recall curves similar to figure 5.3 but also contains the compression ratios using the bitmap encoding.

While the compression ratio improved a lot on Arcene, Fashion-MNIST, MNIST, and SIFT (by 50%, 11%, 14%, and 10%, respectively), the improvement was less pronounced for HAR, MSong, STL, and Trevi (3%, 6%, 3%, and 1%, respectively). This is because, on the latter datasets, a frequent value only exists in less than 30% of the LEP vectors. Despite this, incorporating a bitmap-based compression to this type of dataset is a valuable improvement because the recall does not suffer from it.

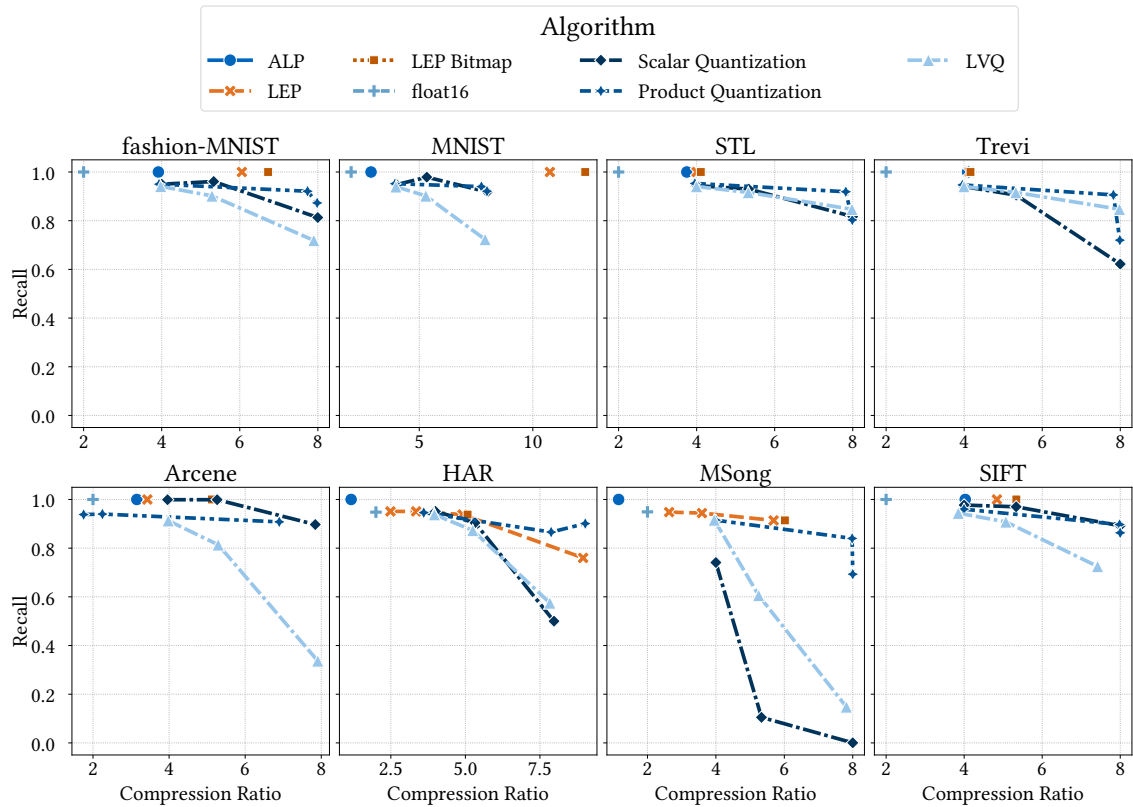


Figure 5.7: The compression ratio - recall curve of LEP with the bitmap encoding on all the datasets where the bitmap encoding is applicable

Non-decimal LEP

For figure 5.8, we calculated the compression ratios and recalls on 4 datasets when using factors that are not powers of 10. We marked the new datapoints with bigger crosses. Recall that the latter was proposed in subsection 4.3 as a way to further tune the compression ratio and recall in LEP. The latter is usable to avoid big gaps in the compression ratio - recall curves.

On Arcene and STL, we multiplied the values by factors between 0 and 1 as they are integer datasets. While this approach can achieve higher compression ratios, the recall changes significantly when it becomes lossy between factors 1 and 0.9. The original plot of HAR in figure 5.3 shows a large gap between the two highest compression ratios. The non-decimal LEP can, therefore, be used to have smaller steps between the different compression ratios. Meanwhile, the recall values all lay on one curve. For MSong, we can see a similar curve. However, we used factors between 1 and 10 on this dataset to get more compression ratios in the range of higher recalls.

While it would also be possible to use factors between 0 and 1 on the float datasets, the recall was already small when using 10^1 on them. Therefore, this would only be relevant for applications that do not need a high recall.

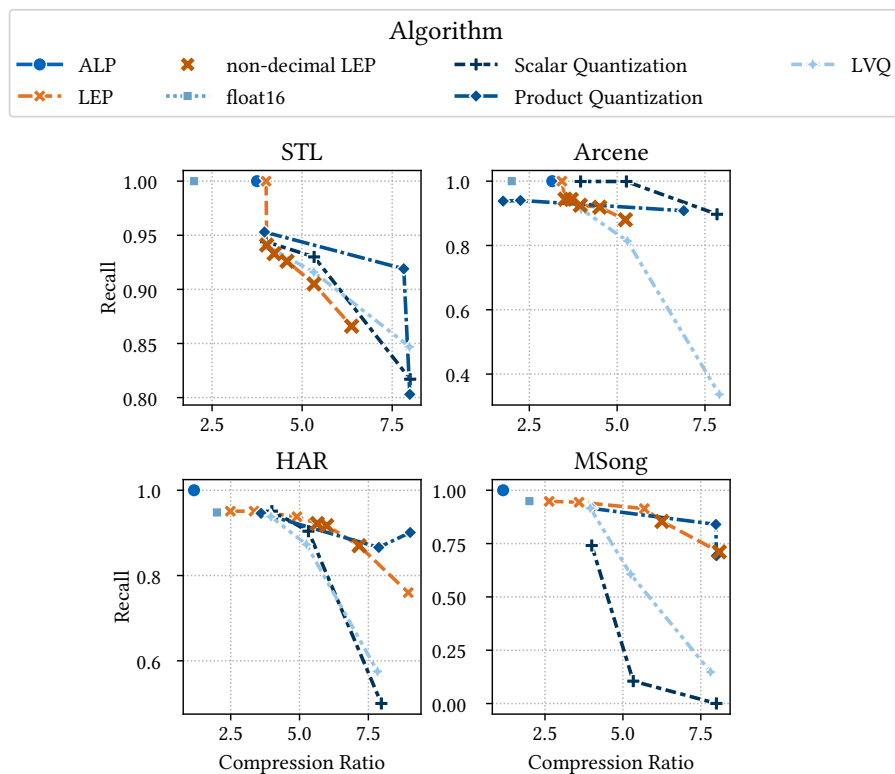


Figure 5.8: The compression ratio - recall curve of LEP using non-decimal LEP on some datasets

5.3 Evaluation of the Encoding Speeds

Table 5.1 shows the runtime and the reached compression ratio for each dataset when doing the sampling proposed in section 4.4 to increase the encoding speed of PFOR. The recall is unaffected because the PFOR is still lossless even when sampling the values.

On all datasets but Fashion-MNIST, MNIST and HAR, at least 97% of the previous compression ratio using no sampling can be reached. Fashion-MNIST and MNIST are datasets with frequently occurring values. Therefore, the decreased compression ratio can be explained by having sampled this value too often, which makes the algorithm choose a bit-width that is too small, leading to a higher number of exceptions. Nevertheless, we can still reach high compression ratios on these datasets with the bitmap encoding. HAR has a highly skewed data distribution, as seen in table 3.1. Therefore, the sampling over 100 LEP vectors might have fallen short of correctly capturing the distribution of the data. The runtime improvement is between 3.117x and 8.647x, being 5.593x faster on average.

| Dataset (LEP Exponent) | CR Using Sam- pling | CR Without Sampling | Share of the Original CR Achieved Using Sam- pling | ET [ms] Using Sam- pling | ET [ms] Without Sampling | Improvement On the ET Achieved Using Sam- pling |
|---------------------------|---------------------------|------------------------|--|--------------------------------|--------------------------------|---|
| Contriever-1M (3) | 3.775 | 3.831 | 0.982 | 68,207 | 273,310 | 4.007 |
| DEEP-1B (3) | 3.500 | 3.530 | 0.992 | 104,228 | 324,926 | 3.117 |
| GloVe25 (2) | 3.460 | 3.567 | 0.970 | 2,334 | 20,181 | 8.647 |
| GloVe50 (2) | 3.573 | 3.684 | 0.970 | 4,405 | 18,531 | 4.207 |
| NYTimes256 (2) | 6.044 | 6.106 | 0.990 | 4,216 | 36,089 | 8.560 |
| Fashion-MNIST (0) | 4.681 | 6.054 | 0.773 | 4,157 | 15,876 | 3.819 |
| MNIST (0) | 7.641 | 10.751 | 0.711 | 3,244 | 12,001 | 3.699 |
| STL (0) | 3.935 | 4.000 | 0.984 | 76,850 | 509,002 | 6.623 |
| Trevi (0) | 4.088 | 4.137 | 0.988 | 32,219 | 197,210 | 6.121 |
| Arcene (0) | 3.417 | 3.428 | 0.998 | 695 | 4,455 | 6.410 |
| GIST (3) | 4.129 | 4.158 | 0.993 | 94,839 | 448,340 | 4.727 |
| HAR (3) | 2.926 | 3.348 | 0.874 | 468 | 2,267 | 5.058 |
| MSong (2) | 3.557 | 3.593 | 0.990 | 41,696 | 208,316 | 4.996 |
| SIFT (0) | 4.716 | 4.840 | 0.974 | 10,655 | 46,954 | 4.407 |
| Random20 (3) | 2.111 | 2.133 | 0.990 | 18 | 151 | 8.389 |
| Random100 (3) | 2.114 | 2.125 | 0.995 | 766 | 5,140 | 6.710 |

Table 5.1: The compression ratio (CR) and encoding time (ET) of sampling vectors at encoding time in the LEP algorithm

6 Conclusion and Future Work

In this thesis, we explored possibilities of compressing vector embeddings lossily with an algorithm we called LEP based on the lossless lightweight floating-point number compression algorithm ALP. We identified often-used datasets for NNS in the literature to develop the algorithm and analyzed them to adapt the ALP algorithm to work lossily on real-world data. The key findings below answer the main research questions formulated in section 1.1.

6.1 Answers to the Research Questions

Q1: Compressing Embeddings Lossily Based on the ALP Algorithm Which compression ratios can be reached when compressing embeddings losslessly using ALP? How is the recall of the nearest neighbor search affected by making ALP lossy?

Compressing embeddings losslessly using the ALP algorithm does not lead to high compression ratios; it can save less than 25% of the space on the datasets with learned features. On the image datasets containing pixels that were originally stored as 8-bit integers, ALP also chooses to bitpack them to 8 bits in most LEP vectors, meaning it only saves a little space.

On the other hand, LEP can reach high recalls of over 90% when having compression ratios of at least 3.4 on all datasets. When we choose the smallest exponents leading to these recalls of over 90%, the average compression ratio is 5.124, and the average recall is 0.956. When sacrificing a bit more recall, much higher compression ratios are achievable. For example, on Contriever-1M, for a recall of 0.947, the compression ratio is 3.831. When allowing a lower recall of 0.890, LEP reaches the compression ratio of 6.186.

Q2: Improving the Compression Ratio on LEP Can the compression ratio be improved by exploiting a data layout for vector embeddings that shares ideas with data layouts in analytical database systems? Which other characteristics do embedding datasets have that can be exploited to amplify the compression? Can we control the compression ratio reached by the lossy algorithm?

To improve the compression ratio on LEP, we tested horizontal and vertical layouts and reordered them in terms of the clusters of an IVF index. On almost all datasets, using the vertical layout and reordering the embeddings in terms of their IVF clusters leads to higher compression ratios than using the horizontal layout and not clustering the data. The amount of improvement depends on the compression ratio before changing the layout and is higher for high compression ratios. For example, on DEEP-1B using LEP exponent 1 and clustering the data, the value of this compression metric improved from 10.257 to 13.031 when using the vertical instead of the horizontal layout.

Incorporating PFOR, as proposed by Zukowski et al. in [75], also leads to higher compression ratios on embedding datasets. Many of the embedding datasets, especially those containing learned features, have a normal distribution. On these sets, opportunities to compress the exceptions further exist because the distance from most exceptions to the rest of the data is small. We propose a scheme that requires only 16 additional bits per most of the exceptions and is lossless. Another characteristic we found in some datasets is that they contain one value very often. This is the case, for example, for the value 0 in Arcene and some image datasets. The bitmap compression described in section 4.2 can exploit these frequently occurring values. For example, on MNIST, the compression ratio improved from 10.751 to 12.308 when leveraging these values.

When using different LEP exponents to influence the compression ratio and recall of LEP, only a limited number of different ratios is reachable. However, unlike the lossless ALP algorithm, LEP does not need to work on the decimal representation of the floating-point numbers as it can tolerate reconstruction errors. Because of this, we can use other factors than powers of 10. This leads to a much higher range of possible factors and, therefore, reachable compression ratios. How to best find a factor for a given compression ratio remains an open question.

Q3: Using Correlated Dimensions Do the embedding datasets contain correlated dimensions? How can they be used to improve the compression ratio?

Some of the embedding datasets contain correlated dimensions, which is the case mostly on datasets with manually chosen features. We used linear regression based on the Pearson coefficient to compress these columns losslessly. This compression led to a slightly improved compression ratio in some datasets but also increased the required computations and transferred data when decompressing. Thus, we did not include this compression technique in the algorithm.

Q4: Comparing LEP to Other Embedding Compression Algorithms How does LEP perform concerning compression ratio and mean squared error compared to other embedding compression algorithms?

When targeting similar recalls, LEP reaches compression ratios similar to or higher than those of the algorithms we compared it to. While quantization approaches fail to reach recalls above 75% on some datasets like NYTimes256 on the compression ratios we were testing, this does not happen with LEP on all datasets we tested. Furthermore, LEP reaches a recall of 100% on integer datasets. For LEP, we can specify the maximal error reached on each value before executing the algorithm. We also can give an upper bound for the MSE. On real-world datasets, this error is usually even lower. Compared to other compression algorithms, LEP has an MSE that is almost always smaller. This is because the values are not changed aggressively.

6.2 Future Work

This thesis answers the research questions defined in section 1.1. These questions focus mainly on the compression ratio reached by LEP. Therefore, questions remain open concerning the LEP algorithm and supplementary work on vector embeddings search.

Using Correlated Dimensions In section 3.6, we only look for linear relationships between dimensions. One could try to use other relationships, like other polynomials on the dimension pairs. These relationships might increase the compression ratio more, making using them more interesting despite the decompression overhead.

Choosing the Bitmap Compression In section 4.2, we propose a second compression scheme that leverages frequently occurring values in a LEP vector. So that the scheme has a noticeable effect, we chose to use it on vectors where a value is repeated at least 300 times. To find an optimal solution, we counted how often every value in the LEP vector occurred. This is computationally expensive. Better solutions that can be tried include counting occurrences on sampled values per LEP vector or only on some LEP vectors. Suppose the latter solution finds no frequently occurring values on all tested LEP vectors, and the dataset has a comparable distribution in all dimensions. In that case, the bitmap compression scheme can be discarded for all LEP vectors.

Measuring Encoding and Decoding Speed This thesis focuses on finding high compression ratios on vector embeddings. Concerning the encoding speed, we propose a sampling scheme that can reach almost the same compression ratios as the exhaustive search for the parameters bit-width and FOR base. However, we only tested this scheme in micro-benchmarks. Furthermore, we think that the runtime for encoding can be improved more by optimizing the code, for example, by using SIMD instructions.

This thesis does not include benchmarks on the decoding speed due to time constraints and the limited scope of the thesis project. The speed is interesting in both micro-benchmarks and search algorithms. When incorporated into a search algorithm, one can see if the compression leads to an overhead because of decompression time or if working on smaller integers resulting from the compression can induce an improvement to the overall search speed.

Using the Compressed Embeddings in a Search Algorithm As mentioned before, when designing the compression algorithm, we kept in mind that the embeddings would eventually be used in a search algorithm. The search can leverage the fact that the data are stored in the vertical layout and clustered. Nonetheless, some changes to the search algorithm would be required. Most embedding searches work on 32-bit floating-point numbers. To save part of the decoding time, we suggest using the integer representation that is obtained after undoing the bitpacking of the compressed data. However, these un-bitpacked values can have different sizes between different LEP vectors. An algorithm would then need different cases dealing with, for example, 8-bit and 16-bit integers. As LEP compresses blocks of 1024 values, accessing single values within them is harder than random access on uncompressed data. Solutions for this can be using a hybrid layout that stores part of the data in the vertical and the rest in the horizontal layout as required by the algorithm. One can also try to work on smaller vectors to have to decompress less data.

Bringing the Compression Algorithm to Production Readiness In the algorithm we propose in this thesis, the LEP exponent or factor influencing the compression ratio and recall are parameters that the user has to choose. However, making this choice is unintuitive, as the actual compression ratio depends on the dataset and cannot be predicted precisely from the chosen parameter. Therefore, letting the user choose the compression ratio would be better. The algorithm can then sample a few values and do a binary search on different factors to find one that leads to approximately the required compression ratio. For users, it can also be interesting to request a recall. Parameters for this could be found by sampling some vectors and executing an exact nearest neighbor search on these vectors, both compressed with different factors and uncompressed. Still, this is computationally expensive, and it is an open question whether there are more efficient algorithms for estimating the recall.

Finally, we did not include LEP into a database management system like DuckDB. Open questions when doing this inclusion are how to store embeddings in the vertical layout in a database table, which datatype can be used for bitpacked values, and what the interface for the user should look like. As mentioned earlier, the user needs options for choosing the LEP exponent, factor, compression ratio, or recall. A new SQL statement could do this.

Bibliography

- [1] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *Proceedings of the VLDB Endowment* 16, 9 (May 2023), 2132–2144. <https://doi.org/10.14778/3598581.3598587>
- [2] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proceedings of the ACM on Management of Data* 1, 4 (Dec. 2023), 230:1–230:26. <https://doi.org/10.1145/3626717>
- [3] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. 2023. Similarity Search in the Blink of an Eye with Compressed Indices. *Proceedings of the VLDB Endowment* 16, 11 (July 2023), 3433–3446. <https://doi.org/10.14778/3611479.3611537>
- [4] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2017. Accelerated Nearest Neighbor Search with Quick ADC. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval (ICMR '17)*. Association for Computing Machinery, New York, NY, USA, 159–166. <https://doi.org/10.1145/3078971.3078992>
- [5] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (Jan. 2020), 101374. <https://doi.org/10.1016/j.is.2019.02.006>
- [6] Artem Babenko and Victor Lempitsky. 2014. Additive Quantization for Extreme Vector Compression. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Columbus, OH, USA, 931–938. <https://doi.org/10.1109/CVPR.2014.124>
- [7] Thierry Bertin-Mahieux, Daniel P. W. Ellis, Brian Whitman, and Paul Lamere. 2011. The Million Song Dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*. University of Miami, Miami, FL, USA, 591–596.
- [8] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1999. When Is “Nearest Neighbor” Meaningful?. In *Database Theory — ICDT'99*, Catriel Beeri and Peter Buneman (Eds.). Springer, Berlin, Heidelberg, Germany, 217–235. https://doi.org/10.1007/3-540-49257-7_15
- [9] Alina Beygelzimer, Sham Kakade, and John Langford. 2006. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning (ICML '06)*. Association for Computing Machinery, New York, NY, USA, 97–104. <https://doi.org/10.1145/1143844.1143857>
- [10] Davis W. Blalock and John V. Guttag. 2017. Bolt: Accelerated Data Mining with Fast Vector Compression. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. Association for Computing Machinery, New York, NY, USA, 727–735. <https://doi.org/10.1145/3097983.3098195>
- [11] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (July 2020), 2649–2661. <https://doi.org/10.14778/3407790.3407851>

- [12] Martin Burtscher and Paruj Ratanaworabhan. 2009. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. *IEEE Trans. Comput.* 58, 1 (Jan. 2009), 18–31. <https://doi.org/10.1109/TC.2008.131>
- [13] Yongjian Chen, Tao Guan, and Cheng Wang. 2010. Approximate Nearest Neighbor Search by Residual Vector Quantization. *Sensors* 10, 12 (Dec. 2010), 11259–11273. <https://doi.org/10.3390/s101211259>
- [14] Arjen P. De Vries, Nikos Mamoulis, Niels Nes, and Martin Kersten. 2002. Efficient k-NN search on vertically decomposed data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, Madison, WI, USA, 322–333. <https://doi.org/10.1145/564691.564729>
- [15] Xavier Delaunay, Aurélie Courtois, and Flavien Gouillon. 2019. Evaluation of lossless and lossy algorithms for the compression of scientific datasets in netCDF-4 or HDF5 files. *Geoscientific Model Development* 12, 9 (Sept. 2019), 4099–4113. <https://doi.org/10.5194/gmd-12-4099-2019>
- [16] Sheng Di, Jinyang Liu, Kai Zhao, Xin Liang, Robert Underwood, Zhaorui Zhang, Milan Shah, Yafan Huang, Jiajun Huang, Xiaodong Yu, Congrong Ren, Hanqi Guo, Grant Wilkins, Dingwen Tao, Jiannan Tian, Sian Jin, Zizhe Jian, Daoce Wang, MD Hasanur Rahman, Boyuan Zhang, Jon C. Calhoun, Guanpeng Li, Kazutomo Yoshii, Khalid Aayed Alharthi, and Franck Cappello. 2024. A Survey on Error-Bounded Lossy Compression for Scientific Datasets. <https://doi.org/10.48550/arXiv.2404.02840>
- [17] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*. ACM, Hyderabad, India, 577–586. <https://doi.org/10.1145/1963405.1963487>
- [18] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvassy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. <https://doi.org/10.48550/arXiv.2401.08281>
- [19] DuckDB Labs 2022. Patas Compression (float/double) (variation on Chimp). Retrieved 2024-08-20 from <https://github.com/duckdb/duckdb/pull/5044>
- [20] Vadim Engelson, Peter Fritzon, and Dag Fritzon. 2000. Lossless compression of high-volume numerical data from simulations. In *Proceedings DCC 2000. Data Compression Conference*. IEEE, Snowbird, UT, USA, 574. <https://doi.org/10.1109/DCC.2000.838221>
- [21] Erik Bernhardsson. [n.d.]. Annoy. Retrieved 2024-07-31 from <https://github.com/spotify/annoy>
- [22] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. 1977. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Software* 3, 3 (Sept. 1977), 209–226. <https://doi.org/10.1145/355744.355745>
- [23] Cong Fu and Deng Cai. 2016. EFANNA : An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph. <https://doi.org/10.48550/arXiv.1609.07228>
- [24] Jianyang Gao and Cheng Long. 2023. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. *Proceedings of the ACM on Management of Data* 1, 2 (June 2023), 137:1–137:27. <https://doi.org/10.1145/3589282>

- [25] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 3 (May 2024), 167:1–167:27. <https://doi.org/10.1145/3654970>
- [26] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 4 (April 2014), 744–755. <https://doi.org/10.1109/TPAMI.2013.240>
- [27] Allen Gersho and Robert M. Gray. 2012. *Vector Quantization and Signal Compression*. Springer Science & Business Media, New York, NY, USA.
- [28] Thomas Glas. 2023. *Exploiting Column Correlations for Compression*. Master’s thesis. Technische Universität München, München, Germany.
- [29] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering*. IEEE Comput. Soc, Orlando, FL, USA, 370–379. <https://doi.org/10.1109/ICDE.1998.655800>
- [30] Mihajlo Grbovic and Haibin Cheng. 2018. Real-time Personalization using Embeddings for Search Ranking at Airbnb. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD ’18)*. Association for Computing Machinery, New York, NY, USA, 311–320. <https://doi.org/10.1145/3219819.3219885>
- [31] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.* 14, 2 (June 1984), 47–57. <https://doi.org/10.1145/971697.602266>
- [32] Yoonho Hwang, Bohyung Han, and Hee-Kap Ahn. 2012. A fast nearest neighbor search algorithm by nonlinear embedding. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Providence, RI, USA, 3053–3060. <https://doi.org/10.1109/CVPR.2012.6248036>
- [33] Ville Hyvönen, Teemu Pitkänen, Sotirios Tasoulis, Elias Jääsaari, Risto Tuomainen, Liang Wang, Jukka Ilmari Corander, and Teemu Roos. 2016. Fast Nearest Neighbor Search through Sparse Random Projections and Voting. In *Proceedings of the 2016 IEEE Conference on Big Data*. IEEE, New York, NY, USA, 881–888. <https://doi.org/10.1109/BigData.2016.7840682>
- [34] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), 1–84. <https://doi.org/10.1109/IEEESTD.2019.8766229>
- [35] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC ’98*. ACM Press, Dallas, TX, USA, 604–613. <https://doi.org/10.1145/276698.276876>
- [36] Intel Corporation 2021. Intel AVX512-FP16 Architecture Specification. Retrieved 2024-08-21 from <https://cdrdv2-public.intel.com/678970/intel-avx512-fp16.pdf>
- [37] Seungdo Jeong, Sang-Wook Kim, Kidong Kim, and Byung-Uk Choi. 2006. An Effective Method for Approximating the Euclidean Distance in High-Dimensional Space. In *Database and Expert Systems Applications*, Stéphane Bressan, Josef Küng, and Roland Wagner (Eds.). Springer, Berlin, Heidelberg, Germany, 863–872. https://doi.org/10.1007/11827405_84
- [38] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (July 2021), 535–547. <https://doi.org/10.1109/TBDATA.2019.2921572>

-
- [39] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (Jan. 2011), 117–128. <https://doi.org/10.1109/TPAMI.2010.57>
- [40] Anthony Ko, Iman Keivanloo, Vihan Lakshman, and Eric Schkufza. 2021. Low-Precision Quantization for Efficient Nearest Neighbor Search. <http://arxiv.org/abs/2110.08919>
- [41] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proceedings of the ACM on Management of Data* 1, 2 (June 2023), 118:1–118:26. <https://doi.org/10.1145/3589263>
- [42] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov. 1998), 2278–2324. <https://doi.org/10.1109/5.726791>
- [43] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-Based Lossless Floating-Point Compression. *Proceedings of the VLDB Endowment* 16, 7 (March 2023), 1763–1776. <https://doi.org/10.14778/3587136.3587149>
- [44] Panagiotis Liakos, Katia Papakonstantinou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment* 15, 11 (July 2022), 3058–3070. <https://doi.org/10.14778/3551793.3551852>
- [45] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J. Elmore. 2021. Decomposed bounded floats for fast compression and queries. *Proceedings of the VLDB Endowment* 14, 11 (July 2021), 2586–2598. <https://doi.org/10.14778/3476249.3476305>
- [46] Hanwen Liu, Mihail Stoian, Alexander van Renen, and Andreas Kipf. 2024. Corra: Correlation-Aware Column Compression. <https://doi.org/10.48550/arXiv.2403.17229>
- [47] Yury A. Malkov and Dmitry A. Yashunin. 2018. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. <https://doi.org/10.48550/arXiv.1603.09320>
- [48] Julieta Martinez, Joris Clement, Holger H. Hoos, and James J. Little. 2016. Revisiting Additive Quantization. In *Computer Vision – ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, Cham, Germany, 137–153. https://doi.org/10.1007/978-3-319-46475-6_9
- [49] Julieta Martinez, Shobhit Zakhmi, Holger H. Hoos, and James J. Little. 2018. LSQ++: Lower Running Time and Higher Recall in Multi-codebook Quantization. In *Computer Vision – ECCV 2018*, Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.). Vol. 11220. Springer International Publishing, Cham, Germany, 508–523. https://doi.org/10.1007/978-3-030-01270-0_30
- [50] Adam C. Mater and Michelle L. Coote. 2019. Deep Learning in Chemistry. *Journal of Chemical Information and Modeling* 59, 6 (June 2019), 2545–2559. <https://doi.org/10.1021/acs.jcim.9b00266>
- [51] Meta [n.d.]. ZStandard. Retrieved 2024-08-20 from <https://github.com/facebook/zstd>
- [52] Meta 2017. Faiss: A library for efficient similarity search. Retrieved 2024-09-04 from <https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/>
-

- [53] Meta Research [n.d.]. Faiss. Retrieved 2024-08-26 from <https://github.com/facebookresearch/faiss>
- [54] Tomàs Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, (ICLR)*. Scottsdale, AZ, USA. <https://doi.org/10.48550/arXiv.1301.3781>
- [55] Milvus [n.d.]. What is Milvus. Retrieved 2024-08-26 from <https://milvus.io/docs/overview.md>
- [56] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11 (Nov. 2014), 2227–2240. <https://doi.org/10.1109/TPAMI.2014.2321376>
- [57] Gonzalo Navarro. 1999. Searching in metric spaces by spatial approximation. In *6th International Symposium on String Processing and Information Retrieval. 5th International Workshop on Groupware (Cat. No.PR00268)*. IEEE, Cancun, Mexico, 141–148. <https://doi.org/10.1109/SPIRE.1999.796589>
- [58] Lushuai Niu, Zhi Xu, Longyang Zhao, Daojing He, Jianqiu Ji, Xiaoli Yuan, and Mian Xue. 2023. Residual Vector Product Quantization for approximate nearest neighbor search. *Expert Systems with Applications* 232 (Dec. 2023), 120832. <https://doi.org/10.1016/j.eswa.2023.120832>
- [59] Mohammad Norouzi and David J. Fleet. 2013. Cartesian K-Means. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Portland, OR, USA, 3017–3024. <https://doi.org/10.1109/CVPR.2013.388>
- [60] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1816–1827. <https://doi.org/10.14778/2824032.2824078>
- [61] Pinecone Systems [n.d.]. Product Quantization: Compressing high-dimensional vectors by 97%. Retrieved 2024-08-26 from <https://www.pinecone.io/learn/series/faiss/product-quantization/>
- [62] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. <https://doi.org/10.1145/3299869.3320212>
- [63] Vijayshankar Raman and Garret Swart. 2006. How to wring a table dry: entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd international conference on Very large data bases (VLDB '06)*. VLDB Endowment, Seoul, Korea, 858–869.
- [64] Josef Sivic and Andrew Zisserman. 2003. Video Google: a text retrieval approach to object matching in videos. In *Proceedings Ninth IEEE International Conference on Computer Vision*. IEEE, Nice, France, 1470–1477 vol.2. <https://doi.org/10.1109/ICCV.2003.1238663>
- [65] The Apache Software Foundation [n.d.]. Parquet. Retrieved 2024-08-21 from <https://parquet.apache.org/>
- [66] Ash Vardanian. 2023. USearch by Unum Cloud. <https://doi.org/10.5281/zenodo.7949416>

- [67] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. 2016. Learning to Hash for Indexing Big Data—A Survey. *Proc. IEEE* 104, 1 (Jan. 2016), 34–57. <https://doi.org/10.1109/JPROC.2015.2487976>
- [68] Weaviate 2023. How to Reduce Memory Requirements by up to 90%+ using Product Quantization. Retrieved 2024-08-26 from <https://weaviate.io/blog/pq-rescoring>
- [69] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik P. A. Lensch. 2016. Efficient Large-scale Approximate Nearest Neighbor Search on the GPU. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Las Vegas, NV, USA, 2027–2035. <https://doi.org/10.1109/CVPR.2016.223>
- [70] Shangyu Wu, Ying Xiong, Yufei Cui, Haolun Wu, Can Chen, Ye Yuan, Lianming Huang, Xue Liu, Tei-Wei Kuo, Nan Guan, and Chun Jason Xue. 2024. Retrieval-Augmented Generation for Natural Language Processing: A Survey. <https://doi.org/10.48550/arXiv.2407.13193>
- [71] Charles S. Zender. 2016. Bit Grooming: statistically accurate precision-preserving quantization with compression, evaluated in the netCDF Operators (NCO, v4.4.8+). *Geoscientific Model Development* 9, 9 (Sept. 2016), 3199–3211. <https://doi.org/10.5194/gmd-9-3199-2016>
- [72] Xianzhi Zeng, Zhuoyan Wu, Xinjing Hu, Xuanhua Shi, Shixuan Sun, and Shuhao Zhang. 2024. CANDY: A Benchmark for Continuous Approximate Nearest Neighbor Search with Dynamic Data Ingestion. <https://doi.org/10.48550/arXiv.2406.19651>
- [73] Haowen Zhang, Yabo Dong, and Duanqing Xu. 2021. Accelerating exact nearest neighbor search in high dimensional Euclidean space via block vectors. *International Journal of Intelligent Systems* 37, 2 (Oct. 2021), 1697–1722. <https://doi.org/10.1002/int.22692>
- [74] Haowen Zhang, Jing Li, Junru Zhang, and Yabo Dong. 2023. Speeding Up K-Means Clustering in High Dimensions by Pruning Unnecessary Distance Computations. <https://doi.org/10.2139/ssrn.4573970>
- [75] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, Atlanta, GA, USA, 59–59. <https://doi.org/10.1109/ICDE.2006.150>
- [76] Kacper Łukawski. 2023. Scalar Quantization: Background, Practices & More. Retrieved 2024-07-31 from <https://qdrant.tech/articles/scalar-quantization/>