Similarity search in the blink of an eye with compressed indices

Cecilia Aguerrebere
Intel Labs
Santa Clara, California
cecilia.aguerrebere@intel.com

Ishwar Singh Bhati
Intel Labs
Hillsboro, Oregon
ishwar.s.bhati@intel.com

Mark Hildebrand
Intel Labs
Hillsboro, Oregon
mark.hildebrand@intel.com

Mariano Tepper
Intel Labs
Hillsboro, Oregon
mariano.tepper@intel.com

Theodore Willke
Intel Labs
Hillsboro, Oregon
ted.willke@intel.com

ABSTRACT

Nowadays, data is represented by vectors. Retrieving those vectors, among millions and billions, that are similar to a given query is a ubiquitous problem, known as similarity search, of relevance for a wide range of applications. Graph-based indices are currently the best performing techniques for billion-scale similarity search. However, their random-access memory pattern presents challenges to realize their full potential. In this work, we present new techniques and systems for creating faster and smaller graph-based indices. To this end, we introduce a novel vector compression method, Locally-adaptive Vector Quantization (LVQ), that uses per-vector scaling and scalar quantization to improve search performance with fast similarity computations and a reduced effective bandwidth, while decreasing memory footprint and barely impacting accuracy. LVQ, when combined with a new high-performance computing system for graph-based similarity search, establishes the new state of the art in terms of performance and memory footprint. For billions of vectors, LVQ outcompetes the second-best alternatives: (1) in the low-memory regime, by up to 20.7x in throughput with up to a 3x memory footprint reduction, and (2) in the high-throughput regime by 5.8x with 1.4x less memory.

PVLDB Reference Format:
Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. Similarity search in the blink of an eye with compressed indices. PVLDB, 16(11): 3433 - 3446, 2023.
doi:10.14778/3611479.3611537

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/IntelLabs/ScalableVectorSearch/blob/main/reproducibility/VLDB2023.md.

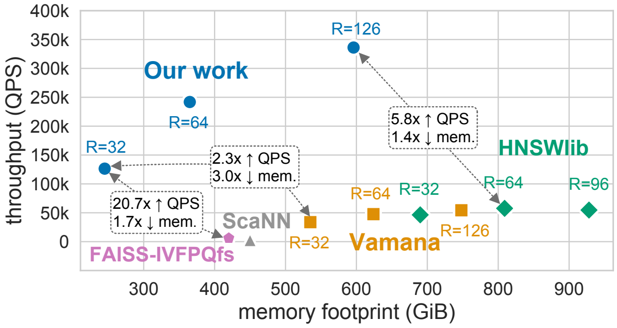


Figure 1: Our contributions enable high-throughput and high-accuracy similarity search with a small memory footprint (results with a 10-recall@10 of 0.9 for deep-96-1B). For graph methods, the memory footprint is a function of the graph out-degree R and the vectors' footprint. Our low-memory configuration (LVQ-8 and $R = 32$) outperforms the current leaders, Vamana [28], HNSWlib [39], FAISS-IVFPQfs [31], and ScaNN [21], by 2.3x, 2.2x, 20.7x, and 43.6x with 3.0x, 3.3x, 1.7x, and 1.8x less memory, respectively. Our highest-throughput configuration (LVQ-8 and $R = 126$) outperforms the second-highest by 5.8x while using 1.4x less memory.

1 INTRODUCTION

In the deep learning era, high-dimensional vectors have become the quintessential data representation for unstructured data, e.g., for images, audio, video, text, genomics, and computer code [e.g., 16, 30, 37, 45, 50]. The representations are generated so that semantically related vectors are close to each other according to a chosen similarity function. A common procedure is to search over these vectors for the nearest neighbors to a given query vector. This enables a wide range of applications, such as image generation [11], NLP [12], question answering [34], recommender systems [38], and ad matching [20]. Datasets with billions of vectors, each with hundreds of dimensions, are increasingly common [52]. Because of the scale and the curse of dimensionality, exact nearest neighbor search is impractical and the literature is focused on approximate methods. Graph-based approximate nearest neighbor methods [5, 28, 39] have been empirically found to offer a better latency-accuracy trade-off than other types of algorithms [55].

Despite requiring fewer memory accesses per query, graph-based search algorithms continue to offer **limited throughput** and a **substantial memory footprint** at very large database sizes making single-machine deployment challenging. Many works have focused on reducing the number of distance computations per query further

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611537

Authors are listed in alphabetical order and have contributed equally to this work.

by changing the parameters of the graph to lower the number of points visited per query. Although this lowers latency and increases query throughput, it does so at the expense of recall and with little reduction in memory consumption. Since applications typically demand a hard and fast lower bound on recall, there are limits to this approach.

The random memory access patterns that come with graph algorithms present further challenges to the system's throughput. The **inability to effectively prefetch** vectors with a hardware prefetcher means that the latency of accessing random vectors cannot be easily hidden and may quickly become a throughput bottleneck. Furthermore, vectors are **difficult to cache** due to the size of the index. Although most of the literature on large-scale similarity search puts more emphasis on the computational intensity of this workload, the simplicity of the distance computation kernel and the aforementioned fetching issues ultimately make the workload memory-bandwidth limited. However, many billion-scale similarity search systems lack the high-performance computing block-and-tackling necessary to wring out enough distance computation performance to put pressure on the memory subsystem.

Gains may be made by performing distance computations on compressed vectors, thereby lowering both computation and memory footprint. However, compression introduces new challenges, including **lowering of recall**. Product Quantization and other lossy compression methods are often used to reduce the memory footprint [28] but incur more **expensive similarity computations** and require the auxiliary **storage of uncompressed vectors** anyway to boost recall during a final re-ranking step. Other methods introduce too much distortion in the distances (e.g., dimensionality reduction and standard scalar quantization), leading to unacceptable accuracy.

In this work, we propose LVQ, a locally-adaptive vector quantizer, that uses a **simple and efficient compression** method to reduce memory pressure and a built-in two-level quantization remainder system that **avoids keeping full precision vectors**. After centering the data, LVQ scales each vector individually (i.e., the local adaptation) and then performs uniform scalar quantization. Its per-vector compression introduces a **negligible accuracy degradation** thanks to its effective usage of all quantization levels. LVQ reduces the bandwidth by up to ~8x compared to a float32-valued vector, greatly accelerating the search. When needed, the second-level quantization remainder is used for a final re-ranking to further boost search recall. Moreover, we show that we can build accurate graphs directly from LVQ-compressed vectors.

We also introduce a new open-source performance library for billion-scale similarity search that removes the barriers limiting the throughput of most graph-based search algorithms and, in our case, allows LVQ to shine. We **streamline memory accesses**, by flattening the memory layout, avoiding any memory indirections, and using a **new custom software prefetcher**. We also accelerate **similarity computations** using AVX instructions, which in the case of LVQ are **blazingly fast** (up to 2.12x faster than with float16-valued vectors). In summary, we present the following contributions:

- **Novel Compression Algorithm**. We present Locally-adaptive Vector Quantization (LVQ), a technique that strikes a balance between effective bandwidth reduction and introducing a minimal decoding overhead for similarity computations (Section 3).
- **Fast Implementation**. The combination of LVQ and our optimized graph search (presented in Section 5 with hyperparameter recommendations) establishes the new state-of-the-art for large-scale similarity search in terms of performance and memory footprint. We present a preview in Figure 1. We backup these claims with an extensive set of experimental results (Section 6).
- **Index construction with LVQ**. LVQ enables building graph indices directly from compressed vectors, releasing memory pressure in this time-consuming step while minimally affecting index quality (Section 4).
- **Open Source Framework**. We open-source a similarity search library[1] to allow the research community to experiment with our algorithms and billion-scale search framework.
- **New Dataset and Generator**. To promote similarity search research in line with modern applications that use deep learning embeddings, we introduce a new dataset with $d = 768$ dimensions, produced using large language models [34]. We open source the code[2] to generate this dataset from a standard corpus. In this paper we use an instance with 10 million vectors.

2 PRELIMINARIES

The similarity search problem (also known as nearest-neighbor search) is defined as follows. Given a vector database $\mathbf{X} = \{\mathbf{x}_i \in \mathbb{R}^d\}_{i=1,\dots,n}$, containing n vectors with d dimensions each, a similarity function, and a query $\mathbf{q} \in \mathbb{R}^d$, we seek the k vectors in $\mathbf{X}$ with maximum similarity to $\mathbf{q}$. Given the size of modern databases, guaranteeing an exact retrieval becomes challenging and this definition is relaxed to allow for a certain degree of error: some retrieved elements may not belong to the ground-truth top k. This relaxation avoids a full linear scan of the database.

Similarity is determined using a similarity function $\mathrm{sim} : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$, where a higher value indicates a higher degree of similarity. This function is symmetric, i.e., $\mathrm{sim}(\mathbf{x}, \mathbf{x}') = \mathrm{sim}(\mathbf{x}', \mathbf{x})$.

Metrics. Search accuracy is measured by k-recall@k, defined by $|S \cap G_t|/k$, where S are the ids of the k retrieved neighbors and G_t is the ground-truth. Unless otherwise specified, we use $k = 10$ in all experiments and 0.9 as the default accuracy value. Search performance is measured by queries per second (QPS).

Scalar quantization. Scalar quantization is a classical technique in signal processing mapping floating-point values to integer values within a specified range. We define the uniform scalar quantization function as

$$Q(x; B, \ell, u) = \Delta \left\lfloor \frac{x - \ell}{\Delta} + \frac{1}{2} \right\rfloor + \ell, \quad \text{where} \quad \Delta = \tfrac{u-\ell}{2^B - 1}, \qquad (1)$$

B is the number of bits used for the code, and u and ℓ are upper and lower bounds, commonly chosen as the maximum and minimum of the values to quantize.

2.1 Graph-based similarity search

Among other similarity search approaches, graph-based methods [5, 28, 39] stand out with their high accuracy and performance for high-dimensional data. They are the state of the art at billion-scale [52].

[1] https://github.com/IntelLabs/ScalableVectorSearch

[2] https://github.com/IntelLabs/DPR-dataset-generator

Algorithm 1: Greedy graph search.

```
Inputs: graph $G = (\mathcal{V}, \mathcal{E})$, query $\mathbf{q}$, number of neighbors $k \in \mathbb{N}$,
        priority queue capacity $W \geq k$, initial candidates $\mathcal{S} \subset \mathcal{V}$,
        similarity function sim
Result: $k$ approximate nearest neighbors to $\mathbf{q}$ in $G$
$Q = \mathcal{S}$ // Initialize candidate set $Q$.
// Initially, no nodes are marked as explored.
while there exists an unexplored node in $Q$ do
    $\mathbf{x}$ = closest unexplored node to $\mathbf{q}$ in $Q$ w.r.t. sim
    Mark $\mathbf{x}$ as explored
    for $\mathbf{x}' \in \mathcal{N}(\mathbf{x})$ do  $Q \leftarrow Q \cup \mathbf{x}'$
    // Limit the size of $Q$ to at most $W$:
    $Q$ = the (at most) $W$ closest nodes to $\mathbf{q}$ in $Q$ w.r.t sim
return $k$ nearest nodes to $\mathbf{q}$ in $Q$ w.r.t. sim
```

Algorithm 2: Neighborhood graph pruning [28].

```
Inputs: graph $G$, $\mathbf{x} \in \mathcal{V}$, set $C$ of out-neighbor candidates for $\mathbf{x}$,
        relaxation factor $\alpha \in \mathbb{R}^+$, out-degree bound $R \in \mathbb{N}$,
        similarity function sim
Result: The new out-neighbors $\mathcal{N}(\mathbf{x})$ of $\mathbf{x}$ in $G$ s. t. $|\mathcal{N}(\mathbf{x})| \leq R$.
$C \leftarrow (C \cup \mathcal{N}(\mathbf{x})) \setminus \{\mathbf{x}\}$ // Add the current out-neighbors
$\mathcal{N}(\mathbf{x}) \leftarrow \emptyset$ // Clear the out-neighbors of $\mathbf{x}$
while $C \neq \emptyset$ do
    $\mathbf{x}^* \leftarrow \underset{\mathbf{x}'' \in C}{\operatorname{argmax}} \operatorname{sim}(\mathbf{x}, \mathbf{x}'')$
    $\mathcal{N}(\mathbf{x}) \leftarrow \mathcal{N}(\mathbf{x}) \cup \{\mathbf{x}^*\}$
    if $|\mathcal{N}(\mathbf{x})| = R$ then break
    for $\mathbf{x}' \in C$ do
        if $\alpha \cdot \operatorname{sim}(\mathbf{x}^*, \mathbf{x}') \geq \operatorname{sim}(\mathbf{x}, \mathbf{x}')$ then  $C \leftarrow C \setminus \{\mathbf{x}'\}$
```

The key idea is that a fast search algorithm is guaranteed to converge to the nearest neighbor by a best-first traversal of the Delaunay graph. However, building a Delaunay graph is too computationally expensive and approximations are used [55]. Many variations exist, e.g., using different edge selection strategies [17, 25] or adding a hierarchy [14, 39]. In this work, we use the graph building introduced by Subramanya et al. [28] for its strong search performance, but our results apply to other graphs-based methods.

In the following discussion, let $G = (\mathcal{V}, \mathcal{E})$ be a directed graph with vertices $\mathcal{V}$ corresponding to elements in a dataset $\mathbf{X}$ and edges $\mathcal{E}$ representing neighbor-relationships between vectors. We denote with $\mathcal{N}(\mathbf{x})$ the set of out-neighbors of $\mathbf{x}$ in G.

Graph search. Graph search involves retrieving the k nearest vectors to query $\mathbf{q} \in \mathbb{R}^d$ with respect to the similarity function sim by using a modified greedy search over G (see pseudo-code in Algorithm 1). The parameter W provides a knob for trading accuracy and performance as increasing W improves the accuracy of the k nearest neighbors at the cost of lower performance by exploring more of the graph. Practical implementations of Algorithm 1 provide optimization opportunities discussed in Section 5.

Graph construction. To build the graph we follow the approach by Subramanya et al. [28]. Starting from an uninitialized graph $G = (V, \emptyset)$ and target maximum degree R, we iteratively perform an update routine for each node $\mathbf{x} \in \mathcal{V}$. For this, we first run Algorithm 1 using the node $\mathbf{x}$ as the query with $W > R$ on the current graph G to obtain C: the $k = W$ approximate nearest neighbors to $\mathbf{x}$. The pruning algorithm [28] shown in Algorithm 2 is run on C to refine the candidate list. The refined candidate list C is used to update the outward adjacency list for $\mathbf{x}$ in G. Finally, we add backward edges $(\mathbf{x}, \mathbf{x}')$ for all $\mathbf{x}'$ in $\mathbf{x}$'s updated neighborhood and prune $\mathbf{x}'$'s edges using Algorithm 2 to the maximum degree R.

Two passes are done through the dataset [28]: one with the relaxation factor $\alpha = 1.0$ and the other with a potentially different α. The optimal values for hyperparameters such as α, R, and W depend on several factors, such as the dataset manifold, its scale and the accuracy-performance trade-off of choice. Nevertheless, we find in practice that the same parameter values work very well for different datasets of similar scale (see Section 6).

3 LOCALLY-ADAPTIVE VECTOR QUANTIZATION

The IEEE 754 format [22] is designed for flexibility, allowing to represent a wide range of very small and very large numbers. However, our empirical analysis of many standard datasets and deep learning embeddings informed us of regularities in the empirical distributions of their component values. In line with modern trends in AI [19], we leverage these regularities for quantization. We explored several 8-bit floating point encodings [53], but found the precision over the small dynamic range of numeric values present in our application to be insufficient for the required search accuracy.

We further found that scalar quantization, in Equation (1), with global bounds for the whole dataset or with bounds computed individually for each dimension do not make a good use of the available bits, as demonstrated in Figure 2. In the following, *global quantization* refers to scalar quantization with global normalization.

We thus introduce Locally-adaptive Vector Quantization (LVQ) that fully utilizes the available range (Figure 2) by changing the slicing direction for computing the quantization bounds. Retaining the simplicity of scalar quantization allows for fast similarity computations while reducing the effective bandwidth.

Definition 1. *We define the Locally-adaptive Vector Quantization (LVQ-B) of vector* $\mathbf{x} = [x_1, \ldots, x_d]$ *with B bits as*

$$Q(\mathbf{x}) = [Q(x_1 - \mu_1; B, \ell, u), \ldots, Q(x_d - \mu_d; B, \ell, u)], \tag{2}$$

where the scalar quantization function Q is defined in Equation (1), $\boldsymbol{\mu} = [\mu_1, \ldots, \mu_d]$ *is the mean of all vectors in* $\mathbf{X}$ *and the constants u and* ℓ *are individually defined for each vector* $\mathbf{x} = [x_1, \ldots, x_d]$ *by*

$$u = \max_j x_j - \mu_j, \qquad \ell = \min_j x_j - \mu_j. \tag{3}$$

LVQ works with mean-centered vectors to homogenize the distributions across dimensions, see Figure 3. The quantization bounds u and ℓ are computed individually for each vector and, hence, are locally adaptive. This normalization ensures that the dynamic range is used efficiently, see Figure 2. Treating all dimensions equally could be problematic in the presence of large variance differences across vector dimensions. Although this scenario is not observed in practice (Figure 3 and Figure 14 in the supplementary material [1]), either in standard datasets or in deep learning embeddings, we empirically show the robustness of LVQ in Appendix A.1 [1].

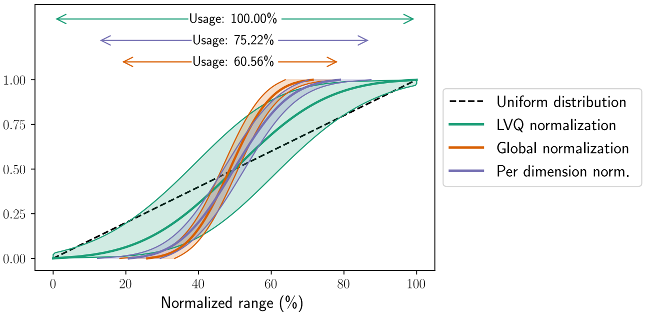


Figure 2: Empirical distributions of the values in each vector for deep-96-1M (we show the mean across vectors ±2σ). For 95% of the vectors, global and per dimension normalization only utilize around 60% and 75% of the available range, respectively. LVQ normalization approximates the uniform distribution better, utilizing the whole range and yielding a more faithful encoding.

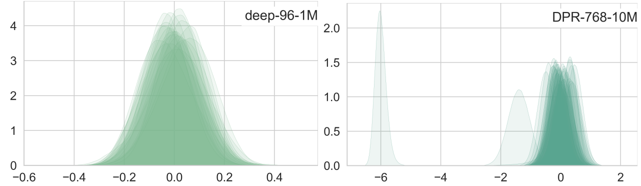


Figure 3: Empirical distributions of vector values in individual dimensions for two prototypical datasets. After demeaning, the values become highly amenable to quantization, as the distributions will not contain regions within the dynamic range with either very high or very small density. Additional datasets are included in Figure 14 in the supplementary material [1].

For each d-dimensional vector compressed with LVQ-B, we need to store the quantized values and the constants u and ℓ. Moreover, to improve search performance, LVQ-compressed vectors can be padded to a multiple of $p = 32$ bytes to be aligned with half cache lines. The footprint in bytes of a vector compressed with LVQ-B is

$$\text{footprint}(Q(\mathbf{x})) = \lceil (d \cdot B + 2B_{const})/8/p \rceil \cdot p, \tag{4}$$

where B_{const} is the number of bits used for u and for ℓ. Typically, we encode them in float16, in which case $B_{const} = 16$.

The compression ratio CR for LVQ is given by

$$\text{CR}(Q(\mathbf{x})) = d \cdot B_{orig} / (8 \cdot \text{footprint}(Q(\mathbf{x}))), \tag{5}$$

where B_{orig} is the number of bits per each dimension of $\mathbf{x}$. Typically, vectors are encoded in float32, thus $B_{orig} = 32$. For example, when using $B = 8$ bits and no padding ($p = 0$), the compression ratio for deep-96-1B ($d = 96$) is 3.84 and 3.98 for DPR-768-10M ($d = 768$).

3.1 Two-level quantization

In graph search, most of the search time is spent (1) performing random memory accesses to retrieve the vectors associated with the out-neighbors of each node and (2) computing the similarity between the query and each vector. After optimizing the compute using AVX instructions, search is heavily dominated by the memory access time. This is exacerbated as the number d of dimensions increases (d is in the upper hundreds for deep learning embeddings).

To reduce the effective memory bandwidth during search, we compress each vector in two levels, each with a fraction of the available bits. After using LVQ for the first level, we quantize the residual vector $\mathbf{r} = \mathbf{x} - \boldsymbol{\mu} - Q(\mathbf{x})$. The scalar random variable $Z = X - \boldsymbol{\mu} - Q(X)$, which models the first-level quantization error, follows a uniform distribution in $[-\Delta/2, \Delta/2)$ (see Equation (1)). Thus, we encode each component of $\mathbf{r}$ using the scalar quantization function

$$Q_{\text{res}}(r; B') = Q(x; B', -\Delta/2, \Delta/2), \tag{6}$$

where B' is the number of bits used for the residual code.

Definition 2. *We define the two-level Locally-adaptive Vector Quantization (LVQ-B_1xB_2) of vector $\mathbf{x}$ as a pair of vectors $Q(\mathbf{x}), Q_{res}(\mathbf{r})$, such that*

- *$Q(\mathbf{x})$ is the vector $\mathbf{x}$ compressed with LVQ-B_1,*
- *$Q_{res}(\mathbf{r}) = [Q_{res}(r_1; B_2), \ldots, Q_{res}(r_d; B_2)]$,*

where $\mathbf{r} = \mathbf{x} - \boldsymbol{\mu} - Q(\mathbf{x})$ and Q_{res} is defined in Equation (6).

No additional constants are needed for the second-level, as they can be deduced from the first-level ones. Given the first-level function in Equation (4), the memory footprint of LVQ-B_1xB_2 is

$$\text{footprint}(Q(\mathbf{x}), Q_{\text{res}}(\mathbf{r})) = \text{footprint}(Q(\mathbf{x})) + d \cdot B_2. \tag{7}$$

3.2 Integrating LVQ into graph-based indices

We use first-level LVQ to search the graph. This improves the search performance by decreasing the effective bandwidth, determined by the number B_1 of bits transmitted from memory for each vector. The reduced number of bits might generate a loss in accuracy. When present, the second level, or compressed residuals, is used for a final re-ranking step, recovering part of the accuracy lost in the first level. Here, we replace Line 6 of Algorithm 1 by a gather operation, that fetches $Q_{\text{res}}(\mathbf{r})$ for each vector $Q(\mathbf{x})$ in $\mathcal{Q}$, recomputes the similarity between the query $\mathbf{q}$ and each $Q(\mathbf{x}) + Q_{\text{res}}(\mathbf{r})$, and finally selects the top-k. Moreover, we can safely build the graph from vectors compressed using LVQ, as we show in the next section.

Adapting to shifts in the data distribution In the case of dynamic indices (supporting insertions, deletions and updates), a compression method should easily adapt to data distribution shifts. Search accuracy can highly degrade over time if the compression model and the index are not periodically updated. Such an update often involves running expensive algorithms (e.g., PQ [33] involves running multiple instances of k-means). For LVQ, the model update is simpler, requiring recomputation of the dataset mean $\boldsymbol{\mu}$ and reencoding of the data vectors, operations that scale linearly with n, and do not require loading the full dataset in memory.

4 THEORETICAL RESULTS ON GRAPH CONSTRUCTION

This section is devoted to showing that we can build a graph with LVQ-compressed vectors without impacting search accuracy, thus accelerating and reducing the footprint of the expensive index construction step. For example, for deep-96-1B (Section 6.1), graph building requires at least 835GiB for a maximum out degree $R = 128$.

Table 1: Memory requirements (graph + vectors) for graph building with full-precision (FP) and with LVQ-B vectors. Depending on the dataset and the graph maximum out-bound degree (R=32,64,128), the memory reduction can reach 6.2x.

	deep-96-1B			text2Image-200-100M			DPR-768-10M		
	Size (GiB)		Ratio	Size (GiB)		Ratio	Size (GiB)		Ratio
R	FP	LVQ-4		FP	LVQ-4		FP	LVQ-4	
32	477	168	2.84	864	216	4.00	298	48	6.20
64	596	287	2.08	983	335	2.93	310	60	5.17
128	834	525	1.59	1222	574	2.13	334	84	3.98

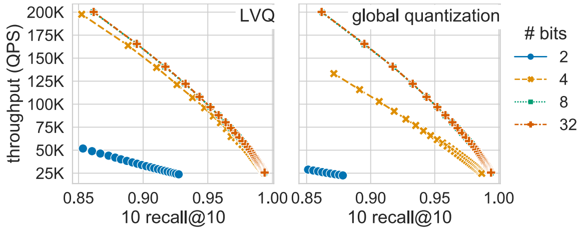


Figure 4: Search performance (conducted with float32-valued vectors to normalize for compute differences) on graphs built with vectors compressed using LVQ (left) and global quantization (right) with different number of bits for deep-96-100M. We observe almost no decrease in throughput in graphs built with 4 or more bits with LVQ (the curves with 8 and 32 bits overlap). In contrast, we observe a sharp drop in throughput for graphs built using global quantization with 4 bits. Thus, we can save memory by building graphs with LVQ-4 or LVQ-8 without affecting the search quality.

Notably, when graphs are built with LVQ-compressed vectors, the search accuracy is almost unchanged even when setting B as low as 8 or 4 bits (see Figure 4). The minimum memory requirements (graph + dataset size) in GiB to construct a graph from full precision and from LVQ with $B = 4$ bits are reported in Table 1, where the memory reduction can be as high as 6.2x. In this section we explain why LVQ makes these savings possible, corroborating that this is indeed logical and even expected.

There are three steps required to generate the adjacency list of each vertex in the graph: building, sorting and pruning the neighbors candidate list. We will begin with the latter, as it is the critical step to make the graph searchable.

4.1 Graph pruning with LVQ

We will now characterize both theoretically and experimentally the errors introduced in the graph-pruning step when building the graph from vectors compressed with LVQ. These errors are mild and LVQ is fully compatible with the graph-pruning rule in Line 8 of Algorithm 2.

Let us consider C the set of candidates for $\mathbf{x}$'s adjacency list in G. The pruning process iterates through C and, at each step, adds to the set of out-neighbors of $\mathbf{x}$ its most similar vector $\mathbf{x}^*$, removing from C all the vector that are closer to $\mathbf{x}^*$ than to $\mathbf{x}$ (Algorithm 2).

When the similarity is Euclidean distance, i.e., $\mathrm{sim}(\mathbf{x}, \mathbf{x}') = -\|\mathbf{x} - \mathbf{x}'\|_2$, the pruning rule in Line 8 of Algorithm 2 becomes

$$\alpha \left\|\mathbf{x}^* - \mathbf{x}'\right\|_2 \leq \left\|\mathbf{x} - \mathbf{x}'\right\|_2 . \tag{8}$$

Geometrically, as shown in Figure 5 (left), this is equivalent to determining the perpendicular bisector hyperplane for $\mathbf{x}$ and $\mathbf{x}^*$, and eliminating from C all vectors $\mathbf{x}'$ that lie on same half-space as $\mathbf{x}^*$. The pruning can be performed by computing

$$\mathrm{sign}(\mathbf{a}^\top \mathbf{x}' - b), \quad \text{with} \quad \mathbf{a} = \frac{\mathbf{x} - \mathbf{x}^*}{\|\mathbf{x} - \mathbf{x}^*\|_2}, \quad b = \frac{\|\mathbf{x}\|_2^2 - \|\mathbf{x}^*\|_2^2}{2\|\mathbf{x} - \mathbf{x}^*\|_2}. \tag{9}$$

and eliminating those vectors $\mathbf{x}'$ for which $\mathrm{sign}(a\mathbf{x}' - b) = -1$.

Proposition 1. *When using Euclidean distance as the similarity function and $\alpha = 1$, the graph pruning rule for full-precision vectors, Equation* (8)*, and the one using vectors compressed with LVQ, i.e.,*

$$\left\|Q(\mathbf{x}^*) - Q(\mathbf{x}')\right\|_2 \leq \left\|Q(\mathbf{x}) - Q(\mathbf{x}')\right\|_2 , \tag{10}$$

are equivalent (in the sense of simultaneously holding) when

$$|\mathbf{a}^\top \mathbf{x}' - b| \cdot \|\mathbf{x} - \mathbf{x}^*\| \geq |E|, \tag{11}$$

where $\mathbf{a}^\top$ and b are defined in Equation (9) *and $E \in \mathbb{R}$ is an error that depends on the quantization error and the vectors $\mathbf{x}$, $\mathbf{x}^*$, and $\mathbf{x}'$.*

The proof is in Appendix B [1].

Classical signal processing theory dictates that under normal conditions the error introduced by a scalar quantization follows a uniform distribution. This uniformity is inherited by the quantization error in LVQ (see Figure 16 in the supplementary material [1]).

Proposition 2. *Under the conditions in Proposition 1 and assuming a uniformly distributed quantization error, the error E is a normally distributed random variable with mean μ_E and variance σ_E^2, given by*

$$\mu_E = \frac{d}{24}(\Delta_x^2 - \Delta_{x^*}^2), \tag{12}$$

$$\sigma_E^2 = \frac{\Delta_{\mathbf{x}}^2}{12}\|\mathbf{x}' - \mathbf{x}\|^2 + \frac{\Delta_{\mathbf{x}^*}^2}{12}\|\mathbf{x}' - \mathbf{x}^*\|^2 + \frac{\Delta_{\mathbf{x}'}^2}{12}\|\mathbf{x} - \mathbf{x}^*\|^2 + \frac{d(\Delta_{\mathbf{x}}^4 + \Delta_{\mathbf{x}^*}^4)}{720} + \frac{d\Delta_{\mathbf{x}'}^2(\Delta_{\mathbf{x}}^2 + \Delta_{\mathbf{x}^*}^2)}{144}. \tag{13}$$

where $\Delta_{\mathbf{x}}$, $\Delta_{\mathbf{x}}^*$, and $\Delta_{\mathbf{x}'}$ are the quantization steps for $\mathbf{x}$, $\mathbf{x}^*$, and $\mathbf{x}'$, respectively, given by Equation (1)*.*

The proof is in Appendix B [1].

Corollary 1. *Let Φ be the normal cumulative distribution function. $|E|$ follows a folded normal distribution parameterized by*

$$\mu_{|E|} = \sigma_E \sqrt{\frac{2}{\pi}} \exp\left(-\mu_E^2/2\sigma_E^2\right) + \mu_E(1 - 2\Phi(-\tfrac{\mu_E}{\sigma_E})), \tag{14}$$

$$\sigma_{|E|}^2 = \mu_E^2 + \sigma_E^2 - \mu_{|E|}^2 . \tag{15}$$

With these theoretical results in hand, we are now ready to characterize Proposition 1 empirically, i.e., the number of bits B (see Definition 1) needed to run the pruning algorithm with minimal errors. We generate triplets of vectors $\mathbf{x}$, $\mathbf{x}^*$ and $\mathbf{x}'$ that may be found during pruning from 1 million vectors taken from the deep-96-100M dataset presented in Table 2. For this, we select a vector $\mathbf{x}$ at random, find its (ground-truth) T nearest neighbors, and among those we first randomly sample $\mathbf{x}^*$, and then $\mathbf{x}'$ from those that are

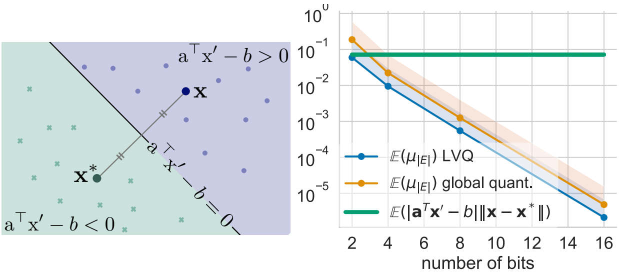


Figure 5: (Left) For $\alpha = 1$, the graph pruning rule (see Algorithm 2) can be interpreted as determining the perpendicular bisector hyperplane for x and x*, and eliminating the candidates (green crosses) that lie on the same (green) half-space as x*. This amounts to eliminating the vectors with sign$(\mathrm{a}^\top\mathbf{x}' - b) = -1$, see Equation (9). (Right) Proposition 1 predicts that, under the green line, two graphs, one built with compressed vectors and one with full-precision, are equally accurate. We compare LVQ (in blue) and global quantization (in orange) with a varying number of bits (expectations computed across 10^6 samples and error bands at $+3\sigma_{|E|}$ for deep-96-100M). Empirically, graphs pruned using vectors compressed with LVQ-4 or LVQ-8 are well within the safe zone, while the errors bands for 4-bit global quantization are close to the threshold. This correlates with the search accuracy in Figure 4 that is only slightly lower for LVQ-4 but severely degraded for 4-bit global quantization.

farther from $\mathbf{x}$ than $\mathbf{x}^*$. The results in Figure 5 (right) show that we can reduce B in LVQ safely to 4 bits without affecting the pruning rule. With 2 bits, the error bars overlap and no guaranties can be given. These results are in agreement with the search accuracy we observe in Figure 4.

4.2 Candidates selection and sorting with LVQ

The candidates list consists of the T closest vectors among those visited in a search. If the compression error is small enough compared to the distance between $\mathbf{x}$ and its T-th nearest neighbor, we could expect similar candidates lists when using compressed or full-precision vectors. To evaluate this, we analyze the T nearest neighbors of an exhaustive search with compressed vectors.

In Figure 6 (left), we present average results for 10^5 vectors chosen at random from the dataset deep-96-100M ($T = 750$). For LVQ with 8 and 16 bits, the recall is almost one. This suggests that there should be no difference between the candidates lists using full-precision or LVQ. At 4 bits, the recall for LVQ is 0.82, suggesting a high degree of agreement between the compressed and uncompressed lists of candidates. For global quantization, we observe a degraded recall of 0.6, pointing to a loss of equivalence between the candidate lists. These results are in agreement with what we observe in Figure 4. The search throughput for graphs built with LVQ using 4 bits is almost unchanged from the baseline (32 bits). For graphs built with global vector quantization using 4 bits, the search throughput suffers significantly.

To assess how LVQ affects the ordering of the candidate list elements, we use Ranked Bias Overlap (RBO) [57], a standard metric

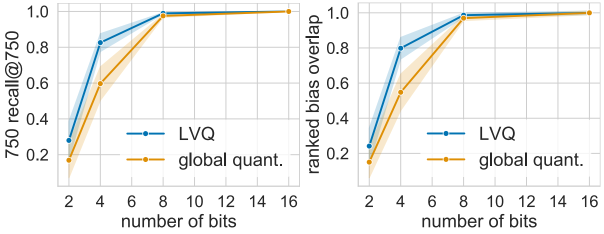


Figure 6: With four or more bits per value, LVQ does not introduce artifacts in the computation and sorting of the candidate lists (of length 750) used for graph pruning (bands at ± one standard deviation). To check the presence of the correct elements, we track 750-recall@750, confirming that LVQ stays above 0.8 for four bits, whereas the global quantization drops to 0.6. To check the order of the elements, we use Ranked Bias Overlap (RBO) [57], obtaining similar numbers.

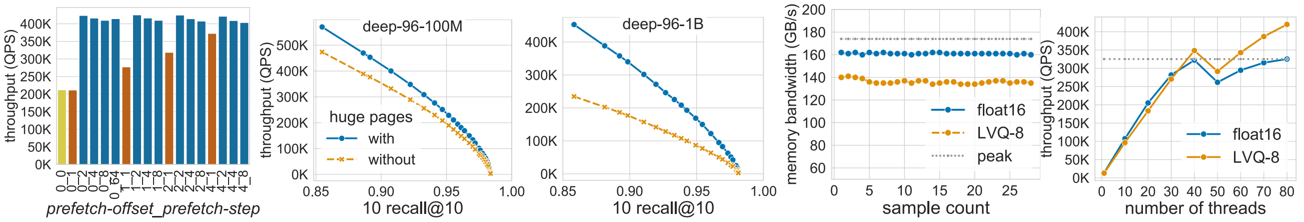
to compare ranked lists. First, we find the two candidate lists, i.e., for LVQ-compressed and full-precision vectors, using exhaustive search. Next, we compute the RBO between the lists sorted according to the similarity between LVQ-compressed and full-precision vectors. Figure 6 shows the results for 10^5 vectors chosen at random in the dataset deep-96-100M. Again, for LVQ with 4, 8, and 16 bits, we observe a high-quality sorting. For global quantization at 4 bits, the sorting gets affected.

5 IMPLEMENTATION CHALLENGES

As with most high-performance algorithms, a fine-tuned implementation is fundamental to realize the full potential of graph-based similarity search. We now describe a set of optimizations that are geared towards putting the system in its *natural* memory bottlenecked regime and improving its performance. To illustrate the discussion with experimental results and ablation studies, we use the deep-96-100M dataset (see Table 2 in Section 6.1) and use the system described in Section 6.2.

Efficient similarity calculations using LVQ with AVX. Computing the similarity between two vectors is a key kernel underpinning similarity search. SIMD vector instructions can be used to efficiently implement distance computations for LVQ-B and LVQ-B_1xB_2. We store compressed vectors as densely packed integers with scaling constants stored inline. When 8-bits are used, native AVX instructions are used to load and convert the individual components into floating-point values which are combined with the scaling constants. The case $B_1 = B_2 = 4$ in LVQ-B_1xB_2 requires a little more work, involving vectorized integer shifts and masking. We fuse the decompression with the distance computation against the query vector. This fusion, combined with loop unrolling and masked operations to tail elements, creates an efficient distance computation implementation that makes no function calls, decompresses the quantized vectors on-the-fly and accumulates partial results in AVX registers.

A notable optimization is the ability to set the dimensionality at compile time (static) versus at runtime (dynamic). As the dimensionality of a dataset is fixed once and for all, setting it statically presents no detrimental aspects and it improves the compiler's ability to



(a) Impact of prefetching

(b) Effect of huge pages

(c) Bandwidth utilization

(d) Core scaling

Figure 7: (a) Our advanced prefetching, parameterized by *prefetch-offset* and *prefetch-step*, provides up to 2x performance gain over no-prefetch (yellow bar). Commonly used prefetch schemes using *prefetch-step*=1 (red bars) show sub-optimal gains. (b) By explicitly utilizing huge pages, we achieve significant performance gains as datasets grow: 20% in deep-96-100M and 90% in deep-96-1B. (c) We reach 90% and 78% of the read-only peak bandwidth for float16 and LVQ-8 data types, respectively. (d) We achieve good performance scaling with the number of threads, obtaining 23.5x and 33x gains over single-thread for float16 and LVQ-8 vectors, respectively. For float16 vectors, performance tops at 40 threads (on a system with 40 cores). Thanks to the reduced bandwidth, LVQ-8 performance continues to grow up to the maximum number (80) of hyperthreaded cores. Unless specified, experiments are done for deep-96-100M with a 10-recall@10 of 0.9.

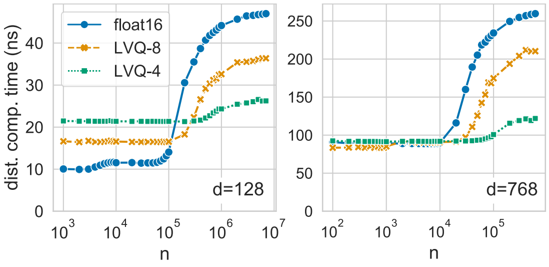


Figure 8: Mean similarity computation times using different encodings for n vectors in d dimensions ($d = 128, 768$ on the left and right). We use a sequential access pattern to minimize any memory side effects. In each case, the curve inflexion marks the point where the vectors do not fit in the L2 cache. When vectors fit in cache, distances with float16 values are 2x faster than those with LVQ-4. When vectors exceed the cache size, LVQ quickly becomes 2.12x faster than float16 due to the reduced number of fetched cache lines.

unroll loops in the similarity function kernel more extensively. We observe up to a 32% performance speedup when using static versus dynamic dimensionality. All in all, this implementation achieves ~2x faster computation performance, as shown in Figure 8.

Advanced prefetching. Prefetching involves proactively moving data that will be accessed soon into the CPU cache. This can either be done automatically in hardware, or manually through software instructions. If done well, it can improve application throughput by lowering memory latency when data is finally accessed and by overlapping computation with memory access. Due to the random data access pattern in graph-based search, hardware prefetchers are ineffective. This leaves software prefetching.

When computing distances between a query and all neighbors of a vertex, we introduce two tunable parameters for prefetching data vectors: *prefetch-offset* that controls the lookahead offset in the list of vectors to prefetch, and *prefetch-step* that sets the number of vectors to prefetch in each iteration.

In Figure 7(a), we observe that our scheme provides up to a 2x speedup over the no-prefetch case (baseline) where both parameters are zero. A simple prefetch strategy (*prefetch-step*=1) which simply prefetches the immediate next vector shows no performance gain over the baseline. However increasing *prefetch-offset* boosts performance (red bars). Furthermore, schemes with *prefetch-step* > 1 show similar performance. Although there is not a universally best combination, we found that *prefetch-offset*=0 and *prefetch-step*=2 works well in many cases. Last, to account for cache-unaligned dimensionality d, we selectively bring additional cache lines with no negative impact on the performance for aligned d.

Optimizing graph search. We now analyze opportunities for optimizing the overall implementation for Algorithm 1. We use a sorted linear buffer to implement the queue Q, storing whether a node has been explored inline with the node id and distance from the query. For values of W common in our datasets (a few dozens), we found this to be faster than using a heap-like data structure due to its hardware friendliness. Common implementations of graph search use an associative data structure to track whether a node has been visited to avoid unnecessary memory accesses and compute. With our optimized similarity computations and smart prefetcher in hand, keeping a visited set carries a performance regression.

The overall advantage of disabling the visited set depends both on the dimensionality d and the CPU micro-architecture. For example, with deep-96-100M on a Cascade Lake server CPU, disabling the visited set can improve throughput by 15-20% whereas on an Ice Lake server, the improvement is capped at 2-3%. We observe no performance difference for the DPR-768-10M dataset (Table 2) with its larger dimensionality ($d = 768$).

Finally, we parallelize the search across queries with each thread being responsible for a subset of the query batch, running a single-thread search routine for each query in the batch. This is a common pattern for graph-based similarity search implementations. Parallelizing the search for an individual query [43], and understanding whether it presents actual benefits, remains as future work.

Memory layout and allocation. Modern computer systems use virtual memory [26] to provide process isolation and address space independence. Virtual memory addresses used by programs are translated to physical addresses through the use of page tables, a process that is accelerated in hardware using a Translation Lookaside Buffer (TLB) to cache recently used translations.

For billion-scale datasets, a TLB miss for each random access in Algorithm 1 is nearly certain when using typical 4096 kB pages. Because the probability of the corresponding page table entry being resident in cache is nearly zero, this miss requires another access to main memory, which degrades performance. Using 2 MB or 1 GB huge pages greatly increases the probability that the missed page-table entry is in the cache [47]. To that end, we avoid graph layouts that involve memory indirections (such as CSR or a list of lists), which further decrease the cache hit rate.

Consequently, we use large contiguous block allocations and implement explicit huge-page allocators. Figure 7(b) demonstrates a 20% and 90% performance gain by using huge pages at 0.90 recall in deep-96-100M and deep-96-1B, respectively.

System utilization. Our algorithm and its implementation achieve high system bandwidth and scale well with the number of cores. Our system's peak read-only memory bandwidth, measured with Intel®'s Memory Latency Checker [53], is 174GB/s per socket. As shown in Figure 7(c), the average memory bandwidth achieved by our implementation during the search is 160GB/s and 135GB/s utilizing more than 90% and 78% of the peak bandwidth for float16-valued and LVQ-8 vectors, respectively. This high bandwidth utilization is the product of the system optimizations described above.

Next, we present the performance scaling with the number of hardware threads used. Our system has 40 cores per socket with 2-way hyper-threading enabled (maximum 80 threads per socket). Figure 7(d) shows that the QPS increases up to 40 threads in both float16 and LVQ-8 data types. In float16, however, the performance scaling slows down between 30 and 40 threads and saturates at 40 threads as the required memory bandwidth approaches the system peak. On the other hand, LVQ-8 reduces the memory bandwidth pressure and scales beyond 40 threads reaching its maximum value at 80 threads, fully utilizing the hyper-threading capabilities. As a result, our technique gains 23.5x and 33x performance over single-thread in float16 and LVQ-8, respectively. In both data types, the performance drops immediately after 40 threads due to hyperthreading. As a result, the threads sharing a core are slower and cap the overall search performance in batch mode. However, increasing the thread usage further leads to higher throughput outweighing the individual thread latency.

6 EXPERIMENTAL RESULTS

In this section, we present different empirical results clarifying the following topics. We first discuss an exhaustive evaluation showing that our optimized graph-based search and LVQ, subsequently denoted as OG-LVQ, establishes a new SOTA for both small and large scale datasets. We also show that this performance benefits also come with memory savings. Then, we address the benefits of LVQ over the standard Product Quantization. Finally, we present an ablation study, where we compare different quantizers under our optimized graph-based search. In all cases, LVQ-compressed vectors are padded to half cache lines ($p = 32$, see Section 3). We report the best out of 5 runs for each method [6].

Table 2: Evaluated datasets, where n (n_q) represent the number of vectors (queries) and d their dimensionality. The space is measured in GiB. We generated DPR-768-10M from [34, 46] (reproducibility details in Section C of the supp. material).

	Dataset	d	n	Encoding	Similarity	n_q	Space
Small scale	gist-960-1M [29]	960	1M	float32	L2	10^3	3.6
	sift-128-1M [29]	128	1M	float32	L2	10^4	0.5
	deep-96-10M [10]	96	10M	float32	cos sim.	10^4	3.6
	deep-96-1M [10]	96	1M	float32	cos sim.	10^4	0.4
	glove-50-1.2M [44]	50	1.2M	float32	cos sim.	10^4	0.2
	glove-25-1.2M [44]	25	1.2M	float32	cos sim.	10^4	0.1
Large scale	DPR-768-10M	768	10M	float32	inner prod.	10^4	28.6
	t2i-200-100M [9]	200	100M	float32	inner prod.	10^4	74.5
	deep-96-100M [10]	96	100M	float32	cos sim.	10^4	35.8
	deep-96-1B [10]	96	1B	float32	cos sim.	10^4	257.6

6.1 Datasets

To cover a wide range of use cases, we evaluate our method on standard datasets of diverse dimensionalities ($d = 25$ to $d = 768$), number of elements ($n = 10^6$ to $n = 10^9$), data types and metrics (see Table 2). In addition, we introduce a new dataset containing 10 million 768-dimensional embeddings generated with the dense passage retriever (DPR) [34] model. This dataset allows us to benchmark our method in a very high-dimensional setting, that has become ubiquitous in retrieval enhanced deep learning and most tasks that make use of large language models. We use text snippets from the C4 dataset [46] to generate: 10 million context DPR embeddings (base set) and 10000 question DPR embeddings (query set). We refer the reader to Appendix C [1] for reproducibility details about this dataset. For deep-96-100M and deep-96-1B, as the vectors have norm one, we compute the cosine similarity using Euclidean distance.

6.2 System setup

We run our experiments on two 2-socket systems. Those in Section 6.3 run on 3rd generation Intel® Xeon® 8360Y @2.40GHz CPUs with 36 cores and 256GB DDR4 memory (@2933MT/s) per socket. All other experiments run on 3rd generation Intel® Xeon® Platinum 8380 @2.30GHz CPUs with 40 cores and 1TB DDR4 memory (@3200MT/s) per socket. Both systems run Ubuntu 22.04.[3]

We ran all experiments in a single socket to avoid introducing performance regressions due to remote NUMA memory accesses.

We use the *hugeadm* Linux utility to preallocate a sufficient number of 1GB huge pages for each algorithm. Our implementation uses huge pages natively to reduce virtual memory overheads (see Section 5). For a fair comparison, we run other methods with system flags enabled to automatically use huge pages for large allocations.

[3] Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. No product or component can be absolutely secure. Your costs and results may vary. Intel technologies may require enabled hardware, software or service activation. ©Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

6.3 Performance on small scale search

We adopt the standard ANN-benchmarks [6] protocol and consider small scale datasets of diverse dimensionality (d=25, 50, 96, 128, 960) and number of vectors ($n = 10^6, \ldots, 10^7$). See details in Table 2. We compare OG-LVQ to the SOTA algorithms for each dataset[4], as well as to other widely adopted approaches, for two query batch sizes: one query at a time (single query mode) and full batch (total number of queries in the dataset). The evaluated algorithms are: Vamana [28], HNSWlib [39], FAISS-IVFPQfs [31]), NGT-qg [24] and ScaNN [21]. NGT-qg is not included in the query batch mode evaluation because the available implementation did not support multi-query processing. Following ANN-benchmarks, we generate Pareto curves of QPS vs. recall for a series of parameter settings. For the graph-based methods (Vamana, HNSWlib and OG-LVQ), we build graphs with $R = 32, 64, 128$[5]. For IVFPQfs, ScaNN and NGT-qg we use the provided configuration settings [7]. For OG-LVQ, we include various LVQ settings (LVQ-8, LVQ-4x4, LVQ-4x8, and LVQ8x8). As explained in Section 6.2, our implementation explicitly uses huge pages when available. However, system flags to automatically use huge pages did not work in the ANN-benchmarks [6] Docker image. Therefore, huge pages were not allocated during these experiments to ensure equal conditions for all methods. When using cosine similarity, we follow the standard approach of normalizing the vectors and running the search using Euclidean distance.

In Table 3, we observe that OG-LVQ outmatches the competition in full query batch mode, where it supersedes its closest competitor by 1.15x to 5.49x across all datasets with their unique sizes and dimensionalities. In single query mode, OG-LVQ still wins in 3 out 5 cases and NGT-qg takes the other two. The performance gains of OG-LVQ are consistent across the entire recall range, as shown in Figure 9 for the deep-96-10M and glove-50-1.2M datasets (all datasets are included in Figure 17 in the supplementary material [1]). Figures 18 and 19 in the supplementary material [1] show similar results for 50 recall@50 and 100 recall@100, respectively.

6.4 Performance on large scale search

We adopt the big-ann-benchmarks [52] framework to run our large-scale studies in full batch query mode. For this study, we consider the datasets in Table 2 with a large footprint. We compare OG-LVQ to four widely adopted approaches: Vamana [28], HNSWlib [39], FAISS-IVFPQfs [31], and ScaNN [21]. We use the following parameter setting to build Vamana graphs for all the datasets: $R = 128$ (we use $R = 126$ for deep-96-1B), $\alpha = 1.2$ and $\alpha = 0.95$ for L2 distance and inner product, respectively. For OG-LVQ, we include various LVQ settings (LVQ-8, LVQ-4x4, LVQ-4x8, and LVQ8x8). For HNSWlib, we build all graphs with a window search size of 200 and $R = 128$[6], except deep-96-1B for which we must reduce R to 96 to fit the working set size in 1TB memory. For FAISS-IVFPQfs, as the build-time is long for deep-96-1B, we pre-build an index with nlist = 32768 and bins = $d/2$. While for t2i-200-100M and DPR-768-10M, indices are built on the fly with combinations of nlist = $\{512, 1024, 4096, 8192\}$ and nbins = $\{d/4, d/2, d\}$. To achieve higher recall rates, we enable re-ranking in FAISS-IVFPQfs and sweep nprobe = $\{1, 5, 10, 50, 100, 200\}$ and k for re-ranking = $\{0, 10, 100, 1000\}$, at runtime. For ScaNN, we use the parameters setting recommended by the authors (n_leaves = $\sqrt{n}$, avq_threshold = 0.2, dims_per_block = 2), as that was the best among several evaluated settings, and vary the runtime parameters (leaves_to_search = $2 - 1000$, reorder = $20 - 1000$). Finally, we did not include NGT [24] in the evaluation as the algorithm designed for large-scale datasets (NGT-QBG) achieves low accuracy saturating at 0.86 recall even for a small 1-million vectors dataset.

Figure 10 shows OG-LVQ's significant performance advantage across recall values for deep-96-1B, with a 6.5x higher throughput over the closest competitor for 10-recall@10 of 0.9. For the datasets that use inner product, the advantage is still present for recall values below 0.97 for t2i-200-100M and 0.95 for DPR-768-10M. There, OG-LVQ achieves 2.0x and 1.8x higher throughput than the closest competitor for 10 recall@10 of 0.9 on t2i-200-100M and DPR-768-10M, respectively. In this case, the OG-LVQ performance for very high recall values is on par with the alternatives. This phenomenon is not due to the quantization error, as it is also present in the graph search with full-precision vectors (not shown in the figure). Understanding this phenomenon will require further studies. Similar results are observed for 50 recall@50 and 100 recall@100 in Figure 20 in the supplementary material [1].

6.5 LVQ: Fast graph search with small footprint

We now show that combining a highly optimized graph-based method (see Section 5) with LVQ provides high search performance with a fraction of the memory. Figure 1 shows the search throughput as a function of the memory footprint (measured as the maximum resident main memory usage during search) of different algorithms for deep-96-1B at 0.9 10-recall@10 (similar results are shown in Figure 21(a) in the supplementary material [1] for deep-96-100M). In the case of the graph-based methods (OG-LVQ, Vamana, HNSWlib), the memory footprint increases with the graph size given by the maximum number of outbound neighbors $R = \{32, 64, 128\}$. In the case of FAISS-IVFPQfs, the memory footprint remains almost constant for all combinations of the considered parameters (nlist={4096, 8192,16384}, nbins={48,96}, nprobe={1,5,10,50,100,200}, k={0,10,100,1000} for re-ranking), increasing by only 7% (from 42GiB to 45GiB for deep-96-100M) when using 48 or 96 PQ segments (nbins), respectively. A similar behavior is observed for ScaNN, as it uses the same index design. LVQ-compressed vectors are padded to half cache lines (p = 32, see Section 3 for details), as it improves performance and has a low impact on the overall memory footprint (e.g., 5% larger footprint for deep-96-1B with R=128).

These results show that OG-LVQ can use a much smaller graph (R = 32) and still outperform its competitors: (A) for deep-96-1B by 2.3x, 2.2x, 20.7x, and 43.6x in throughput with 3x, 3.3x, 1.7x, and 1.8x less memory (Figure 1), and (B) for deep-96-100M by 3.2x, 2.7x, 7.4x, and 11.5x in throughput with 3.1x, 3.3x, 1.8x and 1.9x less memory (Figure 21(a) in the supplementary material [1]), with respect to Vamana, HNSWlib, FAISS-IVFPQfs, and ScaNN, respectively. OG-LVQ's superiority in QPS and memory footprint is consistent across all recall values (see Figure 21(b) in the supplementary material [1]). OG-LVQ, with a memory footprint of 24GiB (LVQ-8 and R = 32),

[4] We consider the SOTA results for single query mode, as those are the ones reported by ANN-benchmarks [7]. Last accessed on Feb. 15 2023.

[5] This corresponds to $M = 16, 32, 64$ in HNSW parameter notation.

[6] This corresponds to $M = 64$ in HNSW parameter notation.

Table 3: The proposed OG-LVQ shows significant gains in small scale datasets at 0.90 10-recall@10, clearly winning 8 out the 10 tested cases (the best throughput in each case is shaded). Note that the alternative with the second-highest throughput is not consistently the same, showing the versatility of OG-LVQ. For the other schemes, we include their raw throughput and the ratio between ours and theirs. The geometric mean across all datasets highlights the overall superiority of OG-LVQ.

	Full query batch									Single query										
	OG-LVQ	Vamana		HNSWlib		FAISS-IVFPQfs		ScaNN		OG-LVQ	Vamana		HNSWlib		FAISS-IVFPQfs		ScaNN		NGT-qg	
	QPS	QPS	Ratio	QPS	Ratio	QPS	Ratio	QPS	Ratio	QPS	QPS	Ratio	QPS	Ratio	QPS	Ratio	QPS	Ratio	QPS	Ratio
deep-96-10M	648 122	75 543	8.58	118 053	5.49	89 550	7.24	95 266	6.80	18 440	6643	2.78	6347	2.91	1895	9.73	3881	4.75	9600	1.92
gist-960-1M	56 717	14 586	3.89	11 313	5.01	34 640	1.64	16 588	3.42	1857	992	1.87	725	2.56	827	2.24	883	2.10	4420	0.42
glove-25-1.2M	1 224 266	1 062 091	1.15	418 850	2.92	703 996	1.74	443 237	2.76	34 118	29 382	1.16	13 304	2.56	7993	4.27	28 230	1.21	25 902	1.32
glove-50-1.2M	558 606	246 095	2.27	117 585	4.75	148 793	3.75	313 337	1.78	17 268	9151	1.89	3237	5.33	3564	4.85	12 582	1.37	15 676	1.10
sift-128-1M	852 705	323 014	2.64	200 634	4.25	464 048	1.84	189 196	4.51	21 969	13 807	1.59	9100	2.41	6852	3.21	4856	4.52	23 117	0.95
Geometric mean			2.97		4.38		2.70		3.42			1.78		3.00		4.29		2.37		1.02

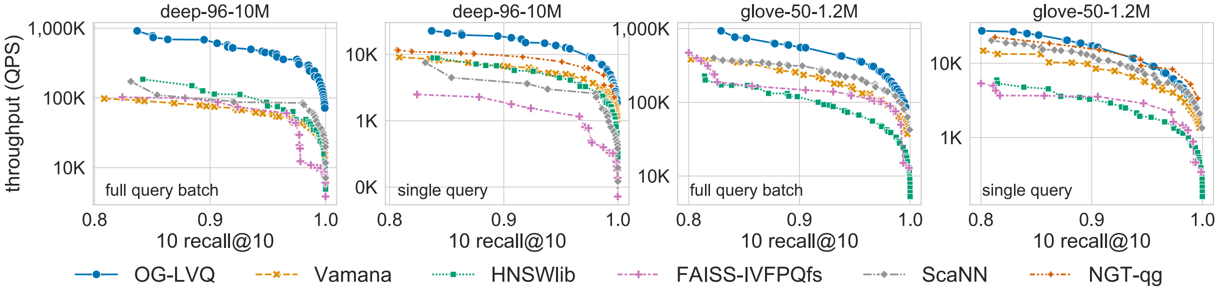


Figure 9: Benchmarking results for small scale datasets (deep-96-10M and glove-50-1.2M). In batch mode (first and third plots), OG-LVQ prevails by a large margin. In single-query mode (second and fourth plots), OG-LVQ takes the lead in one and is second on the other one. Numerical comparisons for 10-recall@10 of 0.9 are shown in Table 3.

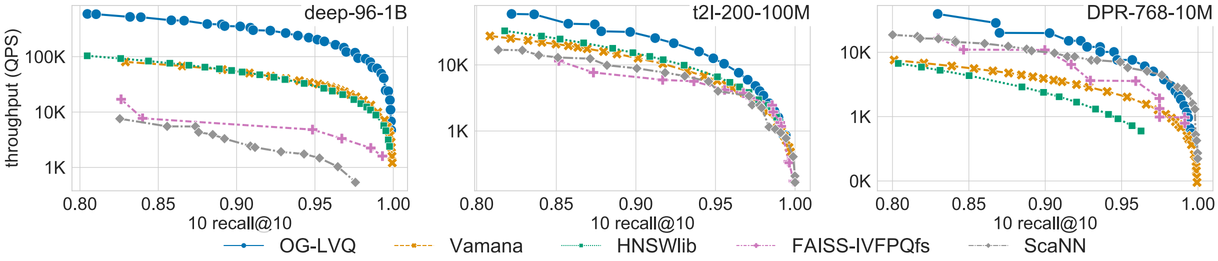


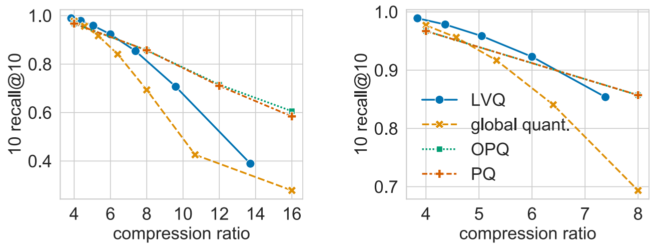
Figure 10: For large scale datasets, the proposed OG-LVQ outperforms its competitors across all datasets and almost across the entire recall range. For a 10-recall@10 of 0.9, the performance of OG-LVQ is 1.8x to 6.5x better than the alternatives. Furthermore, notice that, for larger dimensionalities (second and third plots), the second-best method is not the same in both cases. This highlights the versatility of our approach.

outperforms all its competitors up to recall 0.97. In the high accuracy regime, OG-LVQ with LVQ-4x8 is on par with the competition with a 2.5x smaller footprint than the best alternative.

6.6 LVQ versus PQ for exhaustive search

Product quantization (PQ) [33] is the most popular compression technique for similarity search. PQ is often used at high compression ratios, and is combined with re-ranking using full-precision vectors to achieve a reasonable recall [31]. Subramanya et al. [28] use PQ in this fashion for graphs stored in SSDs. When working with in-memory indices, we have a tough choice: either keep the full-precision vectors in memory (in addition to the compressed codes) and defeat compression altogether, or discard them and experience a severely degraded accuracy. Note that keeping the full-precision vectors in a less expensive storage (e.g., SSD), is not an option as it would severely degrade search throughput. This limits the usefulness of PQ for in-memory graph-based search.

Figure 11 shows the recall achieved by running an exhaustive search with vectors compressed using PQ, OPQ [18] (a PQ variant), LVQ and global quantization for the deep-96-1M dataset (a similar behavior is observed for other datasets). PQ and OPQ perform better for smaller footprints. This occurs because only 1 to 3 bits can be allocated in LVQ to each value, presenting an overly coarse



(a) All compression ratios.

(b) Zoom on smaller ratios.

Figure 11: Exhaustive search accuracy with different vector compression approaches for the deep-96-1M dataset. At lower compression ratios (<6x), see detail on the right, where PQ [33] and OPQ [18] may not require re-ranking, LVQ achieves higher accuracy than both of them (their curves overlap). LVQ comes with the additional advantage of having much faster similarity calculations. At higher compression ratios, re-ranking with full-precision vectors is required to reach a reasonable accuracy, defeating the original purpose of compression. The compression ratio for LVQ is defined in Equation (5), with the same formula for PQ and OPQ, defining their footprint as their number of segments (using 256 centroids per segment).

encoding. On the other hand, PQ and OPQ can exploit correlations across different dimensions, making a better use of the available bits. However, the achieved recall (below 0.7) is not acceptable in modern applications, requiring re-ranking and thus limiting the usefulness of PQ as stated above. At higher footprints, where re-ranking can be avoided, our vector quantization achieves higher accuracy, while introducing almost no overhead for distance computations.

6.7 An ablation study comparing quantizers

We now analyze the search performance advantage that stems from LVQ and compare it to PQ. To assess the quality of vector compression for graph-search, we integrate PQ into our optimized graph-based search and compare the search performance of both compression techniques under the same implementation. We set the number of PQ segments to 96, with 8 bits per segment, as that is the only setting that achieves high enough search recall without the need of re-ranking (see Section 6.6). We also evaluate one and two-level compression schemes using global scalar quantization. For LVQ, we consider three settings: LVQ-8 (one-level with 8 bits), LVQ-4x4 (two-levels with 4 bits each) and LVQ-4x8 (two-levels with 4 and 8 bits). We also evaluate one and two-level compression schemes using global scalar quantization. Finally, we also tried the quantization method by Ko et al. [35], using the suggested parameter settings. This scheme saturates at 0.85 10-recall@10 for deep-96-100M, never reaching our standard of 0.9. At 0.80 10-recall@10, LVQ-8 is 56% faster. We thus omit it from further comparisons.

LVQ achieves higher throughput and higher recall than global scalar quantization. As shown in Figure 12 for the deep-96-100M dataset, the maximum recall achieved by global quantization is 0.96 whereas LVQ goes over 0.98. Note that, with a larger memory footprint, LVQ-4x8 reaches higher accuracy as shown in Figure 13 for deep-96-1B. A similar behavior is observed for other datasets.

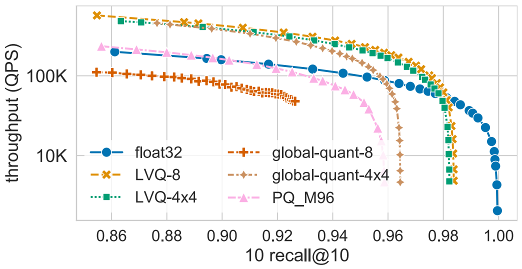


Figure 12: LVQ outcompetes the alternatives for vector compression for deep-96-100M (graph with out-degree $R = 128$). Both variants of LVQ are superior to global quantization with non-vector specific parameters (see Section 3). We use the standard full-precision (i.e., float32-valued) vectors and the ubiquitous Product Quantization (PQ) [33] with 96 segments as our baselines (only this number of segments does not warrant a re-ranking with full-precision vectors at the end of the search). LVQ is clearly superior up to the very high recall of 0.98 (higher recall is achieved with LVQ-4x8, with a slightly higher footprint, as shown in Figure 13). OG-LVQ with LVQ-8 has a 5% larger memory footprint (its vectors are padded to half cache lines) than global quantization with 8 bits and PQ.

Moreover, the storage overhead of using local constants is small for most datasets (e.g., 4% for deep-96-100M). These two aspects confirm the advantage of LVQ over global quantization.

For deep-96-100M, LVQ-4x4 performs slightly worse than LVQ-8, showing that the residual encoding is not the best option in this case (Figure 12). However, as shown in Figure 13 and Table 4, this depends on the dataset. As expected, using two levels has an advantage for higher dimensional datasets as seen for DPR-768-10M. Depending on the dataset, LVQ gives OG-LVQ a QPS boost ranging from 2.6x to 4.7x, vector storage reductions of up to 3.8x and total memory-footprint reduction (considering the space occupied by the graph and the vectors) of up to 2.7x. Table 4 includes float16 encoding as well, confirming the large advantage of LVQ over this compression. For smaller and larger dimensionalities, one or two-level LVQ takes the lead, respectively.

In Figure 12, LVQ-8 outperforms PQ at all recall values with 5.2x more QPS at 0.9 recall for similar compression ratios (4x for PQ vs. 3.84 for LVQ-8). See Section 7 for a detailed explanation.

7 RELATED WORK

The literature on similarity search is vast [36, 49]. Research on the topic evolves quickly, trying to keep up with ever-increasing requirements: more data with larger dimensionality, higher speeds, and a high recall. Trees [13, 41, 51] suffer from the curse of dimensionality. Hashing [23, 27] and learning-to-hash [54] techniques often struggle to simultaneously achieve high accuracy and high speeds. Graph-based methods [5, 14, 17, 25, 28, 39] offer a better latency-accuracy trade-off than other types of algorithms [55].

Product Quantization (PQ) [33] and other related methods [3, 4, 8, 18, 21, 32, 35, 40, 56, 60] were introduced to handle large datasets in settings with limited memory capacity [e.g., 28]. However, when used for high-throughput graph-search, these quantizers do not

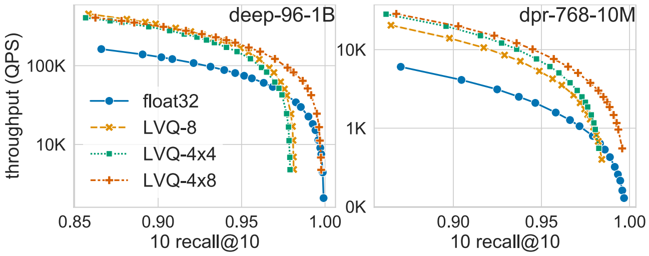


Figure 13: The proposed LVQ increases the performance of graph-baseed search when compared to standard full-precision vectors for deep-96-1B (left) and DPR-768-10M (right). The choice of one vs. two-level LVQ depends on the dimensionality of the dataset. For smaller dimensionalities (d = 96), where there is not such a bandwidth pressure, LVQ-8, with its faster compute prevails, see Section 6.6. For larger dimensionalities (d = 768), the additional bandwidth reduction vastly compensates for the extra compute.

Table 4: The performance of graph search benefits significantly from LVQ. The use of LVQ easily beats the standard baseline of using full-precision (i.e., float32-valued) vectors and even float16-valued vectors, which have recently shown to provide competition-winning performances [52]. The memory-footprint ratio (MR) measures the space occupied by the graph (R = 128) and the float32-valued vectors relative to the space occupied by the graph and the LVQ-compressed vectors. For larger dimensionalities (d = 768), LVQ highly reduces the memory requirements achieving a large MR, and the additional bandwidth reduction from LVQ-4x4 and LVQ-4x8 provides a meaningful performance boost over LVQ-8. The compression ratio (CR) for LVQ is defined in Equation (5). LVQ achieves up to 3.8x vector compression (for improved performance, LVQ-compressed vectors are padded to half-cache lines). The largest QPS, CR, and MR improvements in each case are shaded.

	deep-96-1B			t2i-200-100M			DPR-768-10M		
w.r.t. float32	QPS	CR	MR	QPS	CR	MR	QPS	CR	MR
float16	2.1x	2.0x	1.3x	1.9x	2.0x	1.4x	1.7x	2.0x	1.8x
LVQ-8	2.6x	3.0x	1.4x	2.9x	3.6x	1.8x	3.1x	3.8x	2.7x
LVQ-4x4	2.3x	3.4x	1.4x	2.2x	3.5x	1.8x	4.3x	3.8x	2.7x
LVQ-4x8	2.5x	2.4x	1.3x	3.1x	2.4x	1.6x	4.7x	2.6x	2.1x

enable extremely fast similarity computations in a predominantly random memory access pattern.

For inverted indices [31], the setup for which PQ was designed, the similarity between partitions of the query and each corresponding centroid is generally precomputed to create a look-up table of partial similarities. The computation of the similarity between vectors essentially becomes a set of indexed gather and accumulate operations on this table, which are generally quite slow [42]. This is exacerbated with an increased dataset dimensionality: the lookup table does not fit in L1 cache, which slows down the gather operation. In no small part, the success of recent PQ-based methods [21] can be attributed to Quicker ADC [3], with its optimized lookup operations using AVX shuffle and blend instructions to compute the distance between a query and multiple dataset elements simultaneously. This is enabled by storing these elements in a transposed fashion. This transposition and Quicker ADC by extension are not compatible with the random memory access pattern we see in graph-based similarity search.

Scalar quantization is used for low-precision inference in neural networks [15] to compress the parameter tensors, quantizing each SIMD-sized vector within a tensor individually. They also propose compressing the quantization parameters themselves instead of the residuals as in LVQ. These techniques have not been used for similarity search.

There is existing art in two-level quantization. In [28] the index resides in an SSD instead of in main memory, which enables the use of full-precision vectors. However, SSD indices cannot achieve the performance of their in memory counterparts. In [58], PQ is used for both levels, inheriting the aformentioned issues with PQ. Additionally, both levels are jointly optimized using PQ, which is prohibitive at billion scale.

Dimensionality reduction [2, 59] is an appealing alternative for vector compression that provides an orthogonal reduction to LVQ: they can be combined for stacked gains.

8 CONCLUSIONS

We presented new techniques for creating faster and smaller indices for similarity search. We introduced a novel vector compression method, Locally-adaptive Vector Quantization (LVQ), that simultaneously reduces memory footprint and improves search performance, with minimal impact on search accuracy. LVQ is designed to work optimally in conjunction with graph-based indices, reducing their effective bandwidth while enabling random-access-friendly fast similarity computations. LVQ, combined with key optimizations for graph-based indices in modern datacenter systems, establishes the new state of the art in terms of performance and memory footprint, outcompeting the second-best alternatives for billion scale datasets: (1) in the low-memory regime, by up to 20.7x in throughput with up to a 3x memory footprint reduction, and (2) in the high-throughput regime by 5.8x with 1.4x less memory.

For future work, we plan on studying the impact of LVQ on dynamic similarity search, dimensionality reduction [59] as a preprocessing step, and intra-query parallelism [43].

ACKNOWLEDGMENTS

We would like to thank Kianoosh Zandifar for his invaluable help with our computer systems, and Jawad Khan and Sourabh Dongaonkar for their contributions to an earlier version of our work.

REFERENCES

[1] Cecilia Aguerrebere, Ishwar Bhati, Mark Hildebrand, Mariano Tepper, and Ted Willke. 2023. Similarity search in the blink of an eye with compressed indices. arXiv:2304.04759 [cs.LG]

[2] Nir Ailon and Bernard Chazelle. 2006. Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*. 557–563.

[3] Fabien Andre, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2021. Quicker ADC : Unlocking the Hidden Potential of Product Quantization With SIMD. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43, 5 (may 2021), 1666–1677. https://doi.org/10.1109/tpami.2019.2952606

[4] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache locality is not enough: high-performance nearest neighbor search with product quantization fast scan. *Proceedings of the VLDB Endowment* 9, 4 (Dec. 2015), 288–299. https://doi.org/10.14778/2856318.2856324

[5] Sunil Arya and David M Mount. 1993. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the fourth annual ACM-SIAM symposium on Discrete algorithms*, Vol. 93. 271–280.

[6] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374. https://doi.org/10.1016/j.is.2019.02.006

[7] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. Benchmarking nearest neighbors. http://ann-benchmarks.com/index.html. GitHub code: http://github.com/erikbern/ann-benchmarks/. Accessed: 15 Feb. 2023.

[8] 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

[9] Artem Babenko and Victor Lempitsky. 2021. Benchmarks for Billion-Scale Similarity Search. https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search. Accessed: 15 Feb. 2023.

[10] Artem Babenko and Victor Lempitsky. 22016. Deep billion-scale indexing. http://sites.skoltech.ru/compvision/noimi/. Accessed: 15 Feb. 2023.

[11] Andreas Blattmann, Robin Rombach, Kaan Oktay, Jonas Müller, and Björn Ommer. 2022. Retrieval-augmented diffusion models. *Advances in Neural Information Processing Systems* 35 (2022), 15309–15324.

[12] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *Proceedings of the 39th International Conference on Machine Learning*. PMLR, 2206–2240.

[13] Lawrence Cayton. 2008. Fast nearest neighbor retrieval for bregman divergences. In *Proceedings of the 25th international conference on Machine learning - ICML '08*. ACM Press, Helsinki, Finland, 112–119. https://doi.org/10.1145/1390156.1390171

[14] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. *SPTAG: A library for fast approximate nearest neighbor search*. https://github.com/Microsoft/SPTAG Accessed: 15 Feb. 2023.

[15] Steve Dai, Rangharajan Venkatesan, Haoxing Ren, Brian Zimmer, William J. Dally, and Brucek Khailany. 2021. VS-Quant: Per-vector Scaled Quantization for Accurate Low-Precision Neural Network Inference. *ArXiv* abs/2102.04503 (2021).

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*. 4171–4186. https://doi.org/10.18653/v1/n19-1423

[17] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment* 12, 5 (Jan. 2019), 461–474. https://doi.org/10.14778/3303753.3303754

[18] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2013), 744–755.

[19] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2021. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630* (2021).

[20] Mihajlo Grbovic, Nemanja Djuric, Vladan Radosavljevic, Fabrizio Silvestri, Ricardo Baeza-Yates, Andrew Feng, Erik Ordentlich, Lee Yang, and Gavin Owens. 2016. Scalable semantic matching of queries to ads in sponsored search advertising. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. 375–384.

[21] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, 3887–3896.

[22] IEEE. 1985. IEEE Standard for Binary Floating-Point Arithmetic. https://doi.org/10.1109/IEEESTD.1985.82928 Accessed: 15 Feb. 2023.

[23] 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*. 604–613.

[24] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Nearest Neighbor Search with Neighborhood Graph and Tree for High-dimensional Data. https://github.com/yahoojapan/NGT. Accessed: 15 Feb. 2023.

[25] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. http://arxiv.org/abs/1810.07355 arXiv:1810.07355 [cs].

[26] Bruce Jacob and Trevor Mudge. 1998. Virtual memory: issues of implementation. *Computer* 31, 6 (1998), 33–43. https://doi.org/10.1109/2.683005

[27] Omid Jafari, Preeti Maurya, Parth Nagarkar, Khandker Mushfiqul Islam, and Chidambaram Crushev. 2021. A Survey on Locality Sensitive Hashing Algorithms and their Applications. http://arxiv.org/abs/2102.08942 arXiv:2102.08942 [cs].

[28] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems* 32 (2019).

[29] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/. Accessed: 15 Feb. 2023.

[30] Yanrong Ji, Zhihan Zhou, Han Liu, and Ramana V Davuluri. 2021. DNABERT: pre-trained Bidirectional Encoder Representations from Transformers model for DNA-language in genome. *Bioinformatics* 37, 15 (2021), 2112–2120.

[31] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[32] 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

[33] Herve 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

[34] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 6769–6781. https://doi.org/10.18653/v1/2020.emnlp-main.550

[35] Anthony Ko, Iman Keivanloo, Vihan Lakshman, and Eric Schkufza. 2021. Low-Precision Quantization for Efficient Nearest Neighbor Search. (2021). arXiv:2110.08919

[36] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (Aug. 2020), 1475–1488. https://doi.org/10.1109/TKDE.2019.2909204

[37] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[38] Defu Lian, Haoyu Wang, Zheng Liu, Jianxun Lian, Enhong Chen, and Xing Xie. 2020. Lightrec: A memory and search-efficient recommender system. In *Proceedings of The Web Conference 2020*. 695–705.

[39] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[40] Yusuke Matsui, Yusuke Uchida, Herve Jegou, and Shin'ichi Satoh. 2018. [Invited Paper] A Survey of Product Quantization. *ITE Transactions on Media Technology and Applications* 6, 1 (2018), 2–10.

[41] 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

[42] Douglas Michael Pase and Anthony Michael Agelastos. 2019. Performance of Gather/Scatter Operations. (3 2019). https://doi.org/10.2172/1761952

[43] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. 2022. Speed-ANN: Low-Latency and High-Accuracy Nearest Neighbor Search via Intra-Query Parallelism. (2022). arXiv:2201.13007

[44] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. https://doi.org/10.3115/v1/D14-1162

[45] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PMLR, 8748–8763.

[46] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits

of Transfer Learning with a Unified Text-to-Text Transformer. *The Journal of Machine Learning Research* 21 (2020), 140:1–140:67. http://jmlr.org/papers/v21/20-074.html

[47] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. 2021. Trident: Harnessing Architectural Resources for All Page Sizes in X86 Processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1106–1120. https://doi.org/10.1145/3466752.3480062

[48] Charbel Sakr, Steve Dai, Rangha Venkatesan, Brian Zimmer, William Dally, and Brucek Khailany. 2022. Optimal clipping and magnitude-aware differentiation for improved quantization-aware training. In *International Conference on Machine Learning*. PMLR, 19123–19138.

[49] Larissa C. Shimomura, Rafael Seidi Oyamada, Marcos R. Vieira, and Daniel S. Kaster. 2021. A survey on graph-based methods for similarity searches in metric spaces. *Information Systems* 95 (Jan. 2021), 101507. https://doi.org/10.1016/j.is.2020.101507

[50] Nina Shvetsova, Brian Chen, Andrew Rouditchenko, Samuel Thomas, Brian Kingsbury, Rogerio S Feris, David Harwath, James Glass, and Hilde Kuehne. 2022. Everything at once-multi-modal fusion transformer for video retrieval. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 20020–20029.

[51] Chanop Silpa-Anan and Richard Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Anchorage, AK, USA, 1–8. https://doi.org/10.1109/CVPR.2008.4587638

[52] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. 2022. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. https://big-ann-benchmarks.com/. In *NeurIPS 2021 Competitions and Demonstrations Track*. PMLR, 177–189.

[53] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, and Sri Sakthivelu. 2013. Intel® Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html. Used v.3.10, Last Updated: 07/20/2021.

[54] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. 2018. A Survey on Learning to Hash. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, 4 (April 2018), 769–790. https://doi.org/10.1109/TPAMI.2017.2699960

[55] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 14, 11 (jul 2021), 1964–1978. https://doi.org/10.14778/3476249.3476255

[56] Runhui Wang and Dong Deng. 2020. DeltaPQ: lossless product quantization code compression for high dimensional similarity search. *Proceedings of the VLDB Endowment* 13, 13 (Sept. 2020), 3603–3616. https://doi.org/10.14778/3424573.3424580

[57] William Webber, Alistair Moffat, and Justin Zobel. 2010. A similarity measure for indefinite rankings. *ACM Transactions on Information Systems (TOIS)* 28, 4 (2010), 1–38.

[58] Zhi Xu, Lushuai Niu, Ruimin Meng, Longyang Zhao, and Jianqiu Ji. 2022. Residual Vector Product Quantization for Approximate Nearest Neighbor Search. In *Advances in Knowledge Discovery and Data Mining: 26th Pacific-Asia Conference, PAKDD 2022, Chengdu, China, May 16–19, 2022, Proceedings, Part I*. Springer, 208–220. https://doi.org/10.3390/s101211259

[59] Haokui Zhang, Buzhou Tang, Wenze Hu, and Xiaoyu Wang. 2022. Connecting Compression Spaces with Transformer for Approximate Nearest Neighbor Search. In *Computer Vision – ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XIV*. Springer Nature Switzerland, 515–530. https://doi.org/10.1007/978-3-031-19781-9_30

[60] Ting Zhang, Chao Du, and Jingdong Wang. 2014. Composite quantization for approximate nearest neighbor search. In *International Conference on Machine Learning*. PMLR, 838–846.