Stop Indexing at Full Precision: Revisiting Clustering for Vector Embeddings

Leonardo Kuffo
CWI
Amsterdam, The Netherlands
lxkr@cwi.nl

Peter Boncz
CWI
Amsterdam, The Netherlands
boncz@cwi.nl

ABSTRACT

In this study, we revisit three widely used techniques in vector search and utilize them to optimize vector embedding indexing through clustering: dimensionality reduction, quantization, and dimension pruning. We propose an indexing pipeline in which these techniques are applied *before* clustering, and we focus on how they affect storage footprint, clustering time, and the quality of the resulting centroids for vector search tasks. Our results reveal that using full-precision vectors for clustering is excessive, as even 1-bit codes can achieve near-optimal clustering quality (within 1% of ideal) while reducing storage requirements by 60x and delivering attractive performance gains (Figure 1). We open-source our implementations at https://github.com/cwida/SuperKMeans.

VLDB Workshop Reference Format:
Leonardo Kuffo and Peter Boncz. Stop Indexing at Full Precision: Revisiting Clustering for Vector Embeddings. VLDB 2026 Workshop: The 2nd Workshop on Vector Databases.

1 INTRODUCTION

Nowadays, clustering algorithms (e.g., k-means [49]) are utilized in *approximate vector similarity search* (VSS) to index large collections of high-dimensional vectors [1,12,16,21,29,33,42,57,62,64,67,71,72]. VSS consists of finding the vectors in a collection that are the most similar to a given query vector, based on a distance or similarity metric. When a vector collection is indexed using clustering, the resulting centroids act as entry points to guide queries to the clusters that are most likely to contain their nearest neighbors. Thus, reducing search latency by avoiding accessing the majority of the collection. This indexing method remains a popular option across vector systems due to its scalability and advantages in data management tasks compared to other index families, such as graph-based ones [1,48,51,57,62,64,91,97].

Clustering poses major challenges compared to VSS. First, it is not only memory-bound but also heavily compute-bound [41,42, 54,56]. Furthermore, it requires accessing the entire collection of raw vectors multiple times, whereas VSS prunes most of the vector collection on every query. As a result, clustering remains a critical step that bottlenecks user queries until the index is built. Despite this, most research efforts focus on search algorithms, with few addressing vector indexing via clustering [9,42,54,56,82,85].

Proceedings of the VLDB Endowment. ISSN 2150-8097.

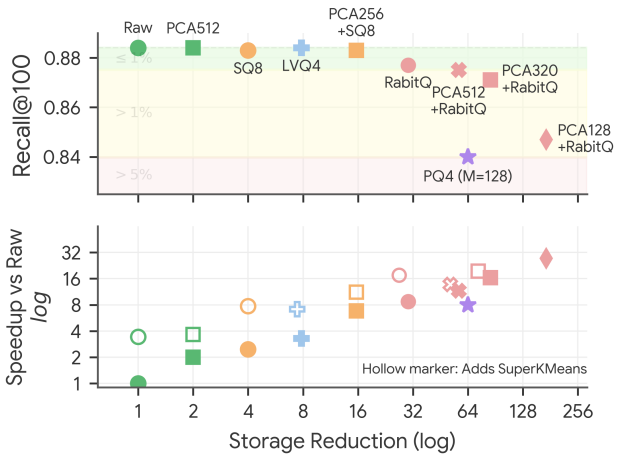

Figure 1: Clustering performance for all competitors in 10M 1024-dimensional Cohere embeddings. Clustering is resilient to approximation techniques, achieving near-optimal clustering quality with 60x storage reduction (top). Clustering speed also improves, even more with SuperKMeans (shown with hollow markers), which brings attractive speedups without hurting index quality (bottom). Green, yellow, and red indicate ≤1%, >1%, and >5% away, resp., from the results of clustering with raw (full-precision) vectors. The evaluation framework is presented in Section 4.

In this work, we revisit three widely used techniques for vector search and use them for vector indexing via clustering: **dimensionality reduction** (JLT [8,9,37], PCA [2,78,79,85,92], Matryoshka vectors [45]), **quantization** (SQ [16], LVQ [3], RabitQ [21,23,72], PQ [17,35]), and **dimension pruning** [15,22,43,67,83,96]. We propose an indexing pipeline that applies these techniques *before* clustering, resulting in attractive performance gains and storage reductions without sacrificing index quality. Our pipeline design *differs* from most systems, where clustering uses full-precision vectors and quantization occurs only after clustering [16,60,72,84].

Our results show that using full-precision vectors for clustering is excessive, as even using 1-bit RabitQ codes of PCA-projected vectors achieves near-optimal clustering quality (less than 1% degradation) with 60x storage reduction. As part of our contributions, we adapt dimension-pruning techniques (SuperKMeans [42], ADSampling [22]) to integrate with quantization schemes to further accelerate clustering without sacrificing quality. Finally, we explore several design decisions across the indexing pipeline that affect performance and the quality of clusters for vector search tasks.

2 PRELIMINARIES

2.1 Vector Search

Nearest Neighbor Search (NNS) consists of finding the closest vectors in a collection to a given query vector, based on a distance or similarity metric. Finding exact answers is often impractical due to the large compute and storage requirements. However, modern applications such as RAG and Recommender Systems powered by NNS typically do not require exact answers as approximate answers are "good enough". This is known as Approximate Nearest Neighbor Search (ANNS), or Vector Similarity Search (VSS). The approximate nature of VSS enables the use of techniques that reduce storage and compute requirements during query time, such as quantization [3,16,21,23,35,93], dimension pruning [22,83,86,96], and dimensionality reduction [2,8,45,47,92].

These techniques are usually combined with *approximate indexes* [21,27,33,35] that guide queries toward the most promising vectors in the collection. Among vector indexes, two main families exist: partition-based [10,33,35,57,62,75,87] and graph-based [7, 27,34,52,76]. While graph-based indexes can answer queries with fewer distance comparisons than partition-based indexes [53], they take a considerably higher amount of time to build [7,42,51,87]. Furthermore, they introduce additional complexity in data management tasks, such as ingestion and updates [10,89,91], as well as when combining VSS with traditional SQL-like filtering [25,50,63]. This makes partition-based indexes an attractive option for large collections and distributed or cloud environments [1,12,48,57,62,77].

2.2 The Role of Clustering in Vector Search

Partition-based indexes, such as the widely used Inverted-Files (IVF), organize a collection of vectors into clusters through clustering methods like k-means [16,33,62]. During query time, the distance metric is first evaluated between the query q and the centroids Y. Then, the vectors within the nearest clusters are chosen for evaluation (Figure 2). By adjusting the number of explored clusters, one can balance search speed with quality [16,84]. Typically, the number of centroids is set around $\sqrt{N}$ [16,84], resulting in higher k values compared to other use cases of clustering [6,58,95].

Despite clustering being orders of magnitude faster than graph-index construction, it remains a crucial bottleneck, as users experience degraded performance or are unable to query the collection until an index is built [51]. Additionally, the clustering of vector embeddings introduces several challenges compared to VSS. First, it requires multiple accesses to the entire vector collection (or a large subsample [9,42]), whereas vector search typically accesses only portions of a collection with each query. Second, unlike vector search, which is mostly data-access bound [41], clustering is also heavily compute-bound due to the higher number of computations needed. On top of that, the increasing demand for vectors as first-class citizens in database systems has created new scenarios in which low-latency, low-memory indexing is essential. For instance, in a situation where an attribute-based filter is applied to a vector column [50], with the resulting vectors then utilized to perform a similarity `JOIN` [88] with a different vector column from another table, the ability to create an index on the fly could enhance the performance of the similarity `JOIN` operator if it can be created quickly enough.

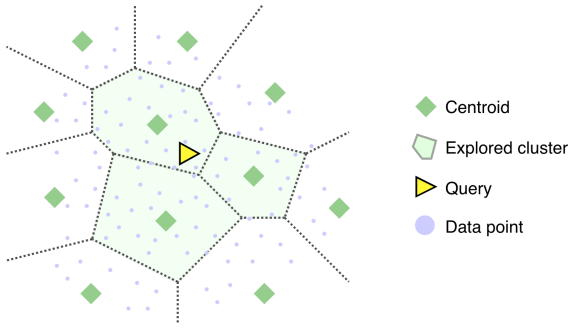

Figure 2: Example of an IVF index search where only the points in the clusters closest to the query are explored. The clusters are defined via k-means.

2.3 Clustering in Vector Systems

The implementation of clustering found in vector systems (e.g., cuVS [68], FAISS [16], Milvus [84], VectorChord [9], Vortex [24], LanceDB [60], Elastic [82]) follows a Lloyd k-means algorithm [49] or a hierarchical variant of it. In brief, k-means follows these steps:

STEP 1. Centroids Initialization: Centroids are initialized by randomly sampling k vectors from the collection [20]. More sophisticated initializations, such as k-means++ [6], have proven ineffective for vector embedding datasets due to their marginal improvements while incurring substantially higher runtime [42].

STEP 2. Determining Assignments: Assignments are determined using a *distance metric* that identifies which point x in the collection X is closest to which centroid y of Y. The L2 Euclidean distance is the most commonly used. This step is the main bottleneck of clustering, as it requires computing pairwise distances between every x in X and every y in Y. The most efficient way to do this is through a General Matrix Multiplication (GEMM) of the data points and centroids ($X \cdot Y$). GEMM routines are highly efficient due to their optimized data access patterns, vectorization (SIMD), efficient cache usage, and multi-threading.

STEP 3. Updating Centroids: Centroids are updated by averaging every vector x assigned to them. When centroids have zero assignments, it is beneficial to split large clusters to achieve a more balanced distribution of points across clusters—a desirable characteristic for partition-based indexes used in VSS [33,57,69].

Termination Conditions and Final Assignments: The algorithm terminates after a predetermined number of iterations of STEPS 2 and 3, typically ranging from 5 to 10 for vector embedding datasets [42]. Then, STEP 2 is executed once more to obtain the final assignments of each point to its nearest centroid.

2.4 Efforts to Optimize Vector Clustering

Research on optimizing clustering of vector embeddings has received far less attention than vector search algorithms. Nevertheless, a few techniques tailored for vector search have seen success when applied to vector clustering [9,42,56,82,85].

Dimensionality reduction aims to represent vectors in a lower-dimensional space, thereby reducing memory footprint and compute requirements [47,78,92]. VectorChord employs a Johnson–Lindenstrauss Transform (JLT) [8] to reduce the dimensionality of the raw vectors before clustering [9]. This reduces the memory footprint of clustering while providing performance gains from fewer computations [8,9,94]. LindormVector [85] accomplishes the same goal with a PCA (Principal Component Analysis) projection that concentrates the vectors' *energy* in the leading dimensions. More recently, embedding models have been trained to generate Matryoshka embeddings [45], which already concentrate energy in the front dimensions. However, the impact of dimensionality reduction on clustering quality has not been thoroughly studied.

Quantization techniques aim to represent each dimension of a vector, or a group of dimensions, with smaller codes while minimizing distortion in the distance metric between points [3,23,61]. PQk-means is one of the few studies that use quantization prior to clustering [56]. PQk-means encodes vectors using Product Quantization [35], effectively reducing memory usage and accelerating distance calculations, albeit at the cost of clustering quality. More recent quantization techniques, such as LVQ [3,4] and RabitQ [21,23,27], have not been used for vector clustering.

Dimension pruning aims to break off distance computations when the distance metric has enough resolution to determine that a point will not be the nearest neighbor of a query [15,22,43,83,90,96]. SuperKMeans integrates dimension pruning into vector clustering by interleaving GEMM routines and pruning kernels, achieving substantial speedups without sacrificing clustering quality [42]. However, it remains unclear whether SuperKMeans can be combined with the previously mentioned techniques.

The impact of approximations on clustering quality has not been thoroughly studied. In most clustering pipelines, vectors are indexed at full precision (`float32`, `float16`) before being projected into a smaller d-dimensional space [47,78,79,92], quantized [3,72,78], and materialized in the index data structure. In this study, we show that approximation techniques can be applied first in the indexing pipeline, saving several round trips to the raw data, reducing memory footprint during clustering, and delivering attractive performance gains, all while mostly preserving clustering quality.

2.5 Keys for the Performance of Clustering

GEMM routines for `float32` are the fuel for high-performance clustering, having undergone decades of optimization. Consequently, clustering speed improves with dimensionality reduction, as d decreases while maintaining `float32` as the data type. Dimension pruning further improves speed by reducing the number of distance calculations. However, the performance of clustering becomes more nuanced when working with quantized vectors. Ideally, the quantization method should facilitate efficient many-to-many distance calculations that leverage modern CPU capabilities for processing smaller data types, such as 8-bit integers. Nonetheless, the performance benefits depend on the availability of SIMD [13,19,41,80,81] or specialized hardware (e.g., Intel's AMX [31,39]) that can efficiently handle 8-bit data types.

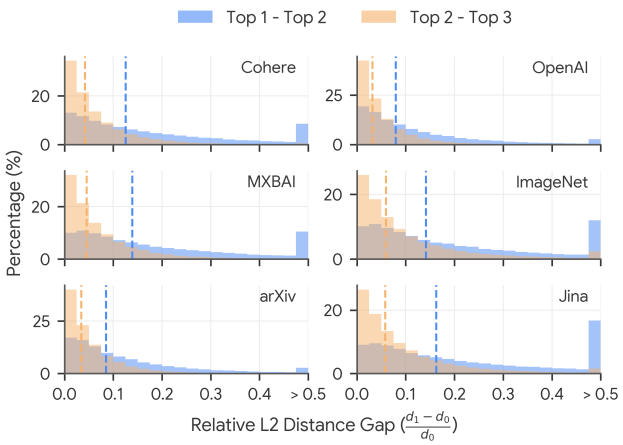

Figure 3: Distribution of the L2 distance gap between points and their 1st and 2nd nearest centroid (blue) and 2nd and 3rd (yellow) across datasets. The nearest centroid is meaningfully separated, making clustering resilient to approximation techniques. The median is shown as a vertical line.

2.6 The Resilience of Clustering

The assignment step in the clustering process (STEP 2) is effectively a series of *top-1* NNS queries with roles reversed: the vector collection acts as the queries, while the centroids become the vector collection to query. Figure 3 shows the distribution of the relative L2 distance gap between vectors and their nearest centroids across several vector embedding datasets. Notably, the gap between the *top-1* and *top-2* nearest centroids (in blue) is 2-3x larger than that of the next neighboring centroids (*top-2* and *top-3*, in yellow), making the *top-1* centroid distinctly identifiable. In contrast, the 2nd and 3rd nearest centroids are typically close to one another. This phenomenon makes vector clustering far more resilient to approximation techniques compared to vector search, as errors introduced by methods such as quantization are less likely to affect the correct assignment to the closest centroid [44]. This observation raises the question: *To what extent can techniques utilized in vector search be applied to vector clustering?* In the remainder of this study, we empirically address this question by examining the effects of several techniques (and their combinations) on the quality of clustering.

3 OUR PIPELINE: APPROXIMATE FIRST, THEN CLUSTER

We propose an indexing pipeline (shown in Figure 4) where dimensionality reduction and quantization occur *before* clustering, and dimension pruning is used during clustering to reduce the number of distance calculations. This design *differs* from most systems, which typically use full-precision vectors for clustering and apply quantization only after the clustering stage [16,60,72,84]. Table 1 presents the techniques we have chosen for our evaluation. Next, we describe how we adapted these techniques for clustering.

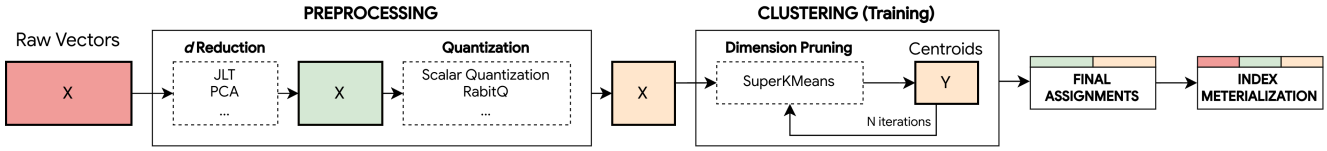

Figure 4: Our proposed pipeline applies vector approximation techniques before clustering, and dimension pruning accelerates distance calculations during clustering. The final assignment of points to clusters can also happen in the projected (in green) or quantized (in yellow) domain. In our pipeline, touching the raw vectors (in red) after preprocessing is never a must.

Table 1: Techniques used in our clustering pipeline.

Type	Name	Representation	Centroids Assignment	Updating Centroids	Storage Reduction	Can be mixed with
Dimensionality Reduction	JLT [8] PCA [2] Matryoshka [45]	$x \in \mathbb{R}^{d'}; d' < d$	GEMM	Averaging	$\frac{d}{d'}$x	All except PQ
Quantization	SQ8 | SQ4 [16]	$x \mapsto x' \in \{0, 1, ..., 255\|15\}^d$	8-bit | 4-bit GEMM	Averaging	4x | 8x	All
	LVQ4 [3]	$x \mapsto x' \in \{0, 1, ..., 15\}^d$	4-bit GEMM + Adjustment	Fused Decode-Averaging	8x	All
	RabitQ [23]	$x \mapsto x' \in \{0, 1\}^d$	FastScan [5] + Adjustment	Fused Decode-Averaging	30x	All
	PQ8 [35,56]	$x \mapsto x' \in \{1, ..., 256\}^M$	FastScan [5]	Sparse voting [56]	$\frac{d}{M} \cdot 4$x	None
	PQ4 [35,56]	$x \mapsto x' \in \{1, ..., 128\}^M$	FastScan [5]	Sparse voting [56]	$\frac{d}{M} \cdot 8$x	None
Dimension Pruning	SuperKMeans [42]	Invariant	Invariant + Pruning	Invariant	Invariant	All except PQ

3.1 Adapting Techniques for Vector Clustering

SQ8 and SQ4: In SQ, each value x_i of a vector x is encoded with a global bias $b = v_{min}$, and a global scaling factor $s = \frac{v_{max} - b}{c_{max}}$, where $c_{max} = 2^B - 1$ and B is the quantization bit-width. Each quantized code $\bar{x}_i$ is defined as $\bar{x}_i = \text{clip}\left(\left\lfloor \frac{x_i - b}{s} \right\rceil, 0, c_{max}\right)$, and reconstructed as: $x_i \approx (\bar{x}_i \cdot s) + b$. Since all vectors are transformed with the same parameters, distance computations can be performed directly in the quantized domain using integer arithmetic, leveraging 8- and 4-bit GEMM routines [26,81]. Finally, updating centroids can be done by averaging each dimension, also in the quantized domain.

LVQ4: In LVQ [3], each value x_i of a vector x is encoded with a per-vector bias $b_x = x_{min}$, and scale $s_x = \frac{x_{max} - b_x}{c_{max}}; c_{max} = 2^B - 1$. Each quantized code $\bar{x}_i$ is defined as $\bar{x}_i = \text{clip}\left(\left\lfloor \frac{x_i - b_x}{s_x} \right\rceil, 0, c_{max}\right)$, and reconstructed as: $x_i \approx (s_x \cdot \bar{x}_i) + b_x$. Since each vector has a different s and b, the codes cannot be used directly for distance calculations. To solve this, LVQ proposes fusing reconstruction of `float32` values with distance calculations. However, by doing so we lose the benefits of using a smaller data type. Furthermore, in many-to-many distance calculations, the reconstruction step is performed multiple times for the same vectors, which hurts efficiency. Hereby, we rewrite the L2 distance $||x - y||^2$ as:

$$\begin{aligned} ||x - y||^2 &\approx \sum_i^d ((s_x \bar{x}_i + b_x) - (s_y \bar{y}_i + b_y))^2 \\ &\approx s_x^2 \sum_i^d \bar{x}_i^2 + s_y^2 \sum_i^d \bar{y}_i^2 - 2 s_x s_y \langle \bar{x}, \bar{y} \rangle \\ &\quad + 2(b_x - b_y)(s_x \sum_i^d \bar{x}_i - s_y \sum_i^d \bar{y}_i) + d(b_x - b_y)^2 \end{aligned} \tag{1}$$

Collecting constant x-only terms into $N_x = s_x^2 \Sigma \bar{x}_i^2 + 2 b_x s_x \Sigma \bar{x}_i + d b_x^2$, and y-only terms into $N_y = s_y^2 \Sigma \bar{y}_i^2 + 2 b_y s_y \Sigma \bar{y}_i + d b_y^2$, and defining $A_x = s_x \Sigma \bar{x} + d b_x$:

$$||x - y||^2 \approx N_x + N_y - 2\left(s_x \cdot s_y \cdot \langle \bar{x}, \bar{y} \rangle + b_y \cdot A_x + b_x \cdot s_y \Sigma \bar{y}_i\right) \tag{2}$$

This derivation allows us to use 4-bit GEMM kernels to compute the highlighted term. The final distances require additional scalar operations, from which N_x and A_x terms can be cached across k-means iterations, while N_y can be computed once at the start of each iteration. Finally, when updating centroids, we cannot solely use LVQ codes, as averaging them is undefined. Thus, we fuse the decoding and averaging of the LVQ codes of the centroid assignments and re-encode the resulting centroids.

RabitQ: In RabitQ, each vector is centered, normalized to the unit sphere, and randomly rotated. Then, each dimension is quantized to 1-bit by only preserving their *sign*. Due to the properties of a random rotation, this quantized vector $\bar{x}$ of signs lives in the codebook $\{+1/\sqrt{d}, -1/\sqrt{d}\}^d$. RabitQ then provides an unbiased estimator of L2 distances and guarantees that the estimator has an asymptotically optimal error bound [23]. Let x_r be a raw data vector and y_r be a raw query, normalized based on a vector m (i.e., the dataset means). Now, let $x := \frac{x_r - m}{||x_r - m||}$ and $y := \frac{y_r - m}{||y_r - m||}$. The L2 distance between x_r and y_r can be expressed as:

$$\begin{aligned} ||x_r - y_r||^2 &= ||(x_r - m) - (y_r - m)||^2 \\ &= ||x_r - m||^2 + ||y_r - m||^2 \\ &\quad - 2 \cdot ||x_r - m|| \cdot ||y_r - m|| \cdot \langle x, y \rangle \end{aligned} \tag{3}$$

From which $||x_r - m||$ can be computed and cached in the first iteration of k-means and $||y_r - m||$ can be precomputed once per centroid, and thus, is amortized by all the pairwise distance calculations. For the term, $\langle x, y \rangle$, RabitQ derives an unbiased estimator. Let $\bar{x}$ denote the 1-bit quantized version of x, then: $\langle x, y \rangle \approx \frac{\langle \bar{x}, y \rangle}{\langle \bar{x}, x \rangle}$, where $\langle \bar{x}, x \rangle$ is the cosine between the original and quantized unit vector, precomputed and stored per data point. Substituting into Equation 3, and recalling that $y := \frac{y_r - m}{||y_r - m||}$, then we derive:

$$||x_r - y_r||^2 \approx ||x_r - m||^2 + ||y_r - m||^2 - 2 \cdot \frac{||x_r - m||}{\langle \bar{x}, x \rangle} \cdot \langle \bar{x},\ y_r - m \rangle \tag{4}$$

The scalar factors $||x_r - m||^2$ and $||x_r - m||/\langle \bar{x}, x \rangle$ depend only on the data points and can be cached across k-means iterations, while $||y_r - m||^2$ can be computed once at the start of each iteration. In RabitQ, the centered residual vector $y_r - m$ is quantized with SQ4 ($\bar{y}$), with scale s and bias b. Then, expanding the SQ4 reconstruction $(y_r - m)_i \approx \bar{y}_i \cdot s + b$ and reconstructing the codebook with the stored bits $\bar{x}_i \in \{0, 1\}$, so that each codebook entry is $(2\bar{x}_i - 1)/\sqrt{d}$, yields the following estimator for $\langle \bar{x}, y_r - m \rangle$:

$$\begin{aligned}\langle \bar{x}, \bar{y} \rangle &\approx \frac{1}{\sqrt{d}} \sum_i^d (2\bar{x}_i - 1)(b + \bar{y}_i \cdot s) \\ &\approx \frac{1}{\sqrt{d}} \left[s \cdot \left(2 \sum_i^d \bar{x}_i \cdot \bar{y}_i - \sum_i^d \bar{y}_i \right) + b \cdot \left(\left(2 \sum_i^d \bar{x}_i \right) - d \right) \right]\end{aligned} \tag{5}$$

The challenge in computing the highlighted term arises from operand asymmetry: $\bar{x}$ is binary while $\bar{y}$ is SQ4. This computation can be done efficiently using in-register 4-bit lookup tables with the `PSHUFB` [18] instruction, which allows for 32 lookups at a time. This method is known as FastScan [5]. The lookup tables can be built on the fly during each iteration of k-means. The other terms are scalar operations performed once per distance calculation. In our vector clustering pipeline, we encode data points (which act as queries) with 1-bit RabitQ and the centroids with SQ4. We took this approach because the data points bound the memory footprint of clustering. Finally, when updating centroids, we fuse the decoding of RabitQ codes with the averaging of centroid assignments, then re-encode the resulting centroid with SQ4. The decoding of RabitQ codes reconstructs each coordinate x_{r_i} as $x_{r_i} \approx m_i + \frac{1}{\sqrt{d}} \cdot ||x_r - m||/\langle \bar{x}, x \rangle \cdot (2\bar{x}_i - 1)$; $\bar{x}_i \in \{0, 1\}$.

PQ8 and PQ4: In PQ [35], the dimensions are divided into M, $\frac{d}{M}$ -dimensional subspaces. Each group M is represented with an 8- or 4-bit code from a codebook trained for each subspace. PQ8 allows for 256 different codes, while PQ4 allows for 16 codes. These codes map to representative centroids in each subspace. For encoding a vector x, each dimension group is assigned the closest centroid code from the codebook of each subspace. As both data and centroids are product-quantized, we use Symmetric Distance Comparisons (code-to-code) to compute assignments. PQ4 can do this efficiently with FastScan [5], while PQ8 uses scalar lookups as its 8-bit codes are incompatible with the `PSHUFB` instruction. Finally, when updating centroids, we adopt the *sparse voting* mechanism introduced in PQk-means [56].

SuperKMeans accelerates distance calculations during centroid assignment by interleaving GEMM routines for the front d' dimensions (set to 12.5% of d) and using progressive-pruning kernels every 64 dimensions on the surviving candidates [42]. A random orthogonal rotation is required for pruning. This transformation distributes the variance more evenly across all dimensions while preserving the L2 distances between vectors, enabling reliable pruning using ADSampling [22]. In SQ8 and SQ4, SuperKMeans is extended by replacing the `float32` GEMM with 8- and 4-bit GEMMs. However, SuperKMeans encounters limitations with LVQ and RabitQ because it requires L2 distances at d', where $d' < d$. Both LVQ and RabitQ derivations (Equations 2 and 3) can be used with d', as the L2 distance can be decomposed as $||x - y||^2 = ||x' - y'||^2 + ||x'' - y''||^2$, where x' refers to the front d' dimensions of x, and x'' to the remaining ones. However, this incurs storage overhead for the adjustment factors needed per dimension segment, as well as the additional overhead of floating-point operations. Hereby, for LVQ4 and RabitQ, we use only 2 pruning checkpoints at 12.5% and 25% of d. For progressive pruning to take place, efficient 1-to-1 distance kernels are also necessary. In SQ8, SQ4, and LVQ4, we use 8- and 4-bit L2 distance kernels [81]. In RabitQ, we use the `POPCNT`-per-bitplane approach described in [23].

Table 2: Vector embedding datasets used for evaluation

Name	Type	Model	# Vectors	Dim.	Size (GB)↑	LID	Dist.
Cohere [11]	Text	EmbedV3-EN	10,000,000	1024	40.96	27.9	
OpenAI [66]	Text	OpenAI	5,000,000	1536	30.72	31.4	
arXiv [65]	Text	InstructorXL	2,253,000	768	6.92	28.9	
Jina [32,40]	Code	Jina	1,374,067	768	4.22	20.5	
MXBAI [32,46]	Text	MXBAI	769,382	1024	3.15	24.5	
ImageNet [14,32]	Image	CLIP	1,281,167	512	2.62	14.6	

The Johnson–Lindenstrauss Transform (JLT) projects vectors to a lower d' while preserving pairwise distances between the points [8,37]. **Principal Component Analysis (PCA)** projects vectors into a lower d' while preserving global variance and concentrating the *energy* of the vectors in the front dimensions. **Matryoshka** vectors are produced by embedding models that already concentrate the energy in the front dimensions [45]. These techniques do not need extra engineering to be integrated in the clustering pipeline, as vectors remain in the `float32` domain.

4 EVALUATION

We experimentally evaluate our indexing pipeline using the vector embedding datasets presented in Table 2. Our evaluation focuses on three aspects: (i) clustering quality, (ii) storage reduction, and (iii) clustering speed. For clustering quality, we assess how well the generated centroids perform in vector search tasks when used as an entry point for an IVF index. To quantify this, we use the *recall@k* metric together with the number of vectors explored during search, which serves as a proxy for the amount of distance computation required. *Recall@k* measures the proportion of the true nearest neighbors (i.e., the ground truth) that are retrieved among the top-k results. In our evaluation, we retrieve the top 100 neighbors (i.e., *recall@100*). In IVF indexes, *recall* can be tuned by varying the number of clusters probed during search. We report results for two probing configurations that correspond to exploring 1% and 3% of the available clusters. For a given *recall* level, exploring fewer vectors is preferable as it indicates that the search requires fewer distance computations.

Additionally, we record the *within-cluster sum of squares* (WCSS), which measures the compactness of clusters by summing the squared distances of each point to its centroid (lower is better). Regarding storage reduction, we report the reduction in storage required to perform the clustering iterations. Finally, in terms of clustering speed, we measure the end-to-end runtime of our clustering pipeline over 10 iterations of k-means, setting the number of clusters to $4\sqrt{N}$ [16,84], where N is the number of vectors in the collection.

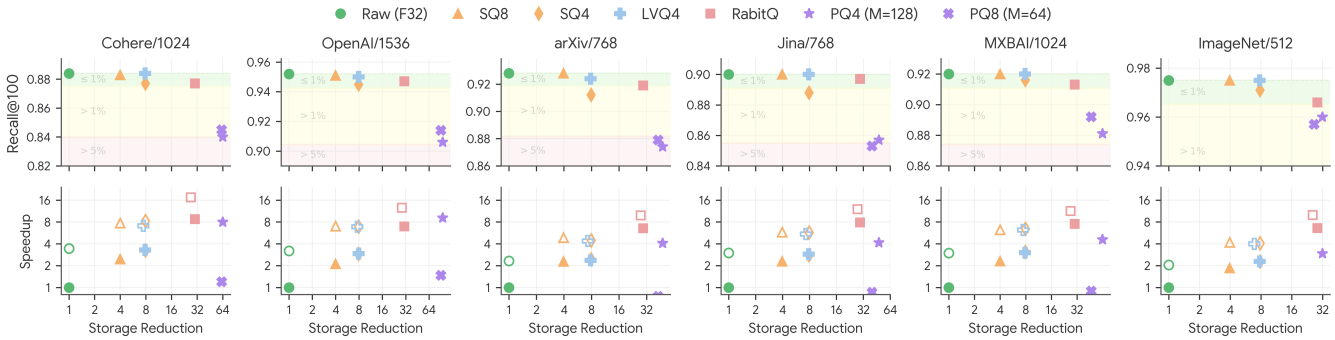

Figure 5: The effect on clustering quality (top) and speedup (bottom) of quantizing vectors before clustering. SQ8, LVQ4, and RabitQ achieve near-optimal clustering quality (within 1% of ideal). SuperKMeans (shown with hollow markers) accelerates clustering up to 17x. SuperKMeans is not shown in the top plot since recall is unaffected. Green, yellow, and red indicate ≤1%, >1%, and >5% away, resp., from the results of clustering raw vectors.

Table 3: Quality of the generated centroids for VSS tasks when clustering quantized vectors. To measure recall, we do top-100 IVF index searches by probing 1% and 3% of the clusters. *Color coding is the same as in Figure 5.*

Quant.	Build Time (s)	Build Time (s) w/ SKM	Recall @100 @1%	Vectors explored @1% ($\times 10^3$)	Recall @100 @3%	Vectors explored @3% ($\times 10^3$)	WCSS ($\times 10^6$)	Cluster Size Std. Dev.
Cohere ($N = 10M$, $d = 1024$, $k = 12649$)								
Raw	640.3 (1.0x)	186.2 (3.4x)	0.884	105.43	0.934	313.32	5.103	354
SQ8	259.3 (2.5x)	82.5 (7.8x)	0.883	105.45	0.933	313.23	5.106	352
SQ4	198.1 (3.2x)	77.4 (8.3x)	0.877	103.68	0.930	308.60	5.697	345
LVQ4	194.8 (3.3x)	90.0 (7.1x)	0.884	106.69	0.934	317.50	5.106	353
RabitQ	**73.3 (8.7x)**	**36.5 (17.5x)**	0.877	107.31	0.930	319.11	5.289	348
PQ4	80.5 (8.0x)	-	0.840	98.01	0.906	290.62	6.694	315
PQ8	532.8 (1.2x)	-	0.845	97.38	0.908	288.74	6.640	323
OpenAI ($N = 5M$, $d = 1536$, $k = 8944$)								
Raw	284.8 (1.0x)	89.0 (3.2x)	0.952	51.86	0.982	153.73	0.993	216
SQ8	134.6 (2.1x)	40.7 (7.0x)	0.951	51.87	0.982	153.75	0.994	216
SQ4	98.4 (2.9x)	41.2 (6.9x)	0.945	51.45	0.978	152.48	1.200	211
LVQ4	97.3 (2.9x)	41.3 (6.9x)	0.950	52.29	0.982	155.42	0.995	217
RabitQ	**41.0 (6.9x)**	**22.6 (12.6x)**	0.947	52.20	0.979	155.03	1.014	218
PQ4	31.3 (9.1x)	-	0.906	51.93	0.964	152.51	1.432	208
PQ8	193.5 (1.5x)	-	0.914	54.28	0.967	152.11	1.361	299
arXiv ($N = 2.25M$, $d = 768$, $k = 6003$)								
Raw	42.9 (1.0x)	18.5 (2.3x)	0.928	24.09	0.980	71.17	0.384	140
SQ8	18.7 (2.3x)	8.8 (4.9x)	0.928	24.04	0.979	71.09	0.385	140
SQ4	17.6 (2.4x)	9.6 (4.5x)	0.912	24.29	0.973	71.69	0.485	140
LVQ4	18.1 (2.4x)	9.8 (4.4x)	0.924	24.06	0.978	71.07	0.386	142
RabitQ	**6.6 (6.5x)**	**4.3 (9.9x)**	0.919	24.27	0.976	71.77	0.401	143
PQ4	10.5 (4.1x)	-	0.874	24.48	0.959	71.04	0.556	141
PQ8	56.6 (0.8x)	-	0.879	24.35	0.960	69.27	0.538	161
Jina ($N = 1.37M$, $d = 768$, $k = 4688$)								
Raw	26.0 (1.0x)	8.7 (3.0x)	0.900	14.19	0.952	42.17	0.992	135
SQ8	11.4 (2.3x)	4.5 (5.8x)	0.901	14.21	0.953	42.18	0.992	135
SQ4	9.3 (2.8x)	4.5 (5.7x)	0.888	15.46	0.943	44.27	1.138	295
LVQ4	9.1 (2.9x)	4.8 (5.5x)	0.901	14.22	0.952	42.25	0.992	136
RabitQ	**3.3 (7.9x)**	**2.2 (12.0x)**	0.897	14.23	0.950	42.20	1.039	137
PQ4	6.3 (4.2x)	-	0.857	14.37	0.928	42.64	1.496	124
PQ8	30.2 (0.9x)	-	0.853	14.75	0.924	43.11	1.488	176
MXBAI ($N = 769K$, $d = 1024$, $k = 3508$)								
Raw	14.3 (1.0x)	4.8 (3.0x)	0.920	8.60	0.970	25.25	101.334	97
SQ8	6.2 (2.3x)	2.3 (6.3x)	0.920	8.62	0.970	25.31	101.361	97
SQ4	4.6 (3.1x)	2.2 (6.5x)	0.916	8.56	0.968	25.12	113.845	95
LVQ4	4.7 (3.0x)	2.3 (6.2x)	0.922	8.72	0.971	25.68	101.447	99
RabitQ	**1.9 (7.5x)**	**1.3 (11.3x)**	0.913	8.86	0.967	26.04	105.737	101
PQ4	3.1 (4.6x)	-	0.881	8.33	0.953	24.12	151.481	88
PQ8	15.9 (0.9x)	-	0.892	8.35	0.956	23.53	141.776	105
ImageNet ($N = 1.28M$, $d = 512$, $k = 4527$)								
Raw	14.3 (1.0x)	7.0 (2.0x)	0.975	12.86	0.995	37.86	0.293	110
SQ8	7.7 (1.9x)	3.4 (4.2x)	0.976	12.88	0.995	37.89	0.293	110
SQ4	6.3 (2.3x)	3.5 (4.1x)	0.971	12.92	0.994	37.99	0.345	112
LVQ4	6.2 (2.3x)	3.6 (4.0x)	0.975	12.93	0.996	38.13	0.294	109
RabitQ	**2.2 (6.6x)**	**1.4 (10.0x)**	0.966	12.81	0.993	37.82	0.326	111
PQ4	4.9 (2.9x)	-	0.960	13.05	0.992	37.89	0.418	106
PQ8	23.4 (0.6x)	-	0.957	12.90	0.990	37.19	0.399	113

Index Materialization: The last step of our pipeline materializes the index to be used for VSS. Notably, the same representations can be used for both VSS and clustering, which avoids a round-trip to the raw vectors during index materialization. Recent studies have proposed vector indexes that utilize not only quantization but also dimensionality reduction [47,78,79,92], aligning with the principles of our pipeline. To ensure a fair comparison across techniques, we compute the *recall@k* based on cluster membership. In other words, we assume that the vectors within the probed clusters are ranked correctly. This helps us circumvent artifacts introduced by VSS pipelines, such as re-ranking. Finally, we materialize the trained centroids for the IVF index as `float32` vectors. Unless stated otherwise, these centroids are computed solely from the reconstruction of the quantized codes of the trained centroids.

Hardware and Software: We used an AMD Zen 5 EPYC 9R45 CPU (4.5 GHz) with 256GB of RAM and 32 cores (`r8a.8xlarge` in AWS). Our implementations extend the C++ SuperKMeans codebase [28]. We use 8- and 4-bit GEMM kernels found in the NumKong library [81] and `float32` GEMM kernels from OpenBLAS [59]. Our experiments use all the available cores.

4.1 Quantization Techniques

Figure 5 (top) shows the quality of the resulting centroids for vector search tasks based on the recall they yield when used as entry points for an IVF index and probing 1% of clusters. Table 3 presents these results in detail, reporting several metrics that characterize clustering quality. **SQ8** always achieves a clustering quality on-par with using raw `float32` vectors in all aspects, while providing 4x storage reduction and up to 8x speedups when paired with SuperKMeans (bottom of Figure 5). **LVQ4** and **RabitQ** achieve near-optimal clustering quality (with no more than 1% difference in recall) with 8x and 30x storage reduction, resp., while mostly maintaining cluster balance and providing significant speedups. RabitQ results in a slightly higher imbalance in cluster size. However, it provides up to 17x acceleration when paired with SuperKMeans' pruning, where the additional metadata to use SuperKMeans degrades storage reduction by around 10%. Finally, LVQ4 outperforms **SQ4** in terms of clustering quality, despite both achieving the same level of storage reduction.

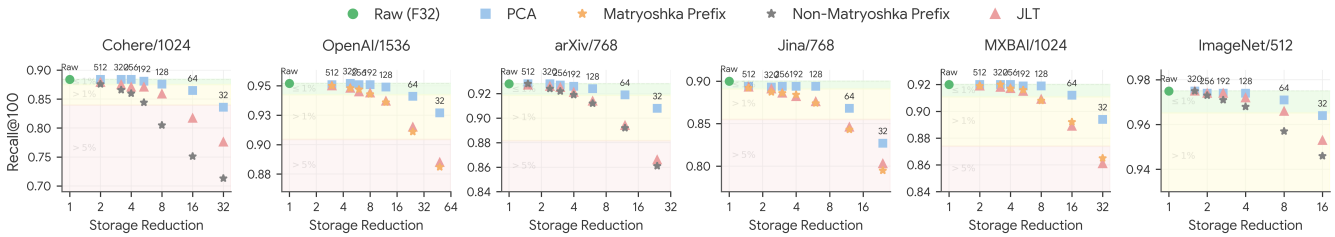

Figure 6: The effect of reducing vector dimensionality on clustering quality. PCA is the best option to maintain the quality of centroids. Dimensionality is shown as labels. *Color coding is the same as in Figure 5.*

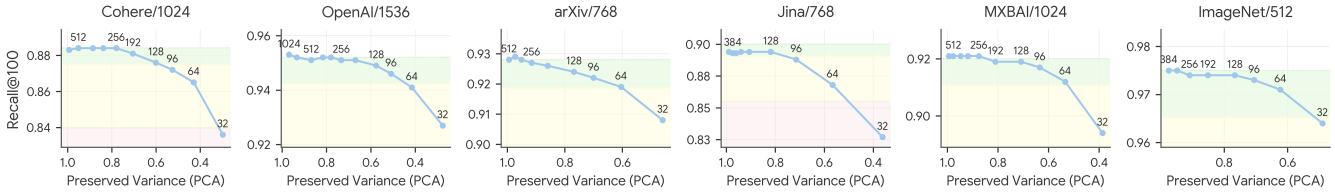

Figure 7: Around 70% of preserved variance after the PCA projection is sufficient to achieve near-optimal clustering quality (within 1%). *Color coding is the same as in Figure 5.*

Table 4: Profiling of clustering in the Cohere dataset. The additional work of quantization methods (encoding and precomputing terms) is negligible, except on PQ4 and PQ8. The phases of SuperKMeans (SKM) are broken down.

Stage	Raw	+SKM	SQ8	SQ4	LVQ4	RabitQ	PQ4	PQ8
Assignments (10 iters.)	635.6	177.3	76.4	71.6	84.3	30.9	52.9	470.9
- 1st Iteration (SuperKMeans)	-	- 31%	- 33%	- 26%	- 23%	- 22%	-	-
- Front d' (SuperKMeans)	-	- 45%	- 35%	- 43%	- 51%	- 51%	-	-
- Pruning (SuperKMeans)	-	- 24%	- 32%	- 33%	- 29%	- 28%	-	-
Precomputing Terms	-	-	-	-	0.1	0.1	-	-
Encoding	-	-	0.6	0.5	0.3	0.4	26.3	59.3
Random Rotation	-	3.2	3.3	3.3	3.2	3.3	-	-
Updating Centroids	2.3	2.3	0.8	0.6	0.7	0.6	0.8	2.1
Other	2.3	3.3	1.4	1.3	1.5	1.1	0.5	0.5
Total	640.3	186.2	82.5	77.4	90	36.5	80.5	532.8

On the other hand, **PQ** vectors provide the highest storage reduction but compromise clustering quality, affecting both recall and the balance of clusters. PQ struggles with clustering quality because it builds a global codebook from raw vectors, whereas PQ is typically applied to the residuals after clustering, allowing vectors to be centered around the clusters' means [16]. A way to improve the quality of PQ-based clustering is to use the raw vectors to update the centroids during the final k-means iteration. This brings the WCSS into the yellow zone, improving recall from 0.84 to 0.85 in the Cohere dataset, albeit at the cost of accessing the raw vectors.

Table 4 breaks down the clustering time into different phases. All techniques, except for PQ4 and PQ8, use SuperKMeans. Notably, the time spent on encoding and precomputing constant terms in SQ, LVQ, and RabitQ is negligible. In contrast, encoding consumes a significant portion of PQ's runtime, limiting its speedup. Finally, in RabitQ and LVQ, a considerable amount of time during the assignment step is consumed by the overhead of computing partial L2 distances of the front d' dimensions.

Closing RabitQ's Gap in Balance: Despite RabitQ achieving exceptional end-to-end recall, the balance of clusters is negatively impacted. Further inspection reveals that when clustering RabitQ codes, outer clusters covering the periphery of the data distribution bleed boundary points into denser in-distribution clusters–explaining the higher number of vectors explored. These outer cluster centroids are the furthest from the dataset mean (m in Equation 4), which is also the root of the issue. Recall that the decoding of RabitQ codes reconstructs each coordinate x_{r_i} as $x_{r_i} \approx m_i + \frac{1}{\sqrt{d}} \cdot \frac{||x_r - m||}{\langle \bar{x}, x \rangle} \cdot (2\bar{x}_i - 1)$; $\bar{x}_i \in \{0, 1\}$. Hereby, the decoded residual sits *on* the sphere of radius $\frac{||x_r - m||}{\langle \bar{x}, x \rangle}$ around m. By Cauchy–Schwarz, $||x_r - m||_1 \leq \sqrt{d}\ ||x_r - m||$, so this radius is always at least $||x_r - m||$. Consequently, each RabitQ-decoded vector has a systematic outward reconstruction bias from the dataset mean. When these decoded vectors are averaged to form a centroid, the bias largely cancels for inner clusters (whose constituent sign patterns are diverse) but survives for peripheral clusters (whose vectors' sign patterns are highly aligned). The result is that peripheral cluster centroids are reconstructed less precisely than central ones, overshooting outward. A simple way to counteract this effect is to use the raw vectors in the last iteration to update the centroids. This brings RabitQ clusters into the green zone across all metrics, albeit at the cost of accessing the raw vectors, which may be unfeasible if they reside in a slower storage unit.

4.2 Dimensionality Reduction Techniques

Figure 6 shows the quality of the resulting centroids for vector search tasks when dimensionality reduction techniques are applied to the vectors prior to clustering. Table 5 presents these results in detail. **PCA** is the most effective method for preserving the quality of the centroids. Figure 7 shows that preserving 60–70% of the total variance after applying PCA is sufficient to achieve clustering quality within 1% of the ideal, while maintaining 80% of the variance (3-4x reduction of d) achieves optimal quality. In contrast, the quality of **JLT** projections degrades much sooner. Using **Matryoshka** prefixes performs comparably to JLT on the OpenAI, Jina, and MXBAI datasets, whose vectors stem from a model that produces Matryoshka representations.

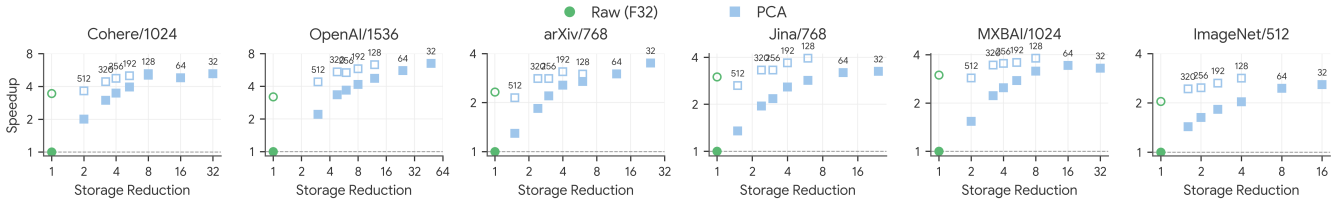

Figure 8: The effect of PCA projections on clustering speed. Speedup stops improving with thinner vectors. However, SuperKMeans makes wider vectors have comparable speedup to those of thinner vectors without the loss in quality.

Table 5: Quality of the generated centroids for VSS tasks as vector dimensionality is reduced. PCA projections achieve the highest quality. Only the datasets with Matryoshka embeddings are shown.

Tech.	Build Time (s)	Build Time (s) w/ SKM	Prep. Time (s)	Recall @100 @1%	Vectors explored @1% ($\times 10^3$)	Recall @100 @3%	WCSS ($\times 10^6$)	Cluster Size Std. Dev.
OpenAI ($N = 5M$, $d = 1536$, $k = 8944$)								
Raw	284.8 (1.0x)	89.0 (3.2x)	-	0.952	51.86	0.982	0.993	216
PCA 512	129.1 (2.2x)	64.9 (4.4x)	8.6	0.951	51.65	0.982	0.993	210
PCA 256	77.3 (3.7x)	53.2 (5.3x)	7.1	0.951	50.86	0.982	0.994	204
PCA 128	60.2 (4.7x)	45.1 (6.3x)	6.4	0.949	49.34	0.981	0.999	197
PCA 64	51.0 (5.6x)	45.1 (6.3x)	6.1	0.941	46.87	0.977	1.013	203
PCA 32	43.8 (6.5x)	37.0 (7.7x)	6.2	0.927	45.57	0.973	1.039	239
JLT 512	96.1 (3.0x)	49.6 (5.7x)	3.2	0.950	51.29	0.981	0.995	208
JLT 256	55.5 (5.1x)	42.4 (6.7x)	1.8	0.945	49.67	0.979	0.997	206
JLT 128	40.2 (7.1x)	37.9 (7.5x)	1.1	0.937	45.43	0.973	1.006	213
JLT 64	30.6 (9.3x)	32.8 (8.7x)	0.7	0.915	38.98	0.958	1.033	320
JLT 32	32.8 (8.7x)	30.5 (9.3x)	0.7	0.885	44.14	0.935	1.113	657
Mat. 512	104.3 (2.7x)	57.8 (4.9x)	-	0.950	51.39	0.981	0.994	210
Mat. 256	63.5 (4.5x)	54.0 (5.3x)	-	0.947	49.53	0.980	0.998	206
Mat. 128	45.8 (6.2x)	50.9 (5.6x)	-	0.936	45.44	0.973	1.006	218
Mat. 64	37.6 (7.6x)	45.1 (6.3x)	-	0.911	38.84	0.954	1.035	330
Mat. 32	39.0 (7.3x)	45.6 (6.2x)	-	0.881	45.47	0.930	1.12	693
Jina ($N = 1.37M$, $d = 768$, $k = 4688$)								
Raw	26.0 (1.0x)	8.7 (3.0x)	-	0.900	14.19	0.952	0.992	135
PCA 512	19.3 (1.4x)	9.9 (2.6x)	2.0	0.894	13.95	0.949	0.806	132
PCA 256	12.0 (2.2x)	7.9 (3.3x)	1.6	0.894	13.87	0.949	0.807	130
PCA 128	9.2 (2.8x)	6.6 (3.9x)	1.4	0.894	13.76	0.949	0.808	132
PCA 64	8.2 (3.2x)	6.2 (4.2x)	1.4	0.868	12.79	0.934	0.824	123
PCA 32	8.0 (3.2x)	5.8 (4.5x)	1.6	0.827	12.29	0.909	0.866	133
JLT 512	17.2 (1.5x)	8.8 (3.0x)	0.8	0.893	13.87	0.948	0.807	131
JLT 256	10.0 (2.6x)	7.4 (3.5x)	0.4	0.886	13.53	0.943	0.811	131
JLT 128	7.4 (3.5x)	7.1 (3.7x)	0.3	0.876	12.68	0.935	0.819	132
JLT 64	6.5 (4.0x)	5.9 (4.5x)	0.2	0.846	11.04	0.909	0.842	158
JLT 32	6.5 (4.0x)	5.7 (4.5x)	0.1	0.803	11.42	0.872	0.912	255
Mat. 512	15.2 (1.7x)	7.9 (3.3x)	-	0.892	13.89	0.947	0.808	131
Mat. 256	8.7 (3.0x)	6.9 (3.8x)	-	0.886	13.51	0.945	0.811	129
Mat. 128	6.2 (4.2x)	7.0 (3.7x)	-	0.874	12.55	0.933	0.82	130
Mat. 64	5.0 (5.2x)	5.8 (4.5x)	-	0.843	11.07	0.906	0.843	156
Mat. 32	4.7 (5.6x)	5.6 (4.7x)	-	0.795	11.42	0.865	0.915	258
MXBAI ($N = 769K$, $d = 1024$, $k = 3508$)								
Raw	14.3 (1.0x)	4.8 (3.0x)	-	0.920	8.60	0.970	101.334	97
PCA 512	9.3 (1.5x)	5.0 (2.9x)	1.4	0.921	8.60	0.970	101.233	97
PCA 256	5.7 (2.5x)	4.0 (3.5x)	1.2	0.921	8.46	0.970	101.354	92
PCA 128	4.5 (3.2x)	3.8 (3.8x)	1.1	0.919	8.20	0.970	101.916	88
PCA 64	4.2 (3.4x)	3.8 (3.7x)	1.1	0.912	7.93	0.967	103.450	81
PCA 32	4.3 (3.3x)	3.8 (3.7x)	1.4	0.894	7.69	0.961	107.044	86
JLT 512	7.6 (1.9x)	3.9 (3.6x)	0.5	0.919	8.51	0.969	101.421	95
JLT 256	4.2 (3.4x)	3.1 (4.5x)	0.3	0.917	8.33	0.968	101.852	92
JLT 128	3.1 (4.6x)	2.6 (5.4x)	0.2	0.909	7.90	0.963	102.667	90
JLT 64	2.6 (5.4x)	2.6 (5.5x)	0.2	0.889	7.11	0.949	104.984	97
JLT 32	2.4 (5.8x)	2.5 (5.6x)	0.2	0.861	6.98	0.928	112.599	154
Mat. 512	6.9 (2.1x)	3.3 (4.3x)	-	0.919	8.50	0.969	101.507	96
Mat. 256	3.9 (3.7x)	2.8 (5.1x)	-	0.917	8.37	0.968	101.793	92
Mat. 128	2.9 (4.9x)	2.6 (5.4x)	-	0.908	7.95	0.964	102.674	89
Mat. 64	2.4 (6.0x)	2.3 (6.2x)	-	0.892	7.19	0.952	105.051	96
Mat. 32	2.6 (5.6x)	2.5 (5.6x)	-	0.865	7.12	0.930	112.6	161

The effect of dimensionality reduction on clustering speed is shown in Figure 8 and Table 5. The preprocessing time for PCA is the highest, while remaining negligible relative to the total clustering time. Notably, if the vector search pipeline requires PCA-projected vectors (e.g., MRQ [92]), this preprocessing would otherwise occur at index materialization time. On the other hand, Matryoshka representations eliminate preprocessing requirements. It is important to note that speedup is sublinear with respect to dimensionality reduction, a known phenomenon in GEMM routines [55]. Remarkably, SuperKMeans delivers modest speedups, ranging from 20% to 2x. Figure 8 shows that SuperKMeans accelerates the clustering of wider vectors to achieve similar speedups to those of thinner vectors without sacrificing quality. Finally, note that SuperKMeans is disabled when $d < 128$, as pruning can be detrimental for performance when approaching this threshold [42].

Dimensionality Reduction and Vector Search: Reducing the dimensionality of the raw vectors will affect index materialization, leading to three possible approaches: 1) recompute the full-dimensional centroids using cluster membership (requires access to raw vectors), 2) unproject centroids back to the original dimensionality (not possible when using Matryoshka vectors), and 3) utilize projected centroids for vector search in the reduced space [85]. The results shown in this section use approach #1. Table 6 shows a comparison of all strategies. Both unprojected and projected centroids yield lower clustering quality than recomputed full-dimensional centroids because energy from discarded dimensions is lost. However, the difference between the approaches is minimal at higher levels of preserved variance, allowing the pipeline to avoid accessing the raw vectors entirely after preprocessing. Previous studies have shown the feasibility of using projected vectors and centroids for vector search [85,92], although these approaches still require storing the residuals of the PCA projection for re-ranking.

Table 6: Quality of the generated centroids when using three different strategies to materialize the centroids: recomputing centroids at full-d (best), unprojecting the centroids to the original dimensionality, and using the projected centroids.

Dim.	Preserved Variance	Recall@100@1%		
		Recomputed	Unprojected	Projected
Cohere ($N = 10M$, $d = 1024$, $k = 12649$)				
Raw	1.00	0.884	0.884	0.884
768	0.99	0.883	0.883	0.883
320	0.84	0.884	0.882	0.882
256	0.78	0.884	0.880	0.880
192	0.71	0.881	0.874	0.874
128	0.60	0.876	0.864	0.864
64	0.43	0.865	0.826	0.826
32	0.30	0.836	0.718	0.718

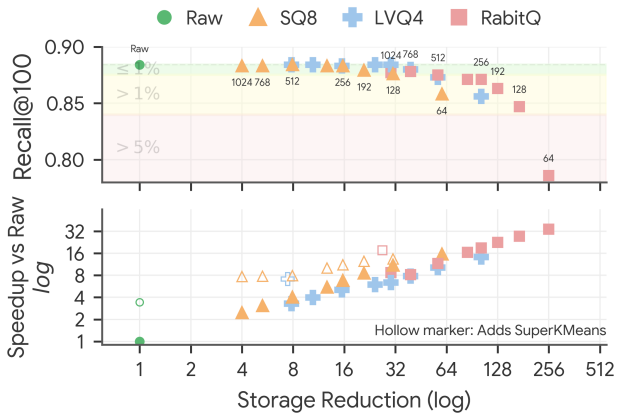

Figure 9: The effect on clustering quality (top) and speedup (bottom) of combining dimensionality reduction and quantization before clustering.

4.3 Mixing Techniques

Figure 9 shows the result of combining techniques in our largest dataset (Cohere/1024). We present results for SQ8, LVQ4, and RabitQ, combined with PCA as dimensionality-reduction method. If maintaining quality equivalent to clustering raw vectors is essential, LVQ with 80% of preserved variance is the best choice. However, if quality can deviate by up to 1% from the optimal value, then RabitQ with 80% of PCA variance becomes the preferred option. For prioritizing speed while still aiming for near-optimal quality, RabitQ combined with SuperKMeans is the optimal choice. It is worth noting that using RabitQ with PCA and SuperKMeans can degrade performance due to the additional floating-point operations required to compute partial L2 distances for pruning, particularly for thinner vectors. In contrast, SQ8 consistently benefits from SuperKMeans because its distance calculations do not require any additional adjustments.

4.4 Other Accelerators

Hierarchical k-Means: Hierarchical k-means [74] has recently emerged as an alternative for clustering large collections of vector embeddings [1,36,42,54,68,82,91] because it reduces the complexity of vanilla k-means from $O(N*k*d)$ to $O(N*\sqrt{k}*d)$ by dividing the clustering process into two phases: MESO- and FINE-CLUSTERING. In the MESO-CLUSTERING phase, a coarser clustering is performed, creating only $\sqrt{k}$ clusters. In the subsequent FINE-CLUSTERING phase, k-means is applied within each meso-cluster, with the number of clusters set to $\sqrt{n_i}$, where n_i is the number of points in the i-th meso-cluster. Finally, an optional refinement iteration of vanilla k-means can be performed to improve clustering quality further. Table 7 shows the quality of the resulting centroids for vector search tasks on our two largest datasets, Cohere and OpenAI, when using hierarchical k-means. The performance gains are remarkable, reaching up to 91x when hierarchical k-means is combined with RabitQ. Note that hierarchical k-means produces clusters with more uniform sizes in dense regions of the space, as indicated by the lower number of vectors explored. Nonetheless, this improved balance negatively affects recall and requires probing more clusters.

Table 7: Quality of the generated centroids for VSS using hierarchical k-means. Hierarchical k-means achieves remarkable speedups while producing more balanced clusters.

Accelerator	Build Time (s)	Recall @100 @1%	Vectors explored @1% ($\times 10^3$)	Recall @100 @3%	Vectors explored @3% ($\times 10^3$)	WCSS ($\times 10^6$)	Cluster Size Std.Dev.
Cohere ($N = 10M$, $d = 1024$, $k = 12649$)							
Vanilla k-means (Raw)	640.3 (1.0x)	0.884	105.4	0.934	313.3	5.10	354
Hierarchical (Raw)	13.6 (47.1x)	0.854	88.1	0.914	262.8	5.26	343
Hierarchical (SQ8)	9.4 (68.1x)	0.853	87.9	0.913	261.6	5.26	348
Hierarchical (LVQ4)	8.2 (78.1x)	0.853	88.4	0.913	263.8	5.26	358
Hierarchical (RabitQ)	**7.0 (91.5x)**	0.844	87.7	0.905	261.7	5.43	339
OpenAI ($N = 5M$, $d = 1536$, $k = 8944$)							
Vanilla k-means (Raw)	284.8 (1.0x)	0.952	51.9	0.982	153.7	0.99	216
Hierarchical (Raw)	11.5 (24.8x)	0.939	46.5	0.977	136.2	1.02	224
Hierarchical (SQ8)	8.3 (34.3x)	0.939	46.7	0.976	137.1	1.02	220
Hierarchical (LVQ4)	7.5 (38.0x)	0.937	46.6	0.975	136.5	1.02	222
Hierarchical (RabitQ)	**6.7 (42.5x)**	0.933	46.4	0.973	136.0	1.04	220

Table 8: Quality of the generated centroids for VSS tasks when using hierarchical k-means and graph-based assignments with HNSW.

Accelerator	Build Time (s)	Build Time (s) w/ SKM	Recall @100 @1%	Vectors explored @1% ($\times 10^3$)	Recall @100 @3%	Vectors explored @3% ($\times 10^3$)	WCSS ($\times 10^6$)	Cluster Size Std.Dev.
Cohere ($N = 10M$, $d = 1024$, $k = 12649$)								
Raw	640.3 (1.0x)	186.2 (3.4x)	0.884	105.4	0.934	313.3	5.10	354
Hierarchical	13.6 (47.1x)	13.4 (47.9x)	0.854	88.1	0.914	262.8	5.26	343
Hierarchical + Refine	28.8 (22.2x)	26.4 (24.2x)	0.866	91.8	0.921	274.4	5.19	331
Graph-based (efs=4)	24.3 (26.4x)		0.861	119.2	0.920	352.7	5.29	453
Graph-based (efs=8)	35.6 (18.0x)		0.875	111.7	0.930	331.0	5.17	371
Graph-based (efs=16)	59.2 (10.8x)		0.881	107.5	0.932	319.1	5.13	362
OpenAI ($N = 5M$, $d = 1536$, $k = 8944$)								
Raw	284.8 (1.0x)	89.0 (3.2x)	0.952	51.9	0.982	153.7	0.99	216
Hierarchical	11.5 (24.8x)	11.3 (25.3x)	0.939	46.5	0.977	136.2	1.02	224
Hierarchical + Refine	17.8 (16.0x)	16.7 (17.1x)	0.945	47.6	0.980	140.5	1.01	213
Graph-based (efs=4)	32.4 (8.8x)		0.934	58.1	0.973	175.6	1.02	254
Graph-based (efs=8)	51.1 (5.6x)		0.946	54.2	0.979	161.6	1.00	221
Graph-based (efs=16)	83.8 (3.4x)		0.95	52.8	0.981	156.9	1.00	219

Graph-based Assignments: Graph-based indexes for VSS have recently been used to determine assignments during clustering [73, 85]. This approach builds a graph-based vector index over the centroids (e.g., HNSW [52]). Consequently, determining assignments becomes a series of top-1 searches on the graph index. Both hierarchical k-means and graph-based assignments help clustering scale to larger datasets. Table 8 compares their performance on Cohere and OpenAI—our two largest datasets. For the graph-based assignment approach, we created an HNSW index using `ef_construction` = 128 and `M`=16. We use a vanilla implementation of HNSW from the FAISS library [70]. Additionally, we incorporated an optimization that starts the graph traversal from the centroid assigned in the previous k-means iteration [85].

Both methods significantly accelerate clustering compared to vanilla k-means. However, graph-based assignments produce less balanced clusters, leading to more vectors explored during VSS. Additionally, the graph-based approach introduces the added complexity of tuning the `ef_search` parameter, which controls search quality. If this value is set too low, it improves speed but compromises clustering quality. Conversely, if set too high, it reduces speed improvements. Note that the graph construction over the centroids takes less than 0.1% of the total runtime.

Sampling: Sampling reduces clustering runtime proportionally to the fraction of vectors sampled from the data [9,42]. Sampling around 20–30% of the data is sufficient to maintain clustering quality and balance [42]. Sampling can be used in conjunction with our proposed pipeline. However, note that the final assignment step and index materialization still need to use all vectors.

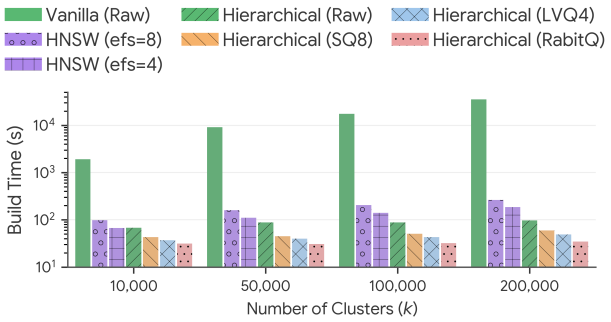

Figure 10: Clustering runtime scaling with the number of clusters. Both hierarchical k-means and graph-based assignments scale sublinearly with k. Hierarchical k-means with RabitQ provides the lowest clustering time.

4.5 Scalability

The optimal ratio of points per cluster varies depending on the application. Recent studies suggest that a lower points-per-cluster ratio (i.e., a higher k) is beneficial for vector search indexes [85]. Consequently, the scalability of our pipeline with respect to k is critical. For this scalability experiment, we scale our Cohere dataset to 50M embeddings [11], totaling 200GB of data, and run our pipeline while progressively increasing the number of clusters to create (k). Additionally, we scale our machine to 512GB of RAM and 64 cores (`r8a.16xlarge` in AWS). Figure 10 shows the results of this experiment. Hierarchical k-means with RabitQ vectors is the fastest approach: even when creating 200K clusters, centroid training remains under 1 minute. Notably, both hierarchical k-means and graph-based assignments scale sublinearly with respect to the number of clusters. This sublinear scalability is essential for indexing large vector collections, which would otherwise require several hours. The latter, combined with the clustering of quantized vectors, facilitates scalability in both storage requirements and runtime, all while maintaining index quality (as shown in Table 7).

5 DISCUSSION

Vector indexing through clustering remains the preferred method for large-scale cloud vector systems [1,12,62,90], where every second of compute counts towards billing. The insights presented in this study show that using full-precision vectors for clustering is excessive: systems can improve the performance of data ingestion pipelines by first applying approximation techniques [85], wherein method selection can be guided by the algorithms used in the vector search pipeline. For instance, systems using LVQ for vector search [30] should apply the encoding before the clustering pipeline. The same applies to dimensionality reduction techniques, as recent studies propose combining them with quantization [47,78,79,85,92]. Ultimately, our design allows the same vector representations to be used for both vector search and clustering, enabling more streamlined indexing and ingestion in vector systems. The latter opens new research directions toward quantization techniques that can efficiently serve a dual purpose: efficient search *and* indexing.

6 CONCLUSIONS AND FUTURE WORK

We have introduced a clustering pipeline for indexing vector embeddings in which approximation techniques commonly used for vector search are applied *before* clustering. Our experiments indicate that clustering is highly resilient to approximation techniques and show that even aggressive encodings (RabitQ), combined with dimensionality reduction (PCA) and dimension pruning (SuperKMeans), can reduce storage by up to 60x and substantially accelerate clustering, all while maintaining near-optimal clustering quality. Another striking feat of our study is the adaptation of quantization techniques for clustering pipelines and their integration with dimension pruning.

In future work, we aim to explore how the availability of specialized hardware units for specific data types (e.g., Intel's AMX) can lead to different trade-offs between speed and storage reduction. Additionally, replicating our study with graph-based indexes (e.g., HNSW, DiskANN) remains a future research opportunity attractive for systems that implement graph-based vector indexes. In the context of the graph-based assignment approach, exploring the performance of different graph indexes or developing novel indexes tailored for this purpose could yield significant benefits. Finally, an evaluation using multi-vector embeddings [38] and classic vector datasets that do not stem from AI embedding models (e.g., SIFT, GIST) would broaden the applicability of our pipeline.

REFERENCES

[1] 2025. turbopuffer: Serverless Vector and Full-Text Search on Object Storage. https://turbopuffer.com/ Accessed: 2025-12-24.

[2] Hervé Abdi and Lynne J Williams. 2010. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics* 2, 4 (2010), 433–459.

[3] Cecilia Aguerrebere, Ishwar Bhati, Mark Hildebrand, Mariano Tepper, and Ted Willke. 2023. Similarity search in the blink of an eye with compressed indices. *arXiv preprint arXiv:2304.04759* (2023).

[4] Cecilia Aguerrebere, Mark Hildebrand, Ishwar Singh Bhati, Theodore Willke, and Mariano Tepper. 2024. Locally-Adaptive Quantization for Streaming Vector Search. *arXiv preprint arXiv:2402.02044* (2024).

[5] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2016. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. In *42nd International Conference on Very Large Data Bases*, Vol. 9. 12.

[6] David Arthur and Sergei Vassilvitskii. 2006. *k-means++: The advantages of careful seeding.* Technical Report. Stanford.

[7] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. Elpis: Graph-based similarity search for scalable data science. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1548–1559.

[8] Christos Boutsidis, Anastasios Zouzias, and Petros Drineas. 2010. Random projections for k-means clustering. *Advances in neural information processing systems* 23 (2010).

[9] Junyu Chen. 2025. *How We Made 100M Vector Indexing in 20 Minutes Possible on PostgreSQL.* https://blog.vectorchord.ai/how-we-made-100m-vector-indexing-in-20-minutes-possible-on-postgresql VectorChord Blog.

[10] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[11] Cohere Labs. 2024. *TREC-RAG 2024 Corpus (MSMARCO 2.1) - Encoded with Cohere Embed English v3).* https://huggingface.co/datasets/CohereLabs/msmarco-v2.1-embed-english-v3 MS MARCO v2.1 / TREC-RAG 2024 corpus embedded using Cohere Embed English v3; contains 113M+ passage embeddings.

[12] Databricks. 2026. *Decoupled by Design: Billion-Scale Vector Search.* https://www.databricks.com/blog/decoupled-design-billion-scale-vector-search Accessed: 2026-05-15.

[13] Chencheng Deng, Weiling Yang, Jianbin Fang, and Dezun Dong. 2025. Demystifying ARM SME to Optimize General Matrix Multiplications. *arXiv preprint arXiv:2512.21473* (2025).

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition.* Ieee, 248–255.

[15] Liwei Deng, Penghao Chen, Ximu Zeng, Tianfu Wang, Yan Zhao, and Kai Zheng. 2024. Efficient data-aware distance comparison operations for high-dimensional approximate nearest neighbor search. *arXiv preprint arXiv:2411.17229* (2024).

[16] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).

[17] Dwango Media Village. 2017. *pqkmeans: Product Quantization k-means clustering.* https://github.com/DwangoMediaVillage/pqkmeans Accessed: 2026-04-30.

[18] Félix Cloutier. n.d.. PSHUFB — Packed Shuffle Bytes. https://www.felixcloutier.com/x86/pshufb x86 instruction reference (SSSE3/AVX/AVX-512); Accessed: 2026-05-06.

[19] Félix Cloutier. n.d.. VPDPBUSD — Multiply and Add Unsigned and Signed Bytes. https://www.felixcloutier.com/x86/vpdpbusd x86 instruction reference; Accessed: 2026-05-06.

[20] Edward W Forgy. 1965. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *biometrics* 21 (1965), 768–769.

[21] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2024. Practical and Asymptotically Optimal Quantization of High-Dimensional Vectors in Euclidean Space for Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2409.09913* (2024).

[22] 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 (2023), 1–27.

[23] 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 (2024), 1–27.

[24] Nicholas Gates, Robert Kruszewski, and Will Manning. 2026. *Vortex.* https://github.com/vortex-data/vortex

[25] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. 2023. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*. 3406–3416.

[26] Google. 2026. ruy: The ruy Matrix Multiplication Library. https://github.com/google/ruy. GitHub repository, accessed 2026-05-31.

[27] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. 2025. SymphonyQG: towards symphonious integration of quantization and graph for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[28] CWI Database Architectures Group. 2026. *SuperKMeans.* https://github.com/cwida/SuperKMeans A super-fast k-means library for high-dimensional vector embeddings.

[29] 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.

[30] Intel. 2025. Scalable Vector Search. https://github.com/intel/ScalableVectorSearch. GitHub repository, accessed 2026-05-15.

[31] Intel Corporation. 2024. What Is Intel® Advanced Matrix Extensions (Intel® AMX)? https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-amx.html Accessed: 2026-05-06.

[32] Elias Jääsaari, Ville Hyvönen, Matteo Ceccarello, Teemu Roos, and Martin Aumüller. 2025. VIBE: Vector Index Benchmark for Embeddings. *arXiv preprint arXiv:2505.17810* (2025).

[33] Elias Jääsaari, Ville Hyvönen, and Teemu Roos. 2024. Lorann: Low-rank matrix factorization for approximate nearest neighbor search. *Advances in Neural Information Processing Systems* 37 (2024), 102121–102153.

[34] 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).

[35] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[36] Yicheng Jin, Yongji Wu, Wenjun Hu, Bruce Maggs, Jun Yang, Xiao Zhang, and Danyang Zhuo. 2026. Curator: Efficient Vector Search with Low-Selectivity Filters. *Proceedings of the ACM on Management of Data* 4, 1 (SIGMOD (2026), 1–27.

[37] William B Johnson, Joram Lindenstrauss, et al. 1984. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary mathematics* 26, 189-206 (1984), 1.

[38] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 39–48.

[39] Hyungyo Kim, Gaohan Ye, Nachuan Wang, Amir Yazdanbakhsh, and Nam Sung Kim. 2024. Exploiting intel advanced matrix extensions (AMX) for large language model inference. *IEEE Computer Architecture Letters* 23, 1 (2024), 117–120.

[40] Daria Kryvosheieva, Saba Sturua, Michael Günther, Scott Martens, and Han Xiao. 2025. Efficient Code Embeddings from Code Generation Models. arXiv:2508.21290 [cs.CL] https://arxiv.org/abs/2508.21290

[41] Leonardo Kuffo and Peter Boncz. 2025. Bang for the Buck: Vector Search on Cloud CPUs. In *Proceedings of the 21st International Workshop on Data Management on New Hardware*. 1–8.

[42] Leonardo Kuffo, Sven Hepkema, and Peter Boncz. 2026. A Super Fast K-means for Indexing Vector Embeddings. *arXiv preprint arXiv:2603.20009* (2026).

[43] Leonardo Kuffo, Elena Krippner, and Peter Boncz. 2025. PDX: A Data Layout for Vector Similarity Search. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[44] Leonardo Kuffo, Ioanna Tsakalidou, Roberta De Viti, Albert Angel, Jiří Iša, and Rastislav Lenhardt. 2026. Semantic Recall for Vector Search. In *Proceedings of the 49th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 3907–3912.

[45] Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, et al. 2022. Matryoshka representation learning. *Advances in Neural Information Processing Systems* 35 (2022), 30233–30249.

[46] Sean Lee, Aamir Shakir, Darius Koenig, and Julius Lipp. 2024. *Open Source Strikes Bread – New Fluffy Embeddings Model.* https://www.mixedbread.ai/blog/mxbai-embed-large-v1

[47] Hui Li, Shiyuan Deng, Xiao Yan, Xiangyu Zhi, and James Cheng. 2025. SAQ: Pushing the Limits of Vector Quantization through Code Adjustment and Dimension Segmentation. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–25.

[48] Zhaoheng Li, Wei Ding, Silu Huang, Zikang Wang, Yuanjin Lin, Ke Wu, Yongjoo Park, and Jianjun Chen. 2025. Cloud-Native Vector Search: A Comprehensive Performance Analysis. *arXiv preprint arXiv:2511.14748* (2025).

[49] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[50] Duo Lu, Helena Caminal, Manos Chatzakis, Yannis Papakonstantinou, Yannis Chronis, Vaibhav Jain, and Fatma Özcan. 2026. An in-depth study of filter-agnostic vector search on a postgresql database system:[experiments & analysis]. *Proceedings of the ACM on Management of Data* 4, 3 (SIGMOD (2026), 1–26.

[51] Vasilis Mageirakos, Bowen Wu, and Gustavo Alonso. 2025. Cracking Vector Search Indexes. *arXiv preprint arXiv:2503.01823* (2025).

[52] 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.

[53] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. Parlayann: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 270–285.

[54] Silvio Martinico, Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. 2026. Efficient Multivector Retrieval with Token-Aware Clustering and Hierarchical Indexing. In *Proceedings of the 49th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 3982–3987.

[55] Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, Joël Falcou, and Jack Dongarra. 2016. High-performance matrix-matrix multiplications of very small matrices. In *European Conference on Parallel Processing*. Springer, 659–671.

[56] Yusuke Matsui, Keisuke Ogaki, Toshihiko Yamasaki, and Kiyoharu Aizawa. 2017. Pqk-means: Billion-scale clustering for product-quantized codes. In *Proceedings of the 25th ACM international conference on Multimedia*. 1725–1733.

[57] Jason Mohoney, Devesh Sarda, Mengze Tang, Shihabur Rahman Chowdhury, Anil Pacaci, Ihab F Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. 2025. Quake: Adaptive Indexing for Vector Search. *arXiv preprint arXiv:2506.03437* (2025).

[58] Kasper Overgaard Mortensen, Fatemeh Zardbani, Mohammad Ahsanul Haque, Steinn Ymir Agustsson, Davide Mottin, Philip Hofmann, and Panagiotis Karras. 2023. Marigold: efficient k-means clustering in high dimensions. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1740–1748.

[59] OpenBLAS Contributors. 2026. *OpenBLAS: An optimized BLAS library.* https://github.com/OpenMathLib/OpenBLAS Version 0.3.33.

[60] Weston Pace, Chang She, Lei Xu, Will Jones, Albert Lockett, Jun Wang, and Raunak Shah. 2025. Lance: Efficient Random Access in Columnar Storage through Adaptive Structural Encodings. *arXiv preprint arXiv:2504.15247* (2025).

[61] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of vector database management systems. *arXiv preprint arXiv:2310.14021* (2023).

[62] Yannis Papakonstantinou, Alan Li, Ruiqi Guo, Sanjiv Kumar, and Phil Sun. 2024. *ScaNN for AlloyDB.* Technical Report. Google Cloud. https://services.google.com/fh/files/misc/scann_for_alloydb_whitepaper.pdf Whitepaper.

[63] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. Acorn: Performant and predicate-agnostic search over vector embeddings and structured data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[64] Jeffrey Pound, Floris Chabert, Arjun Bhushan, Ankur Goswami, Anil Pacaci, and Shihabur Rahman Chowdhury. 2025. MicroNN: An On-device Disk-resident

Updatable Vector Database. In *Companion of the 2025 International Conference on Management of Data*. 608–621.

[65] Qdrant. 2023. *arXiv Titles Instructor XL Embeddings (768-dim, 2.25M vectors).* https://huggingface.co/datasets/Qdrant/arxiv-titles-instructorxl-embeddings 768-dimensional InstructorXL embeddings generated from arXiv paper titles.

[66] Qdrant. 2024. *DBpedia Entities with OpenAI text-embedding-3-large (1536-dim, 1M vectors).* https://huggingface.co/datasets/Qdrant/dbpedia-entities-openai3-text-embedding-3-large-1536-1M 1 million DBpedia entities embedded using OpenAI text-embedding-3-large (1536 dimensions).

[67] Vansh Ramani, Alexis Schlomer, Akash Nayar, Sayan Ranu, Jignesh M Patel, and Panagiotis Karras. 2025. Panorama: Fast-Track Nearest Neighbors. *arXiv preprint arXiv:2510.00566* (2025).

[68] RAPIDS AI. 2025. rapidsai/cuVS: GPU-Accelerated Vector Search and Clustering Library. https://github.com/rapidsai/cuvs. Accessed: 2026-01-21.

[69] RAPIDS AI. 2026. *K-Means — cuVS C++ API Documentation (stable 26.04).* RAPIDS cuVS documentation. Accessed: 2026-01-26.

[70] Meta Research. 2024. *Faiss: A library for efficient similarity search and clustering of dense vectors.* https://github.com/facebookresearch/faiss

[71] Keshav Santhanam, Omar Khattab, Christopher Potts, and Matei Zaharia. 2022. PLAID: an efficient engine for late interaction retrieval. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 1747–1756.

[72] Jifan Shi, Jianyang Gao, James Xia, Tamás Béla Fehér, and Cheng Long. 2026. GPU-Native Approximate Nearest Neighbor Search with IVF-RaBitQ: Fast Index Build and Search. *arXiv preprint arXiv:2602.23999* (2026).

[73] Jack Spalding-Jamieson, Eliot Wong Robson, and Da Wei Zheng. 2025. Scalable k-Means Clustering for Large k via Seeded Approximate Nearest-Neighbor Search. *arXiv preprint arXiv:2502.06163* (2025).

[74] Michael Steinbach, George Karypis, and Vipin Kumar. 2000. A comparison of document clustering techniques. (2000).

[75] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. 2023. SOAR: improved indexing for approximate nearest neighbor search. *Advances in Neural Information Processing Systems* 36 (2023), 3189–3204.

[76] Kento Tatsuno, Daisuke Miyashita, Taiga Ikeda, Kiyoshi Ishiyama, Kazunari Sumiyoshi, and Jun Deguchi. 2024. AiSAQ: All-in-Storage ANNS with Product Quantization for DRAM-free Information Retrieval. *arXiv preprint arXiv:2404.06004* (2024).

[77] Selim Furkan Tekin and Rajesh Bordawekar. 2025. B+ ANN: A Fast Billion-Scale Disk-based Nearest-Neighbor Index. *arXiv preprint arXiv:2511.15557* (2025).

[78] Mariano Tepper, Ishwar Singh Bhati, Cecilia Aguerrebere, Mark Hildebrand, and Ted Willke. 2023. LeanVec: Searching vectors faster by making them fit. *arXiv preprint arXiv:2312.16335* (2023).

[79] Mariano Tepper, Ishwar Singh Bhati, Cecilia Aguerrebere, and Ted Willke. 2024. GleanVec: Accelerating vector search with minimalist nonlinear dimensionality reduction. *arXiv preprint arXiv:2410.22347* (2024).

[80] Unity Technologies. n.d.. Unity.Burst.Intrinsics.Arm.Neon.vdotq_s32 Method. https://docs.unity3d.com/Packages/com.unity.burst@1.6/api/Unity.Burst.Intrinsics.Arm.Neon.vdotq_s32.html Arm NEON dot-product intrinsic (SDOT equivalent); Accessed: 2026-05-06.

[81] Ash Vardanian. [n.d.]. *NumKong: 2000 Mixed Precision Kernels For All.* https://github.com/ashvardanian/NumKong

[82] Thomas Veasey. 2025. *K-means for building vector indices.* https://www.elastic.co/search-labs/blog/k-means-for-vector-indices Elastic Search Labs blog post.

[83] Bohai Wang, Yanhao Wang, Huiqi Hu, Weichen Zhao, and Minghao Zhao. 2026. Distance Comparison Operation Optimization in ANNS: A Survey and Experimental Evaluation.. In *EDBT*. 578–591.

[84] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[85] Yan Wang, Jian Zhou, Sai Huang, Chao Dou, Hanwen Tian, Zhijie Jiang, Zongning Zhang, Xiaoqi Li, Zhencan Peng, Chunhui Shen, et al. 2026. LindormVector: A Distributed Vector Engine on a Cloud-Native Multi-Model NoSQL Database. In *Companion of the International Conference on Management of Data*. 451–463.

[86] Jiuqi Wei, Xiaodong Lee, Zhenyu Liao, Themis Palpanas, and Botao Peng. 2025. Subspace Collision: An Efficient and Accurate Framework for High-dimensional Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–29.

[87] Jiuqi Wei, Xiaodong Lee, Botao Peng, Quanqing Xu, Chuanhui Yang, and Themis Palpanas. 2026. PDET-LSH: Scalable In-Memory Indexing for High-Dimensional Approximate Nearest Neighbor Search with Quality Guarantees. *IEEE Transactions on Knowledge and Data Engineering* (2026).

[88] Jiadong Xie, Jeffrey Xu Yu, and Yingfan Liu. 2025. Fast approximate similarity join in vector databases. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[89] Haike Xu, Magdalen Dobson Manohar, Philip A Bernstein, Badrish Chandramouli, Richard Wen, and Harsha Vardhan Simhadri. 2025. In-place updates of a graph index for streaming approximate nearest neighbor search. *arXiv preprint arXiv:2502.13826* (2025).

[90] Qian Xu, Feng Zhang, Chengxi Li, Lei Cao, Zheng Chen, Jidong Zhai, and Xiaoyong Du. 2025. Harmony: A scalable distributed vector database for high-throughput approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 3, 4 (2025), 1–28.

[91] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. 2023. Spfresh: Incremental in-place update for billion-scale vector search. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 545–561.

[92] Mingyu Yang, Liuchang Jing, Wentao Li, and Wei Wang. 2024. Quantization Meets Projection: A Happy Marriage for Approximate k-Nearest Neighbor Search. *arXiv preprint arXiv:2411.06158* (2024).

[93] Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. 2025. Turboquant: Online vector quantization with near-optimal distortion rate. *arXiv preprint arXiv:2504.19874* (2025).

[94] Alibek Zhakubayev and Greg Hamerly. 2022. Clustering faster and better with projected data. In *Proceedings of the 6th International Conference on Information System and Data Mining*. 1–6.

[95] Alibek Zhakubayev and Greg Hamerly. 2024. Using Annealing to Accelerate Triangle Inequality k-means. In *2024 IEEE 11th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 1–9.

[96] Zhuanglin Zheng, Yuxiang Zeng, Chenchen Liu, Yunzhen Chi, Binhan Yang, and Yongxin Tong. 2026. Distance Comparison Operations Are Not Silver Bullets in Vector Similarity Search: A Benchmark Study on Their Merits and Limits. *arXiv preprint arXiv:2604.02801* (2026).

[97] Jiaxu Zhu, Jiayu Yuan, Kaiwen Yang, Xiaobao Chen, Shihuan Yu, Hongchang Lv, Yan Li, and Bolong Zheng. 2025. An Experimental Evaluation of Hybrid Querying on Vectors. *Proceedings of the VLDB Endowment* 19, 2 (2025), 183–195.