DiskVecLab: A Deployment-Realistic Evaluation Framework for Disk-Based Vector Search

Yuxiang Wang
Beihang University
Beijing, China
yuxiangwang@buaa.edu.cn

Yongxin Tong*
Beihang University
Beijing, China
yxtong@buaa.edu.cn

Shuyuan Li
City Univ. of Hong Kong
Hong Kong, China
shuyuan.li@cityu.edu.hk

Zimu Zhou
City Univ. of Hong Kong
Hong Kong, China
zimuzhou@cityu.edu.hk

Ziyuan He
Beihang University
Beijing, China
hzy_he@buaa.edu.cn

Yu Sun
Nankai University
Tianjin, China
sunyu@nankai.edu.cn

Xianhai Li
Independent Researcher
Beijing, China
smartlxh@gmail.com

Yu Li
Beihang University
Beijing, China
carp84@gmail.com

ABSTRACT

Disk-based vector search has become a critical building block for unstructured data management in modern data centers, where memory constraints make fully in-memory solutions impractical. Despite rapid advances, its performance under realistic deployment conditions and across the full end-to-end pipeline remains insufficiently understood. In this work, we present a systematic evaluation of disk-based vector search that addresses these gaps. We identify three critical components at the core of disk-based vector search, segmentation, in-segment index & search, and quantization, and provide a unified pipeline and taxonomy that capture the design space of state-of-the-art methods. To enable fair and deployment-realistic analysis, we build DiskVecLab, a modular evaluation framework that decouples these components and supports controlled ablations and end-to-end comparisons across diverse storage environments, concurrency regimes, and query distributions. Using DiskVecLab, we conduct an extensive empirical study on six large-scale datasets across diverse deployment environments. Our results reveal that *(i)* segmentation can introduce severe I/O amplification, *(ii)* in-segment optimizations are regime-dependent and often fail to translate into end-to-end gains under I/O saturation or out-of-distribution queries, and *(iii)* no single quantization strategy dominates across memory and I/O configurations. Motivated by these findings, we further distill guidelines and future directions for disk-based vector search.

PVLDB Reference Format:
Yuxiang Wang, Yongxin Tong, Shuyuan Li, Zimu Zhou, Ziyuan He, Yu Sun, Xianhai Li, and Yu Li. DiskVecLab: A Deployment-Realistic Evaluation Framework for Disk-Based Vector Search. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Roy-buaa/DiskVecLab-Artifacts.

*Yongxin Tong is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

High-dimensional vectors have become a primary representation for unstructured data in modern data systems. Across many pipelines, text, images, videos, and other multimodal objects are encoded as embeddings with hundreds to thousands of dimensions, so that semantic similarity can be quantified by vector proximity [8]. Vector search then retrieves (approximate) nearest neighbors over these embeddings, serving as a core primitive for querying and managing unstructured content [38]. It is now widely deployed in *data centers* to support web-scale retrieval augmented generation (RAG) [36], data curation for multimodal large-model training [44, 48], and enterprise analytics such as duplicate detection [43].

Meanwhile, embedding corpora routinely reach tens to hundreds of millions, and recently billions of vectors [15, 16, 26], making it prohibitively expensive to keep all vectors in main memory (DRAM). This pressure is amplified by the *rising cost* of memory provisioning in data center infrastructures [14]. As a result, large-scale systems increasingly rely on *disk-based* approximate nearest neighbor search (ANNS) [12, 26], which stores full-precision vectors (and typically most index structures) on cheaper solid-state drives (SSDs) while retaining only compact metadata and compressed codes in memory. By trading additional I/O for a much smaller DRAM footprint, disk-based ANNS has become a practical building block for billion-scale vector search under realistic memory budgets (*e.g.*, Google Vertex AI [37] and Elasticsearch [1]).

Evaluation Gaps. Disk-based ANNS has seen rapid progress in algorithms and systems [10, 12, 23, 26, 47, 54, 57, 64]. Yet our understanding of their *performance* is incomplete along two dimensions.

First, data-center storage and workload realism is under-explored. Disk-based vector search *in production* is increasingly shaped by the following characteristics.

- **Compute-storage disaggregation**, where SSD storage is decoupled to remote servers for elasticity and fault tolerance, resulting in higher access latency and reduced effective I/O bandwidth [52, 55].
- **High-concurrency execution** (*e.g.*, up to 128 threads [7, 29, 56]) to utilize available computing resources, which can quickly saturate I/O bandwidth.
- **Diverse query distributions**, particularly out-of-distribution (OOD) queries arising from cross-modality retrieval (*e.g.*, text-to-image search [42]), which alter access patterns [11].

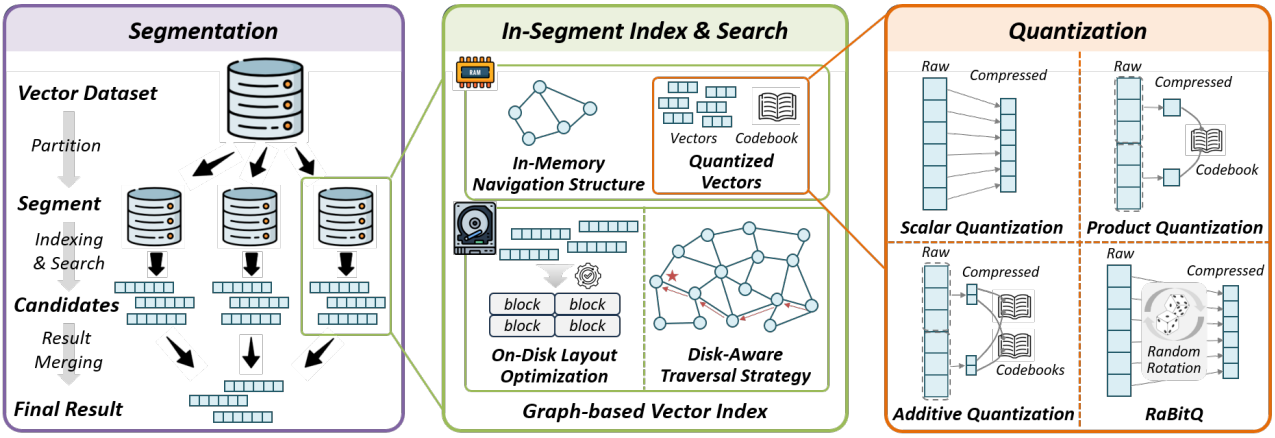


Figure 1: Unified pipeline for disk-based vector search.

However, most studies [12, 23, 26, 57] adopt *simplified* assumptions, *i.e.*, locally attached SSDs, low thread counts, and in-distribution queries, leaving it unclear whether design choices that appear optimal in these settings remain optimal in realistic deployments.

Second, pipeline comparability is incomplete. Compared to in-memory counterparts [31, 33], disk-based vector search must balance large-scale data with disk I/O constraints, which expands and complicates the end-to-end design space. To facilitate systematic analysis, we propose a *unified pipeline* for disk-based ANNS with three key components (see Figure 1).

- **Segmentation** partitions vectors into segments that are indexed and searched independently [10, 54, 57], and is central to improving disk-based scalability and manageability.
- **In-segment index and search** is often graph-based [5, 58] as in-memory ANNS, yet incorporates disk-oriented optimizations such as navigation structures [57], layout adjustments [30, 57, 64], and traversal strategies [23, 62].
- **Quantization** compresses vectors into compact representations for approximate distance computation [19], and is indispensable in disk-based settings to reduce SSD reads.

Yet, prior evaluations [23, 64] often emphasize in-segment index and search while leaving segmentation and quantization under-characterized. As a result, it remains unclear how to select appropriate design choices for each component.

Our Approach. To address these limitations, we propose and implement DiskVecLab, a *modular* evaluation framework for disk-based ANNS pipelines guided by two principles.

- **Deployment-realistic evaluation.** We evaluate disk-based ANNS across a deployment-oriented matrix that varies *storage settings*, *concurrency regimes*, and *query distributions*, so empirical observations reflect data center behaviors.
- **Pipeline-complete evaluation.** DiskVecLab abstracts existing disk-based ANNS into a unified pipeline, and decouples it into clean interfaces to enable *controlled component-level ablations* and *curated end-to-end comparisons*.

Benchmark Implementation. DiskVecLab supports representative segmentation strategies, including natural [54, 57], clustering [12, 27], and tree-based segmentation [4]. It integrates representative quantization families, including scalar [20, 21], product [22, 27], additive [35] and RaBitQ [20, 21], and exposes common hooks for in-segment index [57, 64] and search optimizations [23, 62].

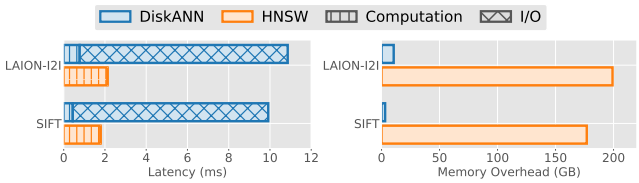
Using DiskVecLab, we conduct a systematic study on *six* large-scale datasets (all ≥50M vectors, including billion-scale datasets), under both *in-* and *out-of-distribution* query settings. We evaluate multiple storage configurations (*locally attached and remote SSDs*) as well as a wide range of concurrency levels, covering both I/O-abundant and I/O-saturated regimes.

Summary of Findings. Our results yield actionable observations that directly inform deployment decisions.

- **Segmentation leads to I/O amplification.** Segmenting the dataset, when coupled with existing merging strategies [13, 41, 57], can cause substantial I/O amplification during search and degrade query performance, while this overhead can be alleviated by adaptively exploring segments.
- **In-segment optimizations are regime-dependent.** Several in-segment optimizations [23, 57, 64] are highly workload- and environment-dependent. Their benefits can shrink under OOD queries or I/O bandwidth saturation, indicating that per-segment performance improvements do not necessarily translate into *deployment-realistic* end-to-end gains.
- **Quantization is not one-size-fits-all.** The best quantization strategy depends on the memory budget and the storage environment, and its impact can shift when the system transitions from compute-limited to I/O-limited regimes.

Contributions. We make the following contributions:

- We identify two gaps for evaluating the performance of disk-based vector search, *i.e.*, *data-center storage/workload realism* and *end-to-end pipeline comparability*.
- We provide a *unified pipeline and taxonomy* that covers the design space of state-of-the-art methods and facilitates systematic analysis of disk-based vector search.
- We implement and open-source DiskVecLab, a *modular* framework that decouples segmentation, in-segment index & search, and quantization, enabling controlled ablations and deployment-realistic end-to-end comparisons.



(a) Latency Breakdown **(b) Memory Overhead**

Figure 2: Comparison of search performance and memory overhead for in-memory HNSW [33] and disk-based DiskANN [26] on 50M-vector LAION-I2I dataset [44] and 100M-vector subset of SIFT dataset [27].

- We provide a systematic empirical study across six large-scale datasets, various storage configurations, concurrency regimes, and query distributions.
- We provide evidence-based guidelines for disk-based vector search, including an adaptive segment exploration strategy to mitigate segmentation-induced I/O amplification.

2 PROBLEM SETTING

2.1 Approximate kNN Search

Let $\mathcal{D} = \{v_1, \ldots, v_N\} \subset \mathbb{R}^d$ be a set of N high-dimensional vectors and $q \in \mathbb{R}^D$ a query vector. Given a distance function $dis(\cdot,\cdot) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ and a result size k, a *k nearest neighbor (kNN)* query returns the k vectors in $\mathcal{D}$ with the smallest distances to q [38].

Exact kNN requires computing $dis(q, v)$ for all $v \in \mathcal{D}$, which incurs a linear scan cost and becomes prohibitive at scale [25]. Therefore, practical systems adopt **approximate k nearest neighbor search (ANNS)**, which trades exactness for efficiency and returns an approximate result set $AkNN(\mathcal{D}, q, k)$ with substantially higher query efficiency [3, 31].

ANNS accuracy is commonly evaluated by *recall rate*:

$$\text{Recall rate} = \frac{|kNN(\mathcal{D}, q, k) \cap AkNN(\mathcal{D}, q, k)|}{k}. \quad (1)$$

Higher recall rate indicates that the approximate results better preserve the true nearest neighbors. The goal of ANNS is to achieve better tradeoff between query efficiency and accuracy.

2.2 Disk-Based Vector Search

Disk-Based Setting. We focus on **disk-based ANNS**, where full-precision vectors and most index structures reside on SSDs, while only compressed vectors and the working set are maintained in DRAM. In contrast to in-memory ANNS [31, 33], where query time is often dominated by distance computations, disk-based ANNS is primarily constrained by I/Os between main memory and storage. Although disk-based solutions incur a much smaller memory footprint, they require a large amount of disk reads, and I/Os can dominate the end-to-end latency (see Figure 2). Accordingly, the optimization target shifts to reducing the *I/Os per query*, while preserving high recall.

Scope. Our scope is *single-node* disk-based ANNS, which commonly acts as the local search engine in distributed deployments [10, 54].

- **Relationship to In-Memory Settings.** Widely used in-memory indexes such as HNSW [33] assume that both vectors and index structures reside entirely in DRAM, making search largely *compute-bound*. In contrast, disk-based setting is dominated by I/O behavior and operates under tight DRAM budgets. These differences motivate a *dedicated* evaluation of disk-based ANNS, rather than extrapolating from in-memory results.
- **Relationship to Distributed Settings.** Another approach to large-scale vector search is a distributed architecture [10, 54]. Distributed and disk-based ANNS address *orthogonal* concerns. Distributed methods scale out across machines, while disk-based methods improve per-node I/O efficiency. In practice, distributed deployments frequently rely on SSD-backed local search engines to control cost, and thus benefit directly from stronger disk-based ANNS [40, 41].

Differences from Existing Benchmarks. Several prior studies benchmark vector search algorithms and systems [3, 5, 31, 34, 58], but they primarily target in-memory solutions. More recently, BigVectorBench [28] compares vector database systems end-to-end, but it does not provide component-level ablation analysis under deployment-realistic environments. In contrast, we target disk-based vector search specifically and benchmark key design choices within a unified pipeline, enabling controlled comparisons and actionable guidance for real deployments.

3 DESIGN CHOICES

3.1 Unified Pipeline for Disk-Based ANNS

Despite the diversity of disk-based ANNS algorithms and systems [12, 23, 26, 57], most designs follow a common end-to-end structure. We abstract them into a *unified* three-stage pipeline (see Figure 1) to make this structure explicit and to support systematic comparisons.

- **Segmentation**: The dataset is partitioned into segments, each with its own in-segment index. Segmentation is largely *disk-specific*. It bounds the working set, enables parallel search, supports segment-level pruning, and simplifies index construction and maintenance.
- **In-Segment Index and Search**: Within each segment, an ANNS index is constructed to generate candidates efficiently. Most methods adopt graph-based primitives [5, 58], but naive traversal incurs highly random disk accesses. Hence, disk-based methods augment the core primitive with *disk-oriented optimizations*, including navigation structures, layout optimizations, and traversal strategies.
- **Quantization**: Vectors are compressed into compact codes that support fast approximate distance computation. Quantization becomes *especially important* when full-precision vectors reside on disk than in memory.

These stages are *complementary* in disk-based vector search. Segmentation provides a disk-centric mechanism to control which data regions are accessed. Quantization provides an in-memory surrogate for distance computation to reduce disk reads. In-segment index/search remains the core ANNS component but must be adapted to mitigate random I/O through disk-aware optimizations.

This unified view also guides our evaluation design. Many recent disk-based ANNS methods adopt similar choices for *segmentation* and *quantization* and focus their novelty on disk-optimized

Table 1: Comparison of design choices in recent disk-based vector search methods.

Method	Segmentation	In-Segment Index and Search			Quantization
		Navigation Structure	*Layout Optimization*	*Traversal Strategy*	
DiskANN [26]	Natural	None	None	Strict best-first search	PQ
Starling [57]	Natural	Vamana	Inter-vector	Strict best-first search	PQ
MARGO [64]	Natural	Vamana	Inter-vector	Strict best-first search	PQ
PipeANN [23]	Natural	Vamana	None	Relaxed best-first search	PQ
Gorgeous [62]	Natural	Vamana	Intra-Vector	Relaxed best-first search	PQ

in-segment search (see Table 1). Therefore, beyond end-to-end comparisons, we include targeted *ablations over segmentation and quantization choices* (see Sec. 5.2 and Sec. 5.4) to quantify their impact on recall-latency tradeoffs and to identify when these choices become bottlenecks under deployment-realistic constraints.

3.2 Segmentation

Segmentation partitions the dataset into multiple segments and treats each segment as an independent search unit. After *data segmentation*, a query typically proceeds in three steps: *(i) segment pruning* selects a subset of segments to probe based on the query, *(ii) in-segment search* retrieves candidates within each probed segment, and *(iii) cross-segment merge* combines local candidates into a global top-k. In disk-based ANNS, segmentation is central because it controls the probing granularity, enables parallelism, and determines whether pruning can be applied *before* incurring SSD reads.

Segmentation Strategies. Most methods differ in whether partitioning leverages *similarity* information.

- **Natural segmentation** partitions vectors by arrival time or identifiers into roughly equal-sized segments, and queries need to probe all the segments. Each segment performs a local search to return its top-k candidates, which are merged and reranked by exact distances to produce the final top-k. This approach is simple, robust to dynamic updates, and widely used in practice [10, 54, 57].
- **Similarity-aware segmentation** groups similar vectors into the same segment for *segment-level pruning*. A common implementation is clustering with segment representatives *e.g.*, k-means by centroids [27]. At query time, the algorithm ranks representatives by distance to the query and probes only *the most promising* segments, thus reducing SSD reads. Tree-based segmentation [4] follows the same principle via hierarchical routing to a small set of candidate leaves, which are then searched and merged.

Segmentation Granularity. Segmentation granularity (the number and size of segments) is orthogonal to the strategy above, but it directly affects disk I/O.

- **Coarse segmentation** uses tens of segments, each containing hundreds of thousands to millions of vectors [10, 54, 57]. Segment-level pruning is limited, and efficiency relies on in-segment index and search (Sec. 3.3). In practice, coarse segmentation is typically combined with graph-based in-segment indexing, and is therefore often referred to as graph-based disk vector search [23, 26, 57].
- **Fine segmentation** uses thousands of smaller segments, each containing hundreds to thousands of vectors [12], and relies heavily on segment-level pruning to improve efficiency. It follows the inverted-file (IVF) paradigm [27] and is often referred to as IVF-based disk vector search [12].

Current Practices. As shown in Table 1, most disk-based ANNS systems adopt *natural segmentation* by default and focus on in-segment index optimization (Sec. 3.3), leaving segmentation design choices underexplored. In Sec. 5.2, we therefore *(i)* quantify the sensitivity of the natural baseline to segmentation granularity, *(ii)* test similarity-aware segmentation can achieve better accuracy-efficiency tradeoffs via segment-level pruning, and *(iii)* evaluate a lightweight *adaptive segment exploration* policy that reallocates query effort across segments (instead of uniform probing) while keeping other components unchanged.

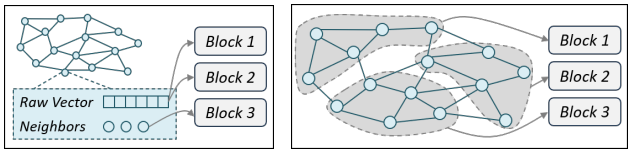
3.3 In-Segment Index and Search

In-segment index/search is the *core retrieval engine* inside each segment. State-of-the-art disk-based ANNS adopts **graph-based indexes**, but naive graph traversal incurs many *random* SSD reads and quickly becomes I/O-dominated. Thus existing methods augment the graph-based index with three disk-oriented optimizations:

- **In-memory navigation structure** to start closer to the target region and shorten disk-resident search paths.
- **On-disk layout optimization** to improve locality at the block/page granularity.
- **Disk-aware traversal strategy** to better coordinate I/O and computation and sustain high I/O utilization.

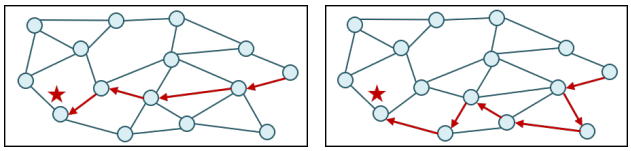
Common Primitive: Graph-based Vector Index. Given a vector set D, a graph-based index constructs a proximity graph G over D, where each vertex corresponds to a vector and edges connect neighbors under a chosen rule. At query time, the search performs greedy (*i.e.*, best-first) exploration by iteratively expanding promising vertices toward the query. Graph-based indexes such as HNSW [33] and NSG [18] demonstrate strong performance in in-memory environments. However, when transferred to disk-resident settings, the inherent properties of graph traversal result in frequent and inter-dependent disk accesses, necessitating disk-aware optimizations for in-segment graph-based indexing and searching.

In-Memory Navigation Structures. A common approach is to build a small navigation structure in DRAM to reduce the portion of the disk graph need to be traversed. Most methods sample a subset of vectors and construct an in-memory graph for coarse routing and entry-point selection. For example, Starling [57] uses Vamana [26]



(a) (b)

Figure 3: Comparison of (a) intra-vector and (b) inter-vector layout optimization over graph-based vector index.



(a) (b)

Figure 4: Comparison of traversal strategies over graph-based vector index: (a) strict and (b) relaxed best-first search.

as navigation graph [26]. Later solutions such as PipeANN [23], MARGO [64], and Gorgeous [62] follow a similar design.

On-Disk Layout Optimization. Layout determines how vectors and auxiliary index structures (*e.g.*, neighbor lists) are stored on SSD. Since data is fetched at fixed-size block/page granularity, co-locating information that tends to be accessed together can improve locality and reduce block reads. Existing techniques generally fall into *intra-vector* and *inter-vector* layout optimization (see Figure 3).

- **Intra-vector layout optimization** organizes the data associated with a *single* vector, mainly the raw vector and its neighbor list (see Figure 3a). A widely used design stores the raw vector and its neighbor list together [23, 26, 57]. Gorgeous [62] proposes a variant that, for each vector v, co-locates not only v and its neighbor list, but also the neighbor lists of v's neighbors to increase useful work per block read.
- **Inter-vector layout optimization** reorders *different* vectors on disk so that vertices likely to be accessed together are physically close and more likely to fall into the same block (see Figure 3b). For example, Starling [57] performs block shuffling to increase the chance that a vertex and its neighbors co-reside in the same block. MARGO [64] extends this idea by incorporating edge access frequency into the objective and using a greedy heuristic for placement.

Disk-Aware Traversal Strategies. Traversal in graph-based index alternates between *(i)* issuing disk reads and *(ii)* computing distances to decide the next read request. To reduce search latency, modern disk-based vector search systems typically overlap these two stages. Existing approaches mainly differ in how aggressively they pipeline computation and I/O, particularly in whether the next disk reads can be issued before the current results are fully processed (see Figure 4).

- **Strict best-first search** selects the next vertices to read strictly by their (fully computed) distances among all discovered candidates, as in-memory algorithms. DiskANN [26] adopts this strategy by expanding a fixed-size batch of vertices in each iteration. Starling [57] and MARGO [64] improve it by overlapping computation with I/O, exploiting the fact that layout optimization can bring multiple useful candidates in the same block.
- **Relaxed best-first search** schedules new reads based on partially processed results to keep the I/O pipeline busy, trading stricter exploration for lower end-to-end latency. For example, PipeANN [23] maintains an I/O request queue and enqueues the best available candidates whenever slots free up. Gorgeous [62] adopts a similar idea at the block level by proactively refilling a block-loading queue to sustain high I/O concurrency.

Current Practices. As summarized in Table 1, disk-oriented optimizations for in-segment search are the most extensively studied and diverse aspect of disk-based ANNS. Prior evaluations [57, 62, 64] have compared different strategies, and showed that in-memory navigation structures and layout optimization improve throughput by reducing I/Os, while relaxed best-first search lowers search latency. However, these conclusions are derived under simplified experimental settings. We revisit these strategies under data-center configurations and quantify how their relative benefits shift across storage environments and concurrency regimes (Sec. 5.3).

3.4 Quantization

Quantization compresses vectors into compact codes for *fast approximate distance evaluation*. In disk-based ANNS, it is widely used to reduce the DRAM footprint of vector storage, thereby avoiding (or amortizing) SSD reads and improving throughput under limited memory budgets. We focus on four representative quantization approaches [16, 19]: *scalar quantization (SQ)*, *product quantization (PQ)*, *additive quantization (AQ)*, and *RaBitQ*.

Scalar Quantization. SQ compresses a vector by quantizing each dimension independently to a low-precision value (*e.g.*, 4-8 bits). Approximate distances are then computed directly between the quantized vector and the query. Classic SQ [39, 54] uses uniform bins per dimension, while LVQ [2] adopt adaptive bin boundaries.

Product Quantization. PQ improves accuracy at a similar memory budget by quantizing at the *subvector* level. It splits a D-dimensional vector into m subvectors and encodes each subvector by the identifier of its nearest centroid in a learned codebook (*a.k.a.*, codeword). PQ enables fast distance estimation via lookup tables. For each query subvector, distances to all centroids are precomputed, and the approximate distance to a database vector is obtained by summing m table lookups. Classic PQ [27] learns codebooks with k-means, while OPQ [22] improves accuracy by applying a learned rotation before codebook construction.

Additive Quantization. AQ generalizes PQ by representing subvectors as the *sum* of multiple codewords, reducing approximation error under high compression. Distance estimation remains lookup-based but accumulates multiple codewords per subvector, increasing computation. Besides, construction of AQ is also more challenging, since selecting good codebook for large-scale dataset is non-trivial. LSQ [35] mitigates this cost by learning codebooks sequentially and encoding greedily, improving scalability while retaining good accuracy.

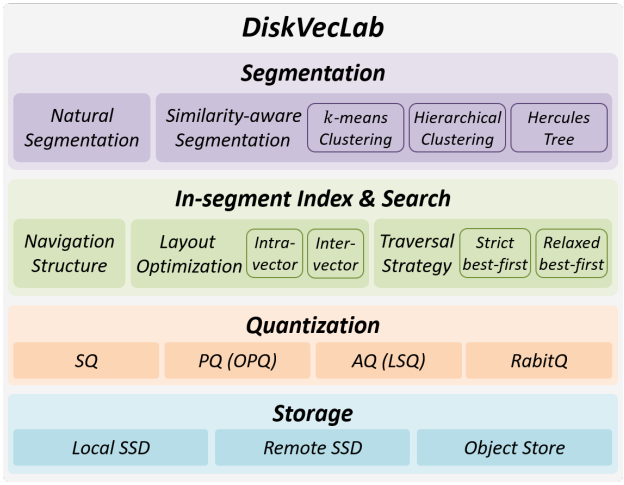


Figure 5: Overview of DiskVecLab framework.

RabitQ. RaBitQ applies a random orthogonal transformation to redistribute information across dimensions before quantization, improving approximation quality under a fixed memory budget [20, 21]. Distance estimation is performed using an unbiased estimator rather than direct reconstruction from quantized values. Compared with conventional scalar quantization, this design achieves higher accuracy while retaining efficient batched computation.

Current Practices. As summarized in Table 1, current disk-based vector search systems predominantly adopt PQ-based methods, while alternative approaches remain largely unexplored. Prior comparisons [20, 21] focus on in-memory indexes and do not capture the interaction between quantization error, graph traversal, and SSD accesses in disk-resident pipelines. Therefore, in Sec. 5.4, we integrate SQ, PQ, AQ and RaBitQ into disk-based pipelines and evaluate them end-to-end across multiple compression rates.

4 DISKVECLAB EVALUATION FRAMEWORK

4.1 Framework Implementation

We implement DiskVecLab (Figure 5), an evaluation framework for disk-based vector search that enables deployment-realistic benchmarking over a broad design space. DiskVecLab decouples the mainstream disk-based vector search pipeline *i.e.*, segmentation, in-segment index & search and quantization, and executes diverse combinations under a unified runtime and storage backend.

Execution Flow. At construction time, DiskVecLab partitions the dataset into segments, builds an in-segment index for each segment, and stores the vector data and index on disk. At query time, it optionally prunes segments, searches the selected segments using a specified traversal strategy, and merges candidates into the final top-k result. This unified execution makes the impact of individual design choices comparable across cross-product combinations.

Implemented Design Space. Based on the above interfaces, DiskVecLab implements representative strategies in each stage.

- **Segmentation.** We support four representative methods: *(i)* **Natural** in Starling [57] and other systems [13, 41]; *(ii)* k**-means Clustering** in IVF indexing [16, 27]; *(iii)* **Balanced Hierarchical Clustering** in SPANN [12] and other systems [50]; and *(iv)* **Hercules Tree** in ELPIS [4].
- **In-segment Index and Search.** Within each segment, DiskVecLab supports graph-based indexes with interchangeable *layout* and *traversal* strategies. We support four layouts: DiskANN [26], Starling [57], MARGO [64], and Gorgeous [62]. On top of these layouts, we support four traversal strategies: two strict best-first searches (DiskANN [26] and Starling [57]) and two relaxed variants (PipeANN [23] and Gorgeous [62]). DiskVecLab also exposes a switch to enable/disable the *in-memory navigation structure*.
- **Quantization.** We support four representative schemes spanning all families: *(i)* **SQ** [16], a simple quantization scheme used in many systems [10, 54]); *(ii)* **OPQ** [22], a widely used PQ scheme in prior systems [26, 49, 65]; *(iii)* **LSQ** [35], an AQ method that achieves high accuracy with compact representations; and *(iv)* **RaBitQ** [20], the state-of-the-art quantization method for in-memory vector search (we integrate its official implementation [32]).

Storage Abstraction. We implement a unified storage abstraction layer with pluggable I/O backends to support multiple deployments, including locally attached SSDs, remote SSDs on disaggregated storage, which are our primary focus. It also supports object storage [51, 61] as an additional backend for extensibility. The layer provides consistent read primitives and unified measurement hooks (*e.g.*, I/O counters and latency breakdowns).

Extensibility and Reproducibility. Building an extensible implementation is non-trivial because many systems hard-code cross-stage dependencies. For example, SPANN [12] is coupled with PQ [27], and extending DiskANN to different quantization formats (*e.g.*, RaBitQ [20]) requires substantial refactoring. In contrast, DiskVecLab isolates each stage behind stable interfaces and executes all pipelines through the same storage backends. Hence, integrating a new strategy typically requires only a small adapter for one interface rather than reworking the full pipeline. We have open-sourced DiskVecLab at https://github.com/Roy-buaa/DiskVecLab-Artifacts.

4.2 Experiment Setup

We base our evaluation on DiskVecLab and conduct experiments under deployment-realistic configurations, as summarized below.

Table 2: Statistics of datasets.

Dataset	Type	Card.	Dim.	Query Distribution
LAION-I2I [44]	Image	50M	768	in-distribution
LAION-T2I [45]	Image/Text	50M	512	out-of-distribution
DEEP [6]	Image	100M	96	in-distribution
Text2Image [17]	Image/Text	100M	200	out-of-distribution
SIFT [27]	Image	1B	128	in-distribution
SpaceV [9]	Text	1B	100	in-distribution

Datasets and Workloads. We evaluate on six widely used large-scale datasets (each containing at least 50 million vectors), reflecting

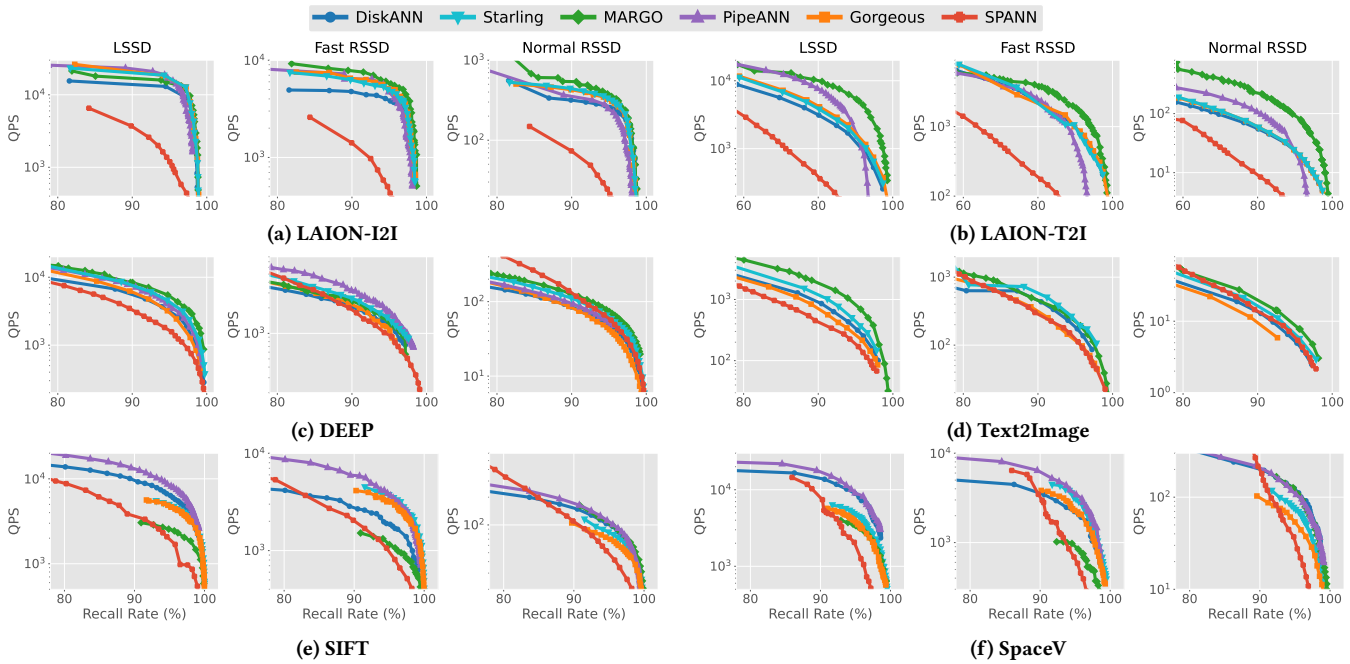


Figure 6: End-to-end search performance across datasets and storage environments.

Table 3: Pipeline settings for each experiment group.

	Segmentation	In-Segment Index & Search	Quantization
Sec. 5.1	Natural (Except SPANN)	Method Default	OPQ (16×)
Sec. 5.2	Varied	DiskANN	
Sec. 5.3	Natural	Varied	
Sec. 5.4		DiskANN	Varied

data-center deployments for retrieval over web-scale image and text corpora [34, 57]. To capture practical workload variation, we include both in-distribution and out-of-distribution query sets. Table 2 shows the dataset statistics. We use inner product distance for Text2Image and Euclidean distance for the remaining datasets.

Metrics. We measure the efficiency-accuracy tradeoff using: *Queries Per Second (QPS)*, *i.e.*, processed queries per second; *Recall Rate*, the accuracy against ground-truth nearest neighbors (Equation (1)); and *Mean I/Os Per Query*, the average number of disk I/O operations (*i.e.*, 4KB block read requests) per query. We additionally report end-to-end *index construction time* and *storage overhead*.

Hardware/Software and Storage Environments. We run all experiments on servers with two Intel Xeon Platinum 8369B CPUs (64 cores and 128 hardware threads in total) and 512 GB DRAM, under three storage environments: (1) *Locally Attached SSDs (LSSD)*, commonly assumed in prior evaluations [12, 26, 57] but less prevalent in production due to limited elasticity and weaker disaster recovery; and (2-3) *Remote SSDs on disaggregated storage (RSSD)* at two performance tiers, *i.e.*, Fast RSSD (up to 400k I/O per second) and Normal RSSD (up to 10k I/O per second), which are widely adopted in production deployments on Alibaba Cloud [46]. These tiers differ in I/O throughput and latency, reflecting practical cloud options across different budget levels. All methods are implemented in C++ and compiled with GCC 13.3 using `-O3` with SIMD enabled, and we require result size $k = 10$. In all experiments, we enable `O_DIRECT` to bypass the operating system page cache and clear the page cache before each run, thereby minimizing the impact of cached data. We systematically investigate different design choices in disk-based ANNS pipelines in Sec. 5, and Table 3 summarizes the complete configuration used in each experiment group.

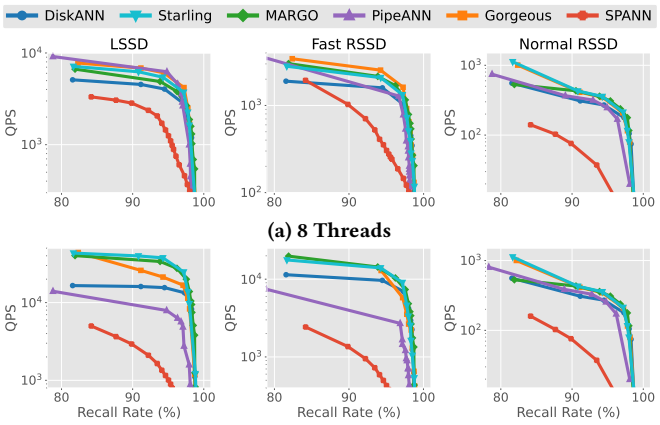
5 EXPERIMENTAL EVALUATION

5.1 End-to-End Performance Comparison under Deployment-Realistic Environment

We start with an end-to-end comparison of six *state-of-the-art* methods in under *deployment-realistic* environments. Since these methods mainly differ in *in-segment optimization* (see Table 1), our goal is not to dissect their designs, but to identify the *environment regimes* where performance tradeoffs deviate from conclusions from the prior experiments. These observations motivate our subsequent investigations in Sec. 5.3 (in-segment optimizations) and the orthogonal design choices in Sec. 5.2 (segmentation) and Sec. 5.4 (quantization). We focus on four questions:

- How do QPS-recall tradeoffs change across different storage environments?
- How does throughput scale with *concurrency*?
- Are the methods sensitive to *query distribution shift*?
- Do *index construction* scale to billion-vector datasets?

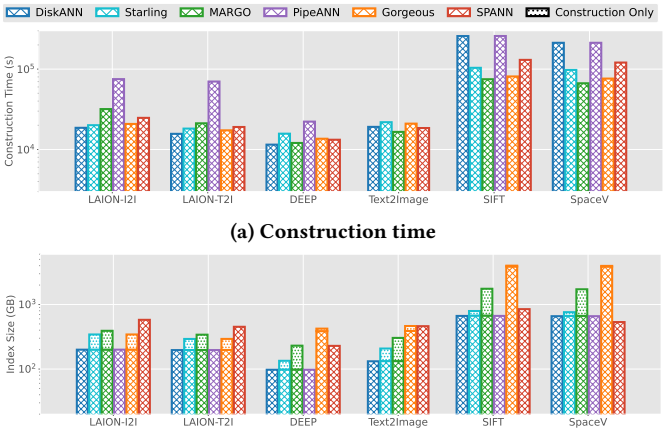
To ensure fair comparison, we use the same graph construction parameters for all methods, following the default settings of the base graph in DiskANN [26]: maximum degree $R = 128$ and candidate



(a) 8 Threads

(b) 128 Threads

Figure 7: End-to-end search performance under different storage environments and concurrency levels.



(a) Construction time

(b) Storage overhead

Figure 8: Construction time and storage across datasets.

neighbor list size $L_{\text{build}} = 125$. For methods that maintain an in-memory navigation graph, we set $R_{\text{mem}} = 24$, $L_{\text{membuild}} = 125$, and the sampling rate $\mu = 0.001$. Method-specific parameters follow the original papers: Starling uses $\beta = 16$ block-shuffling iterations, MARGO uses $nlist = 256$ for graph layout optimization, PipeANN uses an I/O pipeline width of $W = 4$, and SPANN [12] uses $\epsilon_1 = 10$, $\epsilon_2 = 8$, and a maximum replication factor of 8. We report QPS-recall curves by varying the search list size L.

Storage Environments. We evaluate all methods on six datasets across the three storage environments using 32 threads. Figure 6 shows that moving from LSSD to Fast RSSD reduces throughput by 6.0%-74.9%, and further to Normal RSSD by 83.7%-97.9%. Notably, the IVF-based method, SPANN performs relatively better in the Normal RSSD environment, as it incurs fewer random disk I/Os during search and is better suited to I/O-constrained scenarios. However, its performance still lags behind methods that employ in-segment graph indexes. At the billion scale (SIFT and SpaceV), we also observe feasibility constraints: some methods must adopt finer-grained segmentation to ensure their layout optimization step fits within memory. This increases the number of segments and consequently degrades their achievable end-to-end performance.

Concurrency Scaling. We vary concurrency from 8 to 128 threads across all three storage environments to examine the relationship between performance and concurrency. Figure 7 reports results on LAION-I2I [45] dataset under three diverse storage environments. The main observation is that performance stops scaling with increasing thread counts once concurrency becomes high (*i.e.*, ≥ 32 threads), particularly under Normal RSSD setting. For example, at 90% recall on LAION-I2I, MARGO improves by 162.1% from 8 to 32 threads, but only 8.2% from 32 to 128 threads. This is because the I/O bandwidth becomes saturated; under such conditions, increasing computational resources no longer improves efficiency (see Sec. 5.3 for more details).

Query Distribution. We evaluate out-of-distribution (OOD) queries on LAION-T2I [44] and Text2Image [17] to assess robustness under query shift. PipeANN is omitted on Text2Image since it does not support inner product distance. As shown in Figure 6b and Figure 6d, all methods perform worse on OOD queries than on in-distribution queries, consistent with observations in in-memory settings [11, 24]. For example, at the same dataset scale, the throughput of Gorgeous on LAION-T2I drops by up to 88.3% compared to LAION-I2I. Moreover, the benefits of in-segment index optimizations diminish under query shift. For instance, Starling achieves an 1.62× speedup over DiskANN on LAION-I2I, but only 0.95× on LAION-T2I at 95% recall.

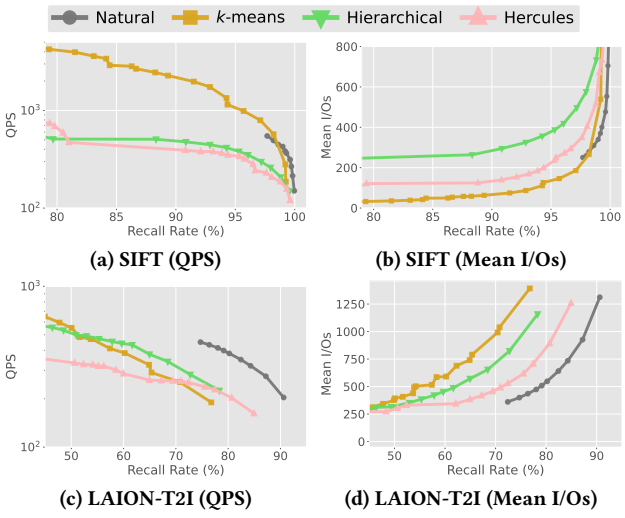
Scaling Costs: Build Time and Storage Overhead. Figure 8 reports construction time and storage overhead (128 threads on Fast RSSD). Most methods complete within 12 hours on million-scale datasets, yet none can finish within this budget on 1B-scale datasets. This limitation is problematic for deployments that require rapid overnight rebuilds (*i.e.*, $T + 1$ scenarios), a common requirement in real-world applications such as knowledge base retrieval [56, 60].

Takeaway 1: Typical data-center settings strongly influence the end-to-end search performance for disk-based vector search: *(i)* Disaggregated storage, especially with limited bandwidth, can cause substantial performance degradation of up to two orders of magnitude. *(ii)* I/O bandwidth saturation limits its scalability with higher concurrency. *(iii)* OOD query shifts generally degrade performance and, in particular, diminish the benefits of in-segment index optimizations. *(iv)* For billion-scale datasets, all methods fail to meet overnight build requirement.

5.2 Evaluation of Segmentation

This section evaluates segmentation choices on the SIFT [27] and LAION-T2I [45]. We build a DiskANN index [26] within each segment and use the list size L to control the in-segment search effort (larger L typically improves recall at the cost of more SSD reads). Following Sec. 3.2, we focus on three questions:

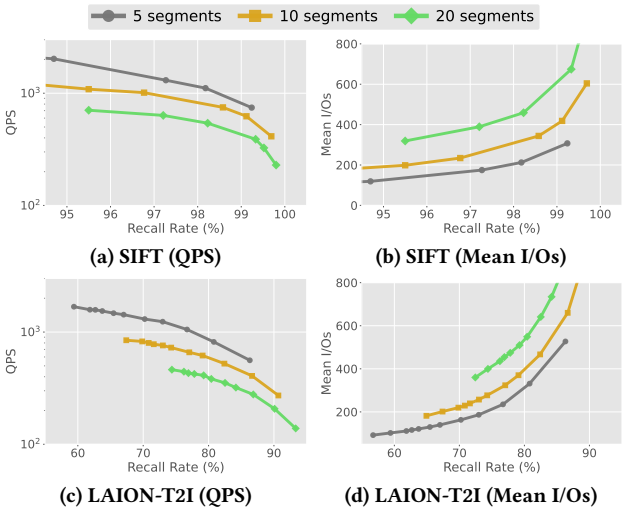
- Under the default *natural segmentation*, how sensitive is the performance to segmentation granularity?



(a) SIFT (QPS) (b) SIFT (Mean I/Os)

(c) LAION-T2I (QPS) (d) LAION-T2I (Mean I/Os)

Figure 9: Effect of segment strategies.



(a) SIFT (QPS) (b) SIFT (Mean I/Os)

(c) LAION-T2I (QPS) (d) LAION-T2I (Mean I/Os)

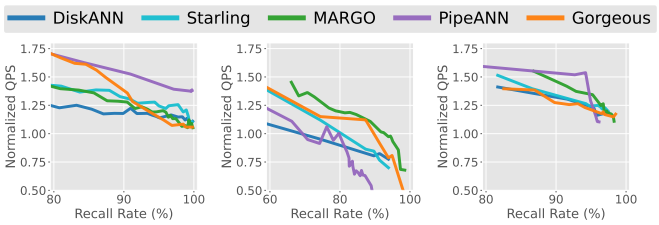
Figure 10: Effect of segment numbers.

- Can *similarity-aware segmentation* improve performance via segment-level pruning?
- Without changing segmentation, can *adaptive segment exploration* outperform uniform probing?

Natural Segmentation: Granularity and I/O Amplification. We first study granularity under natural segmentation, the default in state-of-the-arts (Table 1). We construct $m \in \{5, 10, 20\}$ natural segments and vary L to obtain QPS-recall and mean I/O-recall curves. Figure 10b shows that per-query I/O increases nearly linearly with m, indicating notable I/O amplification from multi-segment search that directly reduces throughput. For example, in LAION-T2I dataset and at 99% recall, increasing m from 10 to 20 raises mean I/Os from 370.3 to 548.8 and reduces QPS from 617.9 to 382.4. This effect is strongest once SSD bandwidth is saturated and I/O becomes the bottleneck (Sec. 5.1).

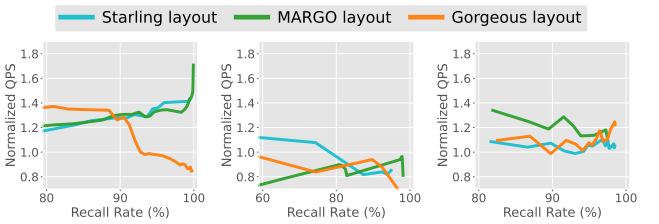
Similarity-Aware Segmentation: When Pruning Helps and Fails. We next evaluate whether similarity-aware partitioning can improve efficiency by pruning segments (Sec. 3.2). We compare four strategies: natural segmentation, k-means clustering, balanced hierarchical clustering, and hercules trees, targeting roughly 20 segments (some methods yield slight deviations). For natural segmentation, queries probe all (or most) segments, so we vary L to trace QPS-recall and mean I/O-recall curves. For similarity-aware strategies, we additionally vary the number of probed segments, and report skyline curves to emulate the best parameter combinations.

Figure 9 shows that similarity-aware segmentation can improve efficiency at moderate recall by concentrating relevant neighbors into a small subset of segments. However, the gains diminish (and may reverse) at very high recall due to false pruning. For example, in SIFT dataset, k-means with pruning improves QPS by 99.0% at 95% recall over natural segmentation, but reduces QPS by 27.0% at 99% recall, because recovering recall forces probing many (often nearly all) segments. Moreover, Hercules trees, while effective in in-memory settings [4], provide limited benefit here because their lower-bound-based computation saving technique is dominated by SSD I/Os in disk-resident vector search.



(a) SIFT (b) LAION-T2I (c) LAION-I2I

Figure 11: Effect of in-memory navigation graph.



(a) SIFT (b) LAION-T2I (c) LAION-I2I

Figure 12: Effect of layout optimization.

> **Takeaway 2:** Similarity-aware segmentation can increase throughput at moderate recall via segment pruning, but at near-exact recall (*e.g.*, ≥99%) the I/O savings vanish and false pruning can make it slower than natural segmentation.

Adaptive Segment Exploration: Cut Wasted I/O without Changing Segmentation. All methods above use *uniform probing*, which applies the same in-segment budget (L) to each probed segment. When many segments are probed (*e.g.*, natural segmentation or high-recall settings), this wastes SSD reads on segments that contribute few candidates to the final top-k.

To quantify the potential benefit of better segment exploration, we further evaluate an oracle-guided probing strategy that allocates search effort according to each segment's contribution to the final answer. Without changing the segmentation scheme or

in-segment search algorithm, the mean I/O cost can be reduced by up to 82.6% on the LAION-T2I dataset under natural segmentation. This result suggests that a large fraction of SSD reads are spent on low-value segments and highlights adaptive segment exploration as a promising direction for disk-based vector search methods.

Takeaway 3: Multi-segment search induces notable I/O amplification in disk-based vector search, roughly proportional to the number of segments, highlighting the need for adaptive segment exploration to avoid redundant SSD reads.

5.3 Evaluation of In-Segment Index and Search

This section evaluates in-segment index and search optimizations, which largely drive the end-to-end performance gaps in Sec. 5.1. Following Sec. 3.3, we study three optimization categories: in-memory navigation, on-disk layout optimization, and disk-aware traversal strategy, under deployment-realistic workload regimes. We focus on three questions:

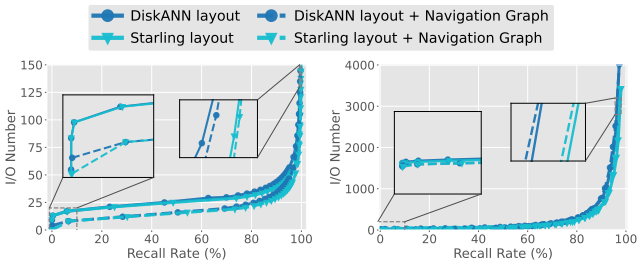
- Does an *in-memory navigation structure* still improve throughput under *OOD* queries?
- Is *on-disk layout optimization* effective at *high vector dimensionality*, where block locality weakens?
- How do *disk-aware traversal strategies* trade off throughput as *concurrency* increases?

We answer these questions using SIFT and LAION workloads under our SSD-backed environments, and report QPS at matched recall.

In-Memory Navigation under Query Shift. We run SIFT [27], LAION-I2I [44], and LAION-T2I [45] under the Fast RSSD setting, where the first two correspond to in-distribution (ID) queries and the last represents an OOD workload. We report normalized QPS, defined as the ratio between the QPS with the in-memory navigation graph enabled and that without it (at matched recall). As shown in Figure 11, the in-memory graph consistently improves throughput under ID queries, consistent with prior findings [57]. However, under OOD queries, the benefit becomes marginal and can even turn negative. For example, on LAION-T2I, the in-memory graph provides only up to a 9.2% improvement in DiskANN, while reducing QPS by up to 42.7% in PipeANN.

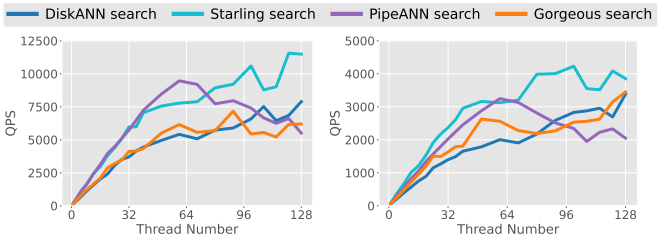
Takeaway 4: In-memory navigation graphs consistently improve throughput for in-distribution queries, but their benefits shrink under out-of-distribution shift (common in cross-modality search) and can become negative for some methods.

On-Disk Layout Optimization: Sensitivity to Dimensionality. We next study on-disk layout optimization on SIFT, LAION-I2I, and LAION-T2I, covering dimensionalities from 128 to 768. Figure 12 reports results under Fast RSSD, with the same in-memory navigation structure enabled for all methods to ensure a fair comparison. Layout optimization delivers strong gains on SIFT (low-dimensional vectors). For example, MARGO [64] achieves the best performance, improving throughput by up to 71.2%. However, the benefit declines rapidly as dimensionality increases. For example, while Starling performs well on SIFT, its gains become marginal, *i.e.*, 12.0% on the 512-dimensional LAION-I2I workload.



(a) SIFT (b) Text2Image

Figure 13: Comparison of navigation structure and layout optimization.



(a) 90% Recall (b) 99% Recall

Figure 14: Comparison of traversal strategies.

This trend follows from block locality. As dimensionality grows, a single vector (together with its neighbor metadata) occupies an increasingly large fraction of a disk block. When the single-vector size approaches a 4KB block, co-locating multiple related vectors within the same block becomes difficult, and a single read is less likely to return multiple useful candidates. As a result, layout optimization offers limited I/O savings at high dimensions. Moreover, layout construction can impose substantial memory overhead. For instance, MARGO [64] runs out of memory when optimizing the layout for 200M vectors even with a massive 512GB DRAM, limiting its practicality at scale.

Takeaway 5: The effectiveness of on-disk layout optimization is dimension-dependent. While it provides notable gains for low-dimensional vectors (*e.g.*, within 200 dimensions), its benefit diminishes as vector size approaches the disk block size.

Navigation vs. Layout Across Recall Regimes. We further examine how navigation and layout optimizations interact across recall regimes, including ID queries on SIFT [27] and OOD queries on Text2Image [17]. By varying the search effort, we record each method's I/O–recall trajectory in Figure 13. Generally, navigation provides larger gains at low and moderate recall, whereas layout becomes more effective as recall approaches near-exact. This behavior aligns with the two-phase nature of graph traversal [53, 59]. The search typically begins with a *converge* phase, routing from entry points toward the query region. Enhancing global routing through navigation primarily reduces wasted reads in this stage (left inset of Figure 13a). However, this benefit largely vanishes for OOD queries (left inset of Figure 13b), where relevant candidates are more scattered across the graph [11, 24], making global navigation

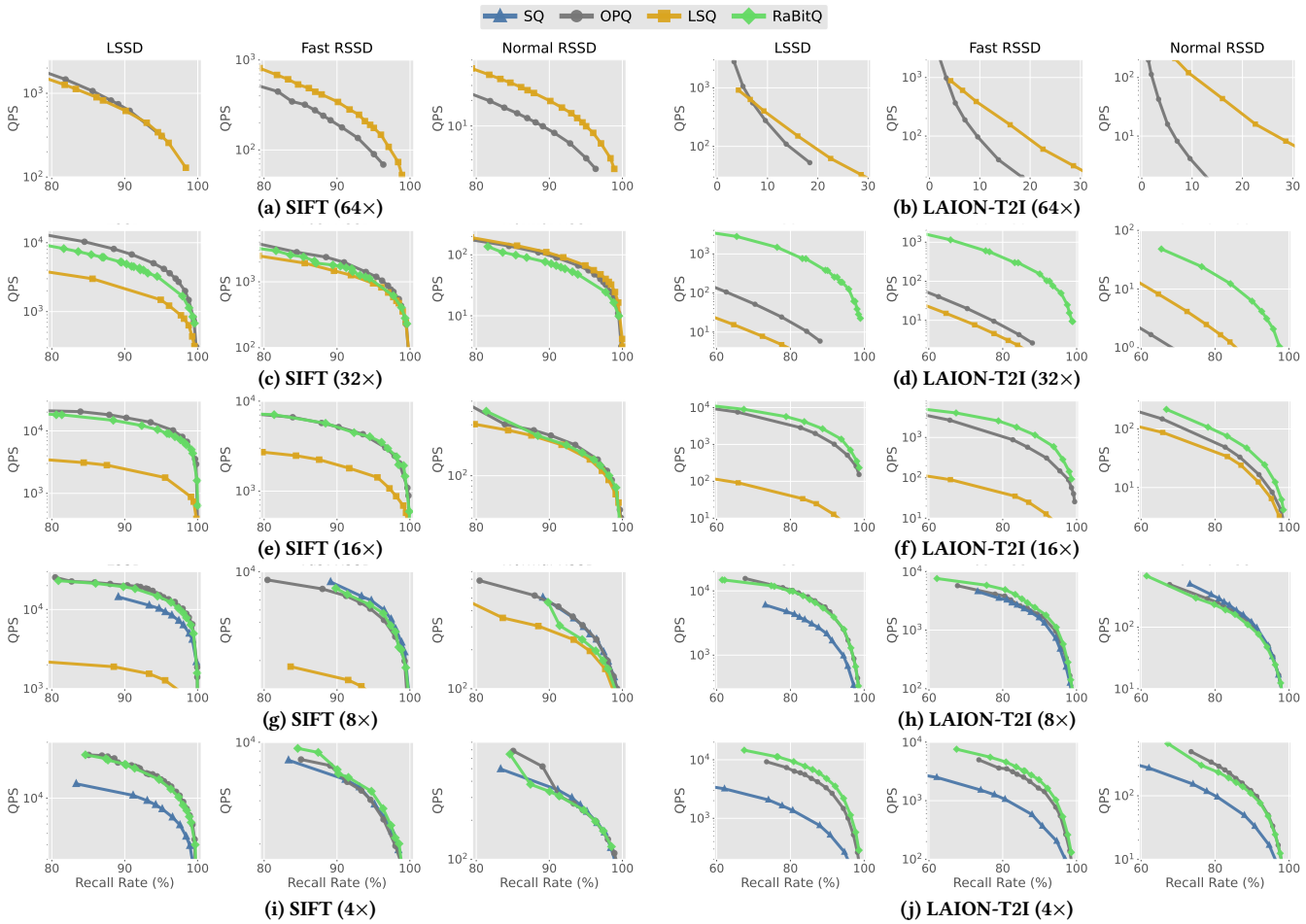


Figure 15: Comparison of quantization schemes under different compression rates and storage environment.

less effective. The traversal then enters the *diverge* phase, performing dense local exploration near the query region. In this stage, layout optimization reduces redundant SSD reads by co-locating frequently co-accessed vertices within same block. This effect is evident for both ID and OOD queries (two right insets of Figure 13) and complements navigation structure to further improve performance.

Traversal Strategy under Concurrency. Finally, we compare four in-segment traversal strategies in DiskVecLab on SIFT under the RSSD setting while varying the number of threads. Each method uses its corresponding on-disk layout, since Starling [57] and Gorgeous [62] couple traversal with their own layouts for best performance. As shown in Figure 14, relaxed best-first search (notably PipeANN) performs strongly under low to moderate concurrency, while requiring no offline layout optimization and supporting arbitrary vector dimensions. For example, at 32 threads and 90% recall, PipeANN improves over DiskANN by 47.5%. However, the advantage diminishes sharply once concurrency exceeds 64 threads. At 128 threads, PipeANN becomes 26.3% worse than DiskANN. The reason is that the benefit of additional compute–I/O overlap decreases once remote-storage bandwidth becomes saturated. Under such conditions, issuing more speculative I/O requests no longer improves latency hiding and instead increases contention for the limited I/O bandwidth, leading to lower throughput.

> **Takeaway 6:** Traversal strategy should be chosen based on concurrency and I/O bandwidth. Relaxed best-first can be advantageous at low to moderate concurrency (less than 64 threads), but once bandwidth saturates at high concurrency, strict best-first becomes more throughput-efficient.

5.4 Evaluation of Quantization

This section evaluates quantization choices in disk-based vector search. Beyond reducing the memory footprint, quantization quality affects candidate ranking and pruning during in-segment traversal, and can thus change how much SSD I/Os a query incurs. We quantify this effect end-to-end on SIFT [27] and LAION-T2I [45] using SQ, OPQ, LSQ, and RaBitQ under all three storage environments and compression ratios from 4× to 64×. We adopt centroid-based normalization for RaBitQ and tune the number of centroids from 64 to 8192 for each setting. We omit SQ and RaBitQ at high compression ratios because the corresponding implementations [16, 32]

Table 4: Construction time for different quantizers.

Dataset	Method	64×	32×	16×	8×	4×
SIFT	SQ	-	-	-	17.2s	29.2s
	OPQ	111.8s	163.8s	354.7s	608.7s	941.8s
	LSQ	1048s	1669s	4998s	30672s	>48h
	RaBitQ	-	62.9s	31.4s	33.4s	30.0s
LAION-T2I	SQ	-	-	-	97.1s	49.5s
	OPQ	248.2s	404.9s	698.5s	1224s	1340s
	LSQ	4389s	16056s	96979s	>48h	>48h
	RaBitQ	-	232.9s	91.6s	95.2s	94.5s

do not support them, and omit configurations whose quantizer construction does not complete within 48 hours (*e.g.*, LSQ at 4×).

Quantizer Construction Cost. Table 4 reports the quantizer construction time. SQ is the fastest, finishing within 2 minutes across all tested compression rates, followed by RabitQ, which completes within 5 minutes. In contrast, LSQ is notably slower, especially at longer codes (lower compression). It takes several hours at 16× and 8×, and does not finish within 48 hours at 4×. This cost can exceed graph construction time and limits LSQ's practicality.

Impact of Quantization on End-to-End Search. Figure 15 compares QPS-recall tradeoffs across compression rates (4×-64×) and storage environments. Quantization codes dominate memory usage in the pipeline, accounting for over 85% of total memory even at 64× compression, and over 90% at lower compression rates. Besides, quantizer choice has a large end-to-end impact on efficiency, with up to a 12.5× QPS gap at the same compression rate.

Across settings, the dominant bottleneck shifts with storage and memory configurations. As I/O throughput decreases, traversal becomes increasingly I/O-bound, and more accurate distance approximation can pay off by avoiding redundant SSD reads. Consequently, no single quantization scheme consistently dominates across all settings. Compared with OPQ, the scheme adopted by most existing disk-based vector search systems, LSQ can achieve higher throughput under very high compression ratios and I/O-limited storage. Its more accurate distance approximation reduces per-query SSD reads, making it up to 2.4× faster than OPQ at 64× compression on Normal RSSD. However, this benefit comes at the cost of substantially higher construction overhead (Table 4). RaBitQ remains competitive across all evaluated settings, often matching or exceeding OPQ by striking a favorable balance between approximation accuracy and computation efficiency, especially under high compression rate, achieving up to 71.8× higher throughput. These results suggest that RaBitQ is a strong alternative to PQ for disk-based vector search systems.

Takeaway 7: Quantization choices in disk-resident vector search are regime-dependent, with the optimal scheme varying across storage environments and compression ratios. Compared with the PQ scheme commonly used in current systems, AQ can deliver higher throughput in memory-constrained and I/O-bound settings, while RaBitQ is highly competitive at high compression ratios.

5.5 Guidelines and Future Directions

Guidelines. Our findings yield a clear set of evidence-based guidelines for disk-based vector search in realistic deployments:

(1) Segmentation: Similarity-aware segmentation improves performance at moderate recall, while natural segmentation is preferable for high recall (Takeaway 2). In this case, it is beneficial to increase segment granularity (*i.e.*, reduce segment count) and adopt adaptive segment exploration to mitigate I/O amplification (Takeaway 3).
(2) In-segment Index & Search: Design choices are deployment-dependent (Takeaway 1). Use navigation structures for ID queries but avoid them for OOD queries (Takeaway 4). Apply layout optimization only for relatively low-dimensional vectors, and avoid it when a single vector approaches the disk block size (Takeaway 5). Choose traversal strategies based on concurrency and I/O bandwidth: relaxed best-first under ample bandwidth, strict best-first under bandwidth saturation (Takeaways 1 and 6).
(3) Quantization: Avoid a one-size-fits-all quantization strategy. Select the quantization scheme based on memory and storage constraints: move from SQ/PQ to AQ as system becomes more I/O-bound, and consider RaBitQ as a competitive option at high compression ratios. (Takeaway 7).

Future Directions. We also outline several promising directions to enhance the practicality of disk-based vector search:

(1) Design more effective adaptive exploration strategy for multiple segments to improve the end-to-end performance.
(2) Enhance in-segment index to better handle I/O-intensive scenarios and improve robustness to OOD queries.
(3) Improve construction efficiency for billion-scale datasets to meet overnight rebuild requirements (Takeaway 1), or develop incremental index update strategies [63, 66].
(4) Extend vector search methods to object stores such as Amazon S3 [51, 61], which serve as a cornerstone of modern analytics systems but pose challenges due to their significantly slower I/O compared to SSDs.

6 CONCLUSION

In this paper, we present DiskVecLab, a modular framework for disk-based vector search that supports controlled component-level analysis and deployment-realistic end-to-end evaluation. Using DiskVecLab, we conduct an extensive study on six large-scale datasets across diverse storage environments, concurrency levels, and query workloads. Based on our study, we derive practical guidelines and identify promising future research directions for efficient and robust disk-based vector search.

ACKNOWLEDGEMENT

This work was partially supported by National Natural Science Foundation of China (NSFC) (Grant Nos. 62425202, 62336003), the Beijing Natural Science Foundation (Z230001), the Fundamental Research Funds for the Central Universities No. JKF-2026007844235, and the State Key Laboratory of Complex & Critical Software Environment (SKLCCSE). Zimu Zhou's research is supported by the RGC of Hong Kong SAR, China (Project No. CityU 11206425).

REFERENCES

[1] Elastic Search: Introducing a new vector storage format DiskBBQ. 2025. https://www.elastic.co/search-labs/blog/diskbbq-elasticsearch-introduction

[2] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. 2023. Similarity Search in the Blink of an Eye with Compressed Indices. In *Proceedings of the VLDB Endowment*. 3433–3446.

[3] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* (2020), 101374.

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

[5] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2025. Graph-based vector search: An experimental evaluation of the state-of-the-art. In *Proceedings of the ACM on Management of Data*. 1–31.

[6] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.

[7] Luiz André Barroso, Urs Hölzle, Parthasarathy Ranganathan, et al. 2013. The datacenter as a computer: Designing warehouse-scale machines. *Margen & Claypool* (2013).

[8] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2013), 1798–1828.

[9] SPACEV1B: A billion-Scale vector dataset for text descriptors. 2021. https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B

[10] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. In *Proceedings of the VLDB Endowment*. 3772–3785.

[11] Meng Chen, Kai Zhang, Zhenying He, Yinan Jing, and X Sean Wang. 2024. RoarGraph: A Projected Bipartite Graph for Efficient Cross-Modal Approximate Nearest Neighbor Search. In *Proceedings of the VLDB Endowment*. 2735–2749.

[12] 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. In *Advances in Neural Information Processing Systems*. 5199–5212.

[13] Qdrant Overview: Segmentation Configuration. 2026. https://qdrant.tech/documentation/overview/#segment-configuration

[14] Counterpoint. 2026. https://counterpointresearch.com/en/insights/Memory-Prices-Surge-Up-to-90-From-Q4-2025

[15] Ishita Doshi, Dhritiman Das, Ashish Bhutani, Rajeev Kumar, Rushi Bhatt, and Niranjan Balasubramanian. 2021. LANNS: a web-scale approximate nearest neighbor lookup system. In *Proceedings of the VLDB Endowment*. 850–858.

[16] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2025. The faiss library. *IEEE Transactions on Big Data* (2025).

[17] Benchmarks for Billion-Scale Similarity Search. 2021. https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search

[18] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. In *Proceedings of the VLDB Endowment*. 461–474.

[19] Jianyang Gao, Yutong Gou, Yuexuan Xu, Jifan Shi, Cheng Long, Raymond Chi-Wing Wong, and Themis Palpanas. 2024. High-Dimensional Vector Quantization: General Framework, Recent Advances, and Future Directions. *Data Engineering Bullentin* (2024), 3.

[20] 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. In *Proceedings of the ACM on Management of Data*. 1–27.

[21] Jianyang Gao and Cheng Long. 2024. RaBitQ: quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. In *Proceedings of the ACM on Management of Data*. 1–27.

[22] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2946–2953.

[23] Hao Guo and Youyou Lu. 2025. Achieving Low-Latency Graph-Based Vector Search via Aligning Best-First Search Algorithm with SSD. In *USENIX Symposium on Operating Systems Design and Implementation*. 171–186.

[24] Zhiyuan Hua, Qiji Mo, Zebin Yao, Lixiao Cui, Xiaoguang Liu, Gang Wang, Zijing Wei, Xinyu Liu, Tianxiao Tang, Shaozhi Liu, et al. 2025. Dynamically Detect and Fix Hardness for Efficient Approximate Nearest Neighbor Search. In *Proceedings of the ACM on Management of Data*. 1–28.

[25] 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.

[26] 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. In *Advances in neural information processing Systems*. 1–11.

[27] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2010), 117–128.

[28] Guoxin Kang, Zhongxin Ge, Jingpei Hu, Xueya Zhang, Lei Wang, and Jianfeng Zhan. 2025. BigVectorBench: Heterogeneous Data Embedding and Compound Queries are Essential in Evaluating Vector Databases. In *Proceedings of the VLDB Endowment*. 1536–1550.

[29] 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.

[30] Huiling Li and Jianliang Xu. 2025. BAMG: A Block-Aware Monotonic Graph Index for Disk-Based Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2509.03226* (2025).

[31] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering* (2019), 1475–1488.

[32] RaBitQ Library. 2025. https://github.com/VectorDB-NTU/RaBitQ-Library

[33] 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* (2018), 824–836.

[34] 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.

[35] Julieta Martinez, Shobhit Zakhmi, Holger H Hoos, and James J Little. 2018. LSQ++: Lower running time and higher recall in multi-codebook quantization. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 491–506.

[36] Lang Mei, Siyu Mo, Zhihan Yang, and Chong Chen. 2025. A survey of multimodal retrieval-augmented generation. *arXiv preprint arXiv:2504.08748* (2025).

[37] Google Vertex AI: Storage optimized Vector Search. 2026. https://docs.cloud.google.com/vertex-ai/docs/vector-search/storage-optimized-vector-search

[38] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *The VLDB Journal* (2024), 1591–1615.

[39] Pinecone. The Vector Database to Build Knowledgeable AI. https://www.pinecone.io/

[40] Qdrant. High-Performance Vector Search at Scale. https://qdrant.tech/

[41] Milvus Data Processing: Data query. 2026. https://milvus.io/docs/data_processing.md

[42] 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*. 8748–8763.

[43] M Ravikanth, Sampath Korra, Gowtham Mamidisetti, Maganti Goutham, and T Bhaskar. 2024. An efficient learning based approach for automatic record deduplication with benchmark datasets. *Scientific Reports* (2024), 16254.

[44] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, et al. 2022. Laion-5b: An open large-scale dataset for training next generation image-text models. In *Advances in neural information processing systems*. 25278–25294.

[45] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. 2021. Laion-400m: Open dataset of clip-filtered 400 million image-text pairs. *arXiv preprint arXiv:2111.02114* (2021).

[46] Alibaba Cloud: Elastic Compute Service. 2026. https://www.alibabacloud.com/help/en/ecs/

[47] Joobo Shim, Jaewon Oh, Hongchan Roh, Jaeyoung Do, and Sang-Won Lee. 2025. Turbocharging Vector Databases using Modern SSDs. In *Proceedings of the VLDB Endowment*. 4710–4722.

[48] Jun Song, Jingyi Ding, Irshad Kandy, Yanghao Lin, Zhongjia Wei, Zilong Zhou, Zhiwei Peng, Jixi Shan, Hongyue Mao, Xiuqi Huang, et al. 2025. Magnus: A Holistic Approach to Data Management for Large-Scale Machine Learning Workloads. In *Proceedings of the VLDB Endowment*. 4964–4977.

[49] 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).

[50] Pinecone: Reimagining the vector database to enable knowledgeable AI. 2024. https://www.pinecone.io/blog/serverless-architecture/

[51] Amazon S3 Vectors. 2026. https://aws.amazon.com/s3/features/vectors/

[52] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an elastic query engine on disaggregated

storage. In *USENIX Symposium on Networked Systems Design and Implementation*. 449–462.

[53] Hongya Wang, Zhizheng Wang, Wei Wang, Yingyuan Xiao, Zeng Zhao, and Kaixiang Yang. 2020. A note on graph-based nearest neighbor search. *arXiv preprint arXiv:2012.11083* (2020).

[54] 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 International Conference on Management of Data*. 2614–2627.

[55] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated database systems. In *Proceedings of the International Conference on Management of Data*. 37–44.

[56] Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Yifan Zhu, and Wenchao Zhou. 2025. Accelerating Graph Indexing for ANNS on Modern CPUs. In *Proceedings of the ACM on Management of Data*. 1–29.

[57] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. In *Proceedings of the ACM on Management of Data*. 1–27.

[58] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. In *Proceedings of the VLDB Endowment*. 1964–1978.

[59] Zeyu Wang, Qitong Wang, Xiaoxing Cheng, Peng Wang, Themis Palpanas, and Wei Wang. 2024. Steiner-hardness: A query hardness measure for graph-based ann indexes. In *Proceedings of the VLDB Endowment*. 4668–4682.

[60] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. In *Proceedings of the VLDB Endowment*. 3152–3165.

[61] Building AI Apps with Amazon S3 Data: Vector Search for Smarter Insights. 2026. https://zilliz.com/data-connectors/zilliz-fivetran-amazon-s3

[62] Peiqi Yin, Xiao Yan, Qihui Zhou, Hui Li, Xiaolu Li, Lin Zhang, Meiling Wang, Xin Yao, and James Cheng. 2025. Gorgeous: Revisiting the Data Layout for Disk-Resident High-Dimensional Vector Search. *arXiv preprint arXiv:2508.15290* (2025).

[63] Song Yu, Shengyuan Lin, Shufeng Gong, Yongqing Xie, Ruicheng Liu, Yijie Zhou, Ji Sun, Yanfeng Zhang, Guoliang Li, and Ge Yu. 2025. A topology-aware localized update strategy for graph-based ann index. *arXiv preprint arXiv:2503.00402* (2025).

[64] Ziyang Yue, Bolong Zheng, Ling Xu, Kanru Xu, Shuhao Zhang, Yajuan Du, Yunjun Gao, Xiaofang Zhou, and Christian S Jensen. 2025. Select Edges Wisely: Monotonic Path Aware Graph Layout Optimization for Disk-Based ANN Search. In *Proceedings of the VLDB Endowment*. 4337–4349.

[65] Minjia Zhang and Yuxiong He. 2019. Grip: Multi-store capacity-optimized high-performance nearest neighbor search for vector search engine. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 1673–1682.

[66] Shurui Zhong, Dingheng Mo, and Siqiang Luo. 2025. Lsm-vec: A large-scale disk-based system for dynamic vector search. *arXiv preprint arXiv:2505.17152* (2025).