Storage-Based Approximate Nearest Neighbor Search: What are the Performance, Cost, and I/O Characteristics?

Zebin Ren
Computer Science Department
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
z.ren@vu.nl

Krijn Doekemeijer
Computer Science Department
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
k.doekemeijer@vu.nl

Padma Apparao
Intel
Portland, Oregon, United States
padma.apparao@intel.com

Animesh Trivedi
IBM Research Europe
Zurich, Switzerland
animesh.trivedi@ibm.com

***Abstract*—Retrieval-augmented generation (RAG) has emerged as an effective method for enhancing large language models by integrating external knowledge sources to reduce the model size, avoid hallucinations, and provide an easier way to update the knowledge than fine-tuning. This external knowledge is commonly managed by vector databases, where the external knowledge is embedded into vectors and retrieved with vector similarity search. As the size of these external knowledge bases grows, the memory requirements for storing vectors and their associated indexes exceed the practical limits of main memory, prompting a shift toward storage-based solutions. Despite the adoption of storage-based solutions in modern vector databases, there have been limited systematic evaluations of the performance characteristics and I/O behavior of state-of-the-practice vector databases with storage-based setups. In this paper, we systematically characterize the performance, scalability, and I/O characteristics of these vector databases on modern SSDs that can deliver millions of I/O operations/s with less than 100 μs latency. We report 22 observations and 3 key findings that indicate: (i) vector databases with storage-based setups do not necessarily indicate lower performance than memory-based setups, for example, the storage-based setup DiskANN outperforms the memory-based setup, IVF, with up to 3.2$\times$ search throughput in Milvus, (ii) state-of-the-practice vector databases with storage-based setups require optimizations on I/O traffic to fully utilize the performance with flash SSDs, the maximum bandwidth achieved in our experiments is 1.7 GiB/s and can not saturate our benchmarked SSD, and (iii) the indexes' search-time parameters affect both performance and I/O characteristics of vector databases, for example, when the parameter `search_list` increases from 10 to 100, the throughput of vector similarity search decreases up to 60.9% and the read bandwidth increases up to 3.3$\times$. We open-source the scripts and traces of this work at: https://zenodo.org/records/16916496.**

***Index Terms*—approximate nearest neighbor search, vector database, I/O workload characterization**

I. Introduction

Vector databases [2], [9], [13], [22], [40], [50], [69], [74] are widely used in information retrieval [5], [25], recommendation systems [31], [39], [60], and e-commerce [50], [74]. The emerging retrieval-augmented generation (RAG) enhances the large language models (LLM) with an external knowledge base to reduce model size, avoid hallucinations, and provide an easier way to update the knowledge than fine-tuning [36], [79]. This knowledge is usually managed by vector databases, where the external knowledge, such as text, images, or videos, is embedded into vectors and retrieved with vector similarity search [47].

Vector databases are specialized databases that manage vector data. A core feature these vector databases provide is approximate nearest neighbor search (ANNS), which finds the approximate k nearest vectors of query vectors in the database according to a user-defined distance metric [18]. To accelerate ANNS, vectors are indexed in these vector databases for this vector similarity search, where these indexes are usually kept in the main memory for fast access [24], [27], [32]–[34], [41], [51], [53], [54], [58]. Modern RAG datasets contain millions of documents and trillions of tokens [28], and the vectors' dimensions exceed one thousand. As a result, the size of the indexes can be several hundred GiBs, making it challenging to fit into main memory [62], [72]. For example, the size of the widely used HNSW (Hierarchical Navigable Small World) index for one billion 96-dimensional vectors is over 700 GiB [62]. This memory overhead becomes more significant when many RAG pipelines adopt local setups that do not split the vector database into multiple servers [52]. Hence, researchers are looking into storing the vectors on storage devices to reduce their memory requirement [29], [59], [68], [72], [75]. Modern NVMe SSDs are able to deliver millions of I/O operations/s with tens of microseconds of latency and easily scale to TiBs in capacity, making it a good fit to store these vector indexes [17]. However, there has been limited systematic study and characterization of the impact of moving these indexes from main memory to storage devices, which affects the performance of vector databases, specifically (i) SSDs have higher latency and larger access granularity than memory, it is unknown how the latency, throughput, and scalability of the vector search queries degrade for the indexes that can not be fully maintained in main memory, (ii) parallel

This work is funded by The Dutch Research Council (NWO) grant number OCENW.KLEIN.561, Netherlands-funded projects NWO OffSense and GFP 6G FNS, and Horizon Graph-Massivizer. Krijn Doekemeijer is funded by the VU PhD innovation program.

I/O requests are required to fully utilize the performance of SSD [63], [64], it is unknown if these vector databases can fully utilize the performance of the modern SSDs and what their I/O characteristics are, and (iii) the parameters that control the behavior of vector indexes affect both the performance and I/O characteristics, which are used to control the trade-off between accuracy, index building time, and search performance.

In this work, we investigate the performance cost and I/O characteristics of four state-of-the-practice vector databases (Milvus [11], Qdrant [13], Weaviate [22], and LanceDB [9]) with a storage-based setup to study and characterize the performance of these vector databases with modern high-performance SSDs. Our investigation is motivated by the three challenges in the previous section. Firstly, the storage devices have higher latency and larger access granularity than memory, creating overheads on the vector search latency, throughput, and scalability. There are multiple solutions to reduce the memory size requirement of vector databases in state-of-the-practice vector databases, such as using storage-friendly indexes (DiskANN) [11] or using `mmap` with memory-based indexes [13]. The lack of quantification on the performance of vector search for different storage-based solutions on modern NVMe SSDs makes it difficult for practitioners to choose the best vector database when using a setup with local SSDs. We demonstrate that *vector databases with storage-based setups do not necessarily have worse performance in throughput and latency than vector databases with memory-based setups*. Secondly, modern NVMe SSDs are able to provide millions of I/O operations per second (IOPS), but achieving this throughput requires parallel I/O requests and a large amount of CPU resources [63], [64], making it challenging to fully utilize the performance of modern SSDs. Thus, we investigate the I/O characteristics of these vector databases and find that the maximum bandwidth achieved during the vector search of Milvus with DiskANN index is 1,720.0 MiB/s, less than the maximum bandwidth in the SSD that is used for the benchmarking (7.2 GiB/s). Thirdly, the indexing algorithms have parameters to control the trade-off among accuracy, performance, and resource usage. These parameters affect the I/O features, such as bandwidth. Thus, it is important to understand how these parameters affect the I/O patterns to fully utilize the performance of modern SSDs. For example, the search-time parameter `search_list` is the number of candidate vectors during the search, where a higher `search_list` indicates more vectors are checked during the search process (§II). We find that although increasing `search_list` from 10 to 100 results in a throughput decrease of up to 60.9% and a bandwidth increase to 3.3×, the bandwidth of the SSD is still not saturated, indicating that using a larger `search_list` does not lead the SSD to be the bottleneck. To summarize, the limited knowledge of the performance, I/O characteristics, and the effect of indexes' parameters motivates us to systematically study the performance and I/O behavior of vector databases with storage-based setups.

In this study, we empirically measure, quantify, and analyze the performance, I/O characteristics, and the performance effect of indexes' parameters of four widely used open-source vector databases in high-dimensional vector space: Milvus [11], Qdrant [13], Weaviate [22], and LanceDB [9]. We specifically motivate and investigate the following research questions (**RQ**) around the I/O workloads of vector databases and make the following key findings (**KF**):

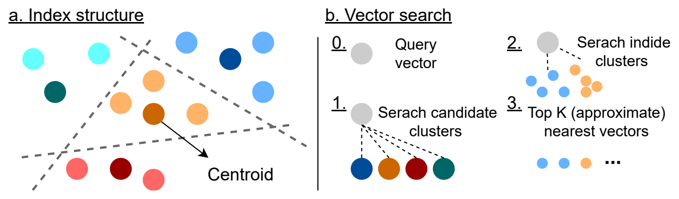
RQ1: What is the latency, throughput, and scalability overhead of using storage-based setups in state-of-the-practice vector databases compared with memory-based setups? This research question focuses on finding the performance cost of using storage-based setups compared with memory-based setups in the evaluated vector databases and compares the latency, throughput, and scalability of storage-based setups in modern vector databases. **KF-1:** We find that storage-based vector search setups do not necessarily perform worse than memory-based setups. For example, although Milvus with DiskANN, a storage-based vector index, has up to 74.1% lower throughput and 96.7% higher latency than the memory-based vector index HNSW, DiskANN outperforms Milvus with IVF, another memory-based index, with up to 3.2× throughput and 44.5% lower latency, and outperforms the other databases in throughput at least three out of the four datasets.

RQ2: What are the I/O characteristics of the state-of-the-practice vector databases with storage-based setups during vector similarity search, and are these vector databases with storage-based setups able to fully utilize the NVMe SSDs? Modern NVMe SSDs are able to provide millions of IOPS with tens of μs latency, but require parallel I/O requests and a large amount of CPU resources to saturate these SSDs. Here, we study the I/O characteristics of Milvus with DiskANN during vector search and find that **KF-2:** the per-query average read bandwidth scales up to 10.1× when the dataset size increases to 10×. Milvus with DiskANN fails to saturate the SSD's bandwidth.

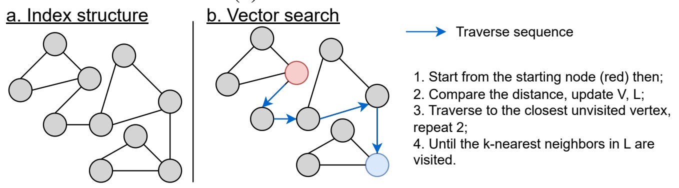
RQ3: How do the indexes' parameters affect vector databases' performance and I/O characteristics? The vector indexes have parameters to tune between accuracy, index size, search performance, and index building time. We investigate how two search-time parameters, `search_list` and `beam_width`, affect the performance and I/O characteristics of DiskANN with Milvus. **KF-3:** Increasing `search_list` from 10 to 100 brings up to 6.5% accuracy increase with the cost of up to 60.9% throughput decrease, and 77.0% latency increase with an up to 6.3× per-query average bandwidth.

Our key contributions in this work are:

- To the best of our knowledge, this is the first-of-its-kind systematic study on state-of-the-practice vector databases with NVMe SSDs, exploring their performance, scalability, and I/O characteristics, resulting in 22 observations and 3 key findings.
- We investigate the I/O characteristics of vector databases with storage-based setups on a modern NVMe SSD and report that the existing storage-based solutions, Milvus with DiskANN, mainly issue 4 KiB random I/O


a. Index structure
Centroid
b. Vector search
0. Query vector
1. Serach candidate clusters
2. Serach indide clusters
3. Top K (approximate) nearest vectors
...

(a) *Cluster-based index.*


a. Index structure
b. Vector search
Traverse sequence
1. Start from the starting node (red) then;
2. Compare the distance, update V, L;
3. Traverse to the closest unvisited vertex, repeat 2;
4. Until the k-nearest neighbors in L are visited.

(b) *Graph-based index.*

Figure 1: *An illustration of cluster-based index and graph-based index.*

requests and fail to fully utilize the performance of modern SSDs.

- To facilitate reproduction, we open-source the implementation of our script and the traces collected during our experiments at https://github.com/atlarge-research/2025-iiswc-vectordb-bench-artifact-public, permanent link at https://zenodo.org/records/16916496.

II. Background

In this section, we present the background on approximate nearest neighbor search, vector databases, and dense vector indexing.

A. Approximate Nearest Neighbor Search

Given a vector dataset $X = [x_1, x_2, ..., x_n] \in \mathbb{R}^d$ and a query vector $q \in \mathbb{R}^d$ where all the vectors in X and q have the same dimension d, the k-nearest neighbor search (NNS) problem is defined as: finding k vectors $K = [i_1, i_2, ..., i_k] \subset X$ that $D(q, i) \leq D(q, j)$ for any $i \in K$ and $j \in X \setminus K$ where D measures the distance between two vectors. However, the vector datasets in real-world applications can contain billions of vectors, making it computationally prohibitive to find the exact K nearest neighbor with brute-force searching the whole dataset [29], [75]. A practical solution is to find the approximate nearest neighbor instead of the exact k nearest neighbor, called approximate nearest neighbor search (ANNS) in vector similarity search. ANNS finds k approximate vectors K' that are close to the query vector q. The accuracy of ANNS is called `recall`, where it is defined as `recall@`$k = \frac{|K \cap K'|}{k}$. To accelerate ANNS, the vectors are indexed for fast similarity search. There are four kinds of indexes: (1) cluster-based indexes, (2) graph-based indexes, (3) tree-based indexes, and (4) hashing-based indexes. Since only the cluster-based indexes and graph-based indexes are used in our benchmarked databases, we explain these two indexes in detail.

B. Dense Vector Indexes

Cluster-based indexes: Fig. 1a shows the general structure of a cluster-based index (left) and the search procedure (right). The vectors are clustered with K-Means clustering, and each cluster is represented by a centroid. The number of clusters is usually referred to as `nlist` and is set as a tunable parameter while building the index. Duplicating the vectors at the border of each cluster into more than one cluster achieves high accuracy at the cost of increased index sizes [29]. The vectors can be compressed, such as product quantization (PQ), to reduce the memory cost [46]. During a query (step 1), the search procedure first identifies the candidate clusters by comparing the query vector to the centroids, where the number of clusters is usually referred to as `nprobe` (step 2). The centroids can be further managed by a graph index to accelerate the procedure of candidate cluster selection. Then, the query vector is compared with all the vectors in the candidate clusters to find the approximate nearest neighbors. The k nearest vectors in the candidate clusters are selected as the final result (step 3).

Graph-based indexes: Fig. 1b shows the general structure of a graph-based index (left) and the search procedure (right). Graph-based indexes construct a proximity graph of the vectors where the vertices in the graph represent the vectors, and the edges connect the vectors that are close to each other [53], [54], [59], [68]. The search procedure starts from a start node and traverses the graph with a best-first strategy (step 1). The start node can be fixed or chosen randomly. Recent works proposed using a smaller sampled graph to select the start vectors to reduce the travel length [59], [68], [72]. The best-first search procedure is shown in step 2 to step 4. During the traversal, the search keeps two sets of vectors: (1) the top-k candidate vectors L, and (2) the vectors that have been traversed V. During each iteration, it selects the vector that is unvisited in L and closest to the query vector, computes the distance between these neighbors and the query vector, and updates the candidate vectors L and visited vectors V (step 2). After the L and V are updated, it repeats step 2 until all the vectors in the candidate set are visited (step 3). When the traversal ends, the vectors in L are the approximate nearest vectors of the query vector (step 4).

Storage-optimized indexes: For billion-level vector datasets, the size of the indexes can reach more than 700 GiBs, posing significant overhead on the memory [62]. Thus, studies propose to store the vector indexes in storage devices instead of main memory [29], [59], [68], [72], [75]. The two kinds of indexes, cluster-based and graph-based indexes, both have storage-based variations.

Cluster-based indexes fit the access granularity since all the vectors in the same cluster can be stored consecutively in the storage devices. However, the vectors near the border of the clusters are replicated into several different clusters to achieve higher accuracy, leading to space amplification. For example, the vectors at the border of clusters can be replicated up to eight times in SPANN [29], [75].

Graph-based indexes are prone to high latency due to their dependency between I/O requests during the graph traversal [59], [68]. In this paper, we only include DiskANN in our benchmarks since it is the only supported storage-based vector index in the selected vector databases [68]. DiskANN is a graph-based index that is specifically designed for SSDs. Specifically, DiskANN stores compressed vectors in memory and the graph index together with the full-precision vectors in the storage device. During the search procedure, the neighbors of the graph vertices are fetched from the storage devices, where the distance is computed using the compressed vectors in the memory. Different from the best-first search in general graph-based indexes, DiskANN utilizes beam search that computes the distance of the top W closest (W=1 in the best-first search strategy) unvisited vectors in the candidate vectors L (`search_list`) since reading a single 4KiB page has the same latency as reading a few pages from the SSDs when the SSD throughput is not the bottleneck. We will show how `search_list` and W, called `beam_width`, affect the throughput, latency, and I/O patterns in §VI.

C. Vector Databases

Vector databases are specialized databases that treat vectors as first-class entities, primarily optimized for vector operations such as ANNS. Different from the ANNS algorithms introduced in the previous section, although vector databases rely on vector indexes to enable efficient vector searches, vector databases are fully functional systems (normally distributed) that offer additional capabilities beyond ANNS, such as transaction management, dynamic vector insertion and deletion, handling auxiliary data (payloads) associated with vectors, and data persistence. Commonly used vector database systems include Milvus [69], Qdrant [13], Weaviate [22], Chroma [2], and LanceDB [9], all of which are production-ready solutions adaptable to various deployment scenarios, ranging from single-machine setups to distributed environments. These databases also support diverse storage backends, including cloud-based solutions such as cloud object storage and local storage devices. In this work, we specifically focus on scenarios where the vector database is deployed on a single machine using high-speed NVMe SSDs.

III. Experiment Setups

In this section, we present details about the benchmarking environment, workloads, and benchmarked vector databases.

A. Software and Hardware Setup

Tab. I shows the detailed hardware environment. We use VectorDBBench [19] as the workload generator and added support for setting the index parameters needed for our benchmarking. All the vectors and their associated indexes are stored in a dedicated NVMe SSD, a Samsung 990 Pro 4 TiB, apart from the operating system and vector database installations, which are stored on a SATA SSD, a Samsung MZ7L31T9. We use a dedicated SSD to store the vector data and their associated indexes to avoid the interference of I/Os from other processes, such as the operating system. Both storage devices are formatted with the `ext4` file system. The performance of the benchmarked vector databases is compared with three metrics: throughput, P99 tail latency, and CPU usage. We use `bpftrace` [7] to collect the I/O traces in the block layer. We probe using the `block_rq_issue` tracepoint and collect the I/O operation's type (i.e., read or write) and request size.

TABLE I: *Details of the benchmarking environment*

Component	***Configuration details***
CPU	Intel(R) Xeon(R) Silver 4416+, 10 cores, hyperthreading disabled, boost disabled.
Memory	256 GiB, DDR5, 4800 MT/s.
Storage device	Samsung MZ7L31T9 and Samsung 990 Pro 4 TiB.
Software	Ubuntu 22.04.4 LTS with kernel 5.15.0-139-generic, Milvus 2.5.11, Qdrant 1.14.1, Weaviate 1.31.0, LanceDB 0.23.0

Before evaluating the performance of vector databases, we measure the raw performance of the NMVe SSD used in the experiments with `fio` [6]. The Samsung 990 Pro is able to deliver a peak random read performance of 324.3 KIOPS with 4 KiB request size on a single CPU core and 1.3 MIOPS with 64 concurrent 4 KiB requests using four CPU cores under the Linux storage stack. 7.2 GiB/s with 128 KiB sequential read using 32 concurrent threads.

B. Workload Patterns and Methodology

We use four vector datasets that VectorDBBench provides in our experiments: Cohere 1M, Cohere 10M, OpenAI 500K, and OpenAI 5M, where the former two contain 1 million and 10 million 768-dimension vectors and the latter two contain 500 thousand and 5 million 1536-dimension vectors. These two dimensions are chosen because they are the two most widely used embedding dimensions in RAG [44], [48], [49].

Each experiment runs for 30 seconds with 1,000 query vectors. If all the queries are used up before the experiment ends, it restarts from the first query vector. Before each run, we write the dirty page cache to the storage devices with `sync` and flush the page cache with `echo 1 | sudo tee /proc/sys/vm/drop_caches` to reduce the impact of the page caching. We repeat each experiment five times and report the average throughput and P99 latency with standard deviation.

C. Vector Databases and Setups

We benchmark four vector databases with different memory-based and storage-based indexes, depending on the indexes that are supported by the benchmarked vector databases. The version of the benchmarked vector databases are shown in Tab. I. We use the official Docker image for Milvus [16], Qdrant [8], and Weaviate [20]. LanceDB is set up with a *local Python setup* since it does not provide a Docker image [15]. We focus on three commonly used dense vector indexes—IVF, HNSW, and DiskANN, and tune their key parameters to achieve recall@10$\geq$0.9 on Milvus and use the same key parameters across the four vector databases to

TABLE II: *The build&search-time parameters and their achieved recall@10 accuracy of the vector indexes used in the benchmarking, where the built-time and search-time parameters are shown with transparent and green backgrounds, the achieved accuracy is shown with grey backgrounds.*

	IVF			HNSW					DiskANN	
	nlist	nprobe	acc	M	efConstruction	efSearch	acc	efSearch (LanceDB)	search_list	acc
Cohere 1M	4,000	25	0.90 (0.73)	16	200	27	0.90	41	10	0.94
Cohere 10M	12,648	17	0.90 (0.72)	16	200	43	0.90	56	10	0.93
OpenAI 500K	2,828	16	0.90 (0.66)	16	200	14	0.90	34	10	0.96
OpenAI 5M	8,944	11	0.90 (0.64)	16	200	10	0.91	38	10	0.98

ensure a fair comparison in §IV and §V, respectively. In §VI, we further investigate how these parameters influence the databases' I/O characteristics.

These vector indexes have two types of parameters: build-time parameters, which determine how the index is constructed and remain fixed after the indexes are constructed, and search-time parameters, which control search behavior and can be modified at search time. Tab. II summarizes the build- and search-time index parameters used in our experiments and their achieved recall@10 on the four vector datasets with Milvus. The built-time and search-time parameters are shown with transparent and green backgrounds, and the achieved accuracy is shown with grey backgrounds. Specifically, for IVF, we set `nlist` to $4\sqrt{n}$, where n is the number of vectors in the dataset, as recommended by faiss [4], and tune `nprobe` to reach recall@10$\geq$0.9. For HNSW, we set `M` to 16 and `efConstruction` to 200 [54] to balance the accuracy and index build time, and tune `efSearch` to reach the target latency. For DiskANN, we tune the `search_list` to reach the target accuracy. However, DiskANN achieves the target accuracy when `search_list` is set to the minimum number, so we keep `search_list` to the minimum value, 10, for all four datasets.

LanceDB only supports IVF and HNSW vector index with quantization. Thus, we use IVF with product quantization and HNSW with scalar quantization in LanceDB. We tune the search parameters separately for LanceDB since quantization has a negative effect on the accuracy. The `efSearch` used for LanceDB is shown in the 'efSearch (LanceDB)' column in Tab. II. For IVF with product quantization, we find that LanceDB has at least 10$\times$ lower performance with the same `nprobe` used in the other database. Thus, we do not further increase the `nprobe` to reach the target accuracy, recall@10$\geq$0.9, and report the achieved accuracy in the parentheses in the 'acc' column of the IVF index.

Below, we detail the specific setups evaluated for each vector database:

Milvus [11]: Milvus is evaluated with all three indexes, IVF (memory-based), HNSW (memory-based), and DiskANN (storage-based).

Qdrant [13]: Qdrant currently only supports HNSW as a dense vector index and uses `mmap` with limited memory resources. We refer to Qdrant with memory-based as `Qdrant-HNSW`. We do not observe a statistically different performance between memory- and storage-based setups since there is enough CPU memory to hold the vectors and their associated indexes. Thus, we do not include Qdrant with a storage-based setup.

Weaviate [22]: Weaviate only supports a memory-based HNSW vector index, which is referred to as `Weaviate-HNSW` in the experiments.

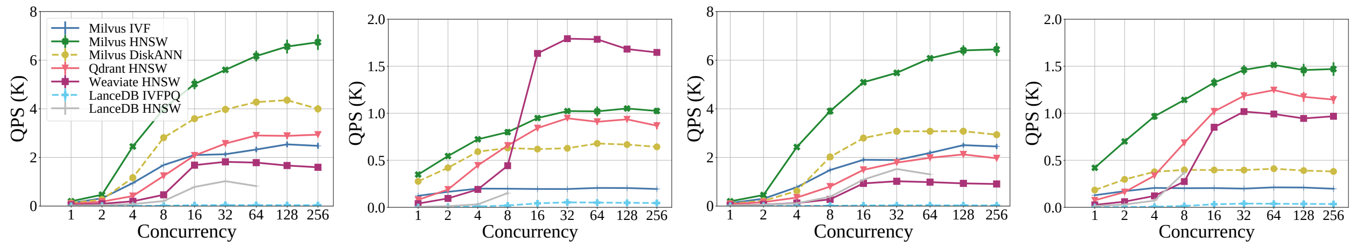
LanceDB [9]: LanceDB provides two indexes: a storage-based index IVF, and a memory-based index HNSW. However, LanceDB only supports quantized vectors with these two vector indexes. Thus, we use IVF with product quantization, `LanceDB-IVF` (storage-based), and HNSW index with scalar quantization, `LanceDB-HNSW` (memory-based).

IV. How Does Storage-based Setup Affect the Performance of Vector Databases

We start our analysis by investigating how storage-based setups affect the performance of vector databases to answer **RQ1**. In this section, we quantify the throughput, latency, and scalability between four different vector databases with different indexes. Specifically, we have five memory-based setups: `Milvus-IVF`, `Milvus-HNSW`, `Qdrant-mem`, `Weaviate-HNSW`, and `LanceDB-HNSW`, and two storage-based setups: `Milvus-DiskANN` and `LanceDB-IVF`. We evaluate the throughput (QPS) and P99 tail latency (μs) to compare the performance of storage-based setups to memory-based setups. We quantify how the indexes and vector databases affect the performance of vector search by comparing the throughput and latency between the same vector databases with different vector indexes and different vector databases with the same vector index. The expectation is that storage-based setups have lower throughput and higher latency than memory-based setups due to the SSDs' lower bandwidth and higher latency than main memory.

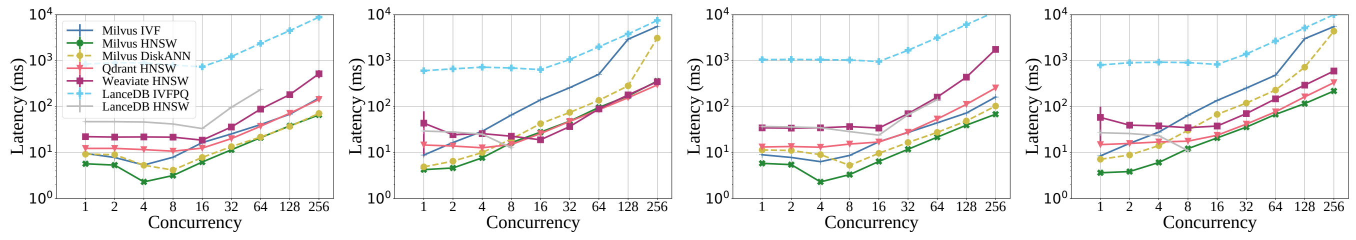
A. Throughput Scalability on Vector Search

Do storage-based indexes have lower throughput than memory-based indexes, and how does this difference scale with increased concurrent queries? To answer this question, we measure the throughput of the benchmarked vector databases with both memory- and storage-based setups under varying levels of concurrency. Fig. 2 shows the throughput in QPS (y-axis, higher is better) of the benchmarked vector databases with different indexes as the number of query threads increases from 1 to 256 on the x-axis. Each query thread issues a single vector search query and waits for the completion before issuing the next query. For LanceDB with



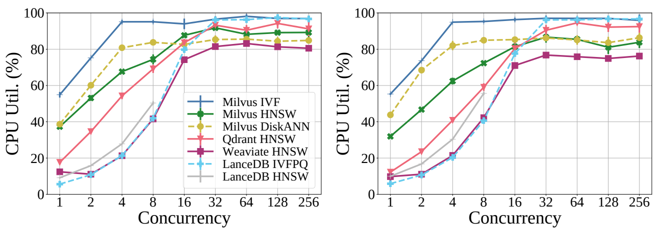
(a) Cohere 1M (b) Cohere 10M (c) OpenAI 500K (d) OpenAI 5M

Figure 2: Throughput scalability with an increasing number of query threads on the four different vector datasets, the storage-based setups are drawn with dashed lines. (Please note (b) and (d) have different scales on the y-axis from (a) and (c).)



(a) Cohere 1M (b) Cohere 10M (c) OpenAI 500K (d) OpenAI 5M

Figure 3: P99 tail latency scalability with an increasing number of query threads on the four different vector datasets, the storage-based setups are drawn with dashed lines.



(a) Cohere 10M (b) OpenAI 5M

Figure 4: Global CPU usage of different vector databases with an increasing number of query threads during vector search, the storage-based setups are drawn with dashed lines, 100% means that all the 20 CPU cores are 100% utlized.

HNSW index, the concurrency does not scale to 256 since higher concurrency leads to out-of-memory errors. LanceDB with the IVF index demonstrates notably lower throughput (under 100 QPS) even at 256 concurrent queries; hence, we exclude it from further discussion.

There are three observations here: Firstly, Milvus demonstrates the highest throughput when using the HNSW index, while its IVF index configuration shows the lowest throughput. The storage-based index, DiskANN, achieves a throughput between these extremes—exhibiting 37.3% to 74.1% lower throughput than the memory-based HNSW index but outperforming IVF by 1.2× to 3.2× throughput with 256 query threads, indicating that **O-1:** *With the same database, the indexing method is an important factor for the throughput of vector search.* Notably, DiskANN achieves higher accuracy (0.93–0.98) compared to both HNSW and IVF (0.90–0.91), as shown in Tab. II.

Secondly, the indexing methods alone do not fully determine the performance of vector databases. Among the databases supporting the HNSW index, Milvus has the highest throughput in three out of the four datasets, achieving 1.2×– 3.3× throughput as Qdrant and 1.5×–7.1× throughput as Weaviate with 256 query threads, with the exception of the Cohere 10M dataset. Specifically, Weaviate surpasses Milvus by 61.8% in throughput with Cohere 10M. **O-2:** *The indexing methods are not the only factor that affects the throughput of the vector search; vector databases with the same index have a throughput difference of up to 7.1×. This indicates that vector databases are a key factor influencing ANNS performance in addition to indexing algorithms.*

Thirdly, among all configurations tested, **O-3:** *LanceDB employing the HNSW index with scalar quantization has the lowest throughput when processing a single concurrent (in-flight) request, highlighting the substantial performance impact of quantization and database implementation choices.* It is worth noting that we use a different setup with LanceDB, an embedded Python library, instead of a database server running in Docker containers like the other databases, so further investigation is needed on how the setup affects the throughput of LanceDB.

How does the throughput change as we increase the number of concurrent query threads? **O-4:** *All evaluated vector databases exhibit superlinear scalability when concurrency increases from 1 to 16 threads with the two small datasets,* Cohere 1M and OpenAI 500K, with throughput ranging from 17.5× to 39.5× on Cohere 1M and from 15.8× to 31.0× on OpenAI 500K with 16 threads as using a single thread. However, when dataset sizes scale by 10× (to Cohere 10M and OpenAI 5M), the throughput scaling behavior varies across databases. Specifically, the throughput of Milvus-IVF and Milvus-DiskANN plateaus after four concurrent threads, showing 6.0× and 1.6× throughput as one thread on the two datasets with Milvus-IVF, respectively, and 2.1× bandwidth for both datasets with Milvus-DiskANN. While Milvus-HNSW generally achieves higher overall throughput than other vector databases with HNSW indexes, it demonstrates limited scalability, only 2.7× to 3.2× improvement, as concurrency increases from 1 to 16 threads. In contrast, Weaviate and Qdrant show greater scalability, achieving throughput increases of up to 41.0× and 14.7× under the same concurrency growth. **O-5:** *Although Milvus-HNSW performs better than the other vector databases with HNSW indexes, it has worse scalability than Qdrant and Weaviate with the two large datasets, Cohere 10M and OpenAI 5M* .

How does the throughput scale as we increase the size of the dataset? Milvus generally exhibits higher throughput than other vector databases. However, it also experiences the highest reduction in throughput when the dataset size increases by 10×, where the throughput reduces to 8.1% and 15.0% on Cohere and 8.2% to 22.8% on OpenAI when the dataset size increases 10×. In contrast, Qdrant demonstrates better scalability than Milvus, with its throughput decreasing to 29.6% and 58.7% of throughput with 10× larger datasets. Remarkably, although Weaviate has the lowest throughput in three out of the four datasets, it even achieves 3.1% and 6.6% throughput increase despite the 10× increase in dataset size. **O-6:** *Although Milvus achieves the highest throughput among the benchmarked vector databases, it shows the highest performance decrease as the dataset size increases. In contrast, the throughput of Weaviate even increases with the same increases in dataset size, indicating that vector databases have a huge influence on the scalability, even with the same vector index.*

B. Latency Scalability on Vector Search

Do storage-based indexes have higher latency than memory-based indexes, and how does this difference scale with increased query concurrency? To answer this question, we measure the P99 tail latency (y-axis, lower is better) with different vector database setups with increasing concurrency on the x-axis, as shown in Fig. 3. Similar to the previous section, we exclude LanceDB with the IVF index from this analysis.

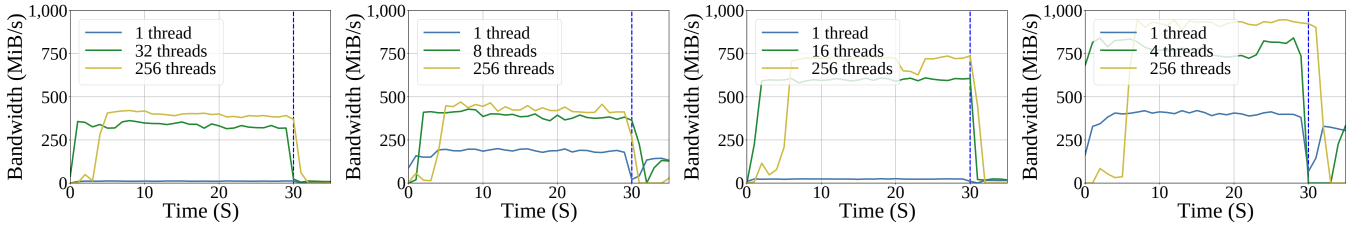
Firstly, with a single thread, Milvus with the storage-based indexes, DiskANN, has 13.1% to 96.7% higher latency than Milvus-HNSW on the four datasets. However, Milvus-DiskANN achieves 3.9% to 44.5% lower latency than Milvus-IVF in three out of the four datasets, with the exception of OpenAI 500K, on which Milvus-DiskANN has 27.3% higher latency than Milvus-IVF. When the number of threads increases to 256, Milvus-DiskANN demonstrates 7.7% and 50.8% higher P99 latency than Milvus-HNSW on the Cohere 1M and OpenAI 500K datasets, respectively. This latency difference becomes even more pronounced, up to 8.6× and 20.1×, when the dataset size increases to 10×. In contrast, DiskANN achieves up to 53.6% lower P99 latency than the IVF index under similar conditions. This leads to a similar observation as the throughput that **O-7:** *vector index is the main factor that affects the tail latency of vector query, but vector databases with storage-based setups do not necessarily have higher latency than memory-based setups.*

Secondly, with the same vector index HNSW, Milvus achieves up to 75.6% and 93.8% lower latency than Qdrant and Weaviate with a single thread. As concurrency increases to 256, Qdrant and Weaviate exhibit 17.6% and 3.2% lower latency than Milvus on the Cohere 10M dataset, whereas Milvus maintains up to 73.2% and 96.1% lower P99 latency than Qdrant and Weaviate on the other three datasets, indicating that **O-8:** *indexing method is not the only factor that affects the P99 tail latency of vector search, vector databases with the same index have a up to 96.1% latency difference.*

Thirdly, as the concurrency increases, the tail latency first stays flat and then increases. **O-9:** *Interestingly, when the dataset size increases 10× with both Cohere and OpenAI, the P99 latency of Milvus-IVF, Milvus-HNSW, and Milvus-DiskANN reduced up to 8.0%, 37.6%, and 47.0% whereas the P99 latency increases up to 96.9% and 18.7% for Weaviate and Qdrant.*

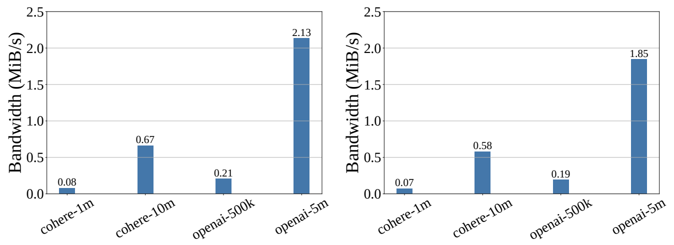
To further investigate if the CPU is the bottleneck during vector search, we record the CPU utilization during the vector search and plot the average global CPU utilization for Cohere 10M and OpenAI 5M in Fig. 4 on the y-axis as the concurrency increases on the x-axis. We find a strong connection between throughput and CPU usage. For Milvus with IVF and DiskANN, both throughput and CPU usage plateau after 4 concurrent query threads, and for Qdrant and Weaviate, this number grows to 32. However, we observe less correlation between CPU usage and P99 latency than throughput, where the P99 latency of Milvus of all three index settings starts to increase from 1 concurrent thread. The P99 latency of Qdrant and Weaviate starts to increase after 8 and 16 threads, before the CPU usage stops increasing.

Answering RQ1 with KF1: We observe that the storage-based setups can have lower throughput and higher latency than memory-based setups (Milvus-HNSW vs. Milvus-DiskANN). However, vector databases with storage-based indexes do not *necessarily* have worse performance than memory-based indexes. For example, Milvus-DiskANN has higher throughput and lower latency than Milvus-IVF, which is the same vector database with memory-based indexes, and the other three vector databases with memory-based indexes in three out of four datasets. We also summarize that although the vector index is an important factor that affects the performance of vector databases (Milvus with different indexes), different vector databases can have a performance difference with the same vector index, with a 7.1× throughput difference in our benchmarked settings.



(a) Cohere 1M (b) Cohere 10M (c) OpenAI 500K (d) OpenAI 5M

Figure 5: Read bandwidth of Milvus with DiskANN index during vector search on the four vector datasets under different query concurrencies, where each thread has one in-flight query.



(a) Concurrency = 1 (b) Concurrency = 256

Figure 6: Average read bandwidth of Milvus with DiskANN index during vector search on the four vector datasets.

V. I/O Workload Characterization

In the previous section, we compared the P99 tail latency, throughput, and scalability between memory-based and storage-based vector database setups. Our analysis indicates that storage-based setups do not necessarily result in lower throughput or higher latency compared to memory-based setups. Given that modern NVMe SSDs are able to deliver millions of I/O operations/s with tens of μs of latency, it is important to investigate whether storage-based setups can fully utilize the potential of these high-performance SSDs. To better understand these observed performance outcomes, this section specifically examines the I/O characteristics of vector databases to answer *RQ2: What are the I/O characteristics of the benchmarked vector databases, and are they able to fully utilize the bandwidth and throughput of modern SSDs?* We focus on DiskANN, given its superior performance over its storage-based counterparts, LanceDB with the IVF index.

A. I/O Bandwidth During Search

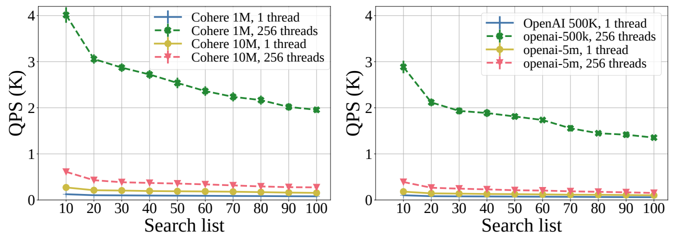
What is the I/O bandwidth of the benchmarked vector databases with storage-based setups, and how does the I/O traffic scale with an increasing number of concurrency? To answer this question, we collect the block-level I/O traces of Milvus-DiskANN with three query concurrencies: (1) concurrency = 1, (2) concurrency = when the throughput plateaus, and (3) concurrency = 256. It is expected that for all four databases, the bandwidth increases as we increase the concurrency since increasing concurrency leads to higher throughput, meaning that more work is completed within a fixed length of time. Consequently, as the vector databases finish more work, it is expected to read more data from the storage device.

Fig. 5 shows the per-second averaged bandwidth (y-axis) of the Milvus-DiskANN with three concurrency levels during vector search. The experiment's start time aligns with the initiation of vector searches, where all the query threads are synced at time 0 on the x-axis. However, due to this synchronization, the I/O traffic at concurrency level 256 begins later than the actual start of the experiment (x = 0). The experiment ends at the 30-second mark, indicated by the horizontal blue line. The read bandwidth remains stable during the search, indicating a stable I/O workload during the vector search. Firstly, the SSD is not saturated during vector similarity search in our hardware setup, where **O-10:** *the maximum bandwidth achieved in the experiments is 658.8 MiB/s, achieved with 256 threads with OpenAI 5M, with 8.9% of the SSD's maximum bandwidth of 7.2 GiB/s.* Secondly, **O-11:** *the average read bandwidth with one thread increases to 16.7× and 17.4× when the dataset size increases 10×, however, this increase reduces to 6.9% and 36.9% with 256 threads.* We hypothesize that this decrease in bandwidth when the number of threads increases from 1 to 256 is caused by the CPU becoming the bottleneck. Thirdly, **O-12:** *increasing concurrency from 1 to 256 results in a substantial bandwidth increase for smaller datasets — 28.8× for Cohere 1M and 22.8× for OpenAI 500K, but the scaling diminishes notably with larger datasets, only 1.8× and 1.9× for Cohere 10M and OpenAI 5M, respectively.*

B. Per-query I/O Load During Vector Search

In this subsection, we analyze the performance at a finer granularity by investigating **how the per-query bandwidth changes with increasing dataset size and concurrency**.

Fig. 6 shows the average read bandwidth per query on the y-axis with two different concurrencies, 1 and 256. Firstly, as concurrency increases from 1 to 256, the per-query bandwidth decreases across all four datasets, with reductions ranging from 9.5% to 13.4%. This indicates that **O-13:** *higher concurrency leads to lower per-query bandwidth.* Secondly, **O-14:** *when the size of the dataset increases by a factor of 10, the per-query average bandwidth of DiskANN increases to 8.4× and 10.1× with the Cohere and OpenAI datasets, respectively.* This indicates that larger datasets require higher per-request bandwidth, raising concerns about whether the SSD will be



(a) Cohere (b) OpenAI

Figure 7: Vector search throughput of Milvus-DiskANN with an increasing value of `search_list`.

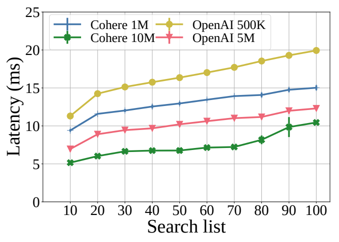


Figure 8: Vector search P99 tail latency of Milvus-DiskANN with an increasing `search_list`.

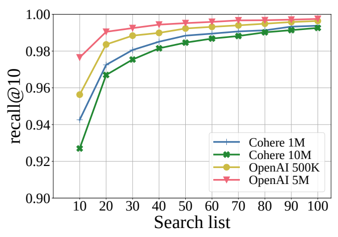


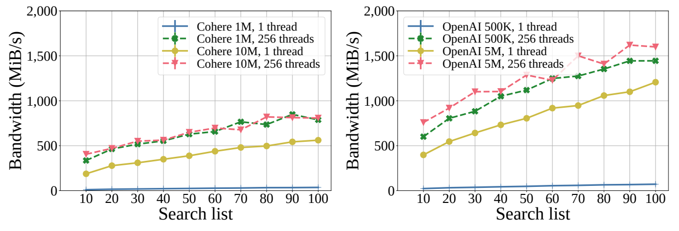
Figure 9: Accuracy of Milvus-DiskANN with an increasing `search_list` (recall@10).

the bottleneck with larger datasets, for example, when the datasets increase to a billion-level. Additionally, we measured the size distribution of I/O requests (not shown in Figure 6) and found that **O-15:** *the I/O requests are dominated by 4 KiB requests (over 99.99% of the I/O requests are 4 KiB for all the lines in Fig. 6)* . Thus, the I/O granularity is not affected by the level of concurrency and the size of the database.

Answering RQ2 with KF2: Our experiments in this section find that the storage-based setup for Milvus, DiskANN, achieves the highest 658.8 MiB/s during vector search, thus can not saturate the SSD. However, further analysis shows that per-query bandwidth exhibits an 8.4× and 10.1× increase when the dataset size increases by a factor of 10, raising concerns that the SSD is the bottleneck with larger vector datasets.

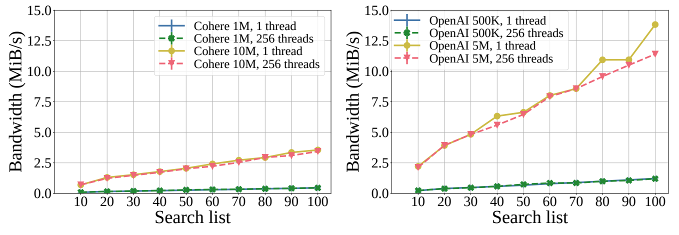
VI. The Effect of Index Parameters

In the previous section, we investigated the I/O characteristics of storage-based setups of vector databases with Milvus-DiskANN. DiskANN offers several configurable parameters that allow users to trade off between the performance of vector search and accuracy, which is crucial for practical deployment. In this section, we address *RQ3: How do search-time parameters affect performance and I/O characteristics?* Specifically, we focus on two search-time parameters of DiskANN that are supported by Milvus: `search_list` and `beam_width` (see §II for the definition of these two parameters).



(a) Cohere (b) OpenAI

Figure 10: Read bandwidth of Milvus with DiskANN during vector search with an increasing `search_list`.



(a) Cohere (b) OpenAI

Figure 11: Average read bandwidth of Milvus with DiskANN during vector search with an increasing `search_list`.

A. The Effect of `search_list`

How does the `search_list` affect the performance and I/O characteristics of DiskANN? `search_list` is the size of the candidate list during the search. Increasing `search_list` leads to more vectors being checked during the vector search, thus resulting in higher search accuracy at the cost of performance and potentially higher I/O overhead, where higher search latency and lower throughput are expected. Fig. 9 shows the accuracy (`recall@10`) of vector similarity search of DiskANN on the y-axis as `search_list` increases on the x-axis (note that the y-axis starts from 0.9). The highest improvement in accuracy occurs when `search_list` increases from 10 to 20, resulting in a gain of 1.0% to 4.3%. When `search_list` increases from 10 to 100, the total accuracy gain ranges from 2.0% to 6.5%. **O-16:** *These results demonstrate that larger `search_list` values provide diminishing returns in accuracy.*

This increased accuracy comes at the expense of reduced throughput and increased latency. Fig. 7 shows the throughput (y-axis) of vector similarity search as `search_list` increases on the x-axis. **O-17:** *With a single thread, the throughput decreases from 36.3% to 43.8% when `search_list` increases from 10 to 100.* **O-18:** *When the number of threads increases to 256, the reduction is more pronounced, with throughput decreasing by 51.2% to 60.9%.* Fig. 8 shows the P99 latency (y-axis) as `search_list` increases with one query thread. **O-19:** *When `search_list` increases from 10 to 100, the P99 latency increases 59.7%, 102.5%, 76.2%, and 77.0% with the four datasets.*

As `search_list` increases, it is expected that the number of vectors checked increases, leading to higher I/O traffic.

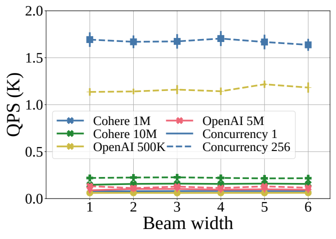


Figure 12: Vector search throughput of Milvus-DiskANN with an increasing `beamwidth`.

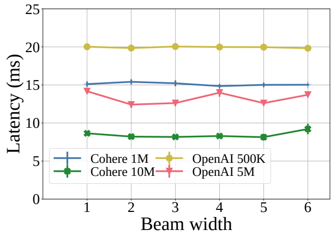


Figure 13: Vector search P99 tail latency of Milvus-DiskANN with an increasing `beamwidth`.

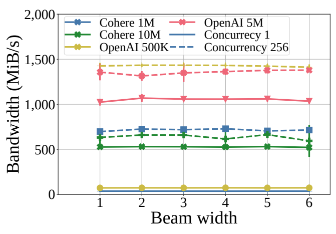


Figure 14: Read bandwidth of Milvus-DiskANN during vector search with an increasing `beamwidth`.

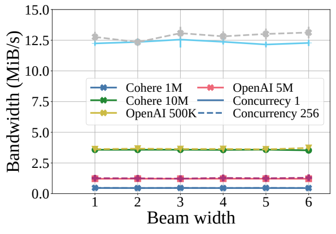


Figure 15: Average read bandwidth of Milvus-DiskANN during vector search with an increasing `beamwidth`.

To further investigate this negative performance effect of increasing `search_list`, we analyze the I/O workloads as `search_list` increases. Fig. 10 and Fig. 11 show the total and per-query average bandwidth on the y-axis as `search_list` increases on the x-axis. **O-20:** *When `search_list` increases from 10 to 100, the total bandwidth increases from 3.0× to 3.3× with a single thread and from 2.0× to 2.4× with 256 threads.* In contrast, the per-query average bandwidth increases from 5.1× to 6.3× with one thread and 4.9× to 5.4× with 256 threads. **O-21:** *Both total bandwidth and per-query average bandwidth increase as `search_list` increases. However, the maximum bandwidth achieved is 1620.0 MiB/s, thus not saturating the SSD .*

B. The Effect of `beam_width`

Next, we investigate how varying `beam_width` affects the performance and I/O characteristics in Milvus with DiskANN. Increasing the `beam_width` allows more I/O operations to be issued in parallel, which generally reduces query latency if the storage devices do not become the bottleneck. Milvus provides `BeamWidthRatio`, which specifies the number of I/O requests that each CPU core can issue per search iteration. For clarity, in this section, we refer to `beam_width` as the number of I/O requests per search iteration per CPU core, which is consistent with the definition of `BeamWidthRatio` in Milvus. To ensure that the system can fully utilize increased parallelism as `beam_width` grows, we set `queryNode.scheduler.maxReadConcurrentRatio`, which is the max read concurrency per CPU core, and `search_list` to 100. This configuration ensures that a sufficient number of candidate vectors are available for parallel processing, preventing the search from becoming bottlenecked by candidate availability rather than by the effects of `beam_width` itself.

Fig. 12 and Fig. 13 show the throughput and P99 tail latency on the y-axis as `beam_width` increases on the x-axis. We observe that both throughput and latency fluctuate as `beam_width` increases. Similarly, Fig. 14 and Fig. 15 depict the total bandwidth and the per-query average bandwidth as `beam_width` increases. The bandwidth also fluctuates without displaying any distinct trend, similar to our earlier performance observations for throughput and latency.

O-22: *We hypothesize that the lack of observable patterns in throughput, latency, and bandwidth as `beam_width` increases is due to limitations in the system configuration. Further investigation is needed to understand system behavior when `beam_width` can be set below the total number of CPU cores.*

Answering RQ3 with KF3: Increasing `search_list` leads to higher accuracy, with the cost of lower throughput, higher latency, and higher bandwidth. Despite the increase in I/O, Milvus with DiskANN does not fully saturate the SSD at any tested `search_list` value. The most significant accuracy improvement occurs when `search_list` increases from 10 to 20, further increasing yield and diminishing returns in accuracy. However, the degradation in performance (throughput and P99 latency) and increased I/O bandwidth do not exhibit the same diminishing returns as accuracy, making it important to tune `search_list` to balance performance and accuracy carefully.

VII. Related Work

Vector management systems: A variety of systems have been developed to manage vector data, such as libraries [3], [10], vector plugins for SQL databases [12], and vector databases [2], [9], [13], [22], [34], [69]. Vector databases are full-fledged systems that provide vector query, insertion, deletion, indexing, and many features that are not directly related to ANNS, such as allowing each vector to be associated with additional data (payload), payload-based filtering, and a distributed setup [2], [9], [13], [22]. In this work, we investigate the performance and I/O characteristics of vector databases with a disk-based setup with vector search workloads.

Approximate Nearest Neighbor Search (ANNS): ANNS is the core of vector similarity search in vector databases, where the vector databases are based on ANNS indexes for quick vector similarity search. There are two widely used ANNS indexes: cluster-based indexes [24], [27], [29], [75], [76] and graph-based indexes [34], [53]–[55], [61], [62]. Cluster-based indexes fit the access granularity of SSDs but have large space amplification due to the replication of the vectors near cluster borders for higher accuracy [29]. Other indexes, such as tree-based indexes [26], [56], [57], [67], [70] and hash-based indexes [37], [42], [43], [66], [71] have scalability or accuracy issues [30], [74], and these indexes are not supported in our benchmarked vector databases [9], [11], [13], [22].

Disk-optimized ANNS indexes: Many studies propose disk-based ANNS indexes to reduce the memory cost. There are two main kinds of disk-based ANNS indexes: cluster-based [29], [75], [78] and graph-based indexes [59], [68], [72]. DiskANN [68] stores an index of product quantization compressed vectors in memory and a graph index of the original vectors in SSDs, and has variants for filtered search and streaming search [38], [45], [65]. Graph-based indexes for storage can be optimized by reducing the search path and increasing the data locality [59], [72]. Cluster-based indexes, such as SPANN [29], store the centroid points in memory and the posting lists (the vectors in the cluster) in storage devices. SPFresh [75] provides an in-place update for vectors with cluster-based indexes. However, these disk-based indexing algorithms are not widely adopted in vector databases. Among our selected vector databases, only Milvus supports DiskANN as a disk-friendly option [11].

Performance measurement of vector databases: Many vector databases provide benchmarking tools with their vector databases, such as Ziliz [19], Qdrant [14], and Weaviate [21]. However, these benchmarks typically report only basic metrics such as throughput, latency, and recall, without analyzing how performance scales with dataset size or query concurrency. ANN Benchmarks [1] compare the performance of different vector databases/libraries but lack storage-based evaluations and in-depth analysis. The main goal of our work is to provide insights into how vector databases, specifically with storage-based setups, scale with concurrency and the size of datasets, and investigate the I/O characteristics of vector databases with storage-based setups. [77] compares vector databases with relational databases enhanced with vector support to identify fundamental limitations in relational databases, while others [23], [73] survey and compare the performance of graph-based in-memory vector search algorithms. [35] focuses on the performance of graph-based ANNS methods on edge devices. Specifically, [30] compares the performance and I/O pattern of DiskANN [68] and SPFresh [75], two state-of-the-art ANNS indexes for vector search. Our work is orthogonal to these benchmarks, where we focus on the performance and I/O characteristics of full-fledged vector databases instead of ANNS algorithms.

VIII. Limitations and Future Work

Our work compares the performance of vector databases in storage-based setups against memory-based setups, investigating the I/O characteristics using four datasets. Our work can be expanded in the following directions.

Firstly, our largest dataset comprises 10 million vectors with 768 dimensions and 5 million vectors with 1536 dimensions, illustrating how performance and I/O characteristics scale when the dataset size increases by a factor of 10. However, existing RAG systems now handle trillions of tokens and billions of vectors, so it would be valuable to investigate performance and I/O scaling at even larger scales, such as billions of vectors.

Secondly, our benchmarks focus on a single-machine setup with local SSD storage. However, benchmarked vector databases, such as Milvus, support distributed setups with compute-storage disaggregation, where index files and vector data reside in remote object storage. Future studies could examine the performance characteristics and scalability of storage-based setups in distributed environments utilizing remote storage.

Thirdly, we have concentrated exclusively on vector similarity search performance without assessing performance and I/O characteristics under workloads involving insertion, deletion, or filtered search operations. We observed that the I/O workloads are dominated by 4 KiB reads during vector search. However, with data mutation operations such as insert and deletion, writes are expected to occur in the I/O workloads. Since NAND SSDs have read-write interference, meaning that the read throughput decreases and the latency increases with concurrent writes. This research can be extended by measuring and analyzing performance and I/O characteristics under such hybrid read-write workloads.

IX. Conclusion

In this work, we investigate the suitability of vector databases for deployment on modern NVMe SSDs by quantifying the performance between memory-based and storage-based setups across different vector databases. We characterize their I/O behaviors and analyze how index parameters influence both performance and I/O characteristics. Our findings reveal three key insights: (1) storage-based setups in vector databases can have lower throughput and higher latency than the memory-based setups within the same vector database, but do not necessarily lead to worse performance than memory-based setups, especially when this comparison is extended to different vector databases, (2) the choice of vector database itself is a major factor influencing search performance, even when using the same vector index, and (3) state-of-the-practice vector databases with storage-based setups require optimizations on I/O traffic to fully utilize the performance with flash SSDs, whereas the parameters of the indexes should also be taken into consideration as a factor that affects the I/O behavior. Future work includes extending performance characterization of storage-based vector databases to more complex workloads that combine insertions, deletions, and searches, as well as integrating state-of-the-art vector indexing techniques into these systems.

Acknowledgment

We thank the IISWC'25 reviewers for their invaluable and constructive feedback. The authors would also like to thank the AtLarge group at Vrije Universiteit Amsterdam for their help with the paper.

References

[1] "ANNS benchmarks," https://ann-benchmarks.com/index.html, Accessed: 2025-09-01.

[2] "Chroma," https://www.trychroma.com, Accessed: 2025-09-01.

[3] "Faiss: A library for efficient similarity search," https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/, Accessed: 2025-09-01.

[4] "faiss: Guidelines to choose an index," https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index?, Accessed: 2025-09-01.

[5] "Find anything blazingly fast with Google's vector search technology," https://cloud.google.com/blog/topics/developers-practitioners/find-anything-blazingly-fast-googles-vector-search-technology, Accessed: 2025-09-01.

[6] "fio - Flexible I/O tester," https://fio.readthedocs.io/en/latest/fio_doc.html, Accessed: 2025-09-01.

[7] "GitHub:bpftrace/bptrace," https://github.com/bpftrace/bpftrace, Accessed: 2025-09-01.

[8] "How to get started with Qdrant locally," https://qdrant.tech/documentation/quickstart/, Accessed: 2025-09-01.

[9] "LanceDB - The database for multimodal AI," https://lancedb.com, Accessed: 2025-09-01.

[10] "Microsoft SPTAG," https://github.com/microsoft/SPTAG, Accessed: 2025-09-01.

[11] "Milvus — High-performance vector database built for scale," https://milvus.io, Accessed: 2025-09-01.

[12] "pgvector," https://github.com/pgvector/pgvector, Accessed: 2025-09-01.

[13] "Qdrant - Vector Database," https://qdrant.tech, Accessed: 2025-09-01.

[14] "Qdrant: vector database benchmarks," https://qdrant.tech/benchmarks/, Accessed: 2025-09-01.

[15] "Quick start: LanceDB," https://lancedb.github.io/lancedb/basic/, Accessed: 2025-09-01.

[16] "Run Milvus in Docker (Linux)," https://milvus.io/docs/install_standalone-docker.md, Accessed: 2025-09-01.

[17] "Samsung 990 pro," https://semiconductor.samsung.com/consumer-storage/internal-ssd/990-pro/, Accessed: 2025-09-01.

[18] "Vector distance metric types," https://milvus.io/docs/metric.md, Accessed: 2025-09-01.

[19] "VectorDBBench - A vector database benchmark tool," https://zilliz.com/vector-database-benchmark-tool, Accessed: 2025-09-01.

[20] "Weavaite: How to install: docker," https://weaviate.io/developers/weaviate/installation/docker-compose, Accessed: 2025-09-01.

[21] "Weaviate: ANN benchmark," https://weaviate.io/developers/weaviate/benchmarks/ann, Accessed: 2025-09-01.

[22] "Weaviate: The AI-native database developers love," https://weaviate.io, Accessed: 2025-09-01.

[23] I. Azizi, K. Echihabi, and T. Palpanas, "Graph-based vector search: An experimental evaluation of the state-of-the-art," *Proc. ACM Manag. Data*, vol. 3, no. 1, pp. 43:1–43:31, 2025. [Online]. Available: https://doi.org/10.1145/3709693

[24] A. Babenko and V. S. Lempitsky, "The inverted multi-index," in *2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, June 16-21, 2012*. IEEE Computer Society, 2012, pp. 3069–3076. [Online]. Available: https://doi.org/10.1109/CVPR.2012.6248038

[25] ——, "Efficient indexing of billion-scale datasets of deep descriptors," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 2055–2063. [Online]. Available: https://doi.org/10.1109/CVPR.2016.226

[26] ——, "Product split trees," in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 6316–6324. [Online]. Available: https://doi.org/10.1109/CVPR.2017.669

[27] D. Baranchuk, A. Babenko, and Y. Malkov, "Revisiting the inverted indices for billion-scale approximate nearest neighbors," in *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XII*, ser. Lecture Notes in Computer Science, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds., vol. 11216. Springer, 2018, pp. 209–224. [Online]. Available: https://doi.org/10.1007/978-3-030-01258-8_13

[28] S. Borgeaud, A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, G. van den Driessche, J. Lespiau, B. Damoc, A. Clark, D. de Las Casas, A. Guy, J. Menick, R. Ring, T. Hennigan, S. Huang, L. Maggiore, C. Jones, A. Cassirer, A. Brock, M. Paganini, G. Irving, O. Vinyals, S. Osindero, K. Simonyan, J. W. Rae, E. Elsen, and L. Sifre, "Improving language models by retrieving from trillions of tokens," in *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvári, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 2022, pp. 2206–2240. [Online]. Available: https://proceedings.mlr.press/v162/borgeaud22a.html

[29] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "SPANN: highly-efficient billion-scale approximate nearest neighborhood search," in *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, M. Ranzato, A. Beygelzimer, Y. N. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021, pp. 5199–5212. [Online]. Available: https://proceedings.neurips.cc/paper/2021/hash/299dc35e747eb77177d9cea10a802da2-Abstract.html

[30] R. Chen, Y. Peng, X. Wei, H. Xie, R. Chen, S. Shen, and H. Chen, "Characterizing the dilemma of performance and index size in billion-scale vector search and breaking it with second-tier memory," *CoRR*, vol. abs/2405.03267, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2405.03267

[31] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*, S. Sen, W. Geyer, J. Freyne, and P. Castells, Eds. ACM, 2016, pp. 191–198. [Online]. Available: https://doi.org/10.1145/2959100.2959190

[32] C. Fu and D. Cai, "EFANNA : An extremely fast approximate nearest neighbor search algorithm based on knn graph," *CoRR*, vol. abs/1609.07228, 2016. [Online]. Available: http://arxiv.org/abs/1609.07228

[33] C. Fu, C. Wang, and D. Cai, "High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 8, pp. 4139–4150, 2022. [Online]. Available: https://doi.org/10.1109/TPAMI.2021.3067706

[34] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *Proc. VLDB Endow.*, vol. 12, no. 5, pp. 461–474, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p461-fu.pdf

[35] A. Ganbarov, J. Yuan, A. Le-Tuan, M. Hauswirth, and D. L. Phuoc, "Experimental comparison of graph-based approximate nearest neighbor search algorithms on edge devices," *CoRR*, vol. abs/2411.14006, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2411.14006

[36] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, Q. Guo, M. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *CoRR*, vol. abs/2312.10997, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2312.10997

[37] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, Eds. Morgan Kaufmann, 1999, pp. 518–529. [Online]. Available: http://www.vldb.org/conf/1999/P49.pdf

[38] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, S. Raz, Y. Lin, Y. Zhang, N. Mahapatro, P. Srinivasan, A. Singh, and H. V. Simhadri, "Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters," in *Proceedings of the ACM Web Conference 2023, WWW 2023, Austin, TX, USA, 30 April 2023 - 4 May 2023*, Y. Ding, J. Tang, J. F. Sequeda, L. Aroyo, C. Castillo, and G. Houben, Eds. ACM, 2023, pp. 3406–3416. [Online]. Available: https://doi.org/10.1145/3543507.3583552

[39] M. Grbovic and H. Cheng, "Real-time personalization using embeddings for search ranking at airbnb," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Y. Guo and F. Farooq, Eds. ACM, 2018, pp. 311–320. [Online]. Available: https://doi.org/10.1145/3219819.3219885

[40] R. Guo, X. Luan, L. Xiang, X. Yan, X. Yi, J. Luo, Q. Cheng, W. Xu, J. Luo, F. Liu, Z. Cao, Y. Qiao, T. Wang, B. Tang, and C. Xie, "Manu: A cloud native vector database management system," *Proc. VLDB*

Endow., vol. 15, no. 12, pp. 3548–3561, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p3548-yan.pdf

[41] B. Harwood and T. Drummond, "FANNG: fast approximate nearest neighbour graphs," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 5713–5722. [Online]. Available: https://doi.org/10.1109/CVPR.2016.616

[42] J. He, W. Liu, and S. Chang, "Scalable similarity search with optimized kernel hashing," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, B. Rao, B. Krishnapuram, A. Tomkins, and Q. Yang, Eds. ACM, 2010, pp. 1129–1138. [Online]. Available: https://doi.org/10.1145/1835804.1835946

[43] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng, "Query-aware locality-sensitive hashing for approximate nearest neighbor search," *Proc. VLDB Endow.*, vol. 9, no. 1, pp. 1–12, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol9/p1-huang.pdf

[44] G. Izacard, M. Caron, L. Hosseini, S. Riedel, P. Bojanowski, A. Joulin, and E. Grave, "Unsupervised dense information retrieval with contrastive learning," *Trans. Mach. Learn. Res.*, vol. 2022, 2022. [Online]. Available: https://openreview.net/forum?id=jKN1pXi7b0

[45] S. Jaiswal, R. Krishnaswamy, A. Garg, H. V. Simhadri, and S. Agrawal, "Ood-diskann: Efficient and scalable graph ANNS for out-of-distribution queries," *CoRR*, vol. abs/2211.12850, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2211.12850

[46] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 1, pp. 117–128, 2011. [Online]. Available: https://doi.org/10.1109/TPAMI.2010.57

[47] Z. Jing, Y. Su, Y. Han, B. Yuan, H. Xu, C. Liu, K. Chen, and M. Zhang, "When large language models meet vector databases: A survey," *CoRR*, vol. abs/2402.01763, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.01763

[48] V. Karpukhin, B. Oguz, S. Min, P. S. H. Lewis, L. Wu, S. Edunov, D. Chen, and W. Yih, "Dense passage retrieval for open-domain question answering," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds. Association for Computational Linguistics, 2020, pp. 6769–6781. [Online]. Available: https://doi.org/10.18653/v1/2020.emnlp-main.550

[49] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[50] J. Li, H. Liu, C. Gui, J. Chen, Z. Ni, N. Wang, and Y. Chen, "The design and implementation of a real time visual search system on JD e-commerce platform," in *Proceedings of the 19th International Middleware Conference, Middleware Industrial Track 2018, Rennes, France, December 10-14, 2018*. ACM, 2018, pp. 9–16. [Online]. Available: https://doi.org/10.1145/3284028.3284030

[51] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 8, pp. 1475–1488, 2020. [Online]. Available: https://doi.org/10.1109/TKDE.2019.2909204

[52] F. Liu, Z. Kang, and X. Han, "Optimizing RAG techniques for automotive industry PDF chatbots: A case study with locally deployed ollama models," *CoRR*, vol. abs/2408.05933, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.05933

[53] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Inf. Syst.*, vol. 45, pp. 61–68, 2014. [Online]. Available: https://doi.org/10.1016/j.is.2013.10.006

[54] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, pp. 824–836, 2020. [Online]. Available: https://doi.org/10.1109/TPAMI.2018.2889473

[55] M. D. Manohar, Z. Shen, G. E. Blelloch, L. Dhulipala, Y. Gu, H. V. Simhadri, and Y. Sun, "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, PPoPP 2024, Edinburgh, United Kingdom, March 2-6, 2024*, M. Steuwer, I. A. Lee, and M. Chabbi, Eds. ACM, 2024, pp. 270–285. [Online]. Available: https://doi.org/10.1145/3627535.3638475

[56] M. McCartin-Lim, A. McGregor, and R. Wang, "Approximate principal direction trees," in *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*. icml.cc / Omnipress, 2012. [Online]. Available: http://icml.cc/2012/papers/348.pdf

[57] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 11, pp. 2227–2240, 2014. [Online]. Available: https://doi.org/10.1109/TPAMI.2014.2321376

[58] J. A. V. Muñoz, M. A. Gonçalves, Z. Dias, and R. da Silva Torres, "Hierarchical clustering-based graphs for large scale approximate nearest neighbor search," *Pattern Recognit.*, vol. 96, 2019. [Online]. Available: https://doi.org/10.1016/j.patcog.2019.106970

[59] J. Ni, X. Xu, Y. Wang, C. Li, J. Yao, S. Xiao, and X. Zhang, "Diskann++: Efficient page-based search over isomorphic mapped graph index using query-sensitivity entry vertex," *CoRR*, vol. abs/2310.00402, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.00402

[60] S. Okura, Y. Tagami, S. Ono, and A. Tajima, "Embedding-based news recommendation for millions of users," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 2017, pp. 1933–1942. [Online]. Available: https://doi.org/10.1145/3097983.3098108

[61] Z. Peng, M. Zhang, K. Li, R. Jin, and B. Ren, "iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, M. M. Dehnavi, M. Kulkarni, and S. Krishnamoorthy, Eds. ACM, 2023, pp. 313–328. [Online]. Available: https://doi.org/10.1145/3572848.3577527

[62] J. Ren, M. Zhang, and D. Li, "HM-ANN: efficient billion-point nearest neighbor search on heterogeneous memory," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/788d986905533aba051261497ecffcbb-Abstract.html

[63] Z. Ren, K. Doekemeijer, N. Tehrany, and A. Trivedi, "Bfq, multiqueue-deadline, or kyber? performance characterization of linux storage schedulers in the nvme era," in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, ICPE 2024, London, United Kingdom, May 7-11, 2024*, S. Balsamo, W. J. Knottenbelt, C. L. Abad, and W. Shang, Eds. ACM, 2024, pp. 154–165. [Online]. Available: https://doi.org/10.1145/3629526.3645053

[64] Z. Ren and A. Trivedi, "Performance characterization of modern storage stacks: POSIX i/o, libaio, spdk, and io_uring," in *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems, CHEOPS 2023, Rome, Italy, 8 May 2023*, J. Acquaviva, S. Ibrahim, and S. Byna, Eds. ACM, 2023, pp. 35–45. [Online]. Available: https://doi.org/10.1145/3578353.3589545

[65] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri, "Freshdiskann: A fast and accurate graph-based ANN index for streaming similarity search," *CoRR*, vol. abs/2105.09613, 2021. [Online]. Available: https://arxiv.org/abs/2105.09613

[66] J. Song, Y. Yang, Z. Huang, H. T. Shen, and R. Hong, "Multiple feature hashing for real-time large scale near-duplicate video retrieval," in *Proceedings of the 19th International Conference on Multimedia 2011, Scottsdale, AZ, USA, November 28 - December 1, 2011*, K. S. Candan, S. Panchanathan, B. Prabhakaran, H. Sundaram, W. Feng, and N. Sebe, Eds. ACM, 2011, pp. 423–432. [Online]. Available: https://doi.org/10.1145/2072298.2072354

[67] R. F. Sproull, "Refinements to nearest-neighbor searching in k-dimensional trees," *Algorithmica*, vol. 6, no. 4, pp. 579–589, 1991. [Online]. Available: https://doi.org/10.1007/BF01759061

[68] S. J. Subramanya, Devvrit, R. Kadekodi, R. Krishaswamy, and H. V.

Simhadri, *DiskANN: fast accurate billion-point nearest neighbor search on a single node*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[69] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, "Milvus: A purpose-built vector data management system," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 2614–2627. [Online]. Available: https://doi.org/10.1145/3448016.3457550

[70] J. Wang, N. Wang, Y. Jia, J. Li, G. Zeng, H. Zha, and X. Hua, "Trinary-projection trees for approximate nearest neighbor search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 2, pp. 388–403, 2014. [Online]. Available: https://doi.org/10.1109/TPAMI.2013.125

[71] J. Wang, S. Kumar, and S. Chang, "Semi-supervised hashing for large-scale search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 12, pp. 2393–2406, 2012. [Online]. Available: https://doi.org/10.1109/TPAMI.2012.48

[72] M. Wang, W. Xu, X. Yi, S. Wu, Z. Peng, X. Ke, Y. Gao, X. Xu, R. Guo, and C. Xie, "Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment," *Proc. ACM Manag. Data*, vol. 2, no. 1, pp. V2mod014:1–V2mod014:27, 2024. [Online]. Available: https://doi.org/10.1145/3639269

[73] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 1964–1978, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1964-wang.pdf

[74] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai, "Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3152–3165, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p3152-wei.pdf

[75] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang, P. Cheng, and M. Yang, "Spfresh: Incremental in-place update for billion-scale vector search," in *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds. ACM, 2023, pp. 545–561. [Online]. Available: https://doi.org/10.1145/3600006.3613166

[76] M. Zhang and Y. He, "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, CIKM 2019, Beijing, China, November 3-7, 2019*, W. Zhu, D. Tao, X. Cheng, P. Cui, E. A. Rundensteiner, D. Carmel, Q. He, and J. X. Yu, Eds. ACM, 2019, pp. 1673–1682. [Online]. Available: https://doi.org/10.1145/3357384.3357938

[77] Y. Zhang, S. Liu, and J. Wang, "Are there fundamental limitations in supporting vector data management in relational databases? A case study of postgresql," in *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 2024, pp. 3640–3653. [Online]. Available: https://doi.org/10.1109/ICDE60146.2024.00280

[78] Z. Zhang, F. Liu, G. Huang, X. Liu, and X. Jin, "Fast vector query processing for large datasets beyond GPU memory with reordered pipelining," in *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, L. Vanbever and I. Zhang, Eds. USENIX Association, 2024, pp. 23–40. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/zhang-zili-pipelining

[79] S. Zhao, Y. Yang, Z. Wang, Z. He, L. Qiu, and L. Qiu, "Retrieval augmented generation (RAG) and beyond: A comprehensive survey on how to make your llms use external data more wisely," *CoRR*, vol. abs/2409.14924, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2409.14924

A. Artifact Appendix

A.1 Abstract

This artifact contains the scripts and traces of *Storage-Based Approximate Nearest Neighbor Search: What are the Performance, Cost, and I/O Characteristics?* We details the environment setup and the procedures to reproduce our results in this appendix.

A.2 Artifact check-list (meta-information)

- **Program:** VectorDBBench, Milvus 2.5.11, Qdrant 1.14.1, Weaviate 1.31.0, LanceDB 0.23.0, Python, Docker, bpftrace.
- **Data set:** Provided by VectorDBBench.
- **Run-time environment:** Ubuntu 22.04 with 5.15.0-142-generic kernel.
- **Hardware:** x86-64 machine, NVMe SSD.
- **Execution:** Command line.
- **Metrics:** Throughput, P99 tail latency, CPU utilization, bandwidth, recall.
- **Output:** I/O traces and vector search throughput and latency.
- **How much disk space required (approximately)?:** 1.5 TiB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** two to three weeks deepening on the CPU.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT license.
- **Data licenses (if publicly available)?:** MIT license for the data collected in this paper.
- **Workflow automation framework used?:** No.
- **Archived (provide DOI)?:** `https://doi.org/10.5281/zenodo.16916495`.

A.3 Description

A.3.1 How to access

The code and traces can be accessed via GitHub link: `https://github.com/ZebinRen/2025-iiswc-vectordb-bench-artifact-public`. Permanent link (Zenodo): `https://doi.org/10.5281/zenodo.16916496`. Due to the file size limitation of GitHub, the I/O traces are only avaiable via Zenodo.

A.3.2 Hardware dependencies

To reproduce the results, a x86-64 base machine with NVMe SSD is required.

A.3.3 Software dependencies

The experiments in this paper are carried out with Ubuntu 22.04 with kernel version of 5.15.0-142-generic. It is expected that the experiments are able to run with other Linux distribution with other kernel versions. We use `bpftrace` to trace the I/O workloads during the vector similarity search, root access is required to use `bpftrace`.

A.3.4 Data sets

The datasets used for benchmarking are provided by VectorDBBench, specific setups are not needed. We use four datasets in this paper: `Cohere 1M`, `Cohere 10M`, `OpenAI 500K` and `OpenAI 5M`. All the traces generated during the benchmarking are accessible via the Zenodo link at `https://doi.org/10.5281/zenodo.16916496`.

A.4 Installation

1. Install bpftrace:

```
$ sudo apt-get install bpftrace
```

If the bpftrace installed from apt does not work, compile it from the source code, please refer to the README of the artifact for instructions on compile bpftrace from source code.

2. Install conda environment (optional):

```
$ wget https://repo.anaconda.com/archive/Anaconda3
    -2024.02-1-Linux-x86_64.sh
$ chmod +x Anaconda3-2024.02-1-Linux-x86_64.sh
$ ./Anaconda3-2024.02-1-Linux-x86_64.sh
$ conda create --name vectordb-bench python=3.11.10
$ conda install pip
```

3. Install VectorDBBench:

```
$ git clone https://github.com/ZebinRen/VectorDBBench-
    dev.git
$ cd VectorDBBench-dev
$ git checkout origin/benchmark-paper
$ pip install -e .
```

4. Install the vector databases' clients:

```
$ pip install vectordb-bench[qdrant]
$ pip install vectordb-bench[weaviate]
$ pip install lancedb pandas
```

5. Install docker: `https://docs.docker.com/engine/install/ubuntu/`.

A.5 Experiment workflow

Before run the experiments, setup the environment variables. To get `PYTHON_BIN` and `VECTORDB_BENCH_BIN`, run `whereis python` and `whereis vectordbbench` when the conda environment is activated. If `bpftrace` is installed with apt, set `BPFTRACE_BIN` to `bpftrace`. If `bpftrace` is compiled from source code, set `bpftrace` to the compiled binary. `DATA_ROOT` is the directory where all the data of the benchmarked vector databases are stored.

```
$ export PYTHON_BIN=/python/path
$ export VECTORDB_BENCH_BIN=/vectordbbench/path
$ export BPFTRACE_BIN=/bpftrace/path
$ export DATA_ROOT=/data/root/path
```

Build the indexes before running the experiments, the following command shows how to build the indexes of `Cohere 1M`. Replace argument `--case-type` with `cohere-10m`, `openai-500k` and `openai-5m` to build the indexes for the three other databases.

```
$ mkdir -p ${DATA_ROOT}/milvus/milvus-ivf-cohere-1m ${
    DATA_ROOT}/milvus/milvus-hnsw-cohere-1m ${
    DATA_ROOT}/milvus-diskann-cohere-1m
$ python3 milvus-ivf-perf.py --case-type cohere-1m --
    run # Milvus IVF
$ python3 milvus-hnsw-perf.py --case-type cohere-1m --
    run # Milvus HNSW
$ python3 milvus-diskann-perf.py --case-type cohere-1m
     --run # Milvus DiskANN
$ mkdir -p ${DATA_ROOT}/qdrant/qdrant-hnsw-mem-cohere
    -1m
$ python3 qdrant-hnsw-perf.py --case-type cohere-1m --
    index-location memory --run # Qdrant
$ mkdir -p ${DATA_ROOT}/weaviate/weaviate-hnsw-cohere
    -1m
$ python3 weaviate-hnsw-perf.py --case-type cohere-1m
    --run # Weaviate
$ mkdir -p ${DATA_ROOT}/lancedb/lancedb-ivfpq-cohere-1
    m ${DATA_ROOT}/lancedb/lancedb-hnsw-cohere-1m
$ python3 lancedb-ivfpq-perf.py --case-type cohere-1m
    --run # LanceDB IVF-PQ
$ python3 lancedb-hnsw-perf.py --case-type cohere-1m
    --run # LanceDB HNSW
```

Experiments for the throughput and latency of different vector databases with different indexes. Use the following commands to reproudce figure 2, 3 and 4.

```
$ cd figure-2-3-4
$ python3 milvus-ivf-perf.py --case-type cohere-1m --
    run
$ python3 milvus-hnsw-perf.py --case-type cohere-1m --
    run
$ python3 milvus-diskann-perf.py --case-type cohere-1m
     --run
$ python3 qdrant-hnsw-perf.py --case-type cohere-1m --
    index-location memory --run
$ python3 weaviate-hnsw-perf.py --case-type cohere-1m
    --run
$ python3 lancedb-ivfpq-perf.py --case-type cohere-1m
    --run
$ python3 lancedb-hnsw-perf.py --case-type cohere-1m
    --run

# Plots
$ python3 plot-performance-all-dbs.py
$ python3 plot-cpu-all-db.py
```

Experiments that traces the I/O trace of Milvus with DiskANN index. Use the following commands to reproduce figure 5 and 6.

```
$ cd figure-5-6
$ sudo -E $PYTHON_BIN milvus-diskann-iotrace.py --case
    -type cohere-1m --concurrency 1 --run
```

Experiments that measures the performance and I/O traffic with different `search_list`. Use the following commands to reproduce figure 7, 8, 9, 10 and 11.

```
$ python3 milvus-diskann-var-klist.py --case-type
    cohere-1m --concurrency 1 --run
$ sudo -E $PYTHON_BIN milvus-diskann-var-klist-io-
    trace.py --case-type cohere-1m --concurrency 1 --
    run
```

Experiments that measures the performance and I/O traffic with different `beam_width`. Use the following commands to reproduce figure 12, 13, 14 and 15.

```
$ cp milvus-configs/user.yaml ${DATA_ROOT}/milvus/
    milvus-diskann-cohere-1m/user.yaml
$ python3 milvus-diskann-var-bwidth.py --case-type
    cohere-1m --concurrency 1 --run
$ sudo -E $PYTHON_BIN milvus-diskann-var-bwidth-io-
    trace.py --case-type cohere-1m --concurrency 1 --
    run
```

A.6 Evaluation and expected results

After the experiment finishes, the plotted graphs are stored under the `figure` subdirectory under the corresponding experiments figures.

We also provide the performances and traces that are used in the paper, to plot the figures with these traces:

```
# Figure 2, 3 and 4
$ cd results/figure-2-3-4
$ python3 plot-performance-all-dbs.py
$ python3 plot-cpu-all-db.py
# Figure 5 and 6
$ cd ../figure-5-6
$ python3 plot-milvus-diskann-iotrace.py
# Figure 7, 8, 9, 10, and 11
$ cd ../figure-7-8-9-10-11
$ python3 plot-diskann-milvus-klist-all.py
$ python3 plot-diskann-milvus-klist-bandwidth.py
# Figure 12, 13, 14, 15 and 16
$ cd ../figure-12-13-14-15-16
$ python3 plot-diskann-milvus-bwidth-all.py
$ python3 plot-diskann-milvus-bwidth-io-trace.py
```

A.7 Experiment customization

The experiments can be augmented with more datasets and vector databases, please refer to VectorDBBench for all the vector databases supported and how to add new datasets at `https://github.com/zilliztech/VectorDBBench`.

A.8 Notes

- We use a dedicated NVMe SSD for the storage of the database to avoid the effect of the operating system and other workloads.
- All the experiments are run in a server with a single NUMA node, we suggest pinning all the benchmark processes and vector databases in a single NUMA node to get stable results.
- The absolute performance of the vector databases may vary with different CPU and NVMe SSD configurations, we expect the relative performance of the vector databases to be stable across different configurations.
- Higher I/O bandwidth is expected if CPUs with higher performance is used.

A.9 Methodology

Submission, reviewing and badging methodology:

- `https://www.acm.org/publications/policies/artifact-review-and-badging-current`
- `https://cTuning.org/ae`