USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

DistVS: Large-scale Vector Search with Compute-Memory Disaggregation

Peiqi Yin, *The Chinese University of Hong Kong;* Xiao Yan, *Wuhan University;* Shiyuan Deng, *Huawei Cloud;* Hui Li, Yifan Zhu, and Xiangyu Zhi, *The Chinese University of Hong Kong;* Jingqi Mao, Ran Xu, and Wenliang Zhang, *Huawei Cloud;* James Cheng, *The Chinese University of Hong Kong*

https://www.usenix.org/conference/nsdi26/presentation/yin

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

DistVS: Large-scale Vector Search with Compute-Memory Disaggregation

Peiqi Yin[†] Xiao Yan[‡,*] Shiyuan Deng[§] Hui Li[†] Yifan Zhu[†]
Xiangyu Zhi[†] Jingqi Mao[§] Ran Xu[§] Wenliang Zhang[§] James Cheng[†]
[†]The Chinese University of Hong Kong [‡]Wuhan University [§]Huawei Cloud

Abstract

Similarity-based vector search, also known as ANNS, underlies many important applications such as content search, recommender system, and retrieval-augmented generation (RAG). However, vector search has a high storage demand due to large datasets and incurs costly IOs for its fine-grained access to the vectors and index. We observe that a compute-memory disaggregation architecture can tackle these challenges and design the DistVS system with a three-tier storage layout. In particular, the compute servers keep the small but low-precision compressed vectors, a more capacious memory server stores larger high-precision compressed vectors along with the index, while the full-precision exact vectors are kept on SSDs. The idea is to progressively prune the vector accesses along the low-high-full precisions from the compute servers to the SSDs, aligning with the storage hierarchy of memory-network-disk with gradually larger capacity but higher IO cost. To effectively utilize the three vector previsions, we design an algorithm called PRESS to conduct vector search. To improve performance, DistVS incorporates system optimizations including asynchronous execution, RDMA IO batching, and decoupled re-ranking. We compare DistVS with state-of-the-art disk-based and distributed vector search systems and show that DistVS consistently outperforms them and usually improves their query throughput by over 40%.

1 Introduction

To manage data with rich semantics (e.g., texts, images, and videos), a common practice is to map them to high-dimensional embedding vectors using machine learning models [8, 24, 38]. A fundamental operation on these vectors is *vector similarity search* or simply *vector search*, which returns the top-k vectors that are the most similar (e.g., measured by Euclidean distance) to a query vector. Since exact vector search [65] requires a linear scan over the entire dataset, which is expensive for high-dimensional vectors, approximate vector search has been widely used instead, which returns most (rather than all) of the top-k similar vectors to trade for efficiency. Efficient vector search is vital for many important applications such as recommender systems [27, 83], contextual search [33, 73], bio-informatics [56, 57], and retrieval-augmented generation (RAG) for large language models (LLMs) [5, 34, 47], as these applications usually require high query throughput and low query latency.

The state-of-the-art index for vector search is *proximity graph*. While there are many variants [14, 39, 61], a common idea is to connect similar vectors in the dataset to form a graph. Query is processed by graph traversals, which compute distances to all neighbors when visiting a node (i.e., a vector)[1] and select the node with the smallest distance (among those whose distances are computed) as the next node to visit.

Motivation. Real-world vector datasets are often large, which may contain billions of vectors and take up TBs of space [50, 51], and thus it is costly to store them in DRAM. To reduce the cost, disk-based vector search systems, such as DiskANN [61] and Starling [70], use SSDs as the primary storage. As shown in Figure 1, they employ a two-tier storage layout, where DRAM keeps a compressed version of all vectors, which are produced by vector quantization techniques such as PQ [30], while the original vectors and their adjacency lists (i.e., the proximity graph index) are stored on SSDs. When visiting each node, these systems fetch its exact vector (to compute exact distance and rank the node in the final search results) and adjacency list (to identify its neighbors) from disk, while approximate distances are computed for its neighbors using the compressed vectors to determine the next node to visit.

As we will show by our profiling results in §2.2, these disk-based vector search systems suffer from two problems. First, they require memory that is 10–20% of the dataset size, and their performance severely degrades when memory falls below the required level. However, for large datasets, even 10–20% of their sizes can still overwhelm common servers. For example, Laion-I2I [50] dataset contains 5 billion vectors and will occupy approximately 14TB storage. It is reported that industrial datasets often scale to 10 billion or even 100 billion vectors [53]. Moreover, there is a trend to use vectors with higher dimensions to enhance their representation power (e.g., 2048 [23] and 4096 [84] rather than 128 and 256), which also results in large dataset sizes. A single server is difficult

*Corresponding author.

[1] We use node and vector interchangeably in subsequent discussions.

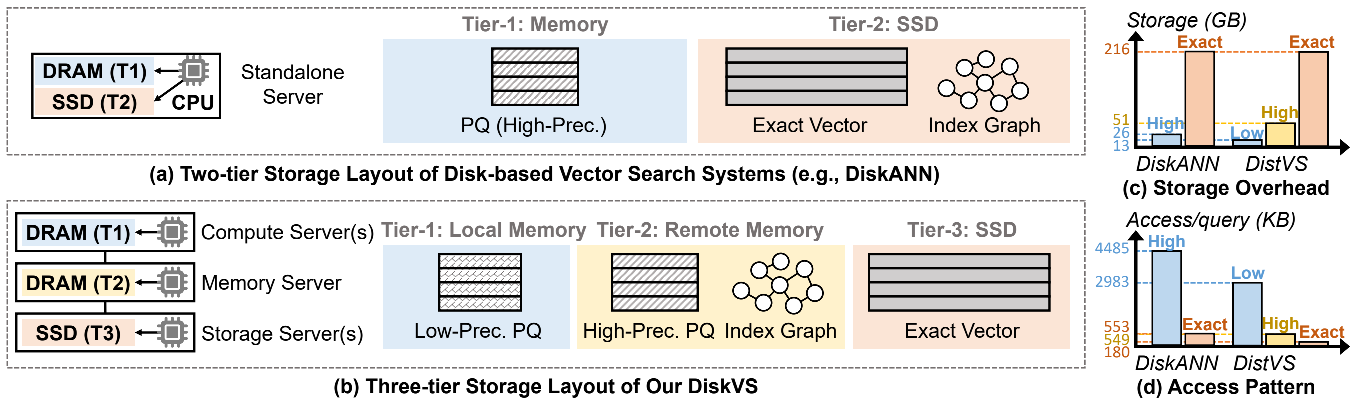

Figure 1: (a) and (b) show the storage layout of a disk-based vector search system (i.e., DiskANN) and our DistVS with compute-storage disaggregation. (c) and (d) report the storage overhead and access volumes of DiskANN and DistVS for different data on the Laion-T2I dataset. We configure the precision of the compressed vectors for DiskANN to reach its highest query throughput.

to handle these large vector datasets by vertically scaling up its local memory.

Second, although compressed vectors are accessed most frequently[2] and kept in memory, the disk IOs for exact vectors and adjacency lists still dominate their query processing time, thus limiting their performance. In addition, to improve query throughput, we may run multiple servers that replicate data in memory to handle multiple queries concurrently. However, due to replication, the aggregate memory of these servers can be even larger than the original dataset.

In recent years, compute-memory disaggregation is becoming popular [54, 68, 76] with the deployment of high-performance fabrics like RDMA and CXL [3]. Compute servers have strong compute power but small memory, while memory servers provide large memory. Compute servers execute tasks by caching hot data in their local memory and accessing memory servers on demand via network. The benefits of compute-memory disaggregation include flexible resource allocation (i.e., by decoupling compute and memory), high resource utilization, and high elasticity (i.e., by adjusting the number of compute servers according to workload). We observe that the compute-memory disaggregation architecture suits vector search as capacious memory servers can meet the high memory demand of vector search and performant networks (e.g., RDMA) allow much faster accesses than SSDs. Moreover, different compute servers can share a common copy of data and index on memory servers to eliminate the memory overhead caused by replication.

Our solution DistVS. Like many compute-memory disaggregation systems [54, 68, 76], our primary design goals are to make most data accesses local on compute servers and reduce the communication between compute servers and memory servers. To this end, our DistVS system adopts the three-tier storage layout as shown in Figure 1(b). Each compute server keeps a low-precision compression of all vectors, which has a high compression ratio w.r.t. the original dataset to fit in the small local memory of compute servers. The memory server keeps the proximity graph index and a high-precision compression of all vectors, which are larger but more accurate than the low-precision vectors on the compute servers. The exact vectors are stored on the SSDs, which can be attached to dedicated storage servers or hosted on compute/memory servers.

To work with the three-tier storage layout, we design the PRESS algorithm (Progressive pREcision Similarity Search) to conduct vector search, which performs coarse-to-fine distance estimation. In particular, when visiting each node during a graph traversal, PRESS first computes distances for all its neighbors using the low-precision vectors on the compute servers, and only the top-ranked neighbors are refined using the high-precision vectors on the memory server to select the next node to visit. Moreover, PRESS only reads the exact vectors to re-rank the candidates that are promising in their high-precision distances. To reduce communication, we push the IDs of the top-ranked neighbors to the memory server to compute their refined distances instead of pulling the high-precision vectors. As such, the compute servers and the memory server only exchange vector IDs.

As shown in Figure 1(d), our PRESS algorithm induces a favorable access pattern across the three storage tiers: low-precision vectors are the smallest in size but hottest in access, the adjacency lists and high-precision vectors are medium in terms of both size and access, while the full-precision vectors are the largest in size but coldest in access. We also observe that a single memory server can keep up with multiple compute servers since the computations pushed to the memory server are lightweight thanks to the pruning using low-precision vectors on the compute servers. As such, a single copy of the high-precision vectors and graph index can be shared across multiple compute servers.

To improve performance, DistVS introduces a suite of sys-

[2] When visiting each node, only one exact vector and adjacency list are fetched but multiple compressed vectors are accessed for its neighbors.

tem designs and optimizations. First, to avoid making the threads idle when queries wait for responses from remote servers, we adopt an *iteration-based asynchronous execution model*, which uses C++20 coroutine to switch a thread to process another query while waiting for responses. Second, we introduce *continuous RDMA IO batching* to aggregate the small network messages generated by our PRESS algorithm for multiple queries, which utilizes the network bandwidth more effectively by sending larger messages. Finally, we decouple the final result re-ranking stage from the graph traversal process and assign it to some dedicated threads, preventing it from blocking the main vector search pipeline. Besides, we discuss how DistVS handles updates to the vector dataset and utilizes computer servers with extremely small memory.

We conduct extensive experiments to evaluate DistVS on five real-world datasets and compare it with state-of-the-art disk-based vector search systems (i.e., DiskANN [61], Starling [70], and PipeANN [21]) and a state-of-the-art distributed vector database (Milvus [67]). The results show that DistVS consistently outperforms all the baselines by achieving higher query throughput while at least matching their query latency. In particular, DistVS improves the query throughput of PipeANN and Starling by an average of 70% and 85%, respectively, while the improvements are even larger over DiskANN and Milvus. Moreover, DistVS usually consumes smaller aggregate memory than the baselines by sharing a memory server over multiple compute servers, and the query throughput of DistVS improves linearly when adding computer servers. Ablation experiments also verify that our storage layout and system optimizations are effective.

To summarize, we make the following contributions.

- We identify high memory overhead and costly disk IOs as the main problems that limit disk-based vector search systems and observe that a compute-memory disaggregation architecture can resolve these problems.
- With compute-memory disaggregation, we design the DistVS system for vector search, which features a three-tier storage layout and the PRESS algorithm to make most data accesses local and reduce communication.
- We introduce system optimizations including asynchronous execution and IO batching to improve performance.
- We comprehensively evaluate DistVS to show its advantages over state-of-the-art baselines and verify our designs.

2 Background and Motivation

2.1 Vector Search and Proximity Graph Index

The quality of approximate vector search is usually measured by recall. In particular, for a query, denote the set of its ground-truth top-k neighbors as $\mathcal{S}$ and the returned approximate top-k neighbors as $\mathcal{S}'$, the recall is $|\mathcal{S} \cap \mathcal{S}'|/k$. Usually, a high recall (e.g., 90% or 95%) is required for good result quality. Indexes

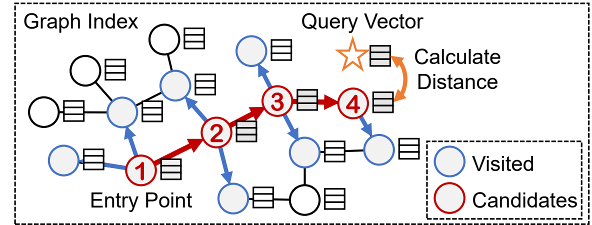

Figure 2: Vector search with proximity graph index.

Algorithm 1: Vector search with compressed vectors

```
1  L_full: Ordered node list (order by exact dist.).
2  L_appr: Ordered node list (order by approx. dist.).
3  D: The length limit for L_appr.
4  def Expand(q, u, L_appr):
5      for each node v in u's adjacency list do
6          Insert {v, ApproxDist(v, q)} into L_appr
7      Reserve the top-D closest nodes in L_appr
8  def Search(q, k):
9      Insert {entry e, ApproxDist(e, q)} into L_appr
10     while L_appr have unvisited nodes do
11         u ← the nearest unvisited node in L_appr
12         Insert {u, ExactDist(u, q)} into L_full
13         Expand(q, u, L_appr)
14     Reserve the top-k closest nodes in L_full
15     return L_full
```

are crucial for the performance of vector search by limiting the distance computations of each query to a small number of vectors. Among the indexes for high-dimensional vectors [7, 10, 14, 29, 39, 66], proximity graph variants [14, 39, 61, 70] are the state-of-the-art in that they require the fewest distance computations to recall a given recall target. As shown in Figure 2, they construct a graph over the vectors in a dataset by connecting each vector to its neighbors. Query is processed via a DFS-style graph traversal: when visiting each candidate node, the distances between its neighbors and the query are computed, and the candidate is marked as visited; then among the nodes whose distances are computed, nearest unvisited node becomes the next candidate; the graph traversal starts with a fixed or random entry node and terminates upon certain conditions (e.g., distance to candidate does not decrease).

Another fundamental technique for vector search is vector quantization. The idea is to compress each vector x into a smaller but lossy representation $\tilde{x}$, and thus the query distance computed with $\tilde{x}$ (i.e., $\|q-\tilde{x}\|$) becomes approximate. The compression ratio, which is the size of the original vector over the compressed vector, can be configured, and a larger compression ratio makes the compressed vector smaller but the approximate distance is also less accurate. Since the compressed vectors are smaller than the original vectors, computing the approximate distance is also faster than exact distance. There are many different vector quantization techniques [4, 16, 17, 30],

and product quantization (PQ) [30] is widely used.

2.2 Disk-based Systems for Vector Search

To reduce the cost of storing large vector datasets in memory, several disk-based vector search systems are proposed, such as DiskANN [61], Starling [70], and PipeANN [21]. They employ the two-tier storage layout in Figure 1(a): the compressed vectors reside in memory, while the index graph and exact vectors are kept on disk. In particular, the exact vector and adjacency list of each node are stored together on the same disk page such that they can be fetched by one disk IO. To work with both compressed and exact vectors, these systems utilize Algorithm 1 to conduct vector search. Specifically, Algorithm 1 maintains two ordered node lists (also called *candidate queues*), i.e., $\mathcal{L}_{appr}$ by approximate distances, and $\mathcal{L}_{full}$ by exact distances. In each iteration, Algorithm 1 selects the nearest unvisited node u from $\mathcal{L}_{appr}$ as the candidate to visit. Then, the exact vector and adjacency list of u are loaded from disk, and the exact distance between u and q is used to update $\mathcal{L}_{full}$. Next, Algorithm 1 computes the approximate distances for u's neighbors and inserts them into $\mathcal{L}_{appr}$. After that, $\mathcal{L}_{appr}$ is pruned to retain only the top-$\mathcal{D}$ nearest nodes, and $\mathcal{D}$ is called the queue size. The search terminates when all nodes in $\mathcal{L}_{appr}$ have been visited, and the top-k nodes in $\mathcal{L}_{full}$ are returned as the final search results. The queue size $\mathcal{D}$ is used to control the search time, with a larger $\mathcal{D}$ taking longer time but providing higher recall for the search results.

A common optimization for these systems is *beam search*, which reads the top-w unvisited nodes in $\mathcal{L}_{appr}$ from disk each time instead of a single node, and w is called beam width. These systems also employ their own optimizations. For instance, DiskANN caches in memory the exact vectors and adjacency lists for the nodes that are within several hops from the entry node. Starling randomly samples vectors from the dataset to build a meta index (which is also proximity graph), keeps the meta index in memory, and searches the meta index to identify good entry points for search on the complete dataset. Moreover, Starling keeps neighboring nodes on the same disk page by using graph reordering such that multiple required nodes can be fetched via a single disk IO. PipeANN improves the efficiency of IO operations with IO-uring [2], and proposes pipelining between disk access and distance computation to avoid CPU idle.

2.3 Profiling and Analysis

In this part, we profile and analyze disk-based vector search systems to provide the motivation and insights for our designs. The detailed settings and dataset statistics of the profiling experiments are provided in §5.1.

Accuracy of the compressed vectors. Figure 3 reports the query throughput of DiskANN when using different compression ratios for the compressed vectors. The results show that

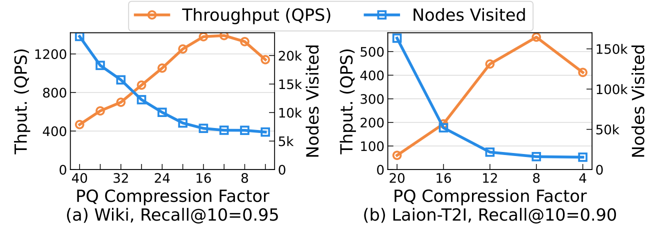

Figure 3: The query throughput of DiskANN when changing the compression ratio of the compressed vectors.

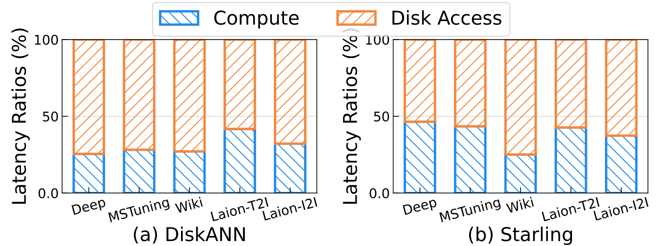

Figure 4: Query processing time decomposition for DiskANN and Starling when achieving 95% recall@10.

to achieve the optimal throughput, the compressed vectors need to be $\frac{1}{16} \sim \frac{1}{8}$ in size compared with the original dataset. Similar patterns are also observed for the other systems, and counting other memory-resident data structures, these systems require a memory that is $10\% - 20\%$ of the dataset. When the compression ratio exceeds the optimal (e.g., due to small memory), the query throughput degrades rapidly. This is because DiskANN uses approximate distances computed over the compressed vectors to select the nodes to visit, and when the approximate distances become inaccurate, vector search visits sub-optimal nodes. Figure 3 also shows that the number of nodes visited increases at large compression ratios. An interesting phenomenon is that query throughput also decreases when the compression ratio is smaller than the optimal. This is because each vector becomes larger, and thus the distance computations become more expensive. Moreover, Figure 3 shows that the number of visited nodes does not decrease when using compression ratios smaller than the optimal.

Overheads of disk IO. In each iteration (i.e., visiting a candidate node), disk-based vector search systems need to load the exact vector and the adjacency list of the candidate from disk. Figure 4 shows that disk IO takes up over 50% of the query processing time for both DiskANN and Starling. Moreover, the disk bandwidth also limits the speed that can be fetched and thus query throughput. Disk IO remains the bottleneck even with high-end SSDs. We experimented with the strongest SSD setting on AWS (i3.metal), which consists of 8 NVMe SSDs combined via RAID-0 and provides 19k Mbps disk bandwidth and 80k IOPS. When reaching a recall@10 of 95% for the Wiki dataset, the query time of Starling is 3.7 ms, and 1.7 ms out of the total time is spent on disk IOs.

The demand for exact vectors. Existing disk-based systems always access the exact vector and adjacency list of a candi-

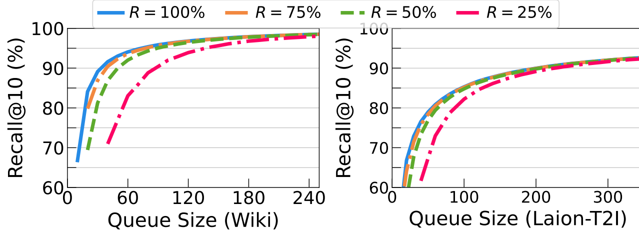

Figure 5: Recall@10 for different re-ranking ratios μ and queue size $\mathcal{D}$ (i.e., the length limit for $\mathcal{L}_{appr}$).

date node together. When visiting each candidate, the adjacency list is certainly required to identify its neighbors for approximate distance computation. The exact vector is used to re-rank the candidate in the final search results, and thus it may not be required if the candidate has a low chance of entering the final search results. We conduct such an experiment in Figure 5, where we delay the exact distance computations until the graph traversal finishes and configure a re-ranking ratio μ such that exact distances are only computed for the nodes ranking top-μ% in approximate distance. The results show that using a re-ranking ratio μ of 50% does not harm recall at high recall regimes.

Insights and takeaways. We summarize the following takeaways to guide the design of our vector search system under the compute-memory disaggregation architecture.

- The compressed vectors used for candidate selection must have sufficient accuracy and thus can be large. We keep these compressed vectors on the memory server since the local memory of the compute servers is usually small.
- Compute-memory disaggregation benefits from performant networks (e.g., RDMA), which is much faster than disk. Besides, network does not suffer from the read amplification of disks caused by 4KB block access, and the memory server can also conduct some computations.
- The demand for exact vectors is lower than the adjacency lists, and re-ranking can be conducted after graph traversal.

3 The PRESS Algorithm and DistVS Workflow

As shown in Figure 1(b), our DistVS keeps three versions of vectors at different precisions, i.e., the low-precision quantized vectors to fit in the limited memory of the compute servers, the high-precision quantized vectors in the memory of the memory server, and the full-precision exact vectors in the SSDs of the storage servers. In this part, we first introduce the PRESS algorithm, which uses the three vector precisions to conduct vector search for the in-memory setting. Then, we discuss how DistVS runs PRESS on distributed servers.

Algorithm 2: Pseudocode for the PRESS Algorithm.

```
1  L_full: Ordered node list (order by exact dist.).
2  L_high: Ordered node list (order by P_high approx. dist.).
3  L_low: Ordered node list (order by P_low approx. dist.).
4  μ: Ratio of selected candidates in L_low.
5  D_full, D_high, D_low: The maximum list depths for L_full,
     L_high, and L_low, respectively.
6  def Search(q, k):
7      L_high insert {entry e, HighApproxDist(e, q)}
8      while L_high have unvisited nodes do
9          u ← the nearest unvisited node in L_high
10         ExpandLow(q, u, L_low)
11         ExpandHigh(q, L_low, L_high)
12     Refine(q, L_high, L_full)
13     return L_full
14 def ExpandLow(q, u, L_low):
15     for node v in u's neighbors list do
16         L_low insert {v, LowApproxDist(v, q)}
17     Reserve the top-D_low closest nodes in L_low
18 def ExpandHigh(q, L_low, L_high):
19     N_u ← μ · Len(L_low)
20     for i in range(N_u) do
21         v ← the nearest unvisited node in L_low
22         L_high insert {v, HighApproxDist(v, q)}
23     Reserve the top-D_high closest nodes in L_high
24 def Refine(q, L_high, L_full):
25     Keep the top-D_full nodes in L_high
26     for node u in L_high do
27         L_full insert {u, ExactDist(u, q)}
28     Reserve the top-k closest nodes in L_full
```

3.1 Vector Search with The PRESS Algorithm

Algorithm 2 details the procedure of PRESS. Guided by the insights from §2.3, Algorithm 2 adopts two key principles. First, the candidate node to visit is selected using the high-precision quantized vectors (i.e., $\mathcal{P}_{high}$) such that the graph traversal does not take detours, while the low-precision quantized vectors (i.e., $\mathcal{P}_{low}$) are used to prune the accesses to the high-precision vectors. Second, re-ranking with the full-precision exact vectors is delayed until the graph traversal finishes to reduce the accesses to the exact vectors. Algorithm 2 uses three ordered lists $\mathcal{L}_{low}$, $\mathcal{L}_{high}$, and $\mathcal{L}_{full}$, which are ranked by $\mathcal{P}_{low}$, $\mathcal{P}_{high}$, and exact distances, respectively.

Search with both low and high precisions. When the search starts, the $\mathcal{P}_{high}$ approximate distance between the query q and the graph entry node e is calculated and inserted into $\mathcal{L}_{high}$ (line 7). For each search iteration, the nearest unvisited node u in $\mathcal{L}_{high}$ is selected (line 9), and two functions `ExpandLow` and `ExpandHigh` are called to update the ordered list (lines 10-11).

Table 1: The number of distance computations conducted on different vectors when using a single compression precision (like DiskANN) and two compression precisions (i.e., our *PRESS*). "Equiv." denotes equivalent computational complexity mapped to exact distance computations: $\frac{N_{low}}{C_{low}} + \frac{N_{high}}{C_{high}} + N_{full}$, where N are the numbers of distance computation, and C represents the compression ratio.

Dataset	Method	Num. Dist. Computations			Equiv.
		Low	High	Full	
Wiki	Low-only	10,484	0	353	681
	High-only	0	5,570	90	438
	PRESS	6,222	465	95	290
LaionT2I	Low-only	25,123	0	770	2,340
	High-only	0	9,096	205	1,342
	PRESS	10,808	1,280	260	910

In particular, `ExpandLow` calculates the approximate distances for u's neighbors with $\mathcal{P}_{low}$ (lines 14-17) and updates $\mathcal{L}_{low}$. Next, a portion of the top-ranking nodes (i.e., μ, which is called the re-ranking ratio) in $\mathcal{L}_{low}$ are selected, and their $\mathcal{P}_{high}$ approximate distances are calculated and updated to $\mathcal{L}_{high}$ (lines 18-23). Afterwards, the top-ranking unvisited node in $\mathcal{L}_{high}$ is selected as the next node to visit. The search terminates when all nodes in $\mathcal{L}_{high}$ are visited.

Delayed re-ranking with pruning. After graph traversal, Algorithm 2 selects the top-$\mathcal{D}_{full}$ nodes from $\mathcal{L}_{high}$ for exact distance computation (lines 24–27) such that they can be re-ranked in the final search results. This is because the nodes that rank low $\mathcal{L}_{high}$ are unlikely to be the top-k neighbors for a query as shown by our profiling in §2.3. The final results are the top-k closest nodes in $\mathcal{L}_{full}$ (line 28).

Setting the parameters. We configure the compression ratio of the high-precision compressed vectors (i.e., $\mathcal{P}_{high}$) as the point that maximizes the query throughput of DiskANN. As shown in Figure 3, this is the highest compression ratio that does not increase the number of visited candidate nodes. Disk-based vector search systems use a single list size (i.e., $\mathcal{D}$) to control search time and recall in Algorithm 1 while our PRESS requires three list sizes $\mathcal{D}_{low}$, $\mathcal{D}_{high}$, and $\mathcal{D}_{full}$ for $\mathcal{L}_{low}$, $\mathcal{L}_{high}$, and $\mathcal{L}_{full}$, respectively. To simplify parameter setting, we use $\mathcal{D}_{low} = 2\mathcal{D}_{high}$ and $\mathcal{D}_{high} = 2\mathcal{D}_{full}$ such that only $\mathcal{D}_{high}$ needs to be configured. Note that $\mathcal{D}_{high} = 2\mathcal{D}_{full}$ implies a re-ranking ratio of 50%, which is shown to work well in Figure 5. For the refinement ratio μ, we set it to 0.3 by default, which is shown to work well empirically and makes the number of high-precision distance computations small.

Profiling PRESS. In Table 1, we compare PRESS with using only the low-precision vectors (i.e., *Low-only*) or only the high-precision vectors (i.e., *High-only*) alongside the exact vectors. The *Equiv.* column can be interpreted as the overall computation complexity of each method. Note that we also enable delayed re-ranking for both *Low-only* and *High-only*.

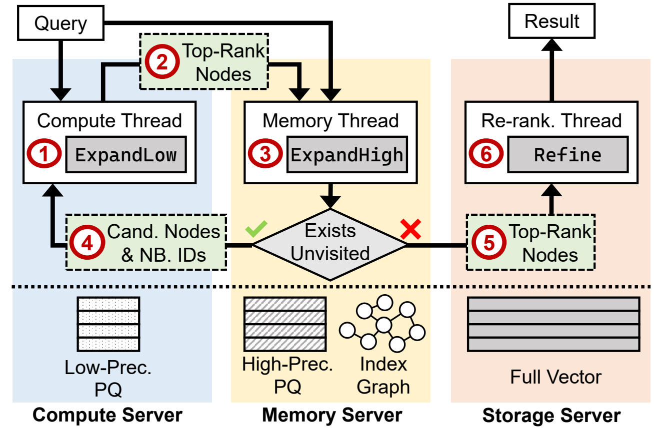

Figure 6: The workflow of DistVS for vector search.

First, we observe that PRESS successfully induces skewed accesses across the three precisions, i.e., $\mathcal{P}_{low}$ vectors receive many more accesses than $\mathcal{P}_{high}$ vectors, which in turn have many more accesses than the full-precision vectors. This is favorable since most of the data accesses (for the $\mathcal{P}_{low}$ vectors) are local on the compute servers, the computations (for the $\mathcal{P}_{high}$ vectors) pushed to the memory server are lightweight, and the accesses to the storage servers are very few. Second, both *High-only* and *Low-only* have higher overall computation complexity than PRESS (see the *Equiv.* column). For *Low-only*, its approximate distances are inaccurate and may lead the graph traversal to detours. This suggests that re-ranking with $\mathcal{P}_{high}$ vectors is necessary. *High-only* can be impractical as the compute servers may not have enough memory to store them, and the memory server will be overloaded if all compute servers push a large number of computations to it. In § 5, we show that pulling the $\mathcal{P}_{high}$ vectors from the memory servers for computation is also less efficient than PRESS.

3.2 Workflow of DistVS for Vector Search

Our DistVS runs on a cluster comprising several compute servers and a memory server. The compute servers process different queries independently and can access the memory server on demand via RDMA network. The SSD storage for exact vectors can be attached to either the aforementioned servers or some separate servers. In our implementation, exact vectors are kept on the SSDs of the compute servers. The lower-half of Figure 6 shows the storage layout of DistVS.

The upper-half of Figure 6 shows how DistVS runs with the PRESS algorithm. The compute server maintains a node list $\mathcal{L}_{low}$ ordered by low-precision distances, while the memory server maintains a node list $\mathcal{L}_{high}$ ordered by high-precision distances. Query processing starts when the memory server adds the distance between the query and entry node to $\mathcal{L}_{high}$. For each search iteration, the memory server pops a batch of candidate nodes (i.e., by the beam width w) from $\mathcal{L}_{high}$, loads their adjacency lists from the index graph, and sends them

to the compute server. The compute server then calculates the $\mathcal{P}_{low}$ approximate distances between the query and the neighbors of these candidates using $\mathcal{P}_{low}$ vectors, inserting the results into $\mathcal{L}_{low}$. Then, a subset of the top-ranking nodes is selected from $\mathcal{L}_{low}$, and their node IDs are sent to the memory server to serve as candidates for the next iteration. The memory server computes their $\mathcal{P}_{high}$ approximate distances to the query and inserts the results into $\mathcal{L}_{high}$ to determine the candidates for the next iteration. The search phase terminates when all the nodes in $\mathcal{L}_{high}$ are visited. Finally, the storage server loads the exact vectors of the top-ranking nodes in $\mathcal{L}_{high}$ from disk, computes their exact distances, re-ranks the search results, and returns the final top-k search results.

3.3 Discussions

The benefits of DistVS. The compute-memory disaggregation architecture of DistVS brings several favorable benefits.

- *Reduced memory consumption*: As shown in Table 1, the computations pushed to the memory server for each query are lightweight with the local pruning on compute servers with low-precision vectors. As such, a memory server can work with multiple compute servers with a single copy of the high-precision vectors. Compared with replicating the memory of the servers to improve query throughput for the disk-based systems, our DistVS usually has a smaller aggregate memory consumption (see § 5.2).
- *Good scalability*: To conduct distributed in-memory vector search, vector databases such as Milvus [67] shard the dataset over machines and build an independent index on each machine. Their query throughput increases only sub-linearly with the number of machines because the pruning power of vector indexes is stronger when the dataset is considered as a whole. DistVS uses a global vector index on the entire dataset, and we show that its query throughput scales linearly with the compute servers in § 5.3.
- *Good elasticity*: For applications that involve vector search, the query workload often fluctuates violently over time [22]. As such, a vector search system should scale in and out its resources quickly to adapt to the query workload. The computer servers of DistVS only load small low-precision vectors and thus can be started or dismissed quickly.

Using low-memory compute servers. When the memory of the compute servers is extremely small, the low-precision vectors will become too inaccurate to guide the pruning for high-precision vectors. In this case, we find that popularity-based vector offloading provides better query performance than reducing the precision of the low-precision vectors to fit in the memory of compute servers. In particular, accesses are skewed over the vectors in a dataset, and the vectors with a high degree in the proximity graph generally receive more accesses [71]. Empirically, we set the size of the low-precision vectors as half of the high-precision vectors; when the memory of compute servers is too small, we offload the low-precision vectors of the low-degree nodes to the memory server. We show in § 5.3 that throughput decreases gracefully with compute server memory when using this strategy.

We further discuss (i) handling index updates, (ii) generality of DistVS, and (iii) failure handling in the Appendix.

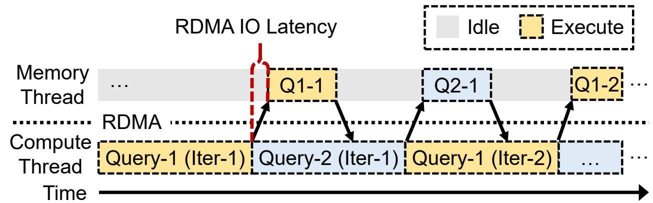

Figure 7: An example of the coroutine execution with a memory thread and a compute thread with two coroutines. The compute thread executes the low-precision `Expand` and the memory thread executes the high-precision `Expand`.

4 System Optimizations

In this section, we discuss the system optimizations for DistVS: iteration-based asynchronous execution to improve CPU utilization (§ 4.1), continuous RDMA IO batching to avoid small messages (§ 4.2), and the re-ranking optimizations for disk accesses (§ 4.3).

4.1 Iteration-based Asynchronous Execution

As illustrated in Figure 6, during each iteration, after computing the low-precision approximate distances, the compute thread invokes a remote function, requesting the memory thread to calculate the high-precision approximate distances for finding the next candidate. During this period, the compute thread remains idle, waiting for a response from the memory thread, as it cannot proceed to the next iteration until it receives the candidate nodes and their adjacency lists. The waiting time consists of two network transmissions and the processing time required to compute high-precision approximate distances, select candidate nodes, and gather their adjacency lists for sending.

To avoid the threads idling, we propose an iteration-based asynchronous execution model, in which the CPU thread processes requests at the granularity of search iterations. After an iteration issues the network message sending, the thread does not block; it switches to another query and resumes the suspended iteration when its reply arrives. This overlaps remote processing with local execution, improves CPU utilization, and reduces head-of-line blocking. Specifically, we leverage coroutine [59], a C++20 feature that enables cooperative multitasking by allowing a CPU thread to suspend execution at certain points and resume later, potentially switching to another execution context. Coroutines are lightweight and more efficient than traditional threading mechanisms, as they avoid the overhead of context switching between OS threads. Each

coroutine is processed by one thread only, and they are cache-friendly. Their execution context (i.e., local variables and stack frames) remains in the CPU cache, reducing memory access latency during context switches.

By assigning two coroutines to each CPU thread, the thread can switch to another coroutine when it sends a message. We show an example of the coroutine switching in Figure 7, suppose the first coroutine handles query q_1 and the second handles query q_2. The compute thread can process one iteration of q_1, and upon sending a message to the memory thread, switch to process an iteration of q_2. By alternating between q_1 and q_2 through coroutine switching, the compute thread can fully utilize CPU resources and avoid idle periods.

4.2 Continuous RDMA IO Batching

For network (RDMA) communication, each coroutine on the compute server maintains its own communication unit and interacts independently with the memory server during each network operation. However, a challenge arises because each message is typically very small (i.e., often containing only tens of node IDs per iteration), resulting in a large number of small messages being sent to the memory server. The limited number of NIC channels can become a bottleneck under high load, preventing full utilization of network bandwidth and potentially increasing network latency, especially when many CPU threads access the system concurrently.

To avoid network bottleneck and improve NIC utilization, we employ continuous IO batching driven by a dedicated communication thread per server. Worker threads exchange messages with the communicator via per-thread, lock-free message queues (send or receive), in which enqueues never block across threads. When messages are ready, workers push descriptors to their send queues; the communicator periodically drains these queues, links work requests (*wr*) into a batch, and transmits them in a single operation using one-sided RDMA WRITE into pre-allocated buffers on the peer, as shown in Figure 8. On the memory server, a matching communication thread receives the batched payload, de-batches it, and dispatches individual requests to coroutine workers; responses follow the symmetric path via per-thread receive queues. It reduces per-message overhead and avoids cross-thread blocking by aggregating many small requests into a single transfer. This continuous IO batching approach enables the system to fully utilize available network bandwidth and prevents the communication channel from becoming a system bottleneck.

The RDMA messages in our design are lightweight, as they only involve node IDs (i.e., top-ranking nodes in $\mathcal{L}_{low}$ and adjacency lists for candidate nodes in $\mathcal{L}_{high}$), and hence their communication volume is independent of vector dimensionality. We have reported the IO volume in Figure 12 and RDMA round-trip latency in Figure 14, which shows that the both communication volume and latency are low.

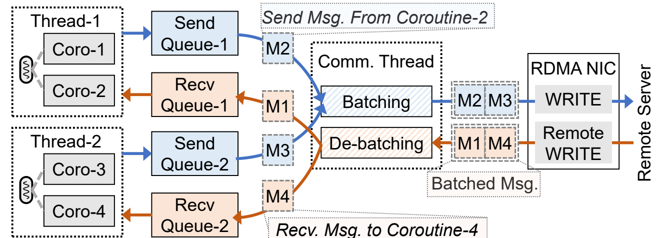

Figure 8: An example of the continuous IO batching with two threads, each thread holds two coroutines.

4.3 Efficient Re-ranking for Final Results

When the search terminates (no unvisited nodes remain in $\mathcal{L}_{high}$), the memory server extracts the top-$\mathcal{D}_{full}$ candidates and dispatches their IDs to the re-ranking thread. The compute server is immediately notified that the search context can be freed and is available for new queries. The re-ranking thread resides on a server with disk storage (either a dedicated disk server or co-located with the compute server). Upon receiving the re-rank request, it reads the exact vectors from disk, computes exact distances, and returns the final top-k results. We separate the compute and re-ranking stages into different threads (or different servers) because the long execution time for exact distance calculation would otherwise block the processing of iterations in other tasks, resulting in longer waiting latency and leaving the memory server idle, which wastes CPU resources.

The re-ranking thread also uses asynchronous processing to overlap disk IO with computation. It leverages the libaio [28] interface: after submitting IO requests, it stores the context in a queue and polls for completions via `io_getevents`. To avoid queuing delays, multiple re-ranking threads are launched in parallel.

Disk vector relayout. Following Starling [70]'s relayout strategy, we reorder the on-disk vector layout to co-locate each node with its likely neighbors within the same page. When two (or more) candidates reside on one page, a single IO fetch serves all of them, eliminating redundant disk accesses. Unlike Starling co-locates nodes' exact vectors and the adjacency lists in the page, our disk pages store vectors only, increasing packing density and the probability of such co-hits, further reducing disk IO. Vector relayout is optional in our system and primarily benefits low-dimensional datasets (e.g., Deep [48] and MSTuning [43]).

5 Experimental Evaluation

We extensively evaluate DistVS and compare with state-of-the-art ANNS systems. The key findings include:

- DistVS consistently outperforms the baselines, achieving higher query throughput with competitive latency.
- DistVS's three-tier design is memory-efficient and scalable. Compute servers use far less memory than the baselines,

Table 2: Statistics of the datasets. **Tot.N** is the number of vectors, **Dim.** is vector dimension, **Metric** is the measure for vector similarity, and **Tar.** is the target recall@10 for search.

Name	Tot. N	Dim.	Metric	Size (GB)	Tar.
Deep [48]	1B	96	L2	357.6	95%
MSTuning [43]	1B	128	L2	372.5	95%
Wiki [13]	100M	384	L2	143.3	95%
Laion-T2I [51]	100M	512	Cosine	191.7	90%
Laion-I2I [50]	100M	768	Cosine	286.4	95%

and a single memory server copy can serve multiple compute servers, reducing the cluster-wide memory footprint.

- Our key algorithm, PRESS, is effective in searching and outperforms alternative designs under the three-tier layout.

5.1 Experiment Settings

Datasets. We evaluate DistVS on five datasets (Table 2). MSTuning-1B [43] and Yandex DEEP-1B [48] are two popular billion-scale datasets from BIGANN benchmarks [43]. Following PipeRAG [31], we construct a 384-dimension *Wiki* dataset using the BGE encoder [8], with 100M base vectors from Wikipedia [13] and 100K queries from MMLU [25]. Laion [50, 51] includes two datasets, with dimensions of 512 and 768, respectively, and we use the first 100M image vectors in Laion for both base datasets. For the query vectors, we sample 10K text vectors for the 512-dimension dataset (Laion-T2I[3]), and sample 10K image vectors for the 768-dimension dataset (Laion-I2I), following ANN benchmarks [6].

Baseline systems. We compare *DistVS* with three SOTA disk-based ANNS systems: *DiskANN* [61], *Starling* [70], and *PipeANN* [21], which we discussed in § 2.2. For *DistVS* and these three systems, we use PQ [30] for vector compression and Vamana [61] as the proximity graph index (with an out-degree of 64). We also evaluate against *Milvus* [67], an in-memory sharding-based system that partitions the dataset across multiple shards. During search, *Milvus* maintains all shards in memory and performs independent searches on each shard before merging results.

Testbed. We conducted all experiments on a cluster of five servers, each equipped with an Intel(R) Xeon(R) Gold 6252N CPU @ 2.30GHz (96 cores) and 756GB DRAM. The servers feature NVIDIA Mellanox ConnectX-5 adapters supporting RDMA over Converged Ethernet (RoCE), and each contains two SATA SSDs configured in RAID-0 [1] to provide 3.5TB storage with 680MB/s bandwidth. The operating system is Ubuntu 22.04.5 with Linux kernel 5.15.0.

For *Starling*, *DiskANN*, and *PipeANN*, each server hosts a disk index and serves queries independently using 8 threads, with each query handled by a single thread. The server maintains in-memory data and loads disk pages from its local SSD. We use the same quantization precision that maximizes *DiskANN*'s throughput. For *PipeANN* and *Starling*, we use the default parameters reported in their paper for the in-memory index. For *DistVS*, we designate one memory server and four compute servers, with re-ranking threads co-located on compute servers and accessing their SSDs. The memory server serves as the memory pool. Both compute and memory servers use 8 threads (including the re-ranking and communication threads), matching the compute resources of the baselines. For *Milvus*, the dataset is partitioned into fixed-size shards, and each server hosts some shards for search.

[3] T2I and I2I refer to "Text to Image" and "Image to Image", respectively.

Table 3: The query throughput of *DistVS* and baselines when achieving 95% recall. *Comparison* is the *throughput improvement* of *DistVS* compared with the best-performing baseline.

Datasets	Deep	MSTuning	Wiki	Laion-T2I	Laion-I2I
DiskANN	5,732	6,113	7,090	805	8,304
PipeANN	9.568	7,638	10,087	1047	12,815
Starling	10,095	10,208	8,921	738	10,469
DistVS	14,292	17,007	12,962	2,207	17,955
Comparison	1.42x	1.67x	1.29x	2.11x	1.40x

Metrics. Our primary metrics are *query throughput* (queries per second, QPS), *query latency* (average processing time), and *memory consumption*. Following prior work [10, 61, 70], we measure search quality using *recall@10* and report the average recall@10 across all queries by default.

5.2 Main Results

Figure 9 reports query throughput and latency for *DistVS* and the baselines. For fair comparison with *DistVS*, we employ five servers, each independently equipped with an SSD, and measure the total throughput as the sum across all servers. *DistVS* adopts the same high-precision quantization setting as *DiskANN*, and the low-precision vectors use half the storage of the high-precision vectors.

Table 3 reports the throughput point of *DistVS* and the baselines (in Figure 9). Across all datasets and baselines, *DistVS* consistently achieves higher throughput. When achieving 95% recall, *DistVS* improves throughput over *DiskANN* by 140%, *Starling* by 85%, and *PipeANN* by 70% on average, respectively. *Starling* outperforms *DiskANN* and *PipeANN* on low-dimensional datasets, as each disk page can store more nodes, making the relayout strategy more effective at reducing disk IOs; for high-dimensional vectors, pages hold few nodes, so the benefit diminishes and performance converges to *DiskANN*. *PipeANN* adopts *DiskANN*'s layout while it exhibits low latency, as it overlaps disk access and distance calculation. *DistVS* achieves the highest throughput, as the search phase is fully memory-resident: the index graph and high-precision vectors are served from the memory server, and disk IO occurs only during final re-ranking. This sidesteps the IO bottleneck that limits the baselines.

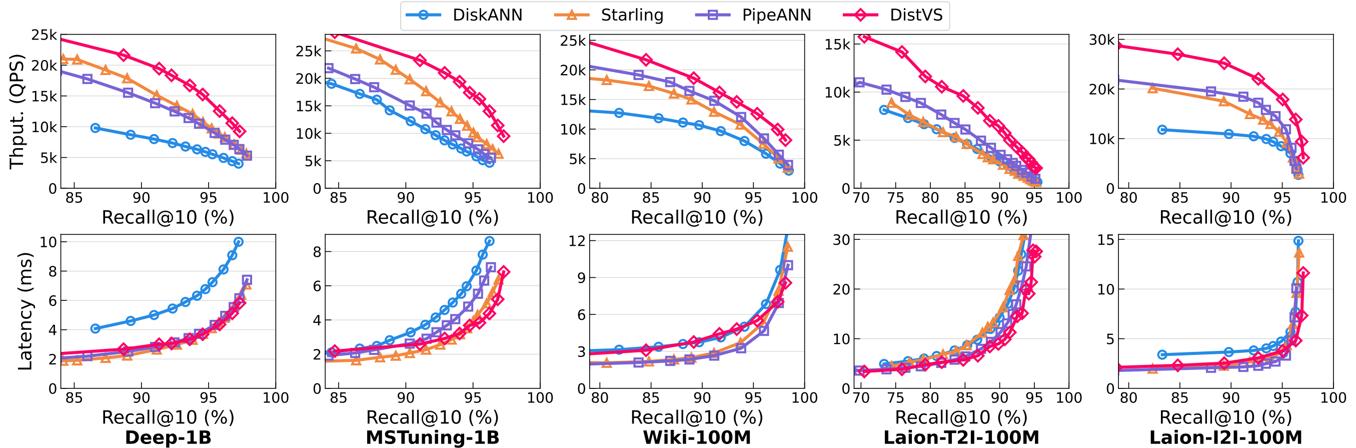

Figure 9: The throughput and latency of *DistVS* and the baselines on the 5-servers cluster. Note that 5-servers cluster for *DistVS* represents using 4 compute nodes and 1 memory node.

Table 4: Memory usage of different systems (GB). ***Comp.*** and ***Mem.*** denote the memory usage in the compute server and memory server of *DistVS*, respectively. For standalone baselines, we aggregate memory usage across five servers; ***Comp.*** aggregates across four compute servers.

Name	DiskANN	PipeANN	Starling	DistVS	
				Comp.	Mem.
Deep [48]	224.7	259.4	562.2	89.2	307.3
MSTuning [43]	243.6	273.4	590.1	99.8	309.2
Wiki [13]	46.6	56.8	137.4	18.0	35.3
Laion-T2I [51]	125.7	140.2	230.5	50.4	50.3
Laion-I2I [50]	96.8	115.3	248.6	39.2	44.5

We show the memory consumption of Figure 9 in Table 4. *DiskANN*, *Starling* and *PipeANN* have fixed per-server footprints as each server hosts a full index; *Starling* and *PipeANN* typically use more memory than *DiskANN* to gain throughput. Note that the memory usage is aggregated across servers. In contrast, *DistVS* is asymmetric: compute servers store only low-precision vectors and therefore have a small footprint, whereas the memory server holds the index graph and high-precision codes and is correspondingly larger. On low-dimensional datasets (e.g., Deep-1B, MSTuning-1B), the memory server's footprint can exceed *Starling*'s because graph storage (largely fixed by degree) dominates the total (index graph for 1B dataset requires 242.1 GB memory). On high-dimensional datasets, this overhead is amortized, and for Laion-I2I *DistVS*'s total memory is even lower than *Starling*'s. Overall, the cluster-wide memory footprint of *DistVS* remains mush smaller than *Starling*. Considering the throughput-to-memory ratio, *DistVS* outperforms the baselines in most cases, which achieve higher throughput and maintain a lower footprint. Only one case fails (*PipeANN* on Deep-1B).

Figure 10 shows disk IO for the three systems on the same target recall (Table 2). *Starling* reduces disk accesses compared to *DiskANN* but still exceeds *DistVS*. Because *DistVS* uses the *PRESS* algorithm with pruned re-ranking, it requires fewer exact vector reads, resulting in much lower disk IO.

Table 5: Queries per dollar performance comparison ($\times10^6$). The setting follows Table 3.

$\times10^6$	Deep	MSTuning	Wiki	Laion-T2I	Laion-I2I
DiskANN	10.27	10.76	15.81	1.65	16.71
PipeANN	16.77	13.20	22.32	2.13	25.44
Starling	14.88	14.76	18.56	1.41	18.99
DistVS	23.66	27.83	29.15	4.70	37.40
Comparison	1.41x	1.89x	1.31x	2.21x	1.47x

For latency, *DistVS* incurs additional overhead from network transfers and cross-machine scheduling inherent to distributed search. Despite these costs, its end-to-end latency remains comparable to *Starling* and *PipeANN*, with similar averages in most cases.

Cost-performance analysis. Table 5 estimates the queries per dollar (QP$) of *DistVS* with the baselines using Google Cloud H4D pricing [11]. Based on costs of $26.337 per CPU core per month, $0.939 per GB of memory per month, and $0.113 per GB of local SSD per month, we calculated QP$ for each server configuration (8 CPU cores, with memory and disk usage from Tables 4 and 2, and throughput from Table 3). We compute QP$ as QPS divided by per-second cost, converting monthly resource costs to per-second rates. *DistVS* consistently outperforms all baselines, improving QP$ by 1.31× to 2.21x over the best baseline, demonstrating more efficient resource utilization and greater cost savings.

Comparing with distributed Milvus. We compare *DistVS* with the sharding-based distributed ANNS system *Milvus* on the 100M Deep dataset, evaluating only this dataset due to *Milvus*'s substantial storage requirements (about 750GB

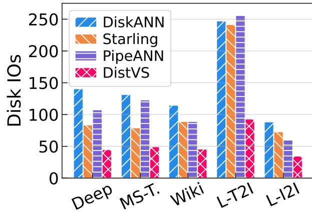

Figure 10: Disk IOs pre request for Figure 9.

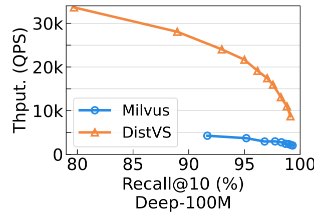

Figure 11: Comparing *DistVS* with distributed *Milvus*.

for 100M 96-dimensional vectors [4]). *DistVS* achieves significantly higher throughput (e.g., 5.84x at 95% recall) because *Milvus*'s fixed-size sharding fragments large datasets into small shards (i.e., 2GB), forcing each query to probe numerous shards and perform redundant computations. As the dataset size grows, more shards are introduced, leading to larger computational redundancy and lower throughput.

5.3 Micro Experiments

In this part, we evaluate the designs of *DistVS*. We use the Laion-T2I dataset by default, and remark that the observations are similar under other configurations.

Three-tier storage layout. To understand the design choice of *DistVS*, we create two *DistVS* variants on the same compute–memory architecture: (i) *DistVS-L* (Low-only), which keeps low-precision vectors on compute servers and follows the original search procedure (Algorithm 1) while fetching the graph from the memory server each iteration. (ii) *DistVS-H* (High-only), which also follows the original algorithm using high-precision vectors, storing a subset of high-precision codes locally on the compute server and placing the remainder on the memory server. The nodes are selected to be stored on the compute server by their popularity. During search iteration, the graph data alongside the missing quantized vectors are fetched from the memory server. All variants use the same memory budget as *DistVS*.

We compare throughput, latency, and per-request RDMA IO volume in Figure 12. *DistVS-L* performs poorly because ranking using low-precision codes is inaccurate, thus more node needs to be visited and computed to reach a high recall. *DistVS-H* also trails *DistVS* since each iteration transfers both graph data and high-precision codes from the memory server, yielding roughly an order-of-magnitude higher RDMA IO volume per request. In contrast, *DistVS* transfers only node IDs (dimension-agnostic), keeping network overhead low and delivering the best throughput and competitive latency.

Scalability. In Figure 13(a), we vary the number of compute servers while keeping one memory server, with all servers using 8 threads. As the worker count increases, system throughput scales nearly linearly while query latency increases only slightly. This indicates good scalability because the network

[4]https://milvus.io/zh/tools/sizing

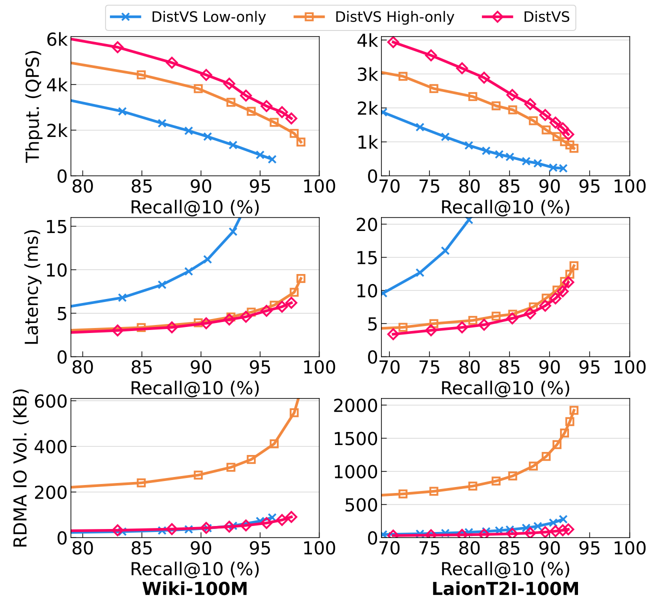

Figure 12: Comparing *DistVS* with two variants under the same memory budget with one compute-memory server pair.

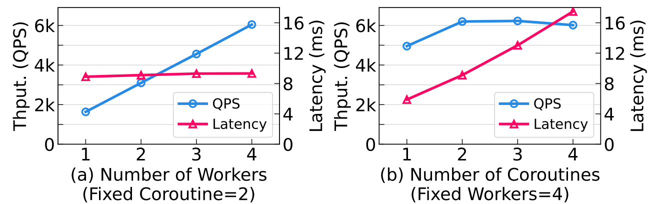

Figure 13: Scalability experiment test on Laion-T2I dataset under the target recall. Left: changing the number of compute servers, each with 8 threads. right: changing the coroutine level, with fixed workers and each with 8 threads.

transfers only node IDs and does not become a bottleneck. The memory server's execution is lightweight, so its limited computational resources are not a constraint either, with most computation still occurring on the compute servers.

Iteration-based asynchronous execution. In Figure 13(b), we show experiments with a fixed number of workers and threads while varying the ratio of coroutines to threads. *DistVS* adopts asynchronous execution to overlap waiting time with execution. By increasing the coroutine-to-thread ratio, more execution can be overlapped; however, this may increase waiting latency. Figure 13(b) shows that without overlapping (i.e., coroutine=1), the system achieves the lowest latency; with each thread holding two coroutines (coroutine=2), it achieves high throughput but increased latency. The latency continues to increase with more coroutines, but the throughput does not continue to improve. This is because the system has already overlapped the waiting time and the CPU is at full load. We therefore use coroutine=2 as our setting, as it achieves good throughput with low latency overhead.

Changing beam widths. Figure 14 reports *DistVS*'s through-

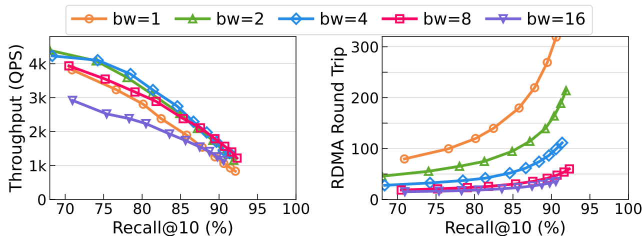

Figure 14: Impact of beam width on *DistVS*: throughput and average RDMA round-trips per request.

put and RDMA round-trips under different beam widths. In particular, beam width controls the number of candidate nodes checked in each iteration. Larger beam widths cut the iteration count and thus reduce RDMA round-trips and transfer overhead. However, overly large beam widths dilute the guidance of *PRESS*, lowering the selectivity of high-precision checks and hurting accuracy, which increases computation to reach the same high recall. Figure 14 shows the result when changing the beam widths from 1 to 16 on Laion-T2I. We found that the RDMA round-trips decrease monotonically with beam width. For throughput at 90% recall, *DistVS* reach peaks at beam width 8; smaller widths suffer excess network overhead, whereas larger widths degrade accuracy and raise compute.

6 Related Work

Systems for vector search. There are two main types of indexes for vector search, i.e., IVF that clusters similar vectors into buckets [7,10,36,46,64,66] and proximity graph that connects similar vectors to form a graph [9, 14, 29, 39–41, 62, 69, 71,82,87]. Since proximity graph requires fewer distance computations to reach the same recall, most systems use proximity graph as the index. Many systems accelerate vector search on CPU. For instance, iQAN [45] and AverSearch [37] parallelize the processing of each query over multiple CPU threads for low latency. SymphonyQG [18] attaches the compressed vectors of its neighbors after a vector to reduce the cache misses during proximity graph traversal. ADSampling [15] and Tribase [75] reduce complexity by pruning unpromising candidates from exact distance computation. Several systems utilize accelerators like GPUs [19, 20, 42, 64, 66, 79] and FPGAs [58, 80, 81]. Developed by NVIDIA, CAGRA [42] implements efficient GPU kernels for building and searching proximity graph. Song [86] observes that data structure management is a major overhead for GPU and proposes to trim the visited node list to reduce such overhead. However, these systems cannot handle large datasets that do not fit in CPU or GPU memory.

Disk-based and distributed systems are proposed to process large vector datasets. As we have introduced, DiskANN [61], Starling [70], and PipeANN [21], and Gorgeous [77] are the state-of-the-art for disk-based systems. Some disk-based consider the case with very limited or even no memory, e.g., LM-DiskANN [44], SPANN [10], and AiSAQ [63], but their performance is also much worse than DiskANN-style systems. Vector databases [26, 35, 60, 67, 72] usually partition the dataset over the servers to conduct distributed vector search such that each shard fits in the memory of a server. However, as we have shown for Milvus in the experiments, partitioning harms the pruning power of vector index and thus query performance. Compared with these systems, our DistVS is the first to utilize the compute-memory disaggregation architecture for vector search. The low-precision vectors on the compute servers effectively prune compute-memory communication, while the high-precision vectors and proximity graph index on the memory server are shared among multiple compute servers and can reduce disk accesses.

Compute-memory disaggregation. The deployment of high-bandwidth and low-latency interconnects (e.g., RDMA, CXL [3]) enables compute–memory disaggregation, allowing systems to scale memory and compute resources independently. Many workloads like graph processing [54, 85], databases [32, 49, 76], and machine learning training [52, 68] adopt compute-memory disaggregation to reduce local DRAM footprint, improve resource utilization, and enhance elasticity. DistVS applies compute–memory disaggregation for vector search, following the general principles of making most data accesses local on the compute servers and reducing compute-memory communication. The insight is that vector search allows pruning with approximate distances, and thus DistVS stores the low-precision, high-precision, and full-precision vectors on compute, memory, and storage servers, respectively. The storage capacity and IO cost progressively increase along this compute-to-storage hierarchy while the access frequency reduces thanks to pruning.

7 Conclusions

In this paper, we propose DistVS, an efficient large-scale vector search system that tackles the high memory and IO costs by leveraging a compute-memory disaggregation architecture. Its novel three-tier storage layout and PRESS algorithm efficiently utilize the storage hierarchy, keeping most operations local. This design, combined with key system optimizations, enables DistVS to consistently outperform state-of-the-art systems in throughput while reducing the memory footprint.

Acknowledgments

We thank the anonymous reviewers for their careful reading and insightful feedback, which substantially improved the quality and clarity of this paper. We are especially grateful to our shepherd, Le Xu, for detailed guidance and constructive suggestions throughout the shepherding process.

References

[1] Linux raid array. https://docs.kernel.org/admin-guide/md.html, 2003.

[2] Efficient io with io_uring. https://kernel.dk/io_uring.pdf, 2019.

[3] Cxl 3.0 specification. https://www.computeexpresslink.org/download-the-specification/, 2022.

[4] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore L. Willke. Similarity search in the blink of an eye with compressed indices. *Proc. VLDB Endow.*, 16(11):3433–3446, 2023.

[5] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. Retrieval-based language models and applications. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts, ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 41–46. Association for Computational Linguistics, 2023.

[6] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.*, 87, 2020.

[7] Artem Babenko and Victor S. Lempitsky. The inverted multi-index. In *2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, June 16-21, 2012*, pages 3069–3076. IEEE Computer Society, 2012.

[8] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. BGE m3-embedding: Multilingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *CoRR*, abs/2402.03216, 2024.

[9] Meng Chen, Kai Zhang, Zhenying He, Yinan Jing, and X. Sean Wang. Roargraph: A projected bipartite graph for efficient cross-modal approximate nearest neighbor search. *Proc. VLDB Endow.*, 17(11):2735–2749, 2024.

[10] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong 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*, pages 5199–5212, 2021.

[11] Google Cloud. Compute engine prices. https://cloud.google.com/compute/all-pricing?hl=zh_cn#compute-optimized-machine-type-family, 2025.

[12] Huawei Cloud. Elastic memory service (ems). https://www.huaweicloud.com/product/ems.html, 2025.

[13] Wikimedia Foundation. Wikimedia downloads, 2007.

[14] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5):461–474, 2019.

[15] Jianyang Gao and Cheng Long. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proc. ACM Manag. Data*, 1(2):137:1–137:27, 2023.

[16] Jianyang Gao and Cheng Long. Rabitq: Quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *Proc. ACM Manag. Data*, 2(3):167, 2024.

[17] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36(4):744–755, 2014.

[18] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. Symphonyqg: Towards symphonious integration of quantization and graph for approximate nearest neighbor search. *arXiv preprint arXiv:2411.12229*, 2024.

[19] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. GGNN: graph-based GPU nearest neighbor search. *IEEE Trans. Big Data*, 9(1):267–279, 2023.

[20] Yuntao Gui, Peiqi Yin, Xiao Yan, Chaorui Zhang, Weixi Zhang, and James Cheng. Pilotann: Memory-bounded GPU acceleration for vector search. *CoRR*, abs/2503.21206, 2025.

[21] Hao Guo and Youyou Lu. Achieving low-latency graph-based vector search via aligning best-first search algorithm with SSD. In *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pages 171–186. USENIX Association, 2025.

[22] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. Manu: A cloud native vector database management system. *Proc. VLDB Endow.*, 15(12):3548–3561, 2022.

[23] Michael Günther, Saba Sturua, Mohammad Kalim Akram, Isabelle Mohr, Andrei Ungureanu, Sedigheh Eslami, Scott Martens, Bo Wang, Nan Wang, and Han Xiao. jina-embeddings-v4: Universal embeddings for multimodal multilingual retrieval, 2025.

[24] Tengda Han, Weidi Xie, and Andrew Zisserman. Video representation learning by dense predictive coding. In *2019 IEEE/CVF International Conference on Computer Vision Workshops, ICCV Workshops 2019, Seoul, Korea (South), October 27-28, 2019*, pages 1483–1492. IEEE, 2019.

[25] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[26] Guoyu Hu, Shaofeng Cai, Tien Tuan Anh Dinh, Zhongle Xie, Cong Yue, Gang Chen, and Beng Chin Ooi. Hakes: Scalable vector database for embedding search service. *arXiv preprint arXiv:2505.12524*, 2025.

[27] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. Embedding-based retrieval in facebook search. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 2553–2561. ACM, 2020.

[28] The Linux Programming Interface. Posix asynchronous i/o (aio). `https://man7.org/linux/man-pages/man7/aio.7.html`, 2008.

[29] Masajiro Iwasaki and Daisuke Miyazaki. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *CoRR*, abs/1810.07355, 2018.

[30] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, 2011.

[31] Wenqi Jiang, Shuai Zhang, Boran Han, Jie Wang, Bernie Wang, and Tim Kraska. Piperag: Fast retrieval-augmented generation via algorithm-system co-design. *CoRR*, abs/2403.05676, 2024.

[32] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. Gaussdb: A cloud-native multi-primary database with compute-memory-storage disaggregation. *Proc. VLDB Endow.*, 17(12):3786–3798, 2024.

[33] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. Embedding-based product retrieval in taobao search. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, pages 3181–3189. ACM, 2021.

[34] Haotian Liu, Kilho Son, Jianwei Yang, Ce Liu, Jianfeng Gao, Yong Jae Lee, and Chunyuan Li. Learning customized visual models with retrieval-augmented knowledge. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pages 15148–15158. IEEE, 2023.

[35] Yi Liu, Fei Fang, and Chen Qian. Efficient vector search on disaggregated memory with d-hnsw. *arXiv preprint arXiv:2505.11783*, 2025.

[36] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. JUNO: optimizing high-dimensional approximate nearest neighbour search with sparsity-aware algorithm and ray-tracing core mapping. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 549–565. ACM, 2024.

[37] Jingjia Luo, Mingxing Zhang, Kang Chen, Xia Liao, Yingdi Shan, Jinlei Jiang, and Yongwei Wu. Efficient graph-based approximate nearest neighbor search achieving: Low latency without throughput loss. *CoRR*, abs/2504.20461, 2025.

[38] Aravindh Mahendran and Andrea Vedaldi. Understanding deep image representations by inverting them. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 5188–5196. IEEE Computer Society, 2015.

[39] Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *CoRR*, abs/1603.09320, 2016.

[40] Magdalen Dobson Manohar, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan 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*, pages 270–285. ACM, 2024.

[41] Naoki Ono and Yusuke Matsui. Relative nn-descent: A fast index construction for graph-based approximate nearest neighbor search. In *Proceedings of the 31st ACM International Conference on Multimedia, MM 2023, Ottawa, ON, Canada, 29 October 2023- 3 November 2023*, pages 1659–1667. ACM, 2023.

[42] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. CAGRA: highly parallel graph construction and approximate nearest neighbor search for gpus. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*, pages 4236–4247. IEEE, 2024.

[43] Big ANN Organizers. Big ann benchmark, 2021.

[44] Yu Pan, Jianxin Sun, and Hongfeng Yu. Lm-diskann: Low memory footprint in disk-native dynamic graph-based ANN indexing. In *IEEE International Conference on Big Data, BigData 2023, Sorrento, Italy, December 15-18, 2023*, pages 5987–5996. IEEE, 2023.

[45] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin 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*, pages 313–328. ACM, 2023.

[46] Jeffrey Pound, Floris Chabert, Arjun Bhushan, Ankur Goswami, Anil Pacaci, and Shihabur Rahman Chowdhury. Micronn: An on-device disk-resident updatable vector database. *arXiv preprint arXiv:2504.05573*, 2025.

[47] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-context retrieval-augmented language models. *Trans. Assoc. Comput. Linguistics*, 11:1316–1331, 2023.

[48] Yandex Research. Benchmarks for billion-scale similarity search, 2021.

[49] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. Persistent memory disaggregation for cloud-native relational databases. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 498–512. ACM, 2023.

[50] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, Patrick Schramowski, Srivatsa Kundurthy, Katherine Crowson, Ludwig Schmidt, Robert Kaczmarczyk, and Jenia Jitsev. LAION-5B: an open large-scale dataset for training next generation image-text models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[51] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. LAION-400M: open dataset of clip-filtered 400 million image-text pairs. *CoRR*, abs/2111.02114, 2021.

[52] Chuanming Shao, Jinyang Guo, Pengyu Wang, Jing Wang, Chao Li, and Minyi Guo. Oversubscribing GPU unified virtual memory: Implications and suggestions. In *ICPE '22: ACM/SPEC International Conference on Performance Engineering, Bejing, China, April 9 - 13, 2022*, pages 67–75. ACM, 2022.

[53] Yang Shi, Yiping Sun, Jiaolong Du, Xiaocheng Zhong, Zhiyong Wang, and Yao Hu. Scalable overload-aware graph-based index construction for 10-billion-scale vector similarity search. In *Companion Proceedings of the ACM on Web Conference 2025, WWW 2025*, pages 1303–1307. ACM, 2025.

[54] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 135–146. ACM, 2013.

[55] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Freshdiskann: A fast and accurate graph-based ANN index for streaming similarity search. *CoRR*, abs/2105.09613, 2021.

[56] Fatima Zohra Smaili, Xin Gao, and Robert Hoehndorf. Onto2vec: joint vector-based representation of biological entities and their ontology-based annotations. *Bioinform.*, 34(13):i52–i60, 2018.

[57] Fatima Zohra Smaili, Xin Gao, and Robert Hoehndorf. Opa2vec: combining formal and informal content of biomedical ontologies to improve similarity-based prediction. *Bioinform.*, 35(12):2133–2140, 2019.

[58] Yifeng Song, Chenjie Liu, Rongrong Zhang, Danyang Zhu, and Zhongfeng Wang. An efficient FPGA implementation of approximate nearest neighbor search. *IEEE Trans. Very Large Scale Integr. Syst.*, 33(6):1705–1714, 2025.

[59] C++ Standard. Coroutines (c++20). https://en.cppreference.com/w/cpp/language/coroutines.html, 2020.

[60] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. Vexless: a serverless vector data management system using cloud functions. *Proceedings of the ACM on Management of Data*, 2(3):1–26, 2024.

[61] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13748–13758, 2019.

[62] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. Rand-nsg: Fast accurate billion-point nearest neighbor search on a single node. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13748–13758, 2019.

[63] Kento Tatsuno, Daisuke Miyashita, Taiga Ikeda, Kiyoshi Ishiyama, Kazunari Sumiyoshi, and Jun Deguchi. Aisaq: All-in-storage ANNS with product quantization for dram-free information retrieval. *CoRR*, abs/2404.06004, 2024.

[64] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Xuecang Zhang, Junhua Zhu, and Yu Zhang. Fusionanns: An efficient cpu/gpu cooperative processing architecture for billion-scale approximate nearest neighbor search. *CoRR*, abs/2409.16576, 2024.

[65] Nimish Ukey, Zhengyi Yang, Binghao Li, Guangjian Zhang, Yiheng Hu, and Wenjie Zhang. Survey on exact knn queries over high-dimensional data space. *Sensors*, 23(2):629, 2023.

[66] Karthik V., Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. BANG: billion-scale approximate nearest neighbor search using a single GPU. *CoRR*, abs/2401.11324, 2024.

[67] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles 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*, pages 2614–2627. ACM, 2021.

[68] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 41–53. ACM, 2018.

[69] Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Yifan Zhu, and Wenchao Zhou. Accelerating graph indexing for ANNS on modern cpus. *CoRR*, abs/2502.18113, 2025.

[70] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. *Proc. ACM Manag. Data*, 2(1):V2mod014:1–V2mod014:27, 2024.

[71] Yichuan Wang, Shu Liu, Zhifei Li, Yongji Wu, Ziming Mao, Yilong Zhao, Xiao Yan, Zhiying Xu, Yang Zhou, Ion Stoica, Sewon Min, Matei Zaharia, and Joseph E. Gonzalez. LEANN: A low-storage vector index. *CoRR*, abs/2506.08276, 2025.

[72] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proc. VLDB Endow.*, 13(12):3152–3165, 2020.

[73] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul N. Bennett, Junaid Ahmed, and Arnold Overwijk. Approximate nearest neighbor negative contrastive learning for dense text retrieval. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[74] Haike Xu, Magdalen Dobson Manohar, Philip A. Bernstein, Badrish Chandramouli, Richard Wen, and Harsha Vardhan Simhadri. In-place updates of a graph index for streaming approximate nearest neighbor search. *CoRR*, abs/2502.13826, 2025.

[75] Qian Xu, Juan Yang, Feng Zhang, Junda Pan, Kang Chen, Youren Shen, Amelie Chi Zhou, and Xiaoyong Du. Tribase: A vector data query engine for reliable and lossless pruning compression using triangle inequalities. *Proc. ACM Manag. Data*, 3(1):82:1–82:28, 2025.

[76] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. Polardb-mp: A multi-primary cloud-native database via

disaggregated shared memory. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago, Chile, June 9-15, 2024*, pages 295–308. ACM, 2024.

[77] Peiqi Yin, Xiao Yan, Qihui Zhou, Hui Li, Xiaolu Li, Lin Zhang, Meiling Wang, Xin Yao, and James Cheng. Gorgeous: Revisiting the data layout for disk-resident high-dimensional vector search. *CoRR*, abs/2508.15290, 2025.

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

[79] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. Gpu-accelerated proximity graph approximate nearest neighbor search and construction. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pages 552–564. IEEE, 2022.

[80] Wei Yuan and Xi Jin. FANNS: an fpga-based approximate nearest-neighbor search accelerator. *IEEE Trans. Very Large Scale Integr. Syst.*, 33(4):1197–1201, 2025.

[81] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, and Yu Wang. DF-GAS: a distributed fpga-as-a-service architecture towards billion-scale graph-based approximate nearest neighbor search. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*, pages 283–296. ACM, 2023.

[82] Ximu Zeng, Liwei Deng, Penghao Chen, Xu Chen, Han Su, and Kai Zheng. LIRA: A learning-based query-aware partition framework for large-scale ANN search. In *Proceedings of the ACM on Web Conference 2025, WWW 2025, Sydney, NSW, Australia, 28 April 2025- 2 May 2025*, pages 2729–2741. ACM, 2025.

[83] Yanhao Zhang, Pan Pan, Yun Zheng, Kang Zhao, Yingya Zhang, Xiaofeng Ren, and Rong Jin. Visual search at alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 993–1001. ACM, 2018.

[84] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

[85] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman P. Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with graphit. In *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020*, pages 158–170. ACM, 2020.

[86] Weijie Zhao, Shulong Tan, and Ping Li. Song: Approximate nearest neighbor search on gpu. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1033–1044. IEEE, 2020.

[87] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *Proc. VLDB Endow.*, 16(8):1979–1991, 2023.

Appendix

A Discussions of DistVS

Handling updates to the dataset. For many applications, the vector dataset will receive vector insertions and deletions. DistVS can easily supports online updates by adapting FreshDiskANN [55], a method to update disk-resident graph index for vector search. Since DistVS stores the index graph in the memory of the memory server, edits can be applied directly in memory without heavy disk IOs. Specifically, for both vector insertions and deletions, the memory node can execute a staged pipeline: (i) updating the in-memory graph structure; (ii) refreshing the associated high-precision quantized vectors; (iii) signaling the storage nodes to update the exact vectors on disk; and (iv) notifying compute nodes to refresh their low-precision quantized vectors. DistVS is also compatible with other updatable graph index implementations [74, 78], since they rely on vector search as the main routine (supported by DistVS) and the memory-resident graph of DistVS is easy to update.

The generality of DistVS. By default, DistVS utilizes Vamanna [61] as the proximity graph index and PQ [30] as the vector quantization method. We are aware that proximity graph has many variants, and some vector quantization methods yield better accuracy than the widely used PQ. Changing the proximity graph and vector quantization method is easy for DistVS and does not require to modify its system designs.

Failure handling. In the event of a memory server failure, compute nodes will lose their in-progress computations. However, the impact is limited: each compute node maintains a bounded number of concurrent requests (threads × coroutines), and individual request execution times are short (several milliseconds). Consequently, the recomputation overhead upon memory server recovery is lightweight, and the system can quickly resume and start the service. The memory server can also be implemented using existing industry-grade memory services, such as Huawei Cloud's Elastic Memory Service (EMS) [12], which provide built-in mechanisms for failure detection and recovery.

B Additional Experiments

Comparing with in-memory Vamana. In Figure 15, we compare *DistVS* with the in-memory version of *DiskANN* on Laion-T2I. For fair comparison, in-memory Vamana uses 40 threads [5], while *DistVS* uses 8 threads per server on five servers. The in-memory Vamana stores all data in DRAM and requires 251.2 GB, which is 2.5× the cluster-wide memory footprint of *DistVS*. At 90% recall, *DistVS* is 17% lower in

[5]On a single server with 192 GB/s memory bandwidth; bandwidth is not the bottleneck for the graph index.

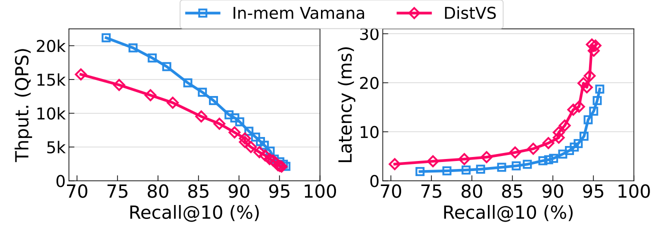

Figure 15: Comparing *DistVS* with in-memory Vamana.

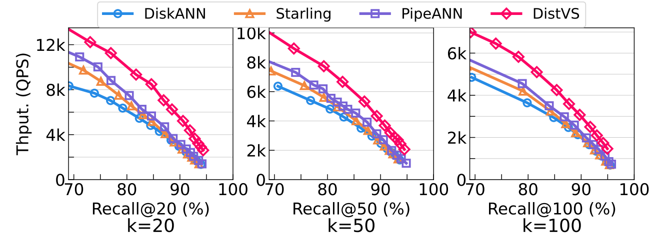

Figure 16: Changing the number of K on Laion-T2I.

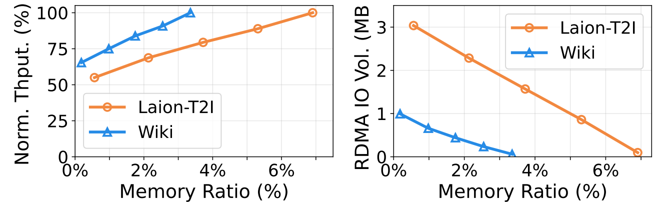

Figure 17: Changing the memory ratio for *DistVS*, comparing throughput (normalized by non-partial throughput) and RDMA IO volume per-request.

throughput, and this is because some threads are allocated to re-ranking and communication rather than query processing. This demonstrates *DistVS*'s high CPU utilization and strong performance with limited memory. *DistVS* also shows longer latency due to disk access and network transfer overhead.

Varying k values. Figure 16 shows performance when varying K rather than using a fixed k=10. *DistVS* consistently outperforms the baselines across different k values. As k increases, all systems show degraded performance because more distance calculations are required to achieve the same accuracy, and more disk accesses are needed. *DistVS* shown larger improvement on smaller k, as less disk accesses are required for re-ranking.

Low memory scenarios. Existing disk-based ANNS systems store the compressed vectors (e.g., PQ codes) in the memory of each server, and this imposes a nontrivial memory requirement (e.g., 6–7% of the dataset size to store PQ data with a compression factor of 16). Significant performance degradations are observed when the memory falls below this level since the compressed vectors become inaccurate. As shown in Figure 3, for DiskANN on Laion-T2I, reducing the memory from 12.5% of dataset size to 6.5% degrades query throughput from around 580 to 200 (i.e., a 65% performance drop), and further reducing the memory usage to 5% decreases

query throughput to around 50 (a 91% performance drop). In comparison, the performance degradations for *DistVS* are significantly smaller when the compute servers have low memory because the high-precision vectors on the memory server can compensate the accuracy of the low-precision vectors on the compute servers. As reported in Figure 17, reducing the memory of compute servers from 12.5% of dataset size to 6.5% incurs only a 23% throughput drop, and *DistVS* can still sustain approximately 52% of its original throughput under an extremely constrained memory setting (less than 0.5% of dataset size).