CoTra: Towards Efficient and Scalable Distributed Vector Search with RDMA

XIANGYU ZHI, The Chinese University of Hong Kong, Hong Kong SAR
MENG CHEN*, Fudan University, China
XIAO YAN†, Institute for Math & AI, Wuhan, Wuhan University, China
BAOTONG LU, Microsoft Research, China
HUI LI, The Chinese University of Hong Kong, Hong Kong SAR
QIANXI ZHANG, Microsoft Research, China
QI CHEN, Microsoft Research, Canada
JAMES CHENG, The Chinese University of Hong Kong, Hong Kong SAR

Similarity-based vector search facilitates many important applications such as search and recommendation, but is limited by the memory capacity and bandwidth of a single machine due to large datasets and intensive data reads. In this paper, we present CoTra, a system that scales up vector search for distributed execution. We observe a tension between *computation and communication* efficiency, which is the main challenge for good scalability, i.e., handling the local vectors on each machine *independently* blows up computation as the pruning power of vector index is not fully utilized, while running a *global index* over all machines introduces rich *data dependencies* and thus extensive communication. To resolve such tension, we leverage the fact that vector search is approximate in nature and robust to *asynchronous execution*. In particular, we run collaborative vector search over the machines with algorithm-system co-designs including clustering-based data partitioning to reduce communication, asynchronous execution to avoid communication stalls, and task push to reduce network traffic. To make collaborative search efficient, we introduce a suite of system optimizations including task scheduling, communication batching, and storage format. We evaluate CoTra on real datasets and compare with four baselines. The results show that when using 16 machines, the query throughput of CoTra scales to 9.8-13.4× over a single machine and is 2.12-3.58× of the best-performing baseline at recall@10≥0.95.

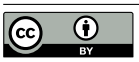
CCS Concepts: • **Information systems** → **Data management systems**; • **Computing methodologies** → **Distributed computing methodologies**.

Additional Key Words and Phrases: Vector search, distributed systems, RDMA

ACM Reference Format:
Xiangyu Zhi, Meng Chen, Xiao Yan, Baotong Lu, Hui Li, Qianxi Zhang, Qi Chen, and James Cheng. 2026. CoTra: Towards Efficient and Scalable Distributed Vector Search with RDMA. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 20 (February 2026), 27 pages. https://doi.org/10.1145/3786634

*Work partially performed during the internship at Microsoft.
†Dr. Xiao Yan is the corresponding author.

Authors' Contact Information: Xiangyu Zhi, xyzhi24@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR; Meng Chen, mengchen22@m.fudan.edu.cn, Fudan University, China; Xiao Yan, yanxiaosunny@whu.edu.cn, Institute for Math & AI, Wuhan, Wuhan University, China; Baotong Lu, baotonglu@microsoft.com, Microsoft Research, China; Hui Li, hli@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR; Qianxi Zhang, qianxi.zhang@microsoft.com, Microsoft Research, China; Qi Chen, cheqi@microsoft.com, Microsoft Research, Canada; James Cheng, The Chinese University of Hong Kong, Hong Kong SAR, jcheng@cse.cuhk.edu.hk.



ACM 2836-6573/2026/2-ART20
https://doi.org/10.1145/3786634

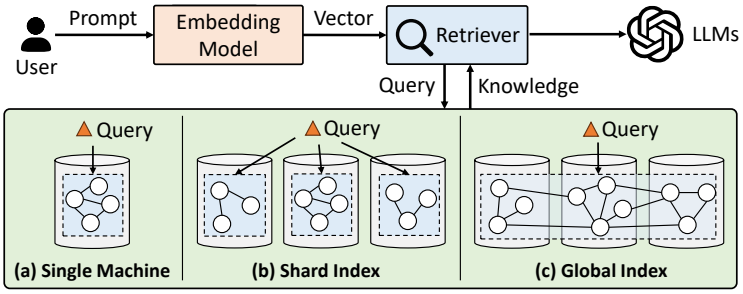


Fig. 1. Vector search in RAG (top) and three designs of large-scale vector search (bottom).

1 Introduction

High-dimensional vector search is fundamental to many modern applications as it enables semantic similarity comparisons across diverse data types from text and images to genomic sequences. Specifically, complex data objects are mapped to vector embeddings [5, 8, 41, 67], and then similarity-based *vector search* is conducted to retrieve top-k most similar vectors (e.g., measured by Euclidean distance) to a query vector [32]. Since exact vector search requires a linear scan of high-dimensional vectors in a vector dataset, *approximate vector search* has been widely adopted to retrieve most (instead of all) of the top-k similar vectors to trade for efficiency. Approximate vector search powers numerous critical tasks like content-based search (e.g., for images and videos) [21, 25, 63], recommender systems (e.g., for e-commerce products and online advertisements) [7, 18, 36, 53], and bioinformatics [4, 29, 45]. Figure 1 shows a popular application, retrieval-argumented generation (RAG) for LLMs [66], where text chunks are embedded as vectors and user prompts act as queries, and retrieved text chunks provide external knowledge to LLMs for response generation.

The state-of-the-art index for vector search is proximity graph [32, 56]. Although there are many variants [15, 34, 38], the core idea is to organize vectors into a graph, where each vector is a node and connects to its similar vectors[1]. Vector search is processed by a *graph traversal*, which starts from a fixed or random entry node, computes similarities for all neighbors when visiting a node, and maintains a *candidate queue* to record the computed similarities and choose the most similar but unvisited node as the next node to visit. For a dataset with N vectors, the number of similarity computations required by the graph traversal is $\log N$ in scale [15, 34, 40].

Motivations and challenges. Scalable vector search requires distributed execution across machines to address both single-machine memory *capacity* and *bandwidth* limitations, especially as billion-scale datasets and high-dimensional vectors (hundreds of dimensions) become commonplace today. More critically, vector search is usually *memory bound* since modern CPUs outpace memory bandwidth and each query accesses numerous vectors [9]. As shown in Figure 2, scaling from 8 to 128 threads on a single machine improves query throughput (QPS) by only 3-4×, far below the ideal 16×. This is because memory bandwidth (204 GB/s for our machine) is saturated and thus increasing the number of threads beyond a point (48 in our experiments) will not improve QPS. Memory bound is influenced not only by memory bandwidth limitations but also by NUMA effects, and no significant performance difference between node binding and interleaved allocation. This is because the large vector datasets typically span multiple NUMA nodes, and due to the random walk nature of graph-based vector search, cross-NUMA memory accesses are inevitable. For the same reason, disk-based systems [7, 50], which keep vectors on SSDs, are also limited by disk

[1]We use "node" and "vector" interchangeably.

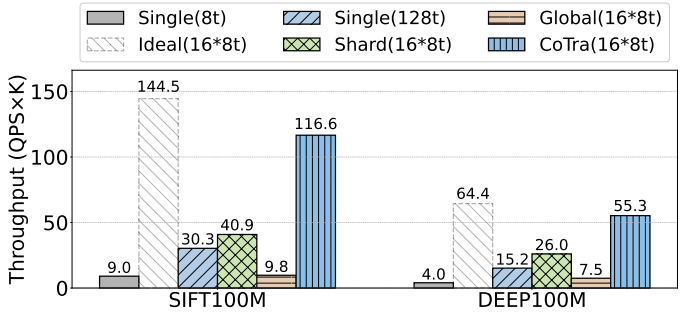
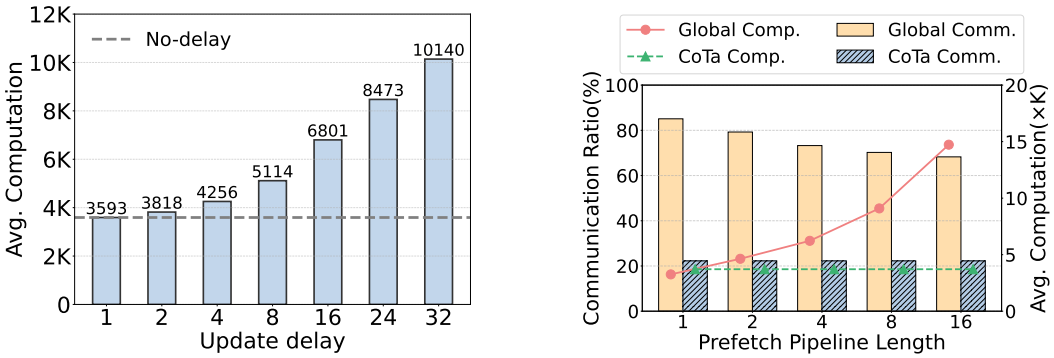


Fig. 2. Query throughput (QPS) at 0.95 recall for top-10 vector search. Single-machine experiments use 8 and 128 threads; distributed solutions (*Shard*, *Global*, and our *CoTra*) use 16 machines (8 threads each). *Ideal* means ideal linear scaling for 16 machines.

bandwidth. In comparison, distributed solutions can leverage the aggregated memory bandwidth of multiple servers for higher throughput. We evaluate the in-memory Vamana index performance on a high-capacity server configuration comprising four NUMA nodes with 192 logical threads and 768GB DRAM. Our experiments demonstrate that memory bandwidth saturation occurs beyond 48 threads, limiting throughput to 160 GB/s (DDR4). As shown in Figure 2, the single-machine in-memory system achieves only a 3.8× throughput improvement at 128 threads compared to 8 threads, whereas the ideal scaling should reach 16×.

However, distributed vector search faces a critical tension between *pruning effect* and *communication efficiency*. The simplest solution is *independent sharding* (called *Shard*), as shown in the middle of Figure 1. Shard assigns vectors across machines, each constructing a local index on its vectors. A query is first broadcast to all machines for local processing, and then the local search results are merged to obtain the global top-k most similar vectors. While simple and adopted by popular vector databases such as Milvus [54] and Weaviate [12], Figure 2 shows that *Shard* only achieves 4-6× QPS gain with 16 machines. This is because the query complexity of proximity graph is *sublinear* w.r.t. dataset size (i.e., $\log N$), and thus partitioning a large dataset into M (i.e., the number of machines) independent sub-datasets means that more similarity computations are used to process each query compared to organizing all vectors in a single index (i.e., $M \log(N/M) > \log N$, where $N \gg M$). To improve *Shard*, Pyramid [11] and Vexless [49] partitions the vectors to machines via K-means, and at query time, only a subset of the partitions (i.e., machines) are checked for each query. We profile this solution Section 3 and find that it can be inefficient, especially at high recall regimes (>0.95), because many partitions need to be checked. Moreover, it introduces additional algorithmic complexity to determine the number of partitions to check.

To fully exploit the pruning power of vector indexes and improve computational efficiency, a natural solution is to utilize a global index on the entire dataset. As shown in the rightmost plot of Figure 1, *Global* keeps some vectors and their adjacency lists on each machine, but the vectors on one machine may connect to the vectors on other machines. Each query is assigned to a machine for processing, and when the graph traversal visits a node, some of its neighbors may reside on remote machines. In this case, network communication is required to fetch these neighbor vectors for computation. However, as shown in Figure 2, the QPS of *Global* is also far below ideal linear scaling. This is because *Global* communicates many vectors over the network, which are large in size and easily saturate network bandwidth. Moreover, although the latency of one-sided remote direct memory access (RDMA) communication (at 2-3 μs) is much shorter than TCP/IP [14, 26],



(a) Update delay impact.

(b) Prefetch solution.

Fig. 3. Impact of *candidate queue update delay* on similarity computations (SIFT100M dataset, 0.95 recall), and the change of communication time ratio and computation count with step size in the prefetch-based method.

it is still over 10× of local memory access. Moreover, since a query needs many communication operations, and the next node to visit can only be determined after updating the candidate queue with remote vectors, the query latency of *Global* can be 10-20× of a single machine.

Our solution: CoTra. We present CoTra, a distributed in-memory vector search system that resolves the tension between pruning effect and communication efficiency. As shown in Figure 2, the QPS of CoTra is significantly higher than both *Shard* and *Global* and is over 0.8× of ideal linear scaling. To our knowledge, CoTra is the first to run a holistic graph index over multiple machines and achieve near-linear scaling w.r.t. single-machine proximity graph.

To avoid the network saturation of *Global*, we start with *computation push*, which pushes the tasks of computing distances to remote machines instead of pulling large vectors. Although computation-push effectively reduces network traffic, its QPS remains low. This is because the remote distance computation tasks have unstable delays, which degrades the candidate queue that is crucial for proximity graph traversal. In particular, if a node u has smaller distance than the current best candidate v but is returned late, the graph traversal will visit v, which is suboptimal and thus wastes computation. This is illustrated in Figure 3a, where we delay the updates to the candidate queue after visiting some nodes and computation escalates with moderate delays (e.g., 16). We also tried several data partitioning techniques but found it difficult to concentrate the vectors required by queries to a single machine to avoid communication. Furthermore, we demonstrate that conventional optimization techniques, such as prefetch-based solution, fail to effectively address these communication bottlenecks as illustrated in Figure 3b. In particular, the pipeline length is the number of prefetch pipelines, and after issuing the prefetch requests, we asynchronously switch to computation tasks to achieve computation-communication overlap. However, this leads to delayed updates of the candidate queue, which in turn reduces pruning efficiency because suboptimal candidates are prefetched. Results shows that the distance computations per query of *Global* grows rapidly with the pipeline length.

The failure of *computation-push* suggests that candidate quality should be maintained when reducing communication, and CoTra achieves this goal with algorithm-system co-designs. In particular, K-means is utilized to partition vectors to machines in a similarity-based manner. For each query, this allows to distinguish machines into *primaries* that host many accessed vectors and *secondaries* that host a few accessed vectors. Moreover, vectors on the primary machines are generally closer to the query. As such, we use different execution strategies for the primary and

secondary machines. The primary machines run *Co-Search* and work like single-machine search but their local candidate queues are synchronized at regular intervals to ensure the quality of the candidates. The secondary machines run *Push-Pull* by sending vectors or computing distances for the primary machines on demand; they may have long delays to update the candidate queue but the degradation is limited since their vectors are usually far from the query.

CoTra also incorporates a suite of system optimizations to improve efficiency. First, as the basic operations, both *Co-Search* and *Push-Pull* are implemented efficiently over RDMA. Second, task scheduling is conducted on each machine to overlap computation and communication and avoid long synchronization delays for the candidate queues. Third, communication tasks on the same machine are carefully batched to reduce the required IO operations. Finally, we tailor the distributed storage format of the proximity graph index for efficient communication and task execution. We also implement a distributed procedure to build the global proximity graph index for large vector datasets.

We evaluate CoTra on four large-scale vector datasets with up to 1 billion vectors, comparing it against *Shard*, *Global*, and a state-of-the-art distributed vector database. The results show that CoTra consistently achieves higher QPS than all baselines and has close-to-linear scaling for QPS when increasing the number of machines. Our ablation studies also validate the effectiveness of our designs.

To summarize, we make the following main contributions.

- We motivate distributed vector search by scaling up both memory capacity and bandwidth. We further explore various design alternatives and, through experiments and profiling, reveal their limitations. This leads us to identify the key scalability challenge as balancing computation and communication efficiency.
- We propose collaborative search to enable fine-grained coordination across the machines for vector search, which maintains pruning effect at low communication cost.
- We tailor a suite of system optimizations for CoTra, which includes task scheduling, communication batching, index storage, and distributed index building.
- We open-source CoTra at https://github.com/xiangyuzhi/CoTra and extensively evaluate its performance, providing insights into the design choices.

2 Background

In this part, we introduce the basics of vector search and RDMA to facilitate the subsequent discussions.

2.1 Vector Search and Proximity Graph Index

Vector search is also known as nearest neighbor search (NNS) and defined as follows.

Definition 2.1. Given a vector dataset $\mathcal{X} = \{x_1, x_2, \cdots, x_N\} \subset \mathbb{R}^d$ with size N, a query vector $q \in \mathbb{R}^d$, and the number of required vectors k, return a set of vectors $\mathcal{S} \subset \mathcal{X}$ that satisfy

$$\|x_i - q\| \leq \|x_j - q\|, \forall x_i \in \mathcal{S}, x_j \in \mathcal{X} - \mathcal{S} \text{ and } |\mathcal{S}| = k.$$

Besides Euclidean distance, the similarity function may also be inner product or cosine similarity. Since embedding vectors usually have a high dimension (e.g., 100 or even >1000), exact NNS requires a linear scan. Thus, approximate NNS (ANNS) is often employed in practice, which returns most rather than all of the top$k$ neighbors to trade for efficiency. The result quality of ANNS is measured by *recall*, which is defined as $|\mathcal{S} \cap \mathcal{S}'|/k$, where $\mathcal{S}'$ is an approximate result set with k vectors. Applications typically require a high recall (e.g., 0.9 or 0.95). ANNS needs to handle large vector datasets (e.g., for RAG with trillions of tokens [46]) and provide high query processing throughput

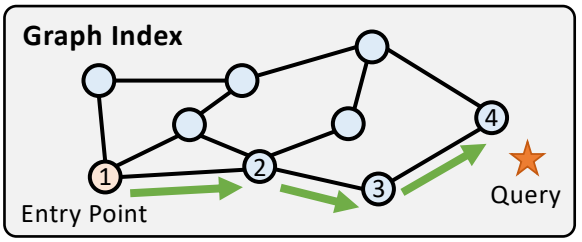


Fig. 4. Vector search with proximity graph index.

Algorithm 1 Graph Traversal for Vector Search

```
1:  Input: Graph G, query q, result count k, queue size L
2:  Output: k similar vectors to q
3:  Initialize a size-L priority queue Q with (v_0, ||v_0 − q||)
4:  while Q has unvisited node do
5:      Read the most similar but unvisited node v in Q
6:      for each neighbor u of v in G do
7:          if distance ||q − u|| is not computed then
8:              Compute ||q − u||
9:              Try to insert (u, ||q − u||) into Q
10: return The k vectors with the smallest distances in Q
```

(QPS) (e.g., for search and recommendation [7, 10]). A short query latency (e.g., 10-100ms) is also required for the quality-of-service (QoS) of user-facing applications.

Proximity graph [19] is the state-of-the-art index for vector search [19] and requires much fewer distance computations to reach the same recall than other indexes (e.g., IVF and LSH). The key idea is to connect each vector with its similar vectors to form a graph, as illustrated in Figure 4. Vector search is conducted by the graph traversal procedure in Algorithm 1. Q is a min priority queue and called the *candidate queue*. Line 3 of Algorithm 1 initializes Q with the entry node (i.e., v_0); lines 4-9 checks the most similar but unvisited node v in Q by computing distances for v's neighbors. Note that the visited nodes can still reside in Q (i.e., line 5 uses read instead of pop, and Q meets the size constraint L by discarding the vectors with large distances. Users configure L to control the quality of search results, with a larger L yielding higher recall at the cost of longer processing time. Proximity graph has good performance because (i) the graph identifies good candidates, i.e., if a vector is similar to the query, its neighbors are also likely to be similar, and (ii) by visiting the most similar vector each time, the graph traversal gradually move to the results of vector search, as shown in Figure 4,

2.2 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a networking technology for high-throughput, low-latency communication between servers. Different from TCP/IP-based networks, which incur significant overhead from kernel processing and data copying, RDMA allows to bypass remote operating systems and directly read from or write to remote memory. Modern RDMA networks achieve bandwidths of 40-400 Gbps [27, 59], far exceeding the 10-25 Gbps of Ethernet, and the end-to-end latencies are in microseconds (i.e., us). RDMA has been widely employed to design efficient distributed systems such as databases [68] and machine learning [22].

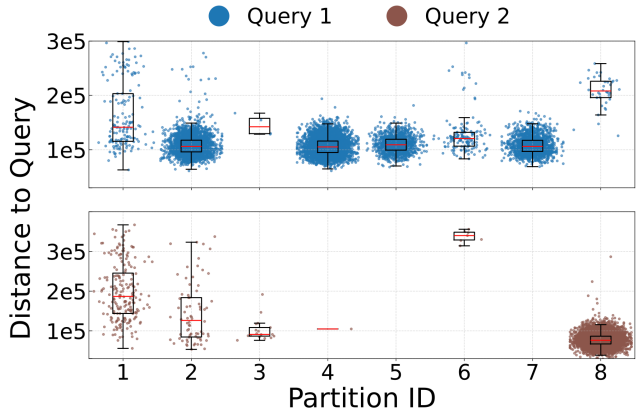


Fig. 5. Access patterns of two sample queries. The dataset is SIFT100M and clustered into 8 partitions. Point clouds denote the number of accessed vectors in each partition, while box plots show the distances of these vectors to query.

RDMA operations are abstracted as *verbs*, a set of low-level primitives including *one-sided* verbs and *two-sided* verbs. One-sided verbs (e.g., `READ`, `WRITE`, and atomic operations like `compare-and-swap /CAS`) execute without remote CPU involvement and thus are ideal for latency-critical tasks. Two-sided verbs (e.g., `SEND/RECV`) resemble TCP/IP semantics, i.e., require the coordination between sender and receiver CPUs, and are typically used for control messages.

The key challenge of using RDMA for vector search is to balance between communication latency and bandwidth utilization. In particular, one-sided RDMA verbs have low latency (e.g., 2-3μs in our cluster when the payload is below 2KB) but can only access contiguous remote memory for read/write operations. However, vector search follows the proximity graph to conduct fine-grained random accesses to the vectors. Moreover, the throughput of RDMA communication is constrained by both network bandwidth and IO operations per second (IOPS) due to the physical limits of NICs [47]. For instance, to fully utilize the 56 Gbps bandwidth in our cluster, the payload must exceed 1KB due to limited IOPS. However, an individual vector may be smaller than 1KB, e.g., a 128-dimension int8 vector takes only 128B.

3 Observations and Insights

Before introducing our collaborative traversal, we present the insights that motivate its designs.

Data locality. We first check how effective K-means partitioning is in improving access locality. If each query mostly accesses the vectors on one machine, and we assign the query to this machine for processing, communication will be small. Figure 5 shows that this is not the case, i.e., although Query 2 mostly accesses the vectors on Partition 8, the vectors accessed by Query 1 are scattered more uniformly over the partitions. By averaging over many queries, we find that 73.8% of the accessed vectors reside on the hottest partition, and note that the hottest partitions are different for queries. As such, if we assign each query to its hottest partition for processing, about 25% of the vector accesses need to go over the network. This will still make network the bottleneck due to the large bandwidth disparity between memory and network. For instance, our machine has a memory and network bandwidth of 204 GBps and 56 Gbps, respectively; to keep up with local in-memory processing and read 25% of the vectors over network, the required bandwidth should be 350 Gbps.

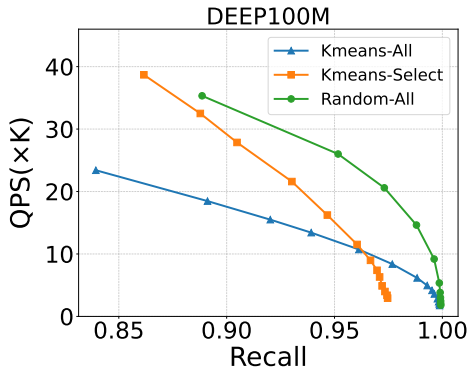


Fig. 6. The performance of K-means-based shard index on DEEP100M dataset partitioned into 16 partitions. *Kmeans-All* denotes searching all partitions, *Kmeans-Select* denotes searching the selected partitions, and *Random-All* denotes searching all randomly partitions.

As such, beside K-means partitioning, other designs are still required to reduce communication costs.

We have also tried more sophisticated data partitioning such as min-cut to partition its nodes over the machines by minimizing the number of cross-partition edges. In particular, the min-cut partitioning first builds a k-nearest neighbor (kNN) graph for the dataset and then uses min-cut to minimize the number of cross-partition edges, where the nodes (i.e., vectors) in each graph partition constitute a sub-dataset. The rationale is that the kNN graph better models the local neighborhood of each vector than the Euclidean distance in K-means partitioning, and by minimizing the number of cross-partition edges, similar vectors are assigned to the same sub-dataset. However, we observe that min-cut partitioning has only marginal improvement over K-means partitioning in localizing the vectors accessed by each query. This is because high-dimension vector space is inherently difficult to partition. As such, we choose K-means due to its simplicity.

In Figure 5, for each query, we can clearly distinguish between hot partitions with many accesses (e.g., Partitions 2, 4, 5, 7 for Query 1, and Partition 8 for Query 2) and cold partitions with a few accesses (e.g., Partitions 1, 3, 6, 8 for Query 1). We call the hot ones *primary partitions* and the cold ones *secondary partitions*.

Insight 1. *K-means partitioning improves data locality and allows to distinguish primary and secondary partitions for each query. However, communication is still the bottleneck.*

Select partitions. With K-means partitioning, Pyramid [11] builds an independent proximity graph on each machine and searches only a subset of the machines for each query. The rationale is that, for each query, some partitions are more likely to contain its results than the other machines. In particular, Pyramid builds a small top-level index by sampling vectors from the dataset and first searches each query on the top-level index. Then, Pyramid computes each machine hosts how many vectors in the top-level search results and checks only the top-ranking machines for the query. Figure 6 compares this strategy (i.e., *Kmeans-Select*) with *Kmeans-All* and *Random-All*, which search all machines but use different data partitioning. The parameters of *Kmeans-Select* are carefully tuned to optimize its performance, and a large query batch size (i.e., 32) is used to reduce potential machine idling.

A surprising phenomenon in Figure 6 is that *Kmeans-All* performs much worse than *Random-All* although they both search all machines. This is because with random partitioning, the ground truth

neighbors of each query tend to be uniformly distributed across the machines. While with K-means partitioning, some machines may have more ground-truth vectors than the other. For instance, when searching top-10 neighbors with 8 machines, each machine may contain 1-2 neighbors with random partitioning but one machine may contain 5 neighbors for K-means partitioning. As such, a larger queue size L needs to be used for *Kmeans-All* to reach the same recall, which introduces more computations.[2]

Figure 6 also shows that *Kmeans-Select* does outperform *Random-All* at low recall regimes (e.g., <0.90) because only some of the machines are checked but the advantage is marginal. Moreover, the query throughput of *Kmeans-Select* drops quickly when increasing the target recall, and this is because many machines need to be checked. *Kmeans-Select* may perform even worse than *Kmeans-All* when the recall is high as some machines may not contain the results for top-level index search but do contain the ground truth neighbors. In these cases, *Kmeans-Select* will never check these machines, and thus increasing the queue size L will only increase computation but not recall. This explains the saturated recall of *Kmeans-Select*. Moreover, the navigation index cannot fully capture the true query distribution. The gap between them makes it difficult to determine the optimal queue size through algorithmic tuning.

Our profiling does not preclude *Kmeans-Select* but it does show that sophisticated algorithm designs (e.g., deciding the machines to check and setting the queue sizes for the machines) are required to make it efficient. In contrast, CoTra uses a global graph structure over distributed vectors. From the algorithm perspective, it resembles classical single-machine vector search, and thus is simple to reason and configure parameters. Moreover, leveraging global structural information helps improve search routing convergence.

Insight 2. *K-means can reduce cross-machine communication, but K-means–based partitioned indexes still struggle to achieve the search efficiency of a global index.*

Candidate quality. As vector search is approximate in nature, we do not really need to faithfully execute Algorithm 1. To reduce the communication cost, we can flexibly control the amount and frequency of the communications. However, these communication designs should limit their impacts on pruning effect, i.e., the number of distance computations required by each query to reach the target recall. In particular, we find that the pruning effect of Algorithm 1 is decided by the candidate queue Q. That is, we may skip or delay some distance computations for the vectors on remote machines; if these computations involves a node (i.e., vector) u that is better than the current best unvisited node v in Q, i.e., $\|q-u\| < \|q-v\|$, visiting v may lead the graph traversal to detours and wastes computation. Figure 3a also shows that computation increases if the candidate quality is degraded by update delays. Conversely, if the candidate queue can produce the same node u to visit as single-machine execution, pruning effect will not be affected.

The above analysis suggests that to avoid degrading the quality of the candidate queue, our communication designs should prioritize high-quality vectors (i.e., those with small distances to the query) and discriminate the low-quality vectors (i.e., those with large distances). Figure 5 shows that the primary and secondary partitions are informative about candidate quality because vectors from the primary partitions (e.g., Partitions 2, 4, 5, 7 for Query 1) generally have smaller distances to the query than vectors from the secondary partitions (e.g., Partitions 1, 3, 6, 8 for Query 1).

Insight 3. *Vectors with small distances to the query should prioritized to keep the quality of the candidate queue. Vectors from the primary partitions are generally closer to the query.*

[2]One may argue that the phenomenon occurs because we use the same queue size L for all machines but using different queue sizes for the machines require additional algorithm designs. Pyramid also uses the same L for all selected machines.

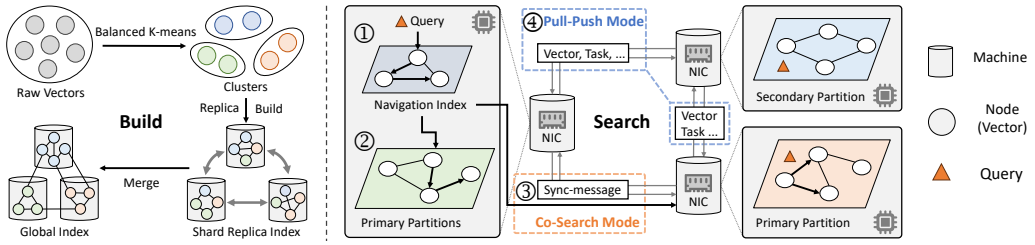


Fig. 7. The execution workflow of CoTra.

4 Distributed Collaborative Traversal

We build CoTra as a system that unleashes the power of distributed vector search over multiple machines by efficiently scaling up both memory capability and bandwidth based on RDMA. It aims to run distributed search on the partitioned global graph with good scalability, i.e., the query processing throughput (QPS) should increase linearly with the number of machines. For this purpose, we design a collaborative graph traversal procedure to run the proximity graph search algorithm (i.e., Algorithm 1) for each query over multiple machines with high efficiency.

Data layout. To fully utilize the pruning power of vector index, CoTra adopts a holistic proximity graph over the entire dataset, and we will discuss how to build the graph in Section 5. As shown in Figure 1(c), each machine keeps a similar number of vectors and the adjacency lists of these vectors in the graph index. Since the graph is holistic, the vectors on one machine may connect to the vectors on other machines. In particular, we partition the vector dataset using balanced K-means and assign each partition to one machine. This is intended to make the vectors on one machine similar to each other and concentrate the vectors accessed by each query to a small number of machines for good locality.

4.1 Collaborative Graph Traversal Overview

Our observations in Section 3 suggest that for each query, the primary partitions with many accessed vectors at close distances and secondary partitions with a few accessed vectors at far distances should be treated differently. Therefore, as shown in Figure 7, we first use a *navigation index* to efficiently distinguish the primary and secondary partitions. Then, we use different modes to coordinate the machines (i.e., partitions) for collaborative graph traversal. In particular, the primary partitions work in *Co-Search* mode by maintaining their own candidate queues, running distance computations on local vectors, and sending distance computation requests to the other partitions (i.e., both primary and secondary). The local candidate queues of the primary partitions are synchronized at regular intervals to ensure *bounded delay* and thus quality of the candidates. In contrast, the secondary partitions work in *Pull-Push* mode by sending vectors or computing distances as requested by the primary partitions without maintaining local candidate queues.

The key idea of our design to **maintain candidate quality at low communication costs**. First, the primary partitions do not exchange vectors and distances because many vectors are accessed on them; instead, the local candidate queues are much smaller for exchange. Second, to ensure candidate quality, the primary partitions are synchronized for bounded delay because their vectors are generally close to the query; tasks on the the secondary partitions have looser synchronization requirements because their vectors are usually far from the query. Third, candidate queues are not maintained on the secondary partitions because a few vectors are accessed on them, and thus the communication costs for exchanging vectors and distances are moderate.

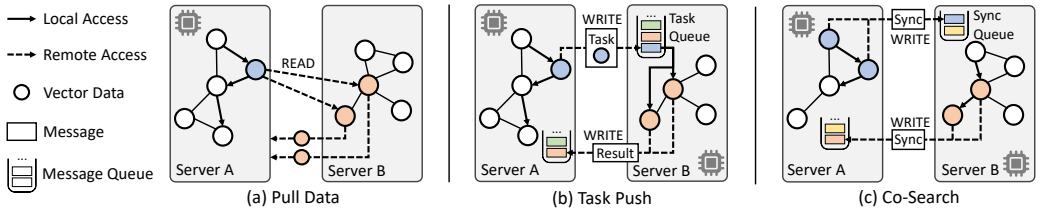


Fig. 8. CoTra communication operations. (a-b): When visiting the blue node, its orange neighbors are required on remote machines; (c): Co-Search conducts parallel traversal and synchronizes query states between servers.

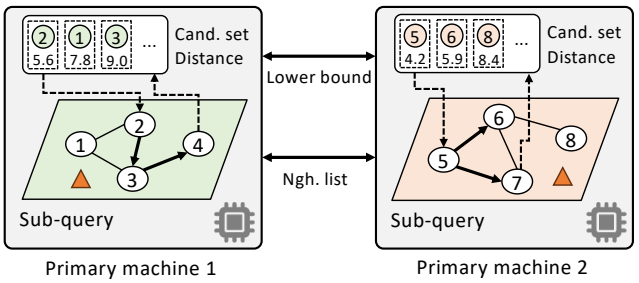


Fig. 9. An example of one query with two sub-queries.

Navigation Index. We sample a small portion (e.g., 1%) of vectors from the dataset and build a proximity graph index on these vectors as the navigation index. The navigation index is replicated over all machines, and each query is first assigned to a random machine for initialization. The machine searches the query with the navigation index to determine its primary and secondary partitions. In particular, we find the top-k neighbors of the query in the navigation index and count how many of these neighbors reside on each partition. A partition is *primary* if it holds more than k/M of these neighbors (M is the number of partitions) and *secondary* otherwise.

To speed up query processing, we use the top-k neighbors from the navigation index to initialize the local candidate queues of the primary partitions. Each primary partition collect the neighbors that reside on them, while the neighbors on the secondary partitions are sent to the primary partition that hosts the largest number of these top-k neighbors.

Collaborative traversal search. A *query* is a request submitted by a user to CoTra to search for the nearest neighbors of a query vector. A *sub-query* involves the state and computation on one machine for a query, and a query can spawn multiple sub-queries on different machines. Algorithm 2 details query processing with our collaborative traversal. In lines 3-4, the navigation index is searched to identify primary and secondary machines and dispatch sub-queries (i.e., candidate queues) to the primary machines (Figure 7 ①→②), which start graph traversal in lines 5-12. Local traversal follows Algorithm 1, i.e., each iteration extracts the most promising candidate from the candidate queue, traverses its neighbors, and packs the computation tasks into different message types, such as computation tasks and synchronization messages, based on the machines where their targeted vectors reside. As shown in Figure 9, each sub-query has a separate candidate queue that contains only the local vectors on that machine, and the sub-queries of the same query synchronize their states for query processing. Each sub-query processes its local tasks while concurrently updating the synchronization messages (line 13). Based on the messages received from the query message

Algorithm 2 Collaborative Traversal Search

```
1:  Input: Graph G, query Q, number of result vectors k
2:  Output: k vectors that are similar to Q
3:  cand_queue ← NavigationIndex(Q)
4:  DispatchSubQuery(cand_queue)
5:  while Changed(cand_queue) or !Terminate(Q) do
6:      Nodes ← cand_queue.top_unvisited()
7:      Machines, Tasks, SyncMsg ← G.getNgh(Nodes)
8:      for machine ∈ Machines do
9:          if machine is primary then
10:             PostSync(SyncMsg) // Co-Search mode
11:         else
12:             PullPush(Tasks) // Pull-Push mode
13:     SyncMsg ← LocalTraversal(Tasks)
14:     for Result, RecvSync ← PullMsg() do
15:         cand_queue.update(Result, RecvSync)
16:     TerminationDetection(Q)
17: return cand_queue.top(k)
```

queues, which contain both remote computation results and synchronization data from other primary machines, a sub-query updates its candidate queue (lines 14-15). A termination detection algorithm (line 16), which will be detailed in Section 5.3, activates when the candidate queue is empty.

4.2 Execution Mode Details

We implement the Pull-Push and Co-Search modes for collaborative traversal efficiently over RDMA. Figure 8 provides an illustration, and we present the details as follows.

Pull-Push mode. We use it pull vectors from and push distance computation tasks to the secondary machines.

- *Pull-Data*: As shown in Figure 8(a), when visiting a candidate, some of its neighbors may reside on a remote machine. CoTra can determine the remote memory addressees of these target vectors according to their IDs and retrieves them via RDMA one-sided primitives. A pre-allocated local buffer is used to store the retrieved vectors, and an asynchronous one-sided READ primitive is adopted to avoid communication stall.
- *Task-Push*: To reduce the bandwidth consumption for transferring large vectors, CoTra can use task-push as shown in Figure 8(b). Specifically, the IDs of the vectors to be computed are written into the remote machine's task queue using one-sided RDMA writes. The remote machine retrieves tasks from the queue, performs computations on the specified vectors, and returns only the distances to the requesting machines. Note that each query only needs to be sent to the machines once.

We use *Pull-Data* when a remote machine has fewer than or equal to two neighbors of the currently visited node and *Task-Push* otherwise. This is because *Pull-Data* does not involve the CPUs of remote machines and thus has shorter latency. This is favorable but it can incur excessive

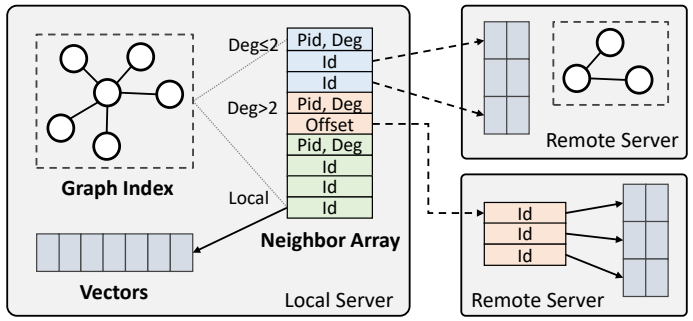


Fig. 10. Optimized graph layout.

bandwidth consumption when there are many remote neighbors. Empirically, we observe that setting the threshold as 2 balances between latency and bandwidth.

Co-Search mode. The primary machines use the Co-Search mode to exchange query states as shown in Figure 8(c). In particular, the exchanged messages include

- new candidates that are inserted into the candidate queue since the last synchronization.
- the current distance upper bound for a node to be added to the candidate queue. A node must have smaller distance to query than the bound to enter the candidate queue.
- vectors whose distances need to be computed on the remote machines since the last synchronization.

These messages grouped by their target machines and packed together for sending. Co-Search conducts synchronization after visiting a fixed number of candidates (set as 4 by default) to ensure good quality for the candidates. Using the synchronization messages, each primary machine updates its local candidate queue and distance upper bound (by taking the minimum of all bounds), and checks if the candidates to be traversed locally have been visited by the other machines. If not, it traverses the candidates to update the local candidate queue as in Algorithm 1. The distance upper bound is used to filter candidates that can be added to candidate queue.

5 Optimizations and Implementations

To enable more efficient execution of CoTra on RDMA-based clusters, we propose a graph storage and scheduling design to further optimize performance.

5.1 RDMA-friendly Graph Layout

To reduce communication costs, we tailor a graph storage format for vector search and RDMA. In particular, we store the proximity graph index in a decoupled manner, where the neighbors of a node on a remote machine are stored on the remote machine. As shown in Figure 10, the adjacency list of each node consists of local edges and pointers to remote edges. To work with the hybrid *Pull-Push* mode, each node stores pointers for high-degree remote neighbor array and IDs for low-degree remote neighbors. Using this storage format, the *Pull-Push* mode can further reduce network bandwidth, as the transmission of large edges is replaced by the pointers.

To conduct efficient memory access, we implement fine-grained prefetching. Vector search on proximity graph conducts many random memory accesses, including local vector reads, graph adjacency list traversals, and visit list operations. We proactively issue prefetch instructions to overlap data transfers with computation. When constructing the graph index, we pre-group each

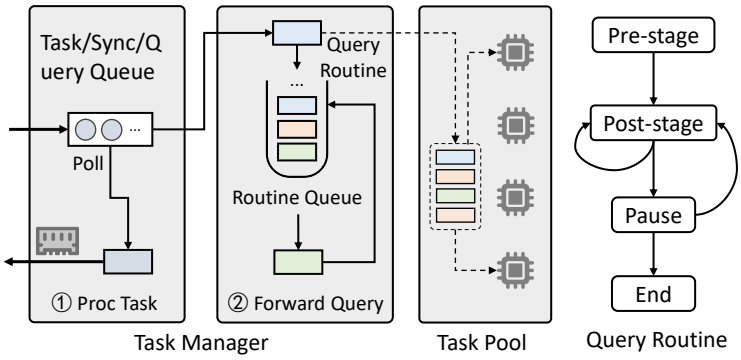


Fig. 11. Query states and task scheduling.

node's neighbors by their respective remote machines and assign metadata to neighbors that reside on the same machine. The metadata includes both the ID of the remote partition and the number of neighbors in that remote partition. Moreover, we employ fixed-size neighbor arrays for inline storage allocation, enabling direct offset calculation from node IDs. This eliminates the initial array access cache misses through predictable memory addressing.

5.2 Task Scheduling for High CPU Utilization

To improve throughput and pipeline computation and communication, CoTra processes multiple queries simultaneously. However, this may lead to increased query latency as some queries can be starved for computation. To balance query latency and throughput, CoTra uses C++ coroutines to break down each query into fine-grained operators, each representing a single hop. By scheduling these operators, CoTra avoids starvation and achieves inter-query load balancing.

Query routine. CoTra creates a query routine for each query, which is a state machine. As shown in Figure 11, each query transits through four states from start to finish:

- *Pre-Stage*: As described in Section 4.1, in this phase, the query searches the navigation index to determine its primary and secondary machines. Sub-queries are then dispatched to the primary machines. Each primary machine creates a query routine and enters the second state.
- *Post-Stage*: In this phase, the query performs local search and traversal, computes distances for candidate nodes, updates the candidate queue, and synchronizes messages with other primary machines. After processing a fixed number of nodes in the current candidate queue, it transits to the next state.
- *Pause*: In this phase, the query enters a suspending state and initiates the distributed termination algorithm. If new synchronization messages received in this period can provide new candidate nodes, the query is reactivated and returns to the post-stage. Otherwise, it waits for the termination token using the termination detection algorithm.
- *End*: When a query successfully detects termination, its local search results are sent back to the originating machine, which merges these results for final top-k vectors.

As shown in Figure 11, we manage queries using a routine queue. Each thread repeatedly performs two procedures, i.e., process remote tasks and advance query routines. When processing remote tasks, as described in Section 4.2, the thread retrieves computation tasks from the task queue and returns the results. After processing the task queue, the thread pops a query routine from the routine queue and executes one step. If the query routine remains in the post-stage state, it is

re-inserted into the routine queue for the next round. If it transits to the Pause or End state, it is not re-inserted into the routine queue.

Load balancing. Due to the dynamic nature of queries, the number of sub-queries handled by each thread may become uneven. To achieve load balancing, we employ a multi-threaded task pool to manage query computations. The tasks created by query routines are placed in a concurrent task queue shared by all threads. After each thread processing their local tasks, it check if the query routine has sub-queries and perform task stealing from the shared queue. This approach allows threads with lighter loads to balance the workload by stealing computation tasks from other threads, ensuring overall load balancing. By controlling the query batch size, the distance computations can be distributed across more threads in parallel, thereby reducing the average processing time.

5.3 Implementation Details

We implemented CoTra with 27K lines of C++ code, including distributed index building and searching. CoTra can be deployed on general RDMA clusters and supports different graph indexes.

Communication batching and pipeline. To efficiently utilize RDMA, we merge small messages (e.g., distances of vectors to the query). By reducing the number of messages, the IOPS bottleneck of NICs is tackled. To support efficient message passing and avoid excessive memory consumption, we design a one-sided write queue based on the circular buffer queue. When a machine receives a certain amount one-sided write operations, it returns the IDs of the released buffers, allowing the remote message buffers to re-enter the circular buffer queue for reuse. By employing a fully asynchronous communication and computation model, our system achieves overlap between communication and computation, avoiding CPU idling caused by network latency.

Distributed query termination. In the *Co-Search* mode, query termination is challenging. Due to the dynamic nature of query processing, a sub-query may still be activated and continue execution after receiving update messages from remote machines even after it has terminated locally. Noticing the similarity between ANNS and the Single Source Shortest Path algorithm, we implement a distributed termination detection algorithm based on Dijkstra-style 2-pass ring termination detection [42]. In particular, during query execution, each sub-query is assigned an identifier flag, and only one sub-query holds the termination token at any given time. When a sub-query completes its execution, it passes the token to the next sub-query. The sub-query that receives the token determines the token's value based on whether new computations have been performed since the last time it held the token and the state of its flag. This process is repeated by each sub-query, and if the same token circulates twice among the sub-queries, the query can be terminated safely.

Distributed index building. We build a proximity graph index for vector datasets that exceed the memory of a single machine following the replica-based partitioning approach of DiskANN [50]. The idea is to partition the dataset via K-means, send each vector to multiple partitions, and merge the local indexes of the partitions such that the graph is connected due to replicated nodes between the partitions. We divide the entire process into three phases i.e., *dispatch*, *build*, and *merge*. In the dispatch phase, each machine collects the vectors destined for every other machine (as determined by K-means on a sample of the dataset) and sends them to the target machines in bulk. Each vector is sent to the S closest machines (with $S = 2$ by default in DiskANN). Each machine independently constructs a local proximity graph on its assigned vectors. In the merge phase, the machines perform a distributed merge operation to de-duplicate the nodes and generate the final graph. CoTra defaults to using Vamana [50] as the graph index due to its good performance. However, CoTra can also construct and search with other proximity graph indexes such as HNSW [34] and

Table 1. The datasets used in the experiments.

Dataset	Num	Dim	Type	Similarity
SIFT	1B/100M	128	uint8	L2
DEEP	1B/100M	96	float32	L2
Text2Image	100M	200	float32	Inner-product
LAION	100M	512	float32	L2

NSG [15]. Our system currently does not deploy index update, but it is relatively easy to integrate update algorithms into our system if needed, as most existing update algorithms for graph-based indexes are based on search results [60].

Using regular network. As CoTra mostly communicates distances instead of raw vectors, we observe its bandwidth consumption to be low. As such, the main disadvantage of Ethernet over RDMA is longer communication latency. This increases synchronization delays and thus degrades candidate quality. As a result, more computations may be required for each query. We leave testing and adapting CoTra to Ethernet for future work.

6 Evaluation

We conduct extensive experiments to compare CoTra with state-of-the-art distributed vector search systems, verify the scalability of CoTra, and present systematic analyses of performance gains, as well as an ablation study.

6.1 Experiment Settings

Datasets. In our experiments, we evaluate our system on four mainstream benchmark datasets [48, 51]: SIFT [31], DEEP [3], Text-to-Image [13], and LAION [44]. To comprehensively assess performance across different scales, we conduct experiments on the SIFT and DEEP datasets using both 100M and 1B vectors, while 100M subsets are used for evaluations on all four datasets. The data distribution of queries in Text2Image differs from the indexed data, and we verify CoTra's capability of handling out-of-distribution data on this dataset. We evaluate our system's search performance on high-dimensional vector datasets by utilizing image embeddings from LAION.

This experimental design enables systematic analysis of system behavior under varying data volumes and different data distribution characteristics. We summarize the dataset configuration in Table 1.

Baselines. We evaluate our system against several representative baseline implementations, including both state-of-the-art single-machine and distributed systems, as well as two RDMA-optimized approaches we developed. The selected baselines include:

- **DiskANN (Single)**[50]: State-of-the-art billion-scale graph solution run Vamana index fully in-memory for best performance. We include the pure memory implementation to demonstrate potential memory bottlenecks in single-machine configurations with multiple CPUs and large memory, highlighting the significance of scalability in distributed systems.
- **Milvus**[54]: The most popular distributed vector database employing fixed-size partitioned graph indexing, evaluated only on the 100M dataset due to substantial storage requirements (approximately required 2TB for 100M vectors, 20TB for 1B vectors[3]). The fixed-size sharding

[3]https://milvus.io/tools/sizing

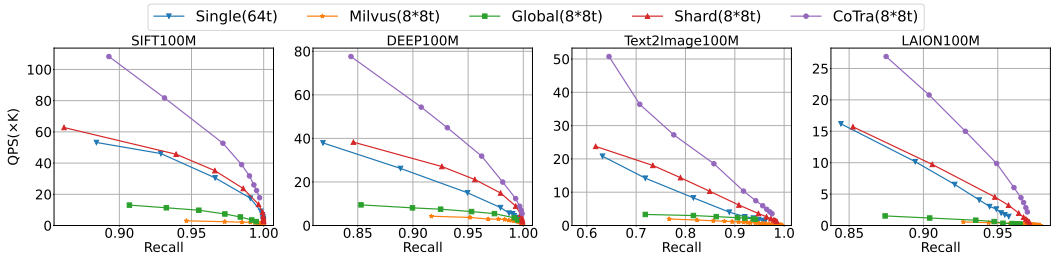


Fig. 12. QPS vs. Recall on four datasets with 100 million nodes each. The distributed baseline uses 8 machines with 8 threads per machine, while the in-memory single-machine baseline uses 64 threads.

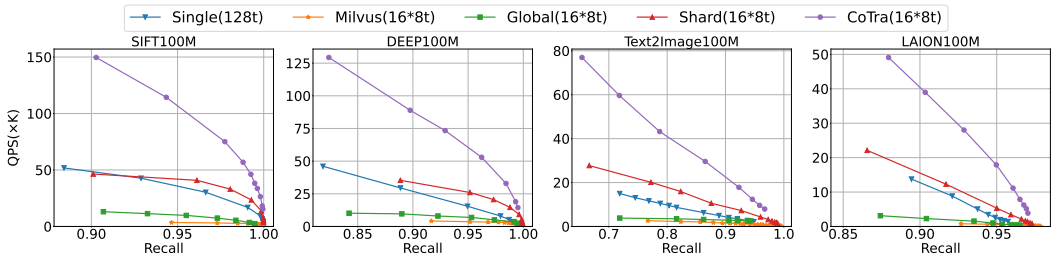


Fig. 13. QPS vs. Recall on four datasets with 100 million nodes each. The distributed baseline uses 16 machines with 8 threads per machine, while the in-memory single-machine baseline uses 128 threads.

approach leads to excessive fragmentation when handling large-scale datasets, consequently requiring the search process to traverse numerous shards and incur substantial computations.

- **Global**: RDMA-based implementation using one-sided READ operations for vector data access with K-means partitioning and holistic graph index. This approach is optimized for direct remote memory access via RDMA, eliminating the need for server-side processing during data retrieval operations.
- **Shard**: Fully independent indexing solution operating in scatter-and-gather mode, where queries are scattered across partitions for independent search and results are gathered afterward through RDMA. This design maximizes parallelism in distributed environments while maintaining complete partition independence.

The baseline graph index types are Vamana [50], while HNSW [34] is used in Milvus. For index construction, we follow the official guidelines and practical recommendations to set the parameters.

Testbed. The experimental evaluation was conducted in two settings. ① The primary testbed comprises a 16-node RDMA cluster, each node equipped with an Intel Xeon Silver 4110 processor (16 cores) and 128 GB of main memory, running on Ubuntu 22.04.4 LTS. The cluster employs 56-Gbps Mellanox MT27500 ConnectX-3 IB RNIC connected to a 56-Gbps InfiniBand switch. ② Another server is a single machine equipped with four Intel Xeon Gold 6252N processors with a total of 768GB of memory, where each NUMA node contains 24 physical cores with 48 logical threads. This server is used to evaluate the in-memory solution, which scales with the number of threads on a single machine.

We use setup ② as our reference point for scalability evaluation, as preliminary testing on smaller-scale datasets confirmed that its performance is comparable to that of individual nodes in

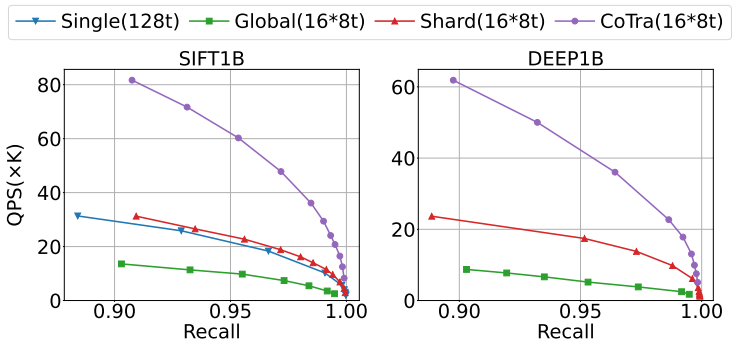


Fig. 14. QPS vs. Recall on two 1-billion-node datasets: distributed (16 machines × 8 threads) vs. single-machine (128 threads). The single-machine baseline omits DEEP due to memory constraints.

setup ①. This validation ensures that meaningful relative performance measurements can be made between the single-machine baseline and distributed configurations. All experimental results are reported as the average of five runs to ensure statistical reliability.

Performance metrics. Following prior works [7, 15, 34, 48, 50], we measure system throughput using queries per second (QPS) that reach a target recall and adopt recall@k (with k=10) as the default accuracy metric, which represents the ratio of search results that are in the ground truth top-k neighbors. We also observed similar performance trends when setting k = 1 and 100 in the micro study.

6.2 Search Performance and Scalability

Performance on 100 million-scale datasets. CoTra demonstrates exceptional scalability and search performance across both 8- and 16-machine configurations. As shown in Figure 12, CoTra achieves a 1.52–2.2× improvement over the best baseline (*Shard*) on 8 machines, across four datasets (recall@10≥0.95). This performance boost is primarily due to enhanced computational efficiency resulting from the integrity of the graph index and fine-grained execution modes (Section 4.2).

Figure 13 illustrates the QPS-recall curves for four 100M datasets on 16 machines. The throughput gains are more substantial compared to the 8-machine setup. Specifically, CoTra performs well on the 512-dimensional LAION dataset, outperforming the best baseline by 3.58× (recall@10 ≥ 0.95), and maintains robust performance on the Text2Image dataset, which exhibits distinct recall-QPS characteristics due to out-of-distribution effects, achieving a 2.23× improvement (recall@10 ≥ 0.95). Improvements of 2.12–2.86× are also observed on the SIFT and DEEP datasets. In contrast, both *Single* and *Global* index configurations fail to scale with increasing computational resources. This is because larger vector sizes exacerbate memory bandwidth bottlenecks in single-machine systems and increase communication overhead in distributed environments.

Across all datasets, the baseline implementation of Shard consistently outperforms Milvus, and CoTra achieves throughput improvements ranging from 8.7× to 33.3× compared to Milvus for recall@10≥0.95 on 8- and 16-machine configurations. This is due to Milvus's fixed-shard partitioning strategy, which differs from our machine-count sharding in the Shard index, resulting in more index shards (referred to as segments in Milvus). As the dataset size increases, more shards are introduced, leading to greater computational redundancy and lower throughput in Milvus.

Performance on 1 billion-scale vector datasets. CoTra also delivers superior performance on billion-scale vector datasets. Figure 14 presents the query throughput (QPS) versus recall on

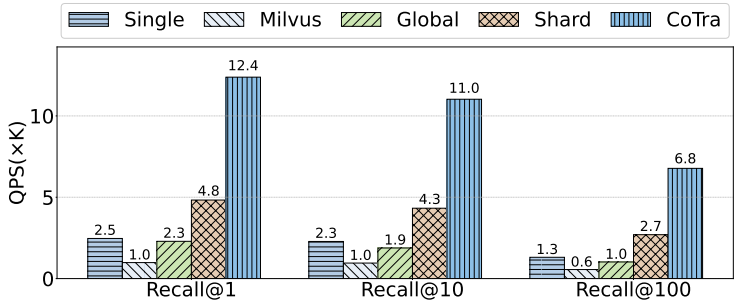


Fig. 15. Throughput at 0.95 recall for different top-k on Text2Image100M dataset: distributed baseline (16 machines × 8 threads) vs. single-machine baseline (128 threads).

Table 2. The throughput (QPS) at 0.95 recall and speedup over the single machine (8 threads) for each dataset and solution.

	8 Machine (64 Threads for Single)			
System	SIFT	DEEP	Text2Image	LAION
Single	30.5K (3.4×)	15.0K (3.7×)	1.8K (1.5×)	2.7K (1.7×)
Milvus	3.0K (0.3×)	3.7K (0.9×)	0.7K (0.6×)	0.4K (0.3×)
Global	6.3K (0.7×)	5.9K (1.4×)	1.4K (1.2×)	1.0K (0.6×)
Shard	35.1K (3.9×)	21.1K (5.1×)	3.3K (2.9×)	4.5K (2.8×)
CoTra	68.9K (7.6×)	32.1K (7.6×)	7.2K (6.2×)	9.9K (6.2×)

	16 Machine (128 Threads for Single)			
System	SIFT	DEEP	Text2Image	LAION
Single	30.3K (3.4×)	15.5K (3.8×)	2.1K (2.1×)	3.1K (1.9×)
Milvus	3.5K (0.4×)	3.6K (0.9×)	0.9K (0.9×)	0.6K (0.4×)
Global	9.7K (1.1×)	7.4K (1.8×)	1.9K (1.9×)	1.3K (0.8×)
Shard	40.8K (4.5×)	26.0K (6.5×)	4.3K (4.4×)	5.0K (3.1×)
CoTra	116.6K (12.7×)	55.3K (13.4×)	9.6K (9.8×)	17.9K (11.2×)

the SIFT1B and DEEP1B datasets using 16 machines. The performance patterns observed on the SIFT and DEEP datasets at the 100M scale are similar to those at the 1B scale, validating the system's scalability. Compared to *Shard* and *Global* approaches, our system achieves 2.2- 2.8× higher throughput when recall ≥ 0.9. The performance gains are pronounced in all recall regimes, demonstrating CoTra's effectiveness for quality-sensitive applications.

The consistent speedup observed across all datasets and system scales further validates the generality of CoTra.

Effect of varying k. To verify that CoTra is capable of handling different values of k in recall@k, we conduct experiments on the Text2Image dataset, which represents a challenging search scenario. Figure 15 shows the throughput performance for $k = 1, 10, 100$, where CoTra outperforms the best

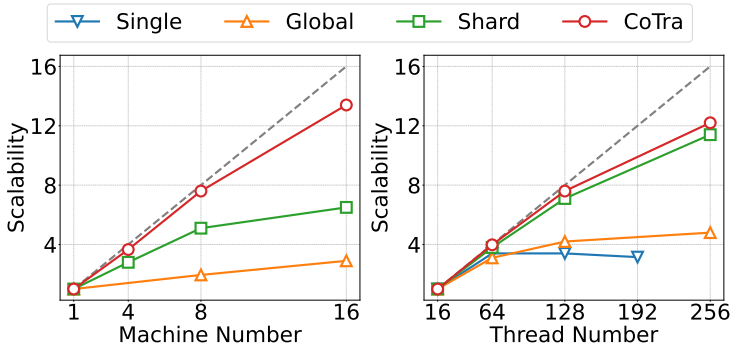


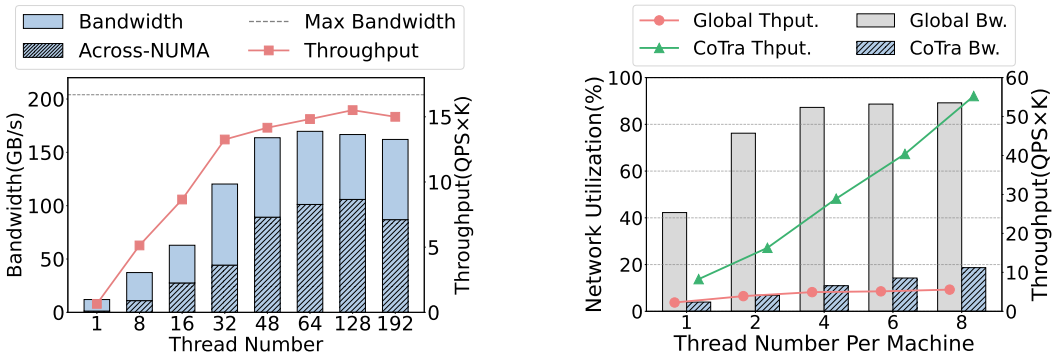
Fig. 16. Scalability over machine numbers (left) and thread numbers (right) on DEEP100M. The gray dashed line shows linear scaling. In the right figure, thread numbers in the distributed setting denote the total threads.

baseline by 2.51–2.58×. Although not shown in the paper due to space limitations, similar patterns are observed across other datasets. These results confirm CoTra's consistent performance under varying retrieval requirements.

Scalability. To demonstrate the scalability of CoTra, we present performance gains in Table 2, where the reported QPS improvements are relative to the in-memory Single baseline with 8 threads. CoTra achieves outstanding scalability, with 6.2- 7.6× improvements on 8-machines and 9.8- 13.4× QPS improvement over the single-machine system.

We also compare the scalability of the system under different cluster sizes of 4, 8, and 16 machines on the same dataset. As shown in Figure 16 (left), CoTra demonstrates good scalability as the number of machines increases, with each machine running the same number of threads by leveraging the proposed techniques. In contrast, the single-machine system hits a clear bottleneck beyond 32 threads due to memory bandwidth limitations. Compared to the Shard index, CoTra outperforms it and requires significantly fewer node accesses and computations per query to achieve equivalent recall levels. A naive application of the global graph index (*Global*), even with RDMA, does not deliver ideal performance because of intensive communication. This efficiency of CoTra stems from the global graph index, which effectively preserves the routing search path across the entire dataset, even in distributed settings, and enhances query traversal, as well as the design of collaborative traversal, which eliminates the redundant computations seen in partitioned approaches (e.g., Shard and Milvus) and reduces communications. We also evaluated the system's scalability under different thread counts. As shown in Figure 16 (right), the single-machine setup encounters a bottleneck beyond 64 threads due to memory bandwidth and NUMA effects. The global-index approach experiences performance degradation after 4 threads per machine (64 threads in total) because of network bandwidth limitations. Both CoTra and Shard achieve good scalability, although a slight drop is observed when increasing the thread count from 128 to 256, primarily due to NUMA effects.

We further analyze the bandwidth bottleneck of the single-machine memory system. As shown in Figure 17a, while both bandwidth constraints and cross-NUMA memory access affect system scalability, the primary bottleneck remains the memory bandwidth capacity. This dual constraint motivates distributed design that addresses capacity through horizontal scaling and bandwidth limitations through distributed communication. As shown in Figure 17b, the global-index-based approach not only suffers from significant latency issues but also experiences limited scalability due to network bandwidth constraints.



(a) Memory bound.

(b) Communication bound.

Fig. 17. Single-machine and global-index bottleneck profiling. (a) Throughput and memory bandwidth vs. thread count on the single machine. (b) Throughput and network bandwidth vs. thread count.

Table 3. Efficiency profiling of different solutions on SIFT100M. *Avg. Comp.*, *Comm. Ratio*, and *Throughput* represent the average query computation count, communication time ratio, and QPS at 0.95 recall, respectively.

Solution	Avg. Comp.	Comm. Ratio	Throughput
Single	3.59K	0.00%	30.3K
Global	3.73K	89.4%	9.7K
Shard	15.6K	<1.00%	40.8K
CoTra	4.33K	22.5%	116.6K

6.3 Micro Experiments and Ablation Study

Search costs breakdown. To understand CoTra's gains, we evaluated the average query computation count (pruning effect) and the proportion of communication time in total execution time (communication efficiency) for different baselines on the SIFT100M dataset across 16 machines.

As shown in Table 3, the single-machine system exhibits the lowest average computation count and communication overhead. However, due to memory bandwidth limitations, its throughput is severely constrained when the number of threads increases. The *Global* approach achieves the second-highest pruning effect because its search logic closely resembles that of the single-machine system. The *Shard* suffers from increased computational operations due to partitioned index construction and search, resulting in around 4× redundant computations and thus limited throughput. In contrast, CoTra maintains relatively high pruning effect (only ~20% redundant computation introduced by delayed updates) while achieving low communication overhead (~22% communication time proportion), ultimately delivering the highest throughput.

Index construction time. The distributed parallel graph construction in CoTra greatly accelerates index building compared to a single machine. We compare the index construction time of different approaches. As shown in Table 4, constructing the LAION100M dataset drops from over 2 days to just 9 hours with 16 nodes. The disk-based setup [50] requires a longer index construction time, while the above search evaluations are performed in-memory. The Shard approach achieves the shortest index construction time, as it does not require data replication or partitioning. In contrast,

Table 4. Index construction time (in hours) for each dataset, where Shard, Milvus, and CoTra use 16 machines, while Single uses 1 machine.

Dataset	Single	Shard	Milvus	CoTra
SIFT100M	10.0	0.8	2.0	1.5
DEEP100M	15.0	1.0	3.5	2.0
T2I100M	25.0	2.0	5.8	4.5
LAION100M	64.0	5.5	12.0	9.0

Table 5. Impact of query batch size on the SIFT100M dataset.

Batch size	1	2	4	8	12
Recall = 0.90	75.2K	124.7K	149.6K	152.2K	151.6K
Recall = 0.95	54.6K	78.4K	114.4K	116.3K	116.9K

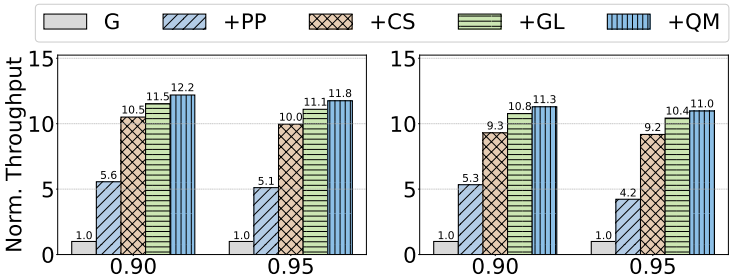


Fig. 18. Speedup over *Global* index at corresponding recall levels. **G** denotes the global index baseline without optimizations, **PP** represents Pull-Push mode, **CS** is Co-Search mode, **GL** indicates graph layout optimization, and **QM** stands for query manager optimization.

Milvus exhibits a longer overall index construction time, primarily due to its extensive partitioning process.

Batch size sensitivity analysis. In our experiments, CoTra adopts a default query batch size of 4 for the best performance. We further evaluate the impact of different batch sizes on the SIFT100M dataset. As shown in Table 5, CoTra achieves near-peak throughput with a relatively small query batch size. Specifically, increasing the batch size from 1 to 4 significantly improves performance, while larger batch sizes yield only marginal gains, indicating that a batch size of 4 is sufficient to fully utilize the system's capacity.

Ablation study. We conducted a comprehensive evaluation of the key optimizations proposed in this work, analyzing their individual contributions to throughput and latency improvements. The evaluated components include: Pull-Push mode (+PP), Co-Search mode (+CS), graph layout optimization (+GL), and query manager (+QM).

As shown in Figure 18. The Pull-Push mode eliminates bulk vector transfers, the bandwidth bottleneck is significantly alleviated, leading to a substantial throughput improvement. The Co-Search mode relaxes query data dependencies, eliminating the need for passive result synchronization. This allows small batch sizes to achieve comparable QPS while drastically improving overall throughput.

The optimized graph layout, combined with our Pull-Push communication mode, achieves further reduction in communication overhead. Finally, the query manager combined with other system optimizations refines computation scheduling through fine-grained workload partitioning and balancing, further improving both latency and throughput in multi-threaded scenarios.

7 Related Work

In-memory systems for vector search. Vector index researches mainly consider the single-machine and in-memory setting [2, 15, 24, 34]. Two types of indexes are the most popular, i.e., *proximity graph* that connects similar vectors to form a graph [34], and *IVF* that partitions the vectors into clusters [24]. Proximity graph requires fewer distance computations than IVF to reach the same recall [34, 57], and thus our CoTra utilizes the proximity graph index to enjoy its high efficiency. Many systems also optimize for in-memory query processing [6, 35, 39]. For instance, graph reordering is used to reduce cache miss caused by the random vector access of proximity graph in [9], while cache miss is tackled with software prefetch in [1]. Both ParlayANN [35] and FAISS [23] provide efficient implementations (e.g., SIMD) for several vector indexes. There are also many vector quantization algorithms [1, 16, 17, 24] that can compress each vector into a smaller vector and accelerate distance computation. Focusing on distributed coordination, our CoTra is orthogonal to these single-machine engine and vector quantization researches and can integrate them to improve performance.

Hybrid systems for vector search. Several systems use disk to store the large vector datasets [43, 52]. For instance, DiskANN [50] keeps the compressed vectors in memory and the exact vectors and proximity graph index on disk. Starling [55] reduces the disk read amplification of DiskANN with graph reordering. SPANN uses the IVF index by keeping the cluster centers in memory and the vectors on disk [7]. These disk-based systems are cost-effective, but throughput is limited by disk bandwidth. Some systems use CPU-GPU joint processing to benefit from the strong computational power of GPUs [37]. For instance, Rummy uses the IVF index and moves clusters from CPU to GPU on demand for computation [65], while Bang uses the proximity graph index and moves the adjacency lists from CPU to GPU to compute approximate distances [28]. For these systems, small GPU memory and PCIe bandwidth are major limiting factors.

Distributed systems for vector search. Most systems conduct distributed vector search by building an independent index on each machine, e.g., Milvus [54], AnalyticDB [58], Weaviate [12], Cassandra [30], and Pyramid [11]. d-HNSW [33] is a distributed system built on a memory disaggregation architecture, but it resembles Pyramid in overall design. In particular, d-HNSW also maintains an independent shard of the dataset on each server and uses a meta-index to search for the potential shards of each query. This simplifies system design but degrades pruning effect, as shown by our profiling. Some systems conduct distributed vector using the IVF index, e.g., Auncel [64] and SPFresh [61]. The benefit is that each machine only needs to scan its local clusters after the top-ranking clusters have been decided. However, the pruning effect of the IVF index is lower than that of the proximity graph. Some systems use new hardware to scale up vector search, e.g., CXL-ANNS [20] adopts the Compute Express Link (CXL) as a memory pool, and DF-GAS [62] utilizes FPGAs for distance computation. To our knowledge, CoTra is the first to conduct distributed vector using a holistic proximity graph index over the entire dataset. This enjoys high pruning effect but requires careful algorithm-system co-designs to handle the vast communications induced by the fine-grained data dependencies of the proximity graph index.

8 Conclusions

We present CoTra, a system that scales up vector search over multiple machines. We find that using a holistic proximity graph index for the entire dataset enjoys high computation efficiency but suffers from the extensive communication caused by rich data dependencies. To reduce the communication cost, we conduct algorithm system co-designs by observing that the efficiency of proximity graph search can be maintained if the quality of the candidates are good. We also incorporate system optimizations including task scheduling, RDMA-friendly graph format optimization, and distributed index building. To our knowledge, CoTra is the first system that achieves close to linear throughput scaling compared to proximity graph on a single machine.

References

[1] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. 2023. Similarity Search in the Blink of an Eye with Compressed Indices. *Proc. VLDB Endow.* 16, 11 (July 2023), 3433–3446. doi:10.14778/3611479.3611537

[2] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence* 37, 6 (2014), 1247–1260.

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

[4] Wout Bittremieux, Pieter Meysman, William Stafford Noble, and Kris Laukens. 2018. Fast open modification spectral library searching through approximate nearest neighbor indexing. *Journal of proteome research* 17, 10 (2018), 3463–3474.

[5] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *arXiv preprint arXiv:2402.03216* (2024).

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

[7] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 5199–5212. https://proceedings.neurips.cc/paper_files/paper/2021/file/299dc35e747eb77177d9cea10a802da2-Paper.pdf

[8] Yiqun Chen and James Zou. 2024. GenePT: a simple but effective foundation model for genes and cells built from ChatGPT. *bioRxiv* (2024), 2023–10.

[9] Benjamin Coleman, Santiago Segarra, Alexander J Smola, and Anshumali Shrivastava. 2022. Graph Reordering for Cache-Efficient Near Neighbor Search. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 38488–38500. https://proceedings.neurips.cc/paper_files/paper/2022/file/fb44a668c2d4bc984e9d6ca261262cbb-Paper-Conference.pdf

[10] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[11] Shiyuan Deng, Xiao Yan, K.W. Ng Kelvin, Chenyu Jiang, and James Cheng. 2019. Pyramid: A General Framework for Distributed Similarity Search on Large-scale Datasets. In *2019 IEEE International Conference on Big Data (Big Data)*. 1066–1071. doi:10.1109/BigData47090.2019.9006219

[12] Etienne Dilocker, Bob van Luijt, Byron Voorbach, Mohd Shukri Hasan, Abdel Rodriguez, Dirk Alexander Kulawiak, Marcin Antas, and Parker Duckworth. [n. d.]. *Weaviate.* https://github.com/weaviate/weaviate

[13] Artem Babenko Dmitry Baranchuk. 2021. Text-to-Image dataset for billion-scale similarity search. Retrieved April 13, 2025 from https://research.yandex.com/datasets/text-to-image-dataset-for-billion-scale-similarity-search

[14] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Low-latency communication for fast DBMS using RDMA and shared memory. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1477–1488.

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

[16] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 3, Article 167 (May 2024), 27 pages. doi:10.1145/3654970

[17] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 4 (2014), 744–755. doi:10.1109/TPAMI.2013.240

[18] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based retrieval in facebook search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2553–2561.

[19] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* (Dallas, Texas, USA) *(STOC '98)*. Association for Computing Machinery, New York, NY, USA, 604–613. doi:10.1145/276698.276876

[20] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 585–600. https://www.usenix.org/conference/atc23/presentation/jang

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

[22] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[23] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547. doi:10.1109/TBDATA.2019.2921572

[24] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128. doi:10.1109/TPAMI.2010.57

[25] Yannis Kalantidis and Yannis Avrithis. 2014. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2321–2328.

[26] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 295–306.

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[28] Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, Jyothi Vedurada, et al. 2024. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. *arXiv preprint arXiv:2401.11324* (2024).

[29] Vladimir Yu Kiselev, Andrew Yiu, and Martin Hemberg. 2018. scmap: projection of single-cell RNA-seq data across data sets. *Nature methods* 15, 5 (2018), 359–362.

[30] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. doi:10.1145/1773912.1773922

[31] Hervé Jégou Laurent Amsaleg. 2010. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/ Accessed: 2025-04-17.

[32] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2019), 1475–1488.

[33] Yi Liu, Fei Fang, and Chen Qian. 2025. Efficient Vector Search on Disaggregated Memory with d-HNSW. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '25)*. 1–8.

[34] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. doi:10.1109/TPAMI.2018.2889473

[35] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 270–285. doi:10.1145/3627535.3638475

[36] Priyanka Nigam, Yiwei Song, Vijai Mohan, Vihan Lakshman, Weitian Ding, Ankit Shingavi, Choon Hui Teo, Hao Gu, and Bing Yin. 2019. Semantic product search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2876–2885.

[37] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2024. Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4236–4247.

[38] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient approximate nearest neighbor search in multi-dimensional databases. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[39] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. 2023. 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* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 313–328. doi:10.1145/3572848.3577527

[40] Liudmila Prokhorenkova and Aleksandr Shekhovtsov. 2020. Graph-based nearest neighbor search: From practice to theory. In *International Conference on Machine Learning*. PMLR, 7803–7813.

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

[42] Michel Raynal. 2013. *Distributed Termination Detection*. Springer Berlin Heidelberg, Berlin, Heidelberg, 367–399. doi:10.1007/978-3-642-38123-2_14

[43] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: efficient billion-point nearest neighbor search on heterogeneous memory. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 895, 13 pages.

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

[45] Konstantin Schütze, Michael Heinzinger, Martin Steinegger, and Burkhard Rost. 2022. Nearest neighbor search on embeddings rapidly identifies distant protein relations. *Frontiers in Bioinformatics* 2 (2022), 1033775.

[46] Rulin Shao, Jacqueline He, Akari Asai, Weijia Shi, Tim Dettmers, Sewon Min, Luke Zettlemoyer, and Pang Wei W Koh. 2024. Scaling retrieval-based language models with a trillion-token datastore. *Advances in Neural Information Processing Systems* 37 (2024), 91260–91299.

[47] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 317–332.

[48] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. 2022. Results of the NeurIPS'21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track*. PMLR, 177–189.

[49] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. 2024. Vexless: A Serverless Vector Data Management System Using Cloud Functions. *Proc. ACM Manag. Data* 2, 3, Article 187 (May 2024), 26 pages. doi:10.1145/3654990

[50] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. 2019. *DiskANN: fast accurate billion-point nearest neighbor search on a single node*. Curran Associates Inc., Red Hook, NY, USA.

[51] Eric S Tellez, Martin Aumüller, and Vladimir Mic. 2024. Overview of the SISAP 2024 Indexing Challenge. In *International Conference on Similarity Search and Applications*. Springer, 255–265.

[52] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. 2024. Scalable Billion-point Approximate Nearest Neighbor Search Using SmartSSDs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 1135–1150. https://www.usenix.org/conference/atc24/presentation/tian

[53] Christophe Van Gysel, Maarten de Rijke, and Evangelos Kanoulas. 2016. Learning latent vector spaces for product search. In *Proceedings of the 25th ACM international on conference on information and knowledge management*. 165–174.

[54] 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. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2614–2627. doi:10.1145/3448016.3457550

[55] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proc. ACM Manag. Data* 2, 1, Article 14 (March 2024), 27 pages. doi:10.1145/3639269

[56] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631* (2021).

[57] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.* 14, 11 (July 2021), 1964–1978. doi:10.14778/3476249.3476255

[58] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: a hybrid analytical engine towards query fusion for structured and unstructured data. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3152–3165. doi:10.14778/3415478.3415541

[59] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 233–251. https://www.usenix.org/conference/osdi18/presentation/wei

[60] Haike Xu, Magdalen Dobson Manohar, Philip A. Bernstein, Badrish Chandramouli, Richard Wen, and Harsha Vardhan Simhadri. 2025. In-Place Updates of a Graph Index for Streaming Approximate Nearest Neighbor Search. arXiv:2502.13826 [cs.IR] https://arxiv.org/abs/2502.13826

[61] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 545–561. doi:10.1145/3600006.3613166

[62] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, and Yu Wang. 2023. 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* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 283–296. doi:10.1145/3613424.3614292

[63] Yanhao Zhang, Pan Pan, Yun Zheng, Kang Zhao, Yingya Zhang, Xiaofeng Ren, and Rong Jin. 2018. Visual search at alibaba. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 993–1001.

[64] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. 2023. Fast, Approximate Vector Queries on Very Large Unstructured Datasets. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 995–1011. https://www.usenix.org/conference/nsdi23/presentation/zhang-zili

[65] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Vector Query Processing for Large Datasets Beyond GPU Memory with Reordered Pipelining. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 23–40. https://www.usenix.org/conference/nsdi24/presentation/zhang-zili-pipelining

[66] Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, Jie Jiang, and Bin Cui. 2024. Retrieval-augmented generation for ai-generated content: A survey. *arXiv preprint arXiv:2402.19473* (2024).

[67] Yifan Zhao, Huiyu Cai, Zuobai Zhang, Jian Tang, and Yue Li. 2021. Learning interpretable cellular and gene signature embeddings from single-cell transcriptomic data. *Nature communications* 12, 1 (2021), 5261.

[68] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA. 1, 2, Article 131 (June 2023), 26 pages. doi:10.1145/3589276

Received July 2025; revised October 2025; accepted November 2025