# Exploiting Modern Hardware Architectures for High-Dimensional Vector Search at Speed and Scale

Minjia Zhang\*, Jie Ren‡\*, Zhen Peng†, Ruoming Jin‡, Dong Li†\*, Bin Ren‡\* \*Microsoft, minjiaz@microsoft.com †Pacific Northwest National Laboratory ‡Kent State University †\*University of California, Merced ‡\*College of William & Mary

#### Abstract

The field of vector search has seen a surge in interest from both researchers and practitioners due to its potential in emerging AI applications. Understanding how to optimize its performance is crucial for numerous tasks, but there remain a lot of challenges in practice. The advent of new hardware architectures and platforms has prompted a reevaluation of the design of large-scale vector search systems. However, current state-of-the-art vector search algorithms have not fully leveraged new hardware architectures to maximize performance.

In this study, we propose design strategies to enhance the computational and memory efficiency of largescale vector search. Our novel search algorithm, <u>iQAN</u>, delivers up to an order of magnitude faster search speeds on multi-core architectures through efficient intra-query parallelism, effectively utilizing the combined computational power of modern multi-core chips. Our new design, HM-ANN, employs a novel form of index that effectively leverages heterogeneous memory, enabling billion-scale vector search at a low cost. This paper delves into the challenges and algorithms associated with <u>iQAN</u> and HM-ANN, with a focus on improvements in computational and memory efficiency. The paper also includes the results of our experiments that demonstrate the outstanding performance of vector search when modern hardware architectures are effectively utilized through our proposed methods. Lastly, the paper explores open questions and future directions for supporting high-dimensional vector search with speed and scale.

# **1** Introduction

#### **1.1** Approximate Nearest Neighbor Search

Finding the top-k nearest neighbors among database vectors for a query has long been a key building block to solve problems such as large-scale information retrieval and image search [33, 38, 49], recommendation [17], entity resolution [27], and sequence matching [10]. As database size and vector dimensionality increase, exact nearest neighbor search becomes expensive and impractical due to latency and memory constraints [11, 12, 62]. Therefore, to reduce the search cost, various approximate nearest neighbor search (ANNS) algorithms have been proposed to improve efficiency substantially while mildly relaxing accuracy constraints, leading to the so-called accuracy-vs-efficiency tradeoffs.

Over the years, a variety of algorithms for Approximate Nearest Neighbor Search (ANNS) have been developed with the goal of enhancing computational and memory efficiency. To improve the compute efficiency, well-designed indexes have been introduced, including tree structure-based [8, 9, 44], hashing-based [26], and proximity graph-based approaches [24, 40]. To improve memory efficiency, various compression algorithms have also been applied to ANNS, such as product quantization-based methods [25, 31, 32, 36, 46]. These methods

can also be combined to improve both compute and memory efficiency simultaneously. For a more detailed understanding and comparison of ANNS algorithms, we recommend several literature that provide excellent surveys [3, 37, 58].

#### **1.2 Modern Applications and Requirements**

ANNS holds significant relevance in contemporary applications, particularly in conjunction with deep learning models, as it facilitates innovative search scenarios. Traditional entity retrieval relies on keyword matching and user behavior signals. However, with the progression of deep learning, it is now possible to construct models that yield vectors with close distances for entity inputs sharing similar "views". With these models, one then can encode unstructured data into embedding vectors in a high dimensional space  $\mathbb{R}^d$  [28, 65]. These vectors capture the similarities between various entities within the latent space. As a result, the nearest embeddings for a specific query often symbolize entities with similar semantics in the latent space. ANNS then emerges as a natural choice for managing these vectors while ensuring both speed and accuracy in retrieval.

Vector-based search has already been integrated into many modern applications. For instance, Web-scale search engines like Google [56] and Bing [14, 55] utilize embeddings for documents (e.g., word2vec [43] and doc2vec [34]) and images (e.g., VGG [54]) to retrieve semantically related entities in response to user queries. Major e-commerce players like Amazon [45] have developed recommendation systems that embed both the product catalog and the search query, recommending products whose embeddings are closest to the embedded search query. YouTube has built search engine that embeds videos to vectors for video recommendation [16]. More recently, vector search has been employed in retrieval augmented generation in large language models (LLMs), where vector search can be used to expand LLMs knowledge by incorporating external data sources [13]. Vector search also presents a fertile ground for exploring future applications. For instance, recent advancements in deep learning have enabled models to capture multimodal relationships, such as through the use of multimodal foundation models [21]. Consequently, the underlying vector search systems can also leverage ANNS to handle multi-modality entities. However, how to effectively handle different modalities and capture the full range of interconnections and relationships among them via ANNS remains an open question. This includes whether various modalities benefit from using the same or different vector search methods, which is an exciting area for future exploration.

As vector search goes to a larger scale, where the dimension scales from  $\sim 100$  to  $\sim 1000$  and the number of vectors scales from millions to billions, the challenge of serving latency becomes more prominent even with novel ANNS algorithms. For instance, online interactive services (e.g., web search engine) often require responses to be returned within a few or tens of milliseconds, as delayed responses could degrade user satisfaction and affect revenue [22]. However, as the number of entities (such as images and documents) grows rapidly and deep learning embeddings expand to higher dimensions (from embedding sentences to full documents), it becomes increasingly difficult to find highly accurate results in large datasets while adhering to latency constraints. Many vector search services, such as text and image search, require intensive computation and may not be feasible due to latency violations. Therefore, how to transform these applications from impossible to ship due to latency violation to well-fitting SLA is crucial for the practical adoption of vector search. Another big requirement for large-scale vector search is cost reduction. Large-scale services deal with a vast volume of requests and could necessitate thousands of machines for a single application. Therefore, decreasing the number of machines while maintaining the same search quality and latency is crucial for reducing the total cost of ownership for the application.

## 1.3 New Hardware Architectures and Opportunities

Existing ANN algorithms have mostly exploited the uni-core CPU infrastructure and standard memory hierarchy. This infrastructure uses processors whose performance increased with Moore's Law, thus limiting the need for high levels of concurrent execution on a single machine. However, processors are no longer providing ever

higher uni-core performance. Meanwhile, the prior infrastructure used DRAM for main memory. However, the main memory capacity is often quite limited to hold a large volume of data. As multi-core processors become ubiquitous and new memory architecture such as heterogeneous memory becomes available, new opportunities for large-scale vector search exist:

- **Design for multi-core**: Modern CPUs are often equipped with high-performance multi-core. Since uni-core speed has pretty much saturated, we need to get better at exploiting a large number of cores by addressing at least two important aspects:
  - 1. Multi-core CPUs provide high concurrency, but as the level of concurrency increases, synchronization among different cores are more likely to block and limit scalability.
  - 2. The performance of multi-core processes also depend on the shared memory bandwidth utilization. According to the roofline model [64], the performance of an application is not only bounded by the compute capability but also the bandwidth performance. So how to make the best utilization of memory bandwidth needs great care.
- Design for modern memory devices: Vector search at large scale is very memory consuming and easily runs out of memory with a few hundred millions of vectors. When the dataset becomes too large to fit on a single machine, one approach is to use the compressed representations of the database points, such as Hamming codes [47] and product quantization [19, 25, 31, 32, 46]. However, the performance of these methods deteriorates rapidly at higher recall targets, because they calculate approximate distance based on compressed vectors instead of on the original data vectors. Another approach is to exploit storage. In DiskANN [55], the authors explore slow storage to achieve billion-scale ANNS in a single machine. However, disk latency is a major problem. While persistent media such as SSD offers lower latency and much higher I/O ops per second than traditional disks, they are still several orders of magnitude slower than DRAM. Based on this assumption, data access to the persistent media during search should be minimized. As a result, DiskANN maintains a copy of compressed data in memory with product quantization [55], which results in loss of in-memory search quality. It then performs a re-ranking using full-precision coordinates stored on SSD, using block-level data accesses but with expensive SSD accessing time. While methods such as DiskANN show promising results, the emergence of Heterogeneous Memory (HM) brings opportunities to significantly improve ANNS. HM combines cheap, slow but extremely large memory with expensive, fast but small memory (e.g., traditional DRAM) to achieve a good balance between production cost, memory performance and capacity. Because of the large memory capacity, HM can use full-precision vectors with accurate distance computation. Since memory access latency/bandwidth of slow memory components in HM is much faster than slow storage such as SSD, it is possible to occasionally access data in slow memory during search without paying the expensive cost of data accesses. That being said, realizing the full performance potential of HM for ANNS is still quite challenging. Although slow memory such as PMM performs  $\sim$ 80X times faster than SSD, it is still  $\sim$ 3X slower than DRAM in terms of random access latency [60]. Therefore, a naive data placement strategy can hurt the search efficiency badly. Therefore, one may still wonder if we can leverage HM for ANNS to achieve both high search accuracy and low search latency, especially when the dataset cannot fit in DRAM (fast memory)?

In this work, we revisit the similarity search problem in light of the recent advances in the field. Two new system optimization methods are introduced, dedicated to improving the efficiency and scaling of vector search while simultaneously delivering high accuracy. They are particularly appropriate for the new hardware architectures discussed above. Specially:

• iQAN [48] is a parallel search algorithm that exploits intra-query parallelism in graph-based vector search to obtain significant latency reduction in vector search on multi-core architectures with high accuracy.

This approach includes a set of optimizations that boost convergence, avoid redundant computations, and mitigate synchronization overhead.

 HM-ANN [51] is a heterogeneous memory-based technique that shatters the memory barrier of deploying large-scale vector search via NVMe memory, enabling billion-scale vector search with low deployment cost. It carefully constructs vector search indices via a memory hierarchy-aware algorithm, hence leading to substantially better search performance as the vectors grow to be larger than the DRAM capacity. It also employs parallel search algorithms to boost the in-memory search efficiency, leading to faster search speed.

In drawing broader lessons from this work, we believe that effectively leveraging multi-core and exploiting the memory hierarchy are the keys to high-performance vector search on modern processors. Further, based on the above methods, we discuss open research problems, including exploring hierarchical parallelism to meet both latency and throughput targets, highly concurrent vector search with addition and deletion, automating the index construction for vector search, and the interactions with modern applications in Section 5.

## 2 Background

The literature on nearest neighbor search is vast, and hence, we focus our attention on the most relevant works here. There has been a lot of work on building effective ANN indices to accelerate the search process. Earlier works focus on space partitioning-based methods. For example, Tree-based methods (e.g., KD-tree [53] and R\* tree [8]) hierarchically split the data space into lots of regions that correspond to the leaves of a tree structure and only search a limited number of promising regions. However, the complexity of these methods becomes no more efficient than brute-force search as the dimension becomes large (e.g., >16) [35]. Prior works also have spent extensive efforts on locality-sensitive hashing-based methods [1, 2, 18, 29], which map data points into multiple buckets with a certain hash function such that the collision probability of nearby points is higher than the probability of others. These methods have solid theoretical foundations. LSH and its variations are often designed for large sparse vectors with hundreds of thousands of dimensions. In practice, LSH-based methods have been outperformed by other methods, such as graph-based approaches, by a large margin on large-scale datasets [4, 24, 40]. More recently, Malkov and Yashunin found graphs that satisfy the small-world property exhibit excellent navigability in finding nearest neighbors. They introduce the Hierarchical Navigable Small World (HNSW) [40], which builds a hierarchical k-NN graph with additional long-range links that help create the small-world property. For each query, it then performs a walk, which eventually converges to the nearest neighbor in logarithmic complexity. Subsequently, Fu et al. proposed NSG, which approximates Monotonic Relative Neighbor Graph (MRNG) [24] that also involves long-ranged links for enhancing connectivity.

# **3** iQAN: Fast and Accurate ANNS via Intra-Query Parallelism on Multi-Core Architecture

Among different vector search methods, the similarity graph-based algorithms have emerged as a remarkably effective class of methods for high-dimensional ANNS, outperforming other approaches on a wide range of datasets to achieve the best accuracy-vs-latency [4, 5, 20, 24, 31, 37, 59, 63]. Despite their promising results, graph-based methods still have challenges that limit their use in real-world scenarios. In particular, as the data size grows, it becomes increasingly challenging to achieve both low latency and high accuracy simultaneously. Existing solutions often resort to inter-query parallelism by dispatching queries across multiple processors or nodes to be processed simultaneously [7, 24]. This approach scales from a throughput perspective, but it does not help reduce query latency because each query still roughly performs the same amount of vector computations to find the nearest neighbors.

#### 3.1 Challenges of ANNS via Intra-Query Parallelism

Another natural idea to reduce latency is to exploit intra-query parallelism on individual nodes with multi-core processors. For example, one may parallelize the node expansion in each iteration step of the sequential search algorithm (referred to as **Node-Expansion-in-Parallel**) because distance computations within a neighborhood expansion iteration do not have dependencies, hoping that multiple worker threads can check the closeness of multiple neighbors in parallel while performing the same computations on each step as the sequential algorithm. Surprisingly, this solution performs quite poorly and may even perform much worse than a well-tuned sequential algorithm, as shown in Fig. 1. There are several challenges in scaling ANNS with intra-query parallelism:

**Challenge 1: Modern multi-core hardware is sensitive to synchronization overhead.** Parallelism boosts compute capacity but may also incur high synchronization overhead, especially if there are complex data dependencies. While parallelizing the distance computation, Node-Expansion-in-Parallel also requires synchronization in between expansion iterations to sort the distance order of all candidates discovered by multiple parallel workers according to their distances to the query point, to decide which node to expand in the next iteration. We have observed that the synchronization is very expensive on a multi-core architecture, and frequent sequential-to-parallel synchronization as in Node-Expansion-in-Parallel can significantly prolong the search process. Fig. 2 shows that as we increase the number of threads, the synchronization overhead accounts for more than 50% of the total search time, becoming a dominating factor in the overall search latency.









Figure 1: EP's latency on Deep100M.





Figure 4: Iteration depths change along with data sizes.

**Challenge 2: Node-Expansion-in-Parallel leads to insufficient computation granularity per worker, leading to sub-optimal memory bandwidth utilization.** Node-Expansion-in-Parallel has low compute intensity because (1) unlike matrix multiplication, the point-wise Euclidean distance computation is an operator with low compute intensity, and (2) the number of neighbors to be expanded in one step is limited, given that similarity graphs naturally have low out-degree to avoid the <u>out-degree explosion problem</u> [24]. As such, further dividing the distance computation within each neighbor expansion iteration leads to insufficient work for each worker.

**Challenge 3: Vector search using graph traversal requires many iterations to converge, resulting in long sequential dependencies between iterations and thus limiting its scalability.** The number of neighborhood expansion iterations depends on the recall target and the graph size. For example, Fig. 3 shows that as the recall target increases, the number of iterations to find the top-100 nearest neighbors on a hundred million scale dataset DEEP100M grows dramatically as the recall target becomes higher (e.g., a 34.6-time increase from 0.9 to 0.999 recall). Fig. 4 shows that as the dataset size increases, the number of iterations to find 1M-vector dataset to 100M-vector dataset). This long sequential dependency makes achieving low latency with high accuracy especially challenging.

### 3.2 Design of iQAN

To address the aforementioned challenges, we introduce  $\underline{iQAN}$ , a parallel search algorithm to accelerate graphbased ANNS on multi-core architectures with three key optimizations: (i) reducing neighbor expansion iteration









Figure 5: Iteration depths to find the *K*-th nearest neighbor (x-axis).

Figure 6: The number of steps for a search to converge.

Figure 7: Aggregated distance computations of BFiS w/ EP and PP, where  $\overline{W} = 64$ .

Figure 8: Dist. compt. increases as iter. depths decrease for PP increasing W.

depth by path-wise parallelism, (ii) reducing redundant distance computation by staged expansion, and (iii) reducing synchronization overhead by redundancy-aware synchronization.

#### 3.2.1 Reduce Iteration Depth by Intra-Query Path-Wise Parallelism

In each search iteration, a Best-First-Search (BFiS) algorithm is often used to perform node expansion to the most promising unchecked candidate [24, 40]. In <u>iQAN</u>, we make a small modification to this process by relaxing the priority order and letting each thread expand a few more nodes (e.g., top W unchecked candidates) in every step as active nodes for expansion. We also relax the synchronization such that a global synchronization is only performed after a few expansion steps. We call this new way of expanding nodes <u>path-wise parallelism (PP)</u>. This small change in algorithm results in a significant reduction in iteration depths for queries, e.g., from a few thousands to tens in some cases.

Why would this change reduce the iteration depth? The multi-node expansion and relaxed synchronizations are equivalent to letting each thread explore paths in a local region instead of a single node's neighbor list before doing a global synchronization. By doing so, it increases the likelihood of finding nearest neighbors in less number of iterations. Fig. 5 shows the comparison results of iteration depths between <u>BFiS</u> and <u>PP</u> on dataset SIFT1M using 10K queries with a 0.90 recall target. We set W to 64. Overall, while <u>BFiS</u> takes 10.1, 69.4, and 88.1 steps to find the top-1, top-50, and top-100 near neighbor, <u>PP</u> only takes 3.4, 5.0, and 5.4 steps on average, respectively, a significant reduction. From the unchecked node's perspective, Fig. 6 shows that <u>PP</u> also takes much fewer steps to converge to a local optimum (i.e., finish examining all the unchecked vertices) than BFiS.

#### 3.2.2 Reduce Redundant Computation by Staged Expansion

Although reducing the iteration depth significantly, does it mean the search process will now get desired speedups on multi-core architectures? The answer is no. The path-wise parallelism reduces iteration depths but at the same time introduces a considerate amount of additional distance computations, especially when the number of parallel workers is large. Fig. 7 shows that to reach the same recall (0.9-0.999), the path-wise parallelism often needs to perform significantly more distance computations than <u>BFiS</u> (1.3-3.5 times). Moreover, we also observe that although the iteration depths continue to decrease by increasing the concurrent expansion width W, the number of distance computations inversely increases, as shown in Fig. 8. The huge amount of redundant computations adversely affects search efficiency as many threads are loading vectors for unnecessary computations, wasting memory bandwidth and compute resources.

To mitigate it, we investigate the usefulness of path-wise parallelism at different search stages: at which stage does the path-wise parallelism reduce the iteration depths the most? We found that overall, in the beginning, since all candidates are far from the query, those early expanded candidates are likely to be discarded by closer ones that are visited later. In other words, candidates expanded and checked at an earlier stage have a high likelihood of becoming unnecessary from a future perspective. As the search moves forward toward the region that has near





100 Dist. Computation 80 Sync. Overhead 400 - 7.5×10<sup>8</sup> C 20 - 7.5×10<sup>8</sup> C



Figure 9: Dist. computation of <u>BFiS</u>, PP w/o and w/ staged expansion.

Figure 10: Number of unchecked candidates after each search step.



Figure 12: A query's average update positions during searching.

neighbors, a larger expansion width that covers more search paths can effectively prevent the search from getting stuck at a local minimum.

Based on these observations, we propose a staged expansion (SE) scheme by gradually increasing the expansion width W and the number of workers every t steps during the search procedure. In practice, we set the starting value of W to 1 and the maximum value as the number of available hardware threads. Then for every t steps (e.g., t = 1) we double the value of W until W reaches its maximum. Fig. 9 shows the comparison results of path-wise parallelism without and with staged expansion. The staged expansion reduces the number of redundant distance computations significantly, leading to distance computations comparable to BFiS. On the other hand, staged expansion is able to preserve the benefits of path-wise parallelism in terms of obtaining reduced iteration depths, as shown in Fig. 10. These results indicate that by performing path-wise parallelism at where they are most effective (i.e., the later phase of the search), the parallel search process can effectively converge with reduced iteration depths and minimal addition of redundant computations among multiple workers.

#### 3.2.3 Reduce Synchronization Overhead by Redundancy-Aware Synchronization

The remaining performance challenge in parallel search resides in the synchronization, as we still need to decide when to do synchronization. However, reducing the synchronization overhead for graph-based ANNS is non-trivial. Fig. 11 shows that as we skip synchronizations in between search iterations (i.e., increasing the interval between two synchronizations), the synchronization overhead (shown as the ratio to the total time) decreases significantly. However, decreasing synchronization increases distance computations, especially when the synchronization intervals become large. This is because as we increase the synchronization interval, it increases the likelihood that individual workers would search their local but unpromising areas without switching to newly identified promising regions found by other workers. As such, one cannot infinitely delay synchronization, and a small set but useful synchronizations are desired to achieve overall high search efficiency without incurring too many redundant computations.

Finding such intervals turns out to be non-trivial since the relative distance of a query to its near neighbors changes all the time at different stages. It is also hard to find one fixed synchronization interval for all queries. To mitigate the synchronization overhead, <u>iQAN</u> performs <u>redundancy-aware synchronization (RAS)</u>, which allows workers to perform a search with low redundant computations by adding a minimal set of global synchronizations. We introduce a metric — <u>update positions</u> — to capture the redundancy during expansion. When a worker thread expands an unchecked candidate, its unchecked neighbors are then inserted into the worker's local queue, and we define the update position as the <u>lowest (best)</u> position of all newly inserted candidates. Thus, <u>the average update position</u> (AUP) is the mean of all update positions of workers. Fig. 12 demonstrates how an example query's AUP changes during the search process without doing any global synchronizations. We observe that the AUP increases gradually to be equal to the local queue capacity and remains flat to the end. When the AUP is close to the queue capacity, it indicates that a majority of workers are searching areas that cannot find promising candidates to update their local results. Therefore, a high AUP indicates that most workers are doing redundant



Figure 13: Latency comparison among HNSW, NSG, and iQAN on Skylake (16T).

computations, and it would benefit from a global synchronization such that all workers can focus on searching for more promising areas that have a higher probability of including closer near neighbors.

## 3.3 Evaluation of <u>iQAN</u>

<u>iQAN</u> offers significant speedups than two state-of-the-art graph-based ANNS, NSG [23, 24] and HNSW [39, 40] over several public datasets, including SIFT1M (128D), GIST1M (960D), DEEP10M (96D), DEEP100M (96D), and SIFT100M (128D). We measure the latency and <u>Recall@100</u> (R@100), which measures the accuracy of finding the top-100 nearest neighbors for every query. We conduct our experiments on a workstation with Xeon Gold 6138 (2.00 GHz) with 20 cores and 128 GB DRAM (Skylake for short).

Fig. 13 compares the latency of HNSW, NSG, and <u>iQAN</u> on Skylake. NSG and HNSW use their sequential search algorithm, whereas <u>iQAN</u> uses 16 threads on Skylake. Across all five datasets, <u>iQAN</u> consistently provides latency speedups over existing sequential-based approaches NSG and HNSW over a wide range of recall targets. In particular, the speedups from <u>iQAN</u> increase as the recall target moves to the high accuracy regime (e.g., from 0.90 to 0.999). Notably, <u>iQAN</u> achieves up to  $12.9 \times$  speedups over NSG on DEEP100M on Skylake, obtaining an incredibly low latency of <5ms or <3ms at the recall target 0.999 by leveraging aggregated multi-core computation and memory bandwidth resources. This enables vector search with very high accuracy on large-scale graphs, even in extremely interactive online applications.

<u>iQAN</u> achieves significant latency speedups mainly for three reasons. First, <u>iQAN</u>'s path-wise parallelism effectively reduces the iteration depths, making the sequential dependencies no longer a major bottleneck. This is particularly critical for a large graph (e.g., DEEP100M) and high recall (e.g., 0.999) as seen in Section 3.2 that the iteration depths increase significantly as we either scale the graph size or increase the recall targets. Second, the reduced iteration depths do not come at the cost of many redundant computations as <u>iQAN</u> leverages staged expansion to effectively avoid redundant computations from doing path-wise parallelism. Third, <u>iQAN</u> significantly reduces the synchronization overhead through redundancy-aware synchronization. It is also worth mentioning that <u>iQAN</u> achieves excellent speedups as we increase the dimensionality of the embedding vectors. <u>iQAN</u> achieves up to  $24.9 \times$  speedups over HNSW on GIST1M on Skylake. This is higher than the speedups we get on a dataset with a similar scale but much smaller dimensionality (e.g., SIFT1M). <u>iQAN</u> is able to achieve better speedups on higher dimensional vectors because as the vector dimension increases, the amount of computation workload for the pair-wise distance computation also increases, which allows <u>iQAN</u> to benefit more from parallel computing.

**Comparison with DiskANN.** Fig. 14 compares the latency of DiskANN [30] (using 1 thread with its inmemory index) and <u>iQAN</u> (using 32 threads) for <u>Recall@1</u> targets. For building its indices of datasets SIFT1M and GIST1M, DiskANN uses L = 125, R = 70,  $\alpha = 2$ , which are the same setting as shown in its paper. For DEEP10M, DiskANN uses L = 100, R = 100,  $\alpha = 1.2$ . Fig. 14 shows that <u>iQAN</u> achieves significant latency speedups over DiskANN, especially for the high recall regime. For example, for recall target 0.999, <u>iQAN</u> has about  $180.5 \times$  average speedup on DiskANN among these three datasets.

**Scaling to billion points.** This experiment is conducted on a machine with a 1.5 TB memory. It is worth mentioning that even 1.5 TB of memory is not enough to build a 100-NN graph with one billion data vectors.



DEEP1B SIFT1B 400 6 400 (ms) NSG's latenc NSG's laten 300 300 g hnnaa -atency 200 200 Latency (ms) Latency (ms Speedup 10 100 Speedup 64 8 16 32 4 8 16 32 2 Number of threads

Figure 14: Recall@1 latency of DiskANN and iQAN.

Figure 15: Performance comparison of <u>iQAN</u> and NSG on two billion-scale datasets SIFT1B (bigann) and DEEP1B.

Therefore, we limit the out-degree of NSG when generating the corresponding NSG index so that the index construction can finish in a reasonable amount of time (e.g.,<10 days). We also note that this is the first time to evaluate an NSG graph at a billion scale as the maximum graph prior work such as NSG evaluated contained less than 100M data points. Fig. 15 compares the latency of <u>iQAN</u> and NSG. <u>iQAN</u> uses up to 64 threads, and the recall target is 0.9. When using 64 threads, <u>iQAN</u> follows the trend of scalability we observed as we increase the graph size and outperform NSG with  $11.5 \times$  and  $16.0 \times$  speedup for SIFT1B and DEEP1B, respectively, confirming the excellent scalability of <u>iQAN</u> on large-scale graphs again.

# 4 HM-ANN: Efficient Billion-Point Vector Search via Heterogeneous Memory

This section describes a large-scale vector search solution built on top of Heterogeneous Memory. The HNSW index is modified to make it HM-aware, and the search algorithm is also modified to make the search more efficient. With these modifications, this solution enables fast and highly accurate billion-scale ANNS on HM.

## 4.1 Design of HM-ANN

The design of HM-ANN generalizes HNSW, whose hierarchical structure naturally fits into HM. Elements in the upper layers consume a small portion of the memory, making them good candidates to be placed in fast memory (small capacity); The bottom-most layer has all the elements and has the largest memory consumption, which makes it suitable to be placed in slow memory. Unlike HNSW, where the majority of search happens in the bottom-most layer, elements in the upper layers now have faster access speed, so it is a reasonable strategy to increase the access frequency of the upper layers. On the other hand, since accessing L0 is slower, it is preferable to have only a small portion of it to be accessed by each query. The key idea of HM-ANN is, therefore, to build high-quality upper layers and make most memory accesses happen in fast memory, in order to provide better navigation for search at L0 and reduce memory accesses in slow memory.

**Notations.** In the rest of the paper, we let V denote the dataset with N = |V| to build the graph; we refer the graph in the layer  $i \in \{0, 1, ..., l\}$  of HM-ANN as  $G_i = (V_i, E_i)$  where  $V_i$  is the vertex set and  $E_i$  is the edge set. We refer  $N_i$  as the number of elements in the layer i, and we have  $N_i = |V_i|$ . Because L0 contains all the elements in the database, we have  $V_0 = V$  and  $N_0 = N$ . Based on the hierarchical structure of HM-ANN, we have  $V_i \subsetneq V_{i-1}$ . Similar to the existing effort [40], we introduce  $M_i$  as the maximum number of established connections for each point v in the layer i. For  $v \in V$ , we let D(v) denote the degree of node v, and  $D(v) = \sum_{u \in V} m(v, u)$  where m(v, u) = 1 if there exists a link between node v and node u.

## 4.1.1 HM-aware Index Construction via Top-Down Insertions and Bottom-up Promotions

To make the ANNS index aware of HM architecture, we generalize the HNSW construction algorithm to include two phases: a top-down insertion phase and a bottom-up promotion phase.

**Top-down insertions.** The top-down insertion phase is the same as HNSW, where we incrementally build a hierarchical graph by iteratively inserting each vector v in V as a node in G. Each node will generate up to M (i.e., the neighbor degree) out-going edges. Among those, M - 1 are short-range edges, which connect vto its M - 1 nearest neighbors according to their pair-wise Euclidean distance to v. The rest is a long-range edge that connects v to a randomly picked node, which may connect other isolated clusters. It is theoretically justified that graphs (e.g., L0) constructed by inserting these two types of edges guarantee to have the small world properties [24, 40, 61].

**Bottom-up promotions.** The goal of the second phase is to build a high-quality projection of L0 elements into the layer 1 (L1), such that the search in L0 can find the true nearest neighbors of the query with only a few hops. Ideally, HM-ANN wants to achieve the goal that performing 1-greedy search in L0 is sufficient to achieve high recall, so that the slowdown caused by accessing the slow memory is minimal. A straightforward way to project the L0 elements into L1 is to randomly select a subset of elements in L0 to be L1, similar to what HNSW already does to build upper layers. However, we observe that such an approach leads to poor index quality. As a result, many searches end up happening in L0 (slow memory), causing long search latency.

HM-ANN uses a high-degree promotion strategy. This strategy promotes elements with the highest degree in L0 into L1. From the layer i ( $i \ge 2$ ) to i + 1, HM-ANN promotes high-degree nodes to the upper layer with a promotion rate of 1/M, where M is the maximum number of neighbors for each element (i.e.,  $M_i = M$ , where i = 2...l). A similar promotion rate setting is used in HNSW [40] and typical skip list [50]. HM-ANN increases search quality in L1 by promoting more nodes from L0 to L1 and setting the maximum number of neighbors for each element in L1 to  $2 \times M$  (i.e.,  $M_1 = 2 \times M$ ). The number of nodes in upper layers ( $N_i$ , where i = 1..l) is decided by available fast memory space.

The high-degree promotion strategy is based on the following observation. The hub nodes of the graph at L0 are those nodes with a large number of connections (i.e., high degree). In the small world navigation algorithm, a higher degree node provides better navigability [6]. Most of the shortest paths between nodes flow through hubs. In other words, the average length of the navigation path (i.e., number of hops) is the smallest, when the adjacent node with the highest degree is selected as the next hop. By promoting the high-degree nodes, the resulting L1 layer allows HM-ANN to effectively reduce the number of search steps in L0, compared with the random promotion strategy.

#### 4.1.2 HM-ANN Graph Search Algorithm

**Fast memory search.** The search in fast memory begins at the entry point in the top layer and then performs a 1-greedy search from the top layer to layer 2, which is the same as in HNSW. To narrow down the search space in L0, HM-ANN performs the search in L1 with a search budget controlled by  $efSearch_{L1}$ .  $efSearch_{L1}$  defines the size of dynamic candidate list in L1. Those candidates in the list are used as entry points for search in L0 (HNSW uses just one entry point), in order to improve search quality in L0.

**Parallel L0 search.** In L0, HM-ANN evenly partitions the candidates from searching L1 and uses them as entry points to perform <u>parallel multi-start 1-greedy search</u> with Thr threads in parallel. The top candidates from each search are collected to find the best candidates. Parallel search makes the best use of memory bandwidth and improves search quality without increasing search time. Thr is determined by peak memory bandwidth constrained by hardware divided by memory bandwidth consumption by one thread, which is easy to calculate.

Different from the SSD-based ANNS [55, 66], the data in slow memory in HM-ANN can be directly accessed by processors, and there is no duplication between fast and slow memories. However, due to high latency and low bandwidth of slow memory, HM-ANN should still make memory accesses in fast memory as many as possible. HM-ANN implements a software-managed cache in fast memory to prefetch data from slow memory to fast memory before the memory access happens. In particular, HM-ANN reserves a space in fast memory ( $\sim 2$ GB) called *migration space*. When searching L1, HM-ANN asynchronously copies neighbor elements of those candidates in *efSearch*<sub>L1</sub> and the neighbor elements' connections in L1 from slow memory to the migration

	BigANN					DEEP1B				
	Indexing			Search		Indexing			Search	
	Graph	Indexing	Promo.	Fast-mem	Slow-mem	Graph	Indexing	Promo.	Fast-mem	Slow-mem
	size	time	rate	usage	usage	size	time	rate	usage	usage
HNSW	475GB	90h	0.02	96GB	490GB	723GB	108h	0.02	96GB	748GB
				(hw caching)					(hw caching)	
NSG	285GB	115h	-	96GB	303GB	580GB	134h	-	96GB	599GB
				(hw caching)					(hw caching)	
HM-ANN	536GB	96h	0.16	96GB	462GB	756GB	117h	0.11	96GB	681GB

Table 1: Indexing time and memory consumption for graph-based methods on billion-scale datasets

space in fast memory. When the search in L0 happens, there is already a portion of to-be-accessed data placed in fast memory, which leads to shorter query time.

## 4.2 Evaluation of HM-ANN

**Billion-scale algorithm comparison.** We compare HM-ANN with the graph- (HNSW and NSG) and quantizationbased algorithms (IMI+OPQ and L&C). For HNSW, we build graphs with *efConstruction* and *M* set to 200 and 48 respectively; For NSG we first build a 100-NN graph using Faiss [42] and then build NSG graphs with R = 128, L = 70 and C = 500. We collect results on NSG and HNSW using Memory Mode since it leads to overall better performance than using first-touch NUMA. For IMI+OPQ, we build indexes with 64- and 80-byte code books on BIGANN and DEEP1B respectively. We present the best search result with search parameters nprobe=128 and ht=30 for BIGANN and with autotuning parameter sweep on DEEP1B. For L&C, we use 6 as the number of links on the base level, and use 36- and 144-byte OPQ code-books. We use the same parameters (*efConstruction*=200 and *M*=48) as HNSW to construct HM-ANN. We set *efSearch*<sub>L0</sub>=2 and vary *efSearch*<sub>L1</sub> to show the latency-vs-recall trade-offs.

Figures 16 (a)-(d) visualize the results. Overall, HM-ANN provides the best latency-vs-recall performance. Figure 16 (a) and (b) show that HM-ANN achieves the top-1 recall of > 95% within 1ms, which is 2x and 5.8x faster than HNSW and NSG to achieve the same recall target respectively. IMI+OPQ and L&C cannot reach a similar recall target, because of precision loss from quantization. As another point of reference, the SSD-based solution, DiskANN [55] (not open-sourced), provides 95% top-1 recall in 3.5ms. In contrast, HM-ANN provides the same recall in less than 1ms, which is at least  $3.5 \times$  faster. We compare the top-100 recall shown in Figures 16 (c) and (d). HM-ANN provides higher performance than all other approaches. For example, it obtains top-100 recall of > 90% within 4 ms, while performing 2.8x and 5x faster than HNSW and NSG with the same recall target respectively. Quantization-based algorithms perform poorly and have difficulties to reach a top-100 recall of 30%.

Table 1 shows the index construction time and index size of HNSW, NSG, and HM-ANN. Among the three, HNSW takes the shortest time to build the graph. HM-ANN takes 8% longer time than HNSW because it takes an additional pass for the bottom-up promotion. However, HM-ANN is still faster to construct than NSG. In terms of memory usage, HM-ANN indexes are 5–13% larger than HSNW, because it promotes more nodes from L0 to L1. In terms of memory usage, HM-ANN consumes less fast memory than HNSW and NSG, which is valuable to reduce production cost [41, 52]. HNSW and NSG use all fast memory because they do not explicitly manage HM and by default using Memory Mode consumes all fast memory. The sum of slow and fast memory consumption can be larger than the index size because there are metadata needed for search that are not counted in the index size.



Figure 16: Query time vs. recall curve in (a) DEEP1B top-1, (b) BigANN top-1, (c) DEEP1B top-100, (b) BigANN top-100, respectively.

# **5** Research Opportunities

In this paper, we have concentrated on the computational and memory efficiency of large-scale vector search. Our  $\underline{iQAN}$  and HM-ANN designs achieve exceptional performance. However, large-scale vector search still requires further attention, and we expect that additional improvements can be made. This section explores the challenges and research opportunities associated with vector search.

**High-performance vector search through hierarchical parallelism.** Inference demand varies across different applications and scenarios, with some being latency-critical, and others being latency-sensitive or throughput-oriented. Additionally, the hardware resources available to each application may also vary significantly. To meet these diverse requirements and make efficient use of all computational resources, we can combine different parallelism approaches:

- Distributed vector search. Libraries such as Milvus [57] allow the development of distributed versions of vector search systems using a cluster.
- Inter-query parallelism. Multithreading enables the use of multiple cores to improve the throughput of answering user requests.
- Intra-query parallelism. Methods such as <u>iQAN</u> enable the system to speed up individual queries by leveraging the aggregated computational capacity of multiple cores.

These techniques can be combined hierarchically to meet given latency, throughput, and cost objectives. The research opportunity here is: <u>Build systems for large-scale vector search that fully exploit parallelism at different levels</u>, providing millisecond-level latency, high throughput, and low hardware cost. Furthermore, while techniques such as <u>iQAN</u> optimize search efficiency by modifying the graph traversal process to leverage intra-query parallelism, the underlying index has not been specifically designed to take advantage of intra-query parallelism. Intuitively, a different index structure could achieve better search efficiency under intra-query parallelism if it naturally helps avoid redundant computations across different threads and also leads to better data locality. The research opportunity here is: <u>Design vector search indices that maximize search efficiency through</u> both inter- and intra-query parallelism.

**Highly concurrent vector search with addition and deletion.** Most existing vector search systems build indices offline and become read-only once deployed. In some applications, data capture may occur more frequently than query processing, or the application may need to index continuous data streams while serving query requests. In these cases, the index organization should be optimized for addition and update in addition to query performance. The complexity arises with concurrent read and update operations (e.g., addition, deletion), as it is challenging to achieve both correctness and speed simultaneously. Concurrent updates and reads can lead to data races, making it difficult to ensure correctness on shared-memory architectures. Lock-based synchronization

can be used to coordinate access to shared-memory data, but while using one lock for the entire index simplifies reasoning about correctness, it also leads to serialized execution, severely impacting scalability. Another solution is to use fine-grained locking, where multiple locks are associated with the index. However, fine-grained locking increases the complexity of operations that access shared data, leading to issues such as deadlock, atomicity violation, and high locking overhead. The research opportunity here is: <u>Build a highly concurrent vector search</u> system that supports robust addition and deletion with high search efficiency.

Automating index construction for vector search. Several advances have been made in support of large-scale vector search, introducing novel algorithms and system optimizations [15, 51, 55, 66]. However, applying these methods to large-scale datasets still requires a significant amount of engineering effort specific to the data, hardware environment, and performance objectives. For instance, deploying a large dataset to a given cloud VM requires a careful selection of vector search indices, with NVMe/SSD/remote storage, the number of cores to use, and multiple index-specific parameters. Correctly choosing the algorithm and tuning the parameters can deliver significant improvements in search performance but also depend on strong system expertise. Therefore, automating the selection and construction of vector search indices would help alleviate the burden on deployment engineers, but it also requires navigating a complex space of choices that grows exponentially with different algorithm choices, each with its own trade-offs, and data sizes and hardware resources. The research opportunity here is: Build a framework and optimization algorithm that automatically finds the optimal index construction strategy for a given dataset to deliver fast search speed with low cost.

# 6 Conclusion

We have designed and implemented <u>iQAN</u> and HM-ANN to enhance the system efficiency of vector search. We have followed the state-of-the-art vector search algorithm, but at every level, we have introduced innovations that stretch prior methods and tailor our system for the newer hardware setting of multi-core processors and heterogeneous memory. Our innovations, such as intra-query path-wise parallelism, staged expansion, redundancy-aware synchronization, and HM-aware index construction, improve the computational and memory efficiency of large-scale vector search. Many of these methods should work well in other settings, such as SSD-based methods and compression-based indices. Finally, we propose several open research directions from a system perspective that have the potential to further improve large-scale vector search. We hope these directions will inspire new research that can advance vector search and make it more efficient, robust, and easy-to-use.

# References

- Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In 2006 47th annual IEEE symposium on foundations of computer science (FOCS'06), pages 459–468. IEEE, 2006. doi: 10.1109/FOCS.2006.49.
- [2] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, <u>Proceedings of the</u> <u>28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS)</u>, volume 28 of <u>NIPS'15</u>, pages 1225–1233, Cambridge, MA, USA, 2015. MIT Press. URL https://proceedings.neurips.cc/ paper/2015/file/2823f4797102ce1a1aec05359cc16dd9-Paper.pdf.
- [3] Alexandr Andoni, Piotr Indyk, and Ilya Razenshteyn. Approximate Nearest Neighbor Search in High Dimensions. arXiv preprint arXiv:1806.09823, 2018.
- [4] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In International Conference on Similarity Search and Applications (SISAP), pages 34–49. Springer, 2017.

- [5] Artem Babenko and Victor Lempitsky. The inverted multi-index. <u>IEEE Transactions on Pattern Analysis and Machine</u> Intelligence, 37(6):1247–1260, 2014.
- [6] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. <u>Science</u>, 286(5439):509–512, 1999. ISSN 0036-8075. doi: 10.1126/science.286.5439.509. URL https://science.sciencemag.org/content/286/5439/509.
- [7] KG Renga Bashyam and Sathish Vadhiyar. Fast scalable approximate nearest neighbor search for high-dimensional data. In <u>2020 IEEE International Conference on Cluster Computing (CLUSTER)</u>, pages 294–302. IEEE, 2020. doi: 10.1109/CLUSTER49012.2020.00040.
- [8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R\*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In SIGMOD 1990, pages 322–331, 1990.
- [9] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. <u>Communications of the</u> ACM, 18(9):509–517, September 1975. ISSN 0001-0782.
- [10] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. <u>Nature Biotechnology</u>, 33 (6):623–630, 2015.
- [11] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When Is "Nearest Neighbor" Meaningful? In International Conference on Database Theory (ICDT), pages 217–235, 1999.
- [12] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in High-Dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases. ACM Computing Surveys, 33(3):322–373, 2001.
- [13] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, <u>International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA</u>, volume 162 of <u>Proceedings of Machine Learning Research</u>, pages 2206–2240. PMLR, 2022.
- [14] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. SPTAG: A library for fast approximate nearest neighbor search, 2018.
- [15] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: highly-efficient billion-scale approximate nearest neighborhood search. In <u>Advances in Neural Information Processing Systems 34</u>: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual, pages 5199–5212, 2021.
- [16] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In Shilad Sen, Werner Geyer, Jill Freyne, and Pablo Castells, editors, <u>Proceedings of the 10th ACM Conference on Recommender</u> Systems, Boston, MA, USA, September 15-19, 2016, pages 191–198. ACM, 2016.
- [17] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: Scalable online collaborative filtering. In Proceedings of the 16th International Conference on World Wide Web (WWW), pages 271–280, 2007.
- [18] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In <u>Proceedings of the Twentieth Annual Symposium on Computational Geometry</u>, SCG '04, pages 253–262, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138857. doi: 10.1145/997817.997857. URL https://doi.org/10.1145/997817.997857.

- [19] Matthijs Douze, Alexandre Sablayrolles, and Hervé Jégou. Link and code: Fast indexing with graphs and compact regression codes. In <u>Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)</u>, pages 3646–3654, 2018.
- [20] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. Return of the lernaean hydra: Experimental evaluation of data series approximate similarity search. <u>Proceedings of the VLDB Endowment</u>, 13(3): 403–420, 2019.
- [21] Nanyi Fei, Zhiwu Lu, Yizhao Gao, Guoxing Yang, Yuqi Huo, Jingyuan Wen, Haoyu Lu, Ruihua Song, Xin Gao, Tao Xiang, et al. Towards artificial general intelligence via a multimodal foundation model. <u>Nature Communications</u>, 13 (1):3094, 2022.
- [22] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In Proceedings of the ACM Conference of the Special Interest Group on Data Communication, SIGCOMM '13, pages 159–170, 2013. ISBN 978-1-4503-2056-6. doi: 10.1145/2486001.2486014. URL http://doi.acm.org/10. 1145/2486001.2486014.
- [23] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Nsg. https://github.com/ZJULearning/nsg, 2019. [Online; accessed 25-April-2022].
- [24] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. <u>Proceedings of the VLDB Endowment (VLDB)</u>, 12(5):461–474, January 2019. ISSN 2150-8097. doi: 10.14778/3303753.3303754. URL https://doi.org/10.14778/3303753.3303754.
- [25] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In 2013 IEEE Conference on Computer Vision and Pattern Recognition, pages 2946–2953, 2013.
- [26] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In <u>VLDB'99</u>, pages 518–529, 1999.
- [27] Johannes Hoffart, Stephan Seufert, Dat Ba Nguyen, Martin Theobald, and Gerhard Weikum. Kore: Keyphrase overlap relatedness for entity disambiguation. In <u>Proceedings of the 21st ACM International Conference on Information and</u> Knowledge Management (CIKM), pages 545–554, 2012.
- [28] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In CIKM '13, pages 2333–2338, 2013.
- [29] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In <u>Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing</u>, STOC '98, pages 604–613, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919629. doi: 10.1145/276698.276876. URL https://doi.org/10.1145/276698.276876.
- [30] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, <u>Advances in Neural Information Processing Systems</u>, volume 32, pages 13771–13781. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/ paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf.
- [31] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. <u>IEEE</u> <u>Transactions on Pattern Analysis and Machine Intelligence</u>, 33(1):117–128, 2011. doi: 10.1109/TPAMI.2010.57.
- [32] Yannis Kalantidis and Yannis S. Avrithis. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In CVPR 2014, pages 2329–2336, 2014.
- [33] Brian Kulis and Kristen Grauman. Kernelized locality-sensitive hashing for scalable image search. In 2009 IEEE 12th International Conference on Computer Vision (ICCV), pages 2130–2137. IEEE, 2009.

- [34] Quoc V. Le and Tomás Mikolov. Distributed representations of sentences and documents. In Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014, volume 32 of JMLR Workshop and Conference Proceedings, pages 1188–1196. JMLR.org, 2014.
- [35] D. T. Lee and C. K. Wong. Worst-case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees. Acta Informatica, 9(1):23–29, March 1977.
- [36] Victor Lempitsky. The inverted multi-index. In CVPR '12, pages 3069–3076, 2012.
- [37] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data experiments, analyses, and improvement. <u>IEEE Transactions on Knowledge</u> and Data Engineering, 32(8):1475–1488, 2020. doi: 10.1109/TKDE.2019.2909204.
- [38] Qin Lv, Moses Charikar, and Kai Li. Image similarity search with compact data structures. In Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management (CIKM), pages 208–217, 2004.
- [39] Yury A Malkov and Dmitry A Yashunin. Hnswlib. https://github.com/nmslib/hnswlib, 2020. [Online; accessed 25-April-2022].
- [40] Yury A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. <u>IEEE Transactions on Pattern Analysis and Machine Intelligence</u>, 42(4):824–836, 2020. doi: 10.1109/TPAMI.2018.2889473.
- [41] Jon Martindale. RAM has never been cheaper, but are the historic prices here to stay? https://www.digitaltrends.com/computing/why-is-ram-so-cheap/, 2019.
- [42] Meta. FAISS. https://github.com/facebookresearch/faiss/tree/master/benchs, 2017.
- [43] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, <u>1st International Conference on Learning Representations, ICLR</u> 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings, 2013.
- [44] Marius Muja and David G. Lowe. Scalable Nearest Neighbor Algorithms for High Dimensional Data. <u>TPAMI 2014</u>, 36(11):2227–2240, 2014.
- [45] Priyanka Nigam, Yiwei Song, Vijai Mohan, Vihan Lakshman, Weitian Allen Ding, Ankit Shingavi, Choon Hui Teo, Hao Gu, and Bing Yin. Semantic product search. In Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis, editors, <u>Proceedings of the 25th ACM SIGKDD International Conference on Knowledge</u> Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019, pages 2876–2885. ACM, 2019.
- [46] Mohammad Norouzi and David J. Fleet. Cartesian K-Means. In CVPR 2013, 2013.
- [47] Mohammad Norouzi, David J. Fleet, and Ruslan Salakhutdinov. Hamming distance metric learning. In <u>Advances</u> in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States, pages 1070–1078, 2012.
- [48] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures. In Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy, editors, Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023, pages 313–328. ACM, 2023.
- [49] James Philbin, Ondrej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman. Object retrieval with large vocabularies and fast spatial matching. In <u>2007 IEEE Conference on Computer Vision and Pattern Recognition</u> (CVPR), pages 1–8. IEEE, 2007.

- [50] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In F. Dehne, J. R. Sack, and N. Santoro, editors, <u>Algorithms and Data Structures</u>, pages 437–449, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. ISBN 978-3-540-48237-6.
- [51] Jie Ren, Minjia Zhang, and Dong Li. HM-ANN: efficient billion-point nearest neighbor search on heterogeneous memory. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, <u>Advances in Neural Information Processing Systems 33</u>: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020.
- [52] Gina Roos. Dram Prices Continue to Climb. https://epsnews.com/2017/08/18/dram-prices-continue-climb/, 2017.
- [53] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In <u>2008 IEEE</u> Conference on Computer Vision and Pattern Recognition (CVPR), pages 1–8. IEEE, 2008.
- [54] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. <u>arXiv</u> preprint arXiv:1409.1556, 2014.
- [55] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Rand-nsg: Fast accurate billion-point nearest neighbor search on a single node. In <u>NeurIPS</u>, pages 13748–13758, 2019.
- [56] Danny Sullivan. Faq: All about the google rankbrain algorithm. https://searchengineland.com/faq-all-about-the-newgoogle-rankbrain-algorithm-234440, 2018.
- [57] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, <u>SIGMOD '21: International</u> Conference on Management of Data, Virtual Event, China, June 20-25, 2021, pages 2614–2627. ACM, 2021.
- [58] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. arXiv preprint arXiv:2101.12631, 2021.
- [59] Runhui Wang and Dong Deng. Deltapq: Lossless product quantization code compression for high dimensional similarity search. Proceedings of the VLDB Endowment, 13(13):3603–3616, 2020.
- [60] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In <u>53rd Annual IEEE/ACM International Symposium on Microarchitecture</u>, MICRO 2020, Athens, Greece, October 17-21, 2020, pages 496–508. IEEE, 2020.
- [61] Duncan J. Watts. <u>Small Worlds: The Dynamics of Networks Between Order and Randomness</u>. Princeton University Press, 1999.
- [62] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In Proceedings of the 24th VLDB Conference, pages 194–205, 1998.
- [63] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. <u>Proceedings of the VLDB</u> Endowment, 13(12):3152–3165, 2020.
- [64] Samuel Williams, Andrew Waterman, and David A. Patterson. Roofline: an insightful visual performance model for multicore architectures. In CACM'09, 2009.
- [65] Hamed Zamani, Bhaskar Mitra, Xia Song, Nick Craswell, and Saurabh Tiwary. Neural Ranking Models with Multiple Document Fields. In WSDM '18, 2018.
- [66] Minjia Zhang and Yuxiong He. GRIP: multi-store capacity-optimized high-performance nearest neighbor search for vector search engine. In <u>CIKM 2019</u>, pages 1673–1682, 2019.