Highly Efficient Disk-based Nearest Neighbor Search on Extended Neighborhood Graph

Cheng Zhang*
Xiamen University
Xiamen, China
zhangcheng@stu.xmu.edu.cn

Jianzhi Wang*
Xiamen University
Xiamen, China
jzwang@stu.xmu.edu.cn

Wan-Lei Zhao†
Xiamen University
Xiamen, China
wlzhao@xmu.edu.cn

Shihai Xiao
Huawei Technologies Ltd.
China
xiaoshihai@huawei.com

Abstract

Nearest neighbor search (NN search) plays a fundamental role in many disciplines. According to recent studies, graph-based search methods show superior performance over other types of methods. In order to accommodate the high dimensionality as well as the growing data-scale, the disk-based NN search in which the index graph and the full-precision vectors are kept in SSD has become a promising direction. This paper optimizes the disk-based NN search from three perspectives. Firstly, an eXtended Neighborhood Graph (XN-Graph) structure is proposed. In contrast to the existing index graphs, the out-edges of the graph neighborhood are collected from much wider coverage of the data space. It therefore reduces the number of hops during NN search, which in turn reduces the search latency. Additionally, a dataset partitioning method called Boundary-adaptive Balanced Partition is proposed to facilitate the graph construction in cases where the system cannot handle large datasets in a single round. Moreover, an efficient hybrid NN search method called In-Memory First Search is proposed. Compared to the existing methods, it considerably reduces the CPU idle times. With the support of XN-Graph, it shows *1.5-3* times lower search latency than SOTA methods. On billion-scale datasets, its QPS is still above *4000* when *Recall@10* is as high as *0.9*.

CCS Concepts

• **Information systems** → **Top-k retrieval in databases**.

Keywords

k-nearest neighbor search, vector similarity search, disk-based approximate nearest neighbor search

*Both authors contributed equally to this research.
†The corresponding author.

ACM Reference Format:
Cheng Zhang, Jianzhi Wang, Wan-Lei Zhao, and Shihai Xiao. 2025. Highly Efficient Disk-based Nearest Neighbor Search on Extended Neighborhood Graph. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '25), July 13–18, 2025, Padua, Italy.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3726302.3729996

1 Introduction

The k-nearest neighbor search (k-NN search) has been extensively studied in the last five decades. It plays a fundamental role in various disciplines, including information retrieval [12, 52], recommendation systems [33, 41], computer vision [8, 40], and the recent Large Language Models (LLMs) [11, 22, 26]. Given a d-dimensional vector dataset $S = \{x_i | x_i \in \mathbb{R}^d\}$ and a query vector $x_q \in \mathbb{R}^d$, the k-NN search algorithm is expected to return the top-k nearest neighbors of x_q under a specified distance metric $dist(\cdot, \cdot) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. Owing to the curse of dimensionality, this issue becomes increasingly challenging as the dimension d grows. In the last few decades, a spectrum of algorithms has been proposed one after another [3, 4, 16, 18, 20, 23, 37, 44]. In general, they can be classified into four categories, namely tree-based methods [4, 27], hash-based methods [23, 30], quantization-based methods [1, 25, 34], and graph-based methods [5, 15, 32]. Among them, graph-based methods have become increasingly popular due to their considerably higher recall and efficiency.

In general, the search procedure of graph-based methods iteratively traverses a pre-built approximate k-NN graph [20], a relative neighborhood graph (RNG) [6, 15, 21, 32], or a Monotonic Search Network (MSNET) [9] using Best-First Search (BFS). Each vertex in these graph indexes represents a unique vector from the dataset S. The search proceeds to the next stage by expanding the neighbors of the closest unvisited vertex in the ranking list, which is usually called one "hop". The newly discovered closer neighbors are used to update the ranking list, allowing the search to progressively approach the true nearest neighbors in each round until no better candidates can be found. Essentially, the search follows a path from a random seed to the neighborhood of the target vector. The path is on a manifold that is formed by the graph. In many real scenarios, the dimension of this manifold is much lower than that of the data. Therefore, the search procedure becomes much more efficient as it traverses a much lower-dimensional space.

To enable efficient access to both the out-edges of any vertex and the corresponding vectors, standard graph-based methods maintain graph indexes and vector data in memory (DRAM). However, with the widespread use of social media, multimedia data have proliferated across various platforms. To facilitate the intelligent search and chatbot, the media data (text, photos and videos) have been largely transformed into high-dimensional vectors via LLMs [17, 45, 50]. With the growing scale of LLMs, the demand for the scalability of search algorithms has been steadily increasing. Current vector datasets often require memory capacities as large as *1,100* GB during the index construction [24]. The reliance on such large memory footprints significantly increases system costs and presents substantial obstacles to the practical deployment of existing methods at scale. To address this challenge, hybrid approximate nearest neighbor search methods that combine limited DRAM with Solid-State Drives (SSDs) have emerged as a viable solution. These methods effectively mitigate the memory constraints while ensuring robust retrieval performance. DiskANN [24] and SPANN [7] are two representative methods in this regard.

DiskANN [24] and its variants [19, 38, 43] store only the compressed vectors in memory, while the graph index and the full-precision vectors are kept on SSDs. During the search, only the neighborhoods of the traversed vertices and the corresponding vectors are loaded. Different from DiskANN, SPANN partitions the dataset into posting lists. A small-scale graph index on the centroids of these posting lists are maintained in memory. When a query arrives, the candidate posting lists are selected via the in-memory graph index. The potential nearest neighbors are discovered from the candidate posting lists using exhaustive search. Only the vectors in these candidate posting lists are loaded from the SSD into memory. Although SPANN demonstrates outstanding performance in single-thread scenarios, its performance significantly drops in multi-thread cases [47]. According to observations, the I/Os in SPANN are often jammed due to the loading of many irrelevant vectors from the candidate posting lists for different threads [47].

Although the hybrid search is a promising solution for high-dimensional large-scale NN search task, there are several latent issues. Firstly, the data scale that these methods are designed to handle is too large. The index cannot be built in memory since we simply cannot hold the entire dataset in memory. As a result, an indispensable step in these methods is to partition the dataset into subsets. The index is built on each subset. The final index of the dataset is the union of these indexes. In order to guarantee high search recall, overlap between different subsets is necessary. However, the overlap results in multiple copies of the same vector appearing in several subsets. This in turn leads to the high extra computational costs during the index construction. For instance, DiskANN takes about *5* days to build an index for a billion-scale dataset on a machine with *16*-core processor and *64* GB of memory, while SPANN takes around *4* days on a *45*-core, *260* GB machine [49].Secondly, compared to in-memory search, the processing bottleneck has been shifted from the distance calculations to the I/O operations between memory and SSD. This processing bottleneck, unfortunately, is not well-addressed in either DiskANN or SPANN. In SPANN, the intensive I/O operations are induced by the excessive loading of irrelevant vectors. While in DiskANN, the employed Beam Search is still stalled by the I/O operations.

This paper reconsiders the whole pipeline of the hybrid NN search. Novel solutions about index construction, dataset partitioning, and hybrid search strategy are proposed. Comprehensive experiments demonstrate that these solutions outperform SOTA methods in both index construction and NN search by a considerable margin. The major contributions of this paper are threefold.

- **A novel graph index called eXtended Neighborhood Graph (XN-Graph)** is proposed. With our construction method, XN-Graph can be built extremely fast. For a million-scale dataset, it typically takes only a few tens of seconds. Moreover, the extended neighborhood structure reduces the number of hops during the search process, which leads to fewer I/O operations in hybrid NN search.
- **A novel dataset partitioning method called Boundary-adaptive Balanced Partition (BBP)** is proposed. It facilitates the graph construction for large datasets which cannot be handled in one round due to the memory limit. Compared to simple partition by k-means clustering, it ensures the connectivity between the divided subsets while minimizing the overlap between them.
- **A novel hybrid search algorithm called In-Memory First Search (IMF Search)** is proposed. This algorithm is specifically designed for the hybrid NN search where the graph index resides on SSDs. It considerably reduces CPU idle time waiting for I/O operations. Moreover, with the support of XN-Graph, it converges faster while achieving higher search recall than SOTA methods.

2 Related Work

Usually, it is expected that the NN search procedure returns top-k nearest neighbors (k-NN search). Given a vector dataset S, a query x_q, a distance metric $dist(\cdot,\cdot)$, and a specified k, k-NN search aims to identify a set of vector points $\widehat{\Gamma}$ [1]

$$\widehat{\Gamma}(x_q, k) = \underset{\Gamma \subseteq S,\, |\Gamma| = k}{\arg\min} \sum_{x \in \Gamma} dist(x, x_q).$$

The performance of k-NN search is evaluated by the recall rate *Recall@k* (or k-recall@k). Given the top-k neighbors returned by an NN search method is $Q(x_q, k)$, its *Recall@k* is calculated as

$$Recall@k = \frac{|\widehat{\Gamma}(x_q, k) \cap Q(x_q, k)|}{k}.$$

In addition, the search procedure is expected to be as efficient as possible. The efficiency is evaluated in terms of latency or throughput, which is usually measured by Queries Per Second (QPS). Despite the diversity of NN search methods, most are driven by the same underlying motivation. Namely, they aim to avoid the comparisons between the query and vectors that are unlikely to be its neighbors. Extensive comparisons over the vector space usually lead to high recall but low QPS. NN search methods aim to make a good trade-off between recall and QPS. In this section, with regard to these two key performance metrics, we focus on analyzing the performance bottlenecks in the representative hybrid NN search methods. For a comprehensive review about the NN search methods, readers are referred to [28, 48].

[1]Without loss of generality, the smaller the distance, the closer the neighbor.

2.1 I/Os Latency in the Existing Hybrid Search

The latency of NN search refers to the total time required to process a query, including CPU computation and memory access overhead (denoted as T_{CPU}). In disk-based NN search, with the involvement of SSDs, I/O time costs must also be considered. Therefore, the total time for disk-based NN search can be summarized as:

$$T_{total} = T_{CPU} + T_{req} + T_{trans}, \tag{1}$$

where T_{req} is the time spent by the operating system handling I/O requests, generally determined by the number of I/O operations; T_{trans} is the time required to transfer data from the SSD to memory, which depends on the total volume of data to be transferred. Owing to the computer architecture, T_{trans} and T_{req} are much higher than T_{CPU} during disk-based NN search. Therefore, we focus on the T_{req} and T_{trans} arising from the I/O operations.

In contrast to in-memory NN search, the graph index in disk-based NN search is stored on SSD. Given the Best-First Search is adopted, each hop requires an SSD access. Although I/O operations load data in small sizes, the large number of I/O operations causes T_{req} to become a performance bottleneck. For this reason, existing disk-based methods abandon the Best-First Search strategy. DiskANN [24] adopts Beam Search, which reads the neighborhoods and vectors of multiple vertices in each I/O operation. Since SSDs organize data on pages and the page size (typically *4* KB) is much smaller than the SSD's bandwidth, the time required to read a small number of random pages is almost the same as reading a single page [24]. This property is fully utilized in DiskANN to reduce both T_{req} and T_{trans}. Nevertheless, due to the large number of I/O operations, T_{req} remains high in DiskANN. As to SPANN [7], it divides the dataset into overlapping subsets and performs an exhaustive search within the subsets which contain the potential nearest neighbors of the query. During exhaustive search, the vectors from these subsets can be loaded in a single operation. This significantly reduces the number of I/O requests, which in turn reduces T_{req}. Whereas, it increases the size of data to be loaded. In the multi-thread scenario, different threads may compete for the limited SSD's bandwidth. As a result, the performance of SPANN drops considerably in a multi-thread context.

In recent literature, a method called Starling [47] has been proposed. It employs a graph reordering scheme to reorganize the graph index, placing nearby vertices on the same SSD page. This allows the search process to load and expand more relevant vectors and their neighbors in each I/O operation, leading to faster convergence and significant decrease in both T_{req} and T_{trans}. Whereas, the high efficiency of Starling is subjected to the cases that multiple vertices can be stored on a single page. When the data dimensionality or the out-degree of the graph index increases, a page can contain only a single vertex, causing significant performance degradation.

2.2 Structure of the Graph Index

It is feasible to perform NN search on a k-NN graph [20, 28]. However, this way is inefficient because the query is compared with many redundant points in the neighborhood. It has been shown in several studies [15, 28, 32, 48] that graphs with diversified neighborhoods are more effective in supporting NN search. The diversification eliminates the redundant neighbors from the k-NN lists. Given vertex x, x_a and x_b are its neighbors, the following diversification rule has been widely adopted.

$$\begin{cases} dist(x, x_a) & < dist(x, x_b) \\ \alpha \cdot dist(x_a, x_b) & < dist(x, x_b), \end{cases} \tag{2}$$

where $\alpha \geq 1.0$ is a hyper-parameter. If Ineqn. **(2)** is satisfied, point x_b is viewed being occluded by x_a and will be eliminated from the x's neighborhood.

For in-memory graph-based methods [14, 32, 35, 39], I/O time is not a concern. In this case, T_{CPU} in Eqn. **(1)**, which is largely related to the number of distance calculations, becomes the primary factor that impacts the search performance. Therefore, in-memory methods aim to minimize the out-degree of points while ensuring sufficient navigability, to achieve a good trade-off between recall and speed. However, when the graph index is maintained on an SSD, such graph structure may not be necessarily effective to support the search. During search process, each expansion step (i.e., one hop) incurs an I/O request to load a vector and its neighborhood. On the one hand, reducing the number of hops during search is time-saving, as I/O operations are significantly more expensive than distance calculations. On the other hand, to maintain a certain level of search recall, the number of distance calculations (i.e., visited neighbors) should not be reduced. As a result, reducing the number of hops in NN search can only be achieved by increasing the average out-degree of the graph.

Although it is possible to increase the out-degree of the graph by relaxing the conditions for graph diversification (i.e., increasing α in Ineqn. **(2)**), the extra edges provide limited benefit since most of them are not "informative" to guide the search. Alternatively, one can perform the graph diversification on a wider neighborhood. Unfortunately, the time cost of building a graph with large neighborhood size is significantly higher. In [24], the proposed index construction method Vamana indeed tries to select the diversified neighbors from a much wider neighborhood. All the vertices that have been visited when retrieving a target vertex are collected for diversification. Compared to the graphs [15, 28, 32, 36] built for in-memory search, graph index built by Vamana results in fewer hops during the search, and consequently, lower I/O latency. However, since the collected vertices only cover a single path from a certain starting point directing to the target vertex, the constructed neighborhoods still lack broad coverage of the data space. Moreover, the construction process is very slow.

In this paper, we aim to optimize the structure of graph index to reduce the search latency due to T_{req} and T_{trans}. Namely, a novel graph index called eXtended Neighborhood Graph (XN-Graph) is proposed. In this graph, the neighborhood of a vertex is composed of the diversified edges that are collected from a much wider neighborhood coverage. This allows us to increase the average out-edge degree for the graph, and therefore reduce the number of hops during the search. Moreover, such a kind of graph can be built in extremely high speed. According to our observation, the construction algorithm is at least *3* times faster than HNSW and Vamana, and *1.5* times faster than RNND [36].

In the following sections, we present how such XN-Graph is constructed for a large-scale dataset. With the support of XN-Graph,

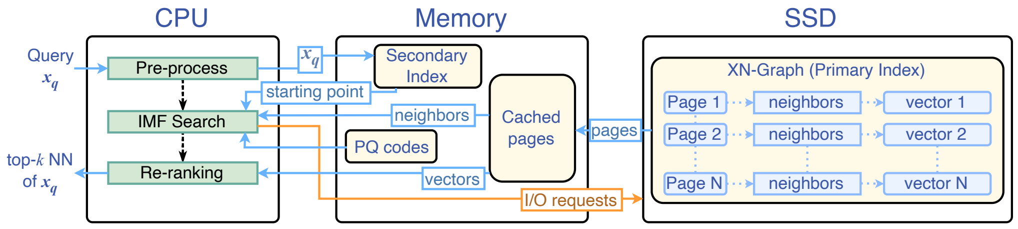


Figure 1: The framework of disk-based NN search on the XN-Graph. XN-Graph along with the dataset vectors are kept in the SSD. The vectors and the graph neighborhoods are loaded from SSD to the memory upon the requests of IMF Search.

we present IMF Search, which is able to achieve considerable speed-up over the SOTA disk-based NN search methods under both the single-thread and multi-thread scenarios.

3 Disk-based Approximate Nearest Neighbor Search

3.1 Framework Overview

Figure 1 illustrates the general framework of our disk-based NN search. It generally follows the pipeline of DiskANN [24]. The graph index along with the full-precision vectors are kept on SSD, while the vectors compressed using product quantization (PQ) [25] for the dataset are maintained in memory. Given a query x_q, two ranking lists $\widehat{Q}(x_q, k)$ and $Q(x_q, k)$ are maintained in memory. $\widehat{Q}(x_q, k)$ is ranked by distances between the query and PQ-compressed vectors (PQ codes). It is used for vertex expansion during the search. $Q(x_q, k)$ is ranked by the true distance between the query and traversed candidates and is updated whenever the vector of an expanded vertex is loaded into memory.

Nevertheless, our disk-based NN search is essentially different from DiskANN [24] in the following four aspects. **1.** A more effective XN-Graph is proposed as the graph index to replace the role of Vamana in DiskANN. **2.** To facilitate large-scale graph construction, instead of relying on k-means, a novel dataset partitioning procedure is proposed to divide the dataset into overlapping subsets. Compared to k-means, it reduces data redundancy while maintaining graph connectivity across different subsets. **3.** A small-scale in-memory graph index is adopted, constructed with less than *1%* of the dataset vectors. This graph index and the corresponding vectors are maintained in memory. When a query arrives, its nearest neighbor in the secondary index is identified using Best-First Search and used as the starting point for disk-based NN search. **4.** Moreover, the Beam Search in DiskANN is replaced by IMF Search. It considerably reduces latency compared to Beam Search, and unlike Starling [47], it requires no specific reorganization of the graph index on each SSD page.

In the following sections, we present the major components of our search framework, namely the eXtended Neighborhood Graph (XN-Graph), Boundary-adaptive Balanced Partition (BBP), and In-Memory First Search (IMF Search) in detail.

3.2 eXtended Neighborhood Graph

For most of the existing graph index construction methods [15, 24, 32], the points of each neighborhood are derived either from a vertex's k-NN list [28, 36] or from a traversed search path to each vertex [15, 24]. The k-NN lists are either pre-constructed as k-NN graphs [10, 13] or generated on-the-fly [32], while the traversed search paths are collected by simulating the graph-based NN search for each vertex [15, 24]. However, the k-NN of a vertex only covers a small region around it. For this reason, the guided hops only occur when the search procedure reaches the close neighborhood of a vertex. This is why it normally takes a series of hops before the search reaches the target vertex. A feasible way to reduce the number of hops is to extend the coverage of the graph neighborhoods. Intuitively, increasing the value of k allows each vertex's k-NN to cover a larger region, but this significantly increases the time cost of index construction. Moreover, augmenting the neighborhood with vertices collected from traversed search paths offers limited benefit. Although these paths span a broader region of the data space, the overall coverage remains limited, as only the vertices along the traversal trail are included.

Considering the drawbacks of existing methods, we aim to design a construction method that is able to collect the neighborhood points from a wider coverage for each vertex. We notice that the classic k-NN graph construction method, NN-Descent [10], actually produces many long edges connecting to distant neighbors during its iterative process. These long edges are generated through *local join*[10] operations across diverse neighborhoods and, compared to existing methods[15, 24], they cover a wider range of traversal paths. However, the long edges are squeezed out when closer neighbors are discovered. When the NN-Descent procedure converges, only the approximate k-NNs are retained. If the intermediate neighbors produced during NN-Descent are retained, it becomes possible to build an index with wider coverage.

Inspired by NN-Descent, a graph index construction procedure called eXtended Neighborhood Descent (XN-Descent) is proposed. The graph index constructed by XN-Descent is called eXtended Neighborhood Graph (XN-Graph). Intermediate neighbors are iteratively collected for each vertex to build the graph index. In order to retain the intermediate neighbors while allowing cross-matching among them, three graphs—G, C and R—are introduced. For each $x_i \in S$, $G[i]$ retains up to λ diversified intermediate neighbors of x_i; $C[i]$ stores the top-ω intermediate nearest neighbors of x_i discovered in previous iterations; and $R[i]$ keeps the reverse nearest neighbors of x_i. Both $G[i]$ and $C[i]$ are sorted in ascending order to ensure that only the close neighbors encountered so far are retained.

In each iteration, all the intermediate neighbors kept in $\widehat{C}[i] = C[i] \cup R[i]$ are evaluated for insertion into $G[i]$. Each $u \in \widehat{C}[i]$ is evaluated with respect to the diversification rule in Ineqn. **(2)**.

Specifically, the distance between u and $\forall v \in G[i]$ is calculated to check whether u is occluded. If u is not occluded by any v, it will be inserted into $G[i]$, and any points in $G[i]$ that are occluded by u are removed. Moreover, as u and v have encountered each other and their mutual distance has been calculated, their intermediate neighborhoods $C[u]$ and $C[v]$ are updated with each other. Note that $C[i]$ is not updated unless x_i appears in the neighborhoods of other vertices.

The complete iterative procedure is presented in Algorithm 1. Each neighborhood in graph C is initialized with ω randomly selected points, which serve as the initial intermediate neighbors for each vertex. In contrast, each neighborhood in graph G is initially empty. During each iteration, the neighbors from $C[i]$ are attempted to be inserted into $G[i]$ (*Lines 11–23*). To this end, each point in $C[i]$ is evaluated against each point $v \in G[i]$ under the diversification rule (Ineqn. **(2)**). Only the points that are not occluded are inserted into $G[i]$. In this way, diversified intermediate neighbors are collected in $G[i]$. Meanwhile, a new edge $\langle u, v\rangle$ is generated as a result of each distance calculation. Since both u and v are intermediate neighbors of x_i, they are likely to be neighbors of each other according to the principle "a neighbor of a neighbor is also likely to be a neighbor" [10]. Thus, the neighborhoods $C[u]$ and $C[v]$ can be updated in this process (*Lines 15–16*). As a result, graph C can be viewed as an approximate ω-NN graph that evolves toward higher quality as the iteration continues, while graph G maintains an XN-Graph by collecting the diversified intermediate neighbors so far produced. As the iteration progresses, neighbors from a wider coverage are incrementally joined into G. Moreover, the graph G remains in a diversified state throughout, as mutual occlusion does not occur between any two points in each $G[i]$.

Discussion Similar to NN-Descent [10], Algorithm 1 improves the quality of the graph neighborhoods by cross-matching within the neighborhood of each vertex. Nevertheless, the cross-matching in Algorithm 1 is undertaken between the set of close neighbors $C[i]$ and the set of diversified intermediate neighbors $G[i]$ collected across different iterations. The iterative cross-matchings can be viewed as a batch NN search for each vertex on an evolving neighborhood graph [10]. Algorithm 1 is apparently more efficient than NN-Descent [10], as searching on a diversified graph has been shown to be more effective [28, 32, 51]. Moreover, compared to NN-Descent [10], no sampling on each neighborhood is required in Algorithm 1, which makes the whole procedure more coherent and thus friendly to multi-thread context. Interestingly, although the primary goal of Algorithm 1 is to build a graph index, it also produces an approximate ω-NN graph C as a by-product.

The merits of Algorithm 1 are at least fourfold. Firstly, a small ω (typically *32 ∼ 40*) is sufficient to build a graph index that achieves significantly wider coverage than a k-NN graph, making the graph index construction extremely efficient. Experiments show that Algorithm 1 always converges faster than NN-Descent. Secondly, compared to Vamana [24] and NSG [15], the extended neighborhood of each vertex is derived from diverse trails of NN search since the iterative cross-matchings are viewed as a batch NN search. Therefore, the constructed XN-Graph covers a wider neighborhood region than both Vamana and NSG. Thirdly, each $G[i]$ keeps both close neighbors and relatively distant neighbors of x_i. It is able to lead the NN search either to a nearby or a more remote region of the data space. Therefore, it largely avoids the NN search being trapped in local regions. Lastly, λ can be increased at low additional costs to include more valuable edges in diverse directions. This will increase the average out-degree of the graph index and reduce the number of hops of NN search, which in turn decreases the I/O latency.

Algorithm 1: eXtended Neighborhood Descent

```
Data: dataset $S$, max. out-degree $\lambda$, num. of candidates $\omega$
Result: Extended Neighborhood Graph $G$
1  begin
2      for $x_i \in S$ do
3          $G[i] \leftarrow \varnothing$;
4          $R[i] \leftarrow \varnothing$;
5          $C[i] \leftarrow \omega$ random points from $S$;
6      end
7      repeat
8          parallel foreach $x_i \in S$ do
9              $\widehat{C}[i] \leftarrow C[i] \cup R[i]$;
10             $R[i] \leftarrow \varnothing$;
11             foreach $u \in \widehat{C}[i]$ do
12                 $R[u] \leftarrow R[u] \cup \{i\}$;
13                 $flag[u] \leftarrow True$;
14                 for $v \in G[i], v \neq u$ do
15                     TryInsert $(u, C[v], \omega)$;
16                     TryInsert $(v, C[u], \omega)$;
17                     $flag[u] \leftarrow False$ if $u$ is occluded by $v$;
18                 end
19                 if $flag[u]$ is True then
20                     Remove points occluded by $u$ from $G[i]$;
21                     TryInsert $(u, G[i], \lambda)$;
22                 end
23             end
24         end
25     until Converged;
26     return $G$;
27 end
28 Function TryInsert $(y, N, L)$
29     $N \leftarrow N \cup \{y\}$;
30     while $|N| > L$ do
31         Remove the points with max distance from $N$;
32     end
33 end
```

3.3 Boundary-adaptive Balanced Partition

With Algorithm 1, we are able to build graph index to support the NN search. However, it requires the whole dataset S to reside in memory. For large-scale search tasks, maintaining the full dataset in memory is usually impractical. In DiskANN [24], the dataset is divided into a number of overlapping subsets by k-means [31]. Each vector is assigned to several nearest centroids, resulting in overlap across subsets. A graph index is constructed on each subset, and the resulting sub-graphs are eventually merged into a complete graph, where the shared vectors across subsets serve as the merging anchors. It is clear to see that the assignment of each vector to multiple clusters guarantees the connectivity of the sub-graphs. However, it also induces data redundancy and extra computation costs. Moreover, given cluster-i shares boundaries with many other clusters, it is possible that an abundance of points from these clusters are partitioned into cluster-i as the secondary assignment. The size of cluster-i becomes prominently bigger than the rest. In this case, the constructed sub-graphs from these subsets connect well to the sub-graph of cluster-i, whereas the connection of the sub-graphs to the others is weakened. Although the sub-graphs can still be merged into one, the navigability of the whole graph has been undermined.

Intuitively, assigning vectors located near cluster boundaries to multiple clusters is sufficient to ensure connectivity between sub-graphs. In contrast, assigning vectors close to cluster centroids to multiple clusters does not contribute to improved connectivity. Given we expect to partition the dataset S into m subsets $\{P_1, \cdots, P_i, \cdots, P_m\}$, the corresponding centroids are $\{\pi_1, \cdots, \pi_i, \cdots, \pi_m\}$. A vector x is viewed as a boundary point if the following inequation holds

$$dist(x, \pi_i) \leq \varepsilon \cdot dist(x, \pi_j), \quad (3)$$

where $\varepsilon \geq 1.0$ is an elastic factor and π_j is the closest centroid to x. Such points are known as active points in closure k-means [46]. Boundary point x that satisfies the above inequation is assigned to cluster π_i. Unlike the modified k-means in [24], x can be assigned to a varying number of nearby clusters. Tuning ε allows us to control both the connectivity and the data redundancy across the subsets.

Nevertheless, such kind of soft-partition still causes imbalanced assignments. Some clusters may keep too many points, which leads to the low connectivity between small-sized clusters and the rest. In order to alleviate this issue, another tuning factor ρ_i is introduced for each centroid π_i. Given π_j is the closest centroid to x, x is also assigned to π_i if the following inequation holds

$$\rho_i \cdot dist(x, \pi_i) \leq \varepsilon \cdot \rho_j \cdot dist(x, \pi_j). \quad (4)$$

Both ρ_i and ρ_j are initialized to 1.0, while ε is initialized to 1.1. When the size of cluster P_i exceeds a predefined threshold, ρ_i is tuned up, resulting in fewer points being assigned to it.

In each round of the k-means like iteration, each vector x is assigned to its closest centroid π_j and the other centroids π_i that satisfy Ineqn. 4. At the end of one iteration, ε and ρ_is are updated with the following empirical formulas.

$$\varepsilon = \quad \varepsilon + (\varepsilon - 1.0) \cdot \frac{E \cdot |S|}{\sum |P_i|}, \quad (5)$$

$$\rho_i = \begin{cases} \rho_i + 0.01 & |P_i| > 1.1 \cdot \frac{E \cdot |S|}{m} \\ \rho_i - 0.01 & |P_i| < 0.9 \cdot \frac{E \cdot |S|}{m} \end{cases}, \quad (6)$$

where $E \in [1.05,\ 1.2]$ is a parameter that represents the expected data redundancy ratio.

This partition procedure is called Boundary-adaptive Balanced Partition (BBP). Compared to the standard k-means, BBP is a soft-partition procedure with multiple objectives. Firstly, it allows soft-assignment of boundary points under redundancy factor E. Secondly, it aims to balance the sizes of clusters. Finally, similar to k-means, it also tries to minimize the sum of distances from the vectors to their assigned centroids. Due to the three competing objectives, one cannot expect it to reach the same optima as k-means. Nevertheless, as a pre-processing step, it well aligns with the practical needs of building graphs for the disk-based search.

3.4 In-Memory First Search

Given a high-dimensional large-scale dataset S, it is partitioned into overlapping subsets by BBP (detailed in Section 3.3). Graph indexes are then built by Algorithm 1 on these subsets, and they are eventually merged into a complete graph index over the entire dataset S. After the construction procedure, both the graph index and the full-precision vectors are stored on SSD due to memory limitations. Specifically, the vector of each vertex is stored along with its neighborhood (as illustrated in the right-most block of Figure 1). If the neighborhood index of a vertex cannot fit within a single page (*4* KB per page), it is stored across several contiguous pages. Now we are ready to process incoming queries using disk-based NN search.

To this end, Best-First Search (BFS) can be employed to progressively locate points closer to the query. At each hop, the top-ranked vertex is expanded with the highest priority. However, BFS becomes inefficient in disk-based scenarios, as the CPU must pause at each hop to wait for I/O operations that load the corresponding neighborhood pages from the SSD. Although Beam Search employed in DiskANN [24] alleviates this issue, considerable CPU idle time is still observed during each round of NN search. To address this issue, two schemes are proposed in our search algorithm. Firstly, the algorithm preemptively sends I/O requests to the SSD to load a few pages containing the points in the ranking list maintained for query x_q. Since I/O requests can be sent concurrently, and reading a small number of random pages from an SSD takes almost the same time as a single page [24], the pre-read operation does not cause any significant additional overhead. Secondly, in the case that the neighborhood of the top-ranked vertex that has not been loaded, the algorithm scans down the ranking list to a vertex whose neighborhood is already in memory, and expands that neighborhood with the highest priority. In short, we propose an In-Memory First Search (IMF Search) as a dedicated strategy for efficient disk-based NN search.

The complete IMF Search procedure is presented in Algorithm 2. Given a query x_q, a starting point x_p is located via the secondary index (as shown in the middle block of Figure 1). Locating a good starting point helps reduce the number of hops required for disk-based NN search. This step is performed prior to the IMF Search. Two ranking lists, $\widehat{Q}$ and Q, are maintained for the query x_q. The list $\widehat{Q}$ keeps the top-ranked vertices whose distances to the query are estimated by asymmetric PQ [25], and serves as the candidate list for expansion. In contrast, list Q keeps the final NN search results and is updated whenever the full-precision vector of an expanded vertex is loaded into memory.

In each hop of NN Search, the IMF Search procedure preemptively sends I/O requests to load τ (typically $\tau = 2, 4$) pages corresponding to the top-ranked vertices in $\widehat{Q}$ (*Line 7*). The loaded pages are cached in a first-in first-out (FIFO) buffer B in memory. Subsequently, the search procedure scans $\widehat{Q}$. The vertex u (as shown in Figure 2), whose neighborhood has been loaded into B and has not been expanded, is selected for expansion (*Lines 11–12*). During

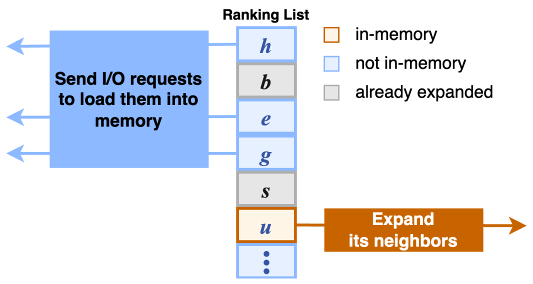


Figure 2: The illustration of In-Memory First Search (IMF Search). At this moment, vertex *u* will be expanded since it is the most top-ranked point whose neighborhood has been loaded into a cache (shown as "cached pages" in Figure 1). In contrast to Beam Search [24], we do not wait for the I/O to load the corresponding page for Vertex *h*, *e* or *g*.

Algorithm 2: In-Memory First Search

```
Data: query point $x_q$, starting point $x_p$, num. of pre-loaded
      points $\tau$
Result: $x_q$'s approximate nearest neighbors
1  begin
2      $Q \leftarrow \{x_p\}$; /**ranking list for the final search result**/;
3      $\widehat{Q} \leftarrow \{x_p\}$; /**ranking list with PQ distances**/;
4      $B \leftarrow \emptyset$; /**FIFO cache for the loaded points**/;
5      $V \leftarrow \emptyset$; /**mark the expanded points**/;
6      repeat
7          Send I/O request to load $\tau$ num. of top-ranked points
           in $\widehat{Q}$ into $B$;
8          if $(\widehat{Q} \cap B) \backslash V == \emptyset$ then
9              Waiting for I/O completed;
10         end
11         $u \leftarrow \arg\min_{x \in (\widehat{Q} \cap B) \backslash V} dist(x, x_q)$;
12         Expand $u$'s neighbors and update $\widehat{Q}$;
13         $Q \leftarrow Q \cup \{u\}$;
14         $V \leftarrow V \cup \{u\}$;
15     until Converged;
16     return $Q$;
17 end
```

expansion, asymmetric PQ distances between the query x_q and the neighbors of u are calculated, and $\widehat{Q}$ is updated accordingly. At the same time, u is inserted into Q since the full-precision vector of u is now available. Since pre-read and vertex expansion happen concurrently, it significantly reduces the CPU idle time. In the IMF Search procedure, CPU pauses occur only when no unexpanded vertex in $\widehat{Q}$ has its neighborhood cached in B.

Table 1: Overview of Datasets

Dataset	Data type	Dim.	Scale	Query
SIFT	float/uint8	128	$10^6 \sim 10^9$	10^4
DEEP	float	96	$10^6 \sim 10^8$	10^4
LAION	float	768	$10^6 \sim 10^8$	10^3

4 Experiments

This section evaluates the effectiveness of our disk-based NN search method by comparing it with SOTA disk-based approaches, including DiskANN [2], SPANN [3], and Starling [4]. To demonstrate the efficiency of our method, the construction time is reported on several high-dimensional and large-scale datasets. Additionally, the ablation studies show the effectiveness of BBP and IMF Search.

4.1 Experiment Setup

All the experiments are conducted on a workstation running Ubuntu 22.04 LTS, equipped with dual *20* cores *2.50* GHz CPUs, *256* GB of memory, and a *4* TB NVMe SSD with a maximum sequential read/write speed of *3500* MB/s. Three public datasets covering a variety of scales—SIFT [29], DEEP [2], and LAION [42]—are adopted for evaluation. The small-scale configuration (memory-based) contains millions of vectors, while the large-scale configuration (disk-based) contains hundreds of millions to billions of vectors. The brief information about these datasets is summarized in Table 1. *Recall@k* is used to measure the performance of *k*-NN search. Unless otherwise specified, we set $k = 10$ as the parameter for recall. The search speed is measured by Latency and Queries Per Second (QPS). Latency is measured as the query response time in milliseconds, while QPS is used to measure the throughput of a search method per second. The performance of our method is evaluated in both single-thread and multi-thread contexts. Latency is adopted as the efficiency measure for the single-thread scenario, while QPS is adopted for the multi-thread scenario.

4.2 Disk-based NN Search

In this experiment, the performance of our method (given as Disk XN-Graph) is studied in comparison to three disk-based methods DiskANN, SPANN, and Starling. For DiskANN and Starling, we set the parameters as $R = 64$ and $L = 256$ to construct their indexes. For SPANN, we follow the recommended parameters provided in its GitHub repository. Due to memory limitations, we were unable to build the index of Starling on SIFT1B, and SPANN failed to construct indexes for both SIFT1B and LAION100M.

The search performances in single-thread and multi-thread contexts are shown in Figure 3 and Figure 4 respectively. As shown in the figures, Disk XN-Graph exhibits *1.5* to *3* times lower latency than the others when their *Recall@10* is on the same level. Similar improvement in QPS is observed under the *16*-thread scenario. Furthermore, we also compare the time costs of graph construction spent by all four methods in Figure 5. The construction time of Disk XN-Graph is *8* to *20* times lower than that of DiskANN or Starling.

[2]https://github.com/microsoft/DiskANN
[3]https://github.com/microsoft/SPTAG
[4]https://github.com/zilliztech/starling

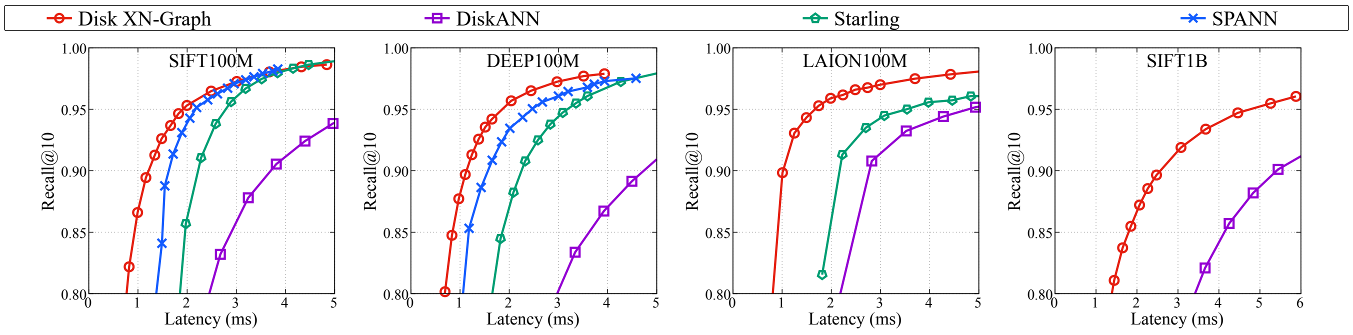


Figure 3: Recall@10 vs. latency in single-thread scenario.

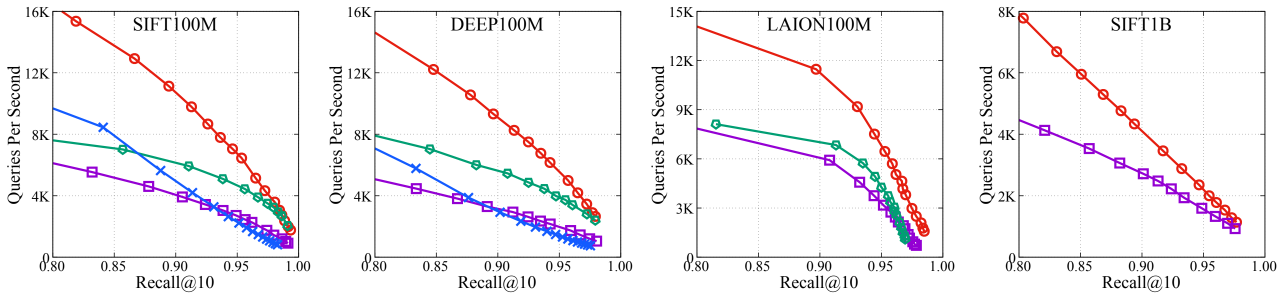


Figure 4: QPS vs. Recall@10 in *16* threads scenario.

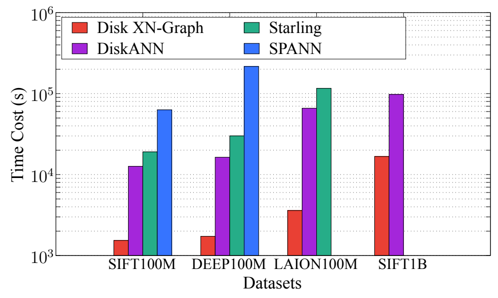


Figure 5: The comparison of graph index construction time costs among *4* methods.

Compared to SPANN, the time cost is even further reduced. In the following sections, we further analyze the contribution of each of our proposals to the superior performance of Disk XN-Graph.

4.3 All-in-memory NN Search

In this experiment, the NN search is conducted on two million-scale datasets. Both the graph index and the vectors are kept in memory. The effectiveness of XN-Graph is compared to HNSW[5], Vamana [24], and RNND [6] with the same Best-First Search procedure. To ensure fairness in the comparison, we adopt the recommended parameters from their GitHub repositories and set the maximum out-edges for all methods to *64*.

[5]https://github.com/nmslib/hnswlib
[6]https://github.com/mti-lab/rnn-descent

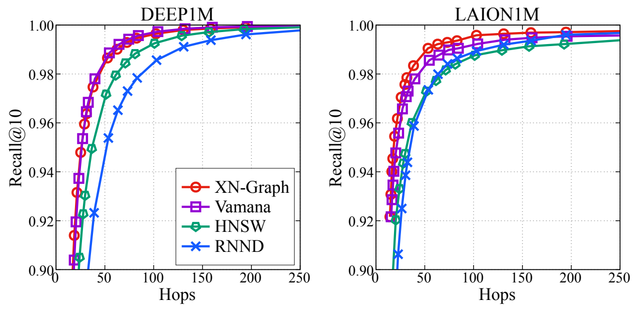
The search performance is evaluated in terms of both "hops vs. recall" (Figure 6(a)) and "latency vs. recall" (Figure 6(b)). The former studies whether XN-Graph helps to save the number of hops while maintaining the same level of search quality. The latter shows XN-Graph induces no extra comparisons when enlarging the coverage of the neighborhood. As shown in the figures, the number of hops in XN-Graph is significantly fewer than that of HNSW and RNND. It also outperforms Vamana slightly. Moreover, as shown in Figure 7, the construction of XN-Graph is at least *5* times faster than HNSW and Vamana, while about *1.5* times faster than RNND. Due to the small value of ω in Algorithm 1, it even considerably outperforms KGraph [7], which is the original implementation of NN-Descent [10].

4.4 Ablation Study

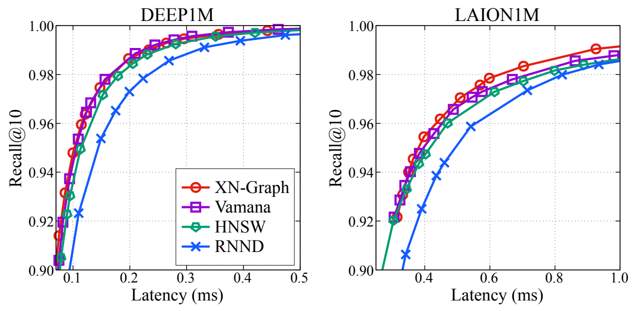
In this section, a series of experiments are conducted on the DEEP100M dataset to show the behavior of BBP (presented in Section 3.3) and the effectiveness of IMF Search (presented in Section 3.4).

In order to study the behavior of BBP, two sets of experiments are pulled out. In the first set of experiments, we study the relation between the data redundancy introduced by BBP and the search latency. We fix the number of partitions to *10* and vary the redundancy rate parameter E gradually from *1.05* to *2.0*. The trend of search latency when varying E is shown in Figure 8(a). The results indicate that when E increase from *1.05* to *1.1*, the search latency undergoes a significant drop. This is simply because the connectivity between different subsets is increased due to the higher E. However,

[7]https://github.com/aaalgo/kgraph



(a) Hops vs. Recall



(b) Latency vs. Recall

Figure 6: Latency and hops vs. Recall@10 comparison for all-in-memory methods in a single-threaded scenario.

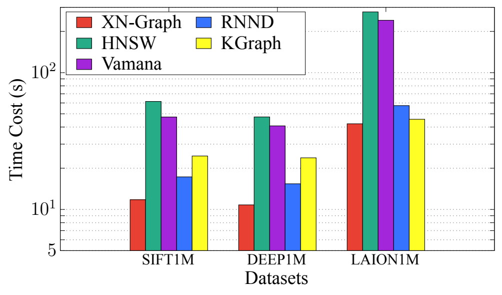
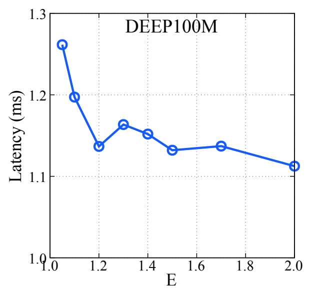


Figure 7: The comparison of index construction time.

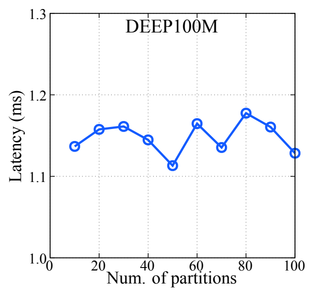
when $E \geq 1.2$, the variance in latency is minor. Therefore, we fix E to *1.2* for all the experiments. In the second set of experiments, we fix E at *1.2* and vary the number of partitions from *10* to *100*. The trend is shown in Figure 8(b). As we can see, the number of partitions has minor impact on the search performance as long as the redundancy rate is properly set.

In order to validate the effectiveness of IMF Search, we compare it with the Best-First Search (BFS) and Beam Search in terms of recall and latency. Beam Search is tested with Beam Widths *2* and *4*, while the parameter τ of IMF Search is also configured to *2* and *4*. The disk-based NN search for all three search methods is pulled out on the same XN-Graph. The results are shown in Figure 9.

As shown in the figure, BFS shows much higher latency than Beam Search and IMF Search. It is not surprising as BFS incurs



(a) Varying redundancy rates



(b) Varying the num. of partitions

Figure 8: The search performance trend when varying the redundancy rate and the number of partitions in BBP in a single-thread scenario.

significant CPU idle time waiting for SSD responses. Although Beam Search reduces CPU idle time to some extent, it still suffers from noticeable CPU pause. In contrast, IMF Search reduces CPU idle time to even lower level since it will expand the in-memory vertex first when the top-ranked neighborhoods are on the way of loading. This leads to the substantially lower latency than Beam Search when they are under the same beam width configurations.

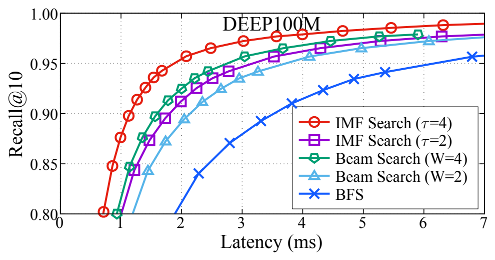


Figure 9: Comparison of search efficiency between three methods in a single-thread scenario.

5 Conclusion

We have presented a novel solution for the disk-based NN search. The innovation covers both the offline graph index construction and the online search that is specifically designed for disk-based NN search. For graph index construction, the XN-Graph is proposed. It is able to collect diversified neighbors from a much wider coverage. Experiments show that it achieves the best performance on the disk-based as well as the all-in-memory NN search. Moreover, such graph index can be built at very high speed. To the best of our knowledge, it is the most efficient graph index construction method in the literature. For the online disk-based NN search, a novel algorithm called IMF Search is proposed. Compared to the widely used Beam Search, the priority of expansion has been shifted from top-ranked vertices to the vertices just in memory. Such innovation leads to *1.5-3* times lower latency. Overall, our method outperforms the SOTA methods on the billion-scale search task by at least *50%* even on the high recall level.

References

[1] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2019. Quicker ADC: Unlocking the hidden potential of product quantization with SIMD. IEEE transactions on pattern analysis and machine intelligence 43 (2019), 1666–1677.

[2] 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.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In Proceedings of the ACM SIGMOD international conference on Management of data. 322–331.

[4] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. Commun. ACM 18 (1975), 509–517.

[5] Patrick Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. 2023. FINGER: Fast Inference for Graph-based Approximate Nearest Neighbor Search. In Proceedings of the ACM Web Conference 2023. 3225–3235.

[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. Advances in Neural Information Processing Systems 34 (2021), 5199–5212.

[8] Jian Cheng, Cong Leng, Jiaxiang Wu, Hainan Cui, and Hanqing Lu. 2014. Fast and Accurate Image Matching with Cascade Hashing for 3D Reconstruction. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 1–8.

[9] D.W. Dearholt, N. Gonzales, and G. Kurup. 1988. Monotonic Search Networks For Computer Vision Databases. In Twenty-Second Asilomar Conference on Signals, Systems and Computers, Vol. 2. 548–553.

[10] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In Proceedings of the 20th international conference on World wide web. 577–586.

[11] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. In Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. 6491–6501.

[12] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Qian Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. 1995. Query by image and video content: the QBIC system. Computer 28 (1995). https://doi.org/10.1109/2.410146

[13] Cong Fu and Deng Cai. 2016. EFANNA : An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph. https://arxiv.org/abs/1609.07228

[14] Cong Fu, Changxu Wang, and Deng Cai. 2021. High Dimensional Similarity Search with Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. IEEE Transactions on Pattern Analysis and Machine Intelligence 44, 8 (2021), 4139–4150.

[15] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. Proceedings of VLDB Endowment 12, 5 (2019), 461–474.

[16] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. Proceedings of the ACM on Management of Data 2 (2024), 1–27.

[17] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. https://arxiv.org/abs/2312.10997

[18] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2946–2953.

[19] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. 2023. Filtered-DiskANN: Graph algorithms for approximate nearest neighbor search with filters. In Proceedings of the ACM Web Conference 2023. 3406–3416.

[20] Kiana Hajebi, Yasin Abbasi-Yadkor, Hossein Shahbazi, and Hong Zhang. 2011. Fast Approximate Nearest-Neighbor Search with k-Nearest Neighbor Graph. In Proceedings of International Joint Conference on Artificial Intelligence. 1312–1317.

[21] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In Proceedings of IEEE Conference on Computer Vision and Pattern Recognition. 5713–5722.

[22] Oz Huly, Idan Pogrebinsky, David Carmel, Oren Kurland, and Yoelle Maarek. 2024. Old IR Methods Meet RAG. In Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval. 2559–2563.

[23] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In Proceedings of the 13th Annual ACM Symposium on Theory of Computing. 604–613.

[24] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. Advances in Neural Information Processing Systems 32 (2019).

[25] 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.

[26] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollar, and Ross Girshick. 2023. Segment Anything. In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV). 4015–4026.

[27] Wuchao Li, Chao Feng, Defu Lian, Yuxin Xie, Haifeng Liu, Yong Ge, and Enhong Chen. 2023. Learning Balanced Tree Indexes for Large-Scale Vector Retrieval. In Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. 1353–1362.

[28] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement. IEEE Transactions on Knowledge and Data Engineering 32 (2020), 1475–1488.

[29] David G Lowe. 2004. Distinctive image features from scale-invariant keypoints. International journal of computer vision 60 (2004), 91–110.

[30] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In Proceedings of the 33rd International Conference on Very Large Data Bases. 950–961.

[31] James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In Proceedings of the fifth Berkeley symposium on mathematical statistics and probability. 281–297.

[32] 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.

[33] Yitong Meng, Xinyan Dai, Xiao Yan, James Cheng, Weiwen Liu, Jun Guo, Benben Liao, and Guangyong Chen. 2020. PMD: An optimal transportation-based user distance for recommender systems. In Advances in Information Retrieval: 42nd European Conference on IR Research, ECIR 2020, Lisbon, Portugal, April 14–17, 2020, Proceedings, Part II 42. 272–280.

[34] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. IEEE Transactions on Pattern Analysis and Machine Intelligence 36 (2014), 2227–2240.

[35] Javier Vargas Munoz, Marcos A Gonçalves, Zanoni Dias, and Ricardo da S Torres. 2019. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. Pattern Recognition 96 (2019), 106970.

[36] Naoki Ono and Yusuke Matsui. 2023. Relative NN-Descent: A Fast Index Construction for Graph-Based Approximate Nearest Neighbor Search. In Proceedings of the 31st ACM International Conference on Multimedia. 1659–1667.

[37] Ezgi Can Ozan, Serkan Kiranyaz, and Moncef Gabbouj. 2016. Competitive Quantization for Approximate Nearest Neighbor Search. IEEE Transactions on Knowledge and Data Engineering 28 (2016), 2884–2894.

[38] Yu Pan, Jianxin Sun, and Hongfeng Yu. 2023. LM-DiskANN: Low Memory Footprint in Disk-Native Dynamic Graph-Based ANN Indexing. In 2023 IEEE International Conference on Big Data (BigData). 5987–5996.

[39] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. Proc. ACM Manag. Data 1, 1 (2023), 1–27.

[40] Bryan Russell, Antonio Torralba, Ce Liu, Rob Fergus, and William Freeman. 2007. Object Recognition by Scene Alignment. In Advances in Neural Information Processing Systems.

[41] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In Proceedings of the 10th International Conference on World Wide Web. 285–295.

[42] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, Patrick Schramowski, Srivatsa Kundurthy, Katherine Crowson, Ludwig Schmidt, Robert Kaczmarczyk, and Jenia Jitsev. 2022. LAION-5B: An open large-scale dataset for training next generation image-text models. In Advances in Neural Information Processing Systems, Vol. 35. Curran Associates, Inc., 25278–25294.

[43] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. https://arxiv.org/abs/2105.09613

[44] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. 2023. SOAR: Improved Indexing for Approximate Nearest Neighbor Search. In Advances in Neural Information Processing Systems. 3189–3204.

[45] Di Wang, Jing Zhang, Bo Du, Minqiang Xu, Lin Liu, Dacheng Tao, and Liangpei Zhang. 2024. Samrs: Scaling-up remote sensing segmentation dataset with segment anything model. Advances in Neural Information Processing Systems 36 (2024), 8815–8827.

[46] Jing Wang, Jingdong Wang, Qifa Ke, Gang Zeng, and Shipeng Li. 2015. Fast approximate k-means via cluster closures. Multimedia Data Mining and Analytics (2015), 373–395.

[47] 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 (2024), 1–27.

[48] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. In Proceedings of the VLDB Endowment, Vol. 14. 1964 – 1978. Issue 11.

[49] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In Proceedings of the 29th Symposium on Operating Systems Principles. 545–561.

[50] Diji Yang, Jinmeng Rao, Kezhen Chen, Xiaoyuan Guo, Yawen Zhang, Jie Yang, and Yi Zhang. 2024. IM-RAG: Multi-Round Retrieval-Augmented Generation Through Learning Inner Monologues. In Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval. 730–740.

[51] Wan-Lei Zhao, Hui Wang, and Chong-Wah Ngo. 2022. Approximate k-NN Graph Construction: A Generic Online Approach. IEEE Transactions on Multimedia 24 (2022), 1909–1921.

[52] Chun Jiang Zhu, Tan Zhu, Haining Li, Jinbo Bi, and Minghu Song. 2019. Accelerating Large-Scale Molecular Similarity Search through Exploiting High Performance Computing. In 2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). 330–333. https://doi.org/10.1109/BIBM47256.2019.8982950