Select Edges Wisely: Monotonic Path Aware Graph Layout Optimization for Disk-based ANN Search

Ziyang Yue
Huazhong University of Science and Technology

Bolong Zheng*
Huazhong University of Science and Technology

Ling Xu
Shuyi Technology

Kanru Xu
Shuyi Technology

Shuhao Zhang
Huazhong University of Science and Technology

Yajuan Du
Wuhan University of Technology

Yunjun Gao
Zhejiang University

Xiaofang Zhou
HKUST

Christian S. Jensen
Aalborg University

ABSTRACT

Approximate nearest neighbor (ANN) search is a critical problem in various real-world applications. However, as one of the most promising solutions to ANN search, graph-based indexes often suffer from high memory consumption. Although a few studies attempt to alleviate this issue by storing the index on inexpensive disk storage, they still face challenges such as insufficient data locality and low efficiency when optimizing the graph layout on disk. Therefore, we propose MARGO, a monotonic path-aware graph layout optimization method for disk-based ANN search. First, we present the essence of graph layout optimization in disk-based ANN search, and design a monotonic path-aware objective function that weighs the edges based on their importance in monotonic paths, supported by rigorous theoretical analysis. Second, we propose a greedy algorithm that prioritizes high-weight edges to accommodate more monotonic paths. To enhance efficiency, MARGO introduces a two stage decoupling method that processes intra-cluster edges in parallel first, followed by inter-cluster edges. Third, we develop a weight computation strategy that computes edge weights on-the-fly during index construction with almost no additional overhead. A comprehensive experimental study demonstrates that MARGO improves search efficiency by up to 26.6% while maintaining the same recall, and accelerates the graph layout optimization by up to 5.5×.

PVLDB Reference Format:
Ziyang Yue, Bolong Zheng, Ling Xu, Kanru Xu, Shuhao Zhang, Yajuan Du, Yunjun Gao, Xiaofang Zhou, and Christian S. Jensen. Select Edges Wisely: Monotonic Path Aware Graph Layout Optimization for Disk-based ANN Search. PVLDB, 18(11): 4337 - 4349, 2025.
doi:10.14778/3749646.3749697

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CodenameYZY/MARGO.

*Bolong Zheng is the corresponding author

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749697

1 INTRODUCTION

Recent advancements in Large Language Model (LLM) revolutionize various domains, from natural language processing [6] to computer vision [40]. LLM demonstrates remarkable performance in generating, understanding, and reasoning with human language, which is powered by vectorized data representations. Vast cross-modal data, such as text, images, and videos, are embedded into high dimensional vectors, correlating by the distances between their vector representations. Approximate nearest neighbor (ANN) search plays a critical role in LLM by facilitating the discovery of relevant knowledge for inference [3, 24]. It is also a fundamental function in various traditional tasks, including image search [31], information retrieval [20], and recommendation systems [27]. So far, various ANN search methods are proposed, including tree-based [1, 5, 12], hashing-based [21, 39, 44, 46, 48, 49], quantization-based [16, 22, 47], and graph-based [29, 30, 33, 45] methods. Among these methods, graph-based methods are regarded as one of the most promising directions, exhibiting both high efficiency and accuracy in ANN search [4, 25, 26, 38].

Despite the advantages, graph-based methods, or alternatively graph-based indexes, often suffer from excessive memory footprint. This is because they require storing both the graph index and raw vectors in memory. As a result, graph-based indexes struggle to fit into main memory and cannot scale well as the number of vectors increases. For instance, it requires up to 1100GB of memory to build an in-memory Vamana graph for billion-scale datasets [37], which goes beyond what a single machine can accommodate. To address this issue, prominent vector databases [17, 42] distribute and index the vectors onto different search nodes, instead of building an entire index, where each search node performs ANN search independently, and the results from all nodes are merged to generate final ANN search results. However, it still suffers from high cost due to extensive memory usage in commercial databases. Therefore, it is critical to develop a disk-based graph index that stores vectors and performs ANN search on inexpensive disk storage.

Existing disk-based graph indexes focus on reducing random disk I/Os in ANN search to improve efficiency. DiskANN [37] employs Product Quantization [22] and uses approximate distances to guide the search path. Since the computation of such approximate distances does not require loading the vectors from the disk, a large

number of random disk I/Os are hence avoided. However, it sequentially stores vectors on the disk based on IDs, resulting in poor data locality. Starling [43] enhances data locality by optimizing the layout of the graph index. It stores each vertex and its neighbors in a same page as much as possible. This approach ensures that a single random I/O can access more relevant vertices. Despite the improvements, they still face two major challenges:

- **Insufficient data locality.** When optimizing the graph layout, Starling only considers the benefit of storing a pair of neighbors in the same page from the perspective of a single hop in the search path, so that each pair of neighbors is supposed to be equally important. This lacks the perspective of the entire search path and results in a tunnel vision. In fact, the importance varies across pairs of neighbors, among which those frequently passed by search paths offer greater potential in reducing random disk I/Os. Therefore, it is essential to design a graph layout optimization strategy that takes the entire search path into account.
- **Low efficiency of graph layout optimization.** Starling optimizes the graph layout in an iterative manner, where the number of iterations controls the trade-off between data locality and efficiency. To achieve a high data locality, a large number of iterations is required, which leads to low efficiency. The efficiency of graph layout optimization can be greatly enhanced if it is completed in a single pass.

To address above challenges, we propose MARGO, a monotonic path-aware graph layout optimization method for disk-based ANN search that achieves both high data locality and efficiency. We begin by rethinking the graph layout optimization (GLO) problem from a novel perspective: its essence lies in selecting edges, not vertices. Based on this perspective, MARGO employs a monotonic path-aware objective function that aims to accommodate most important edges. Each edge is delicately weighed through rigorous theoretical analysis, to quantify its importance in the entire search path. Next, we prove that the GLO problem is equivalent to the minimum n_p-cut problem with equal-size constrains on subsets, which is known to be NP-hard. We introduce a greedy algorithm as the solution, which prioritizes high-weight edges to accommodate more monotonic paths. To improve efficiency and fully leverage the parallel capabilities of multi-core processors, we propose two stage decoupling, where intra-cluster edges are processed in parallel first, followed by the inter-cluster edges. Finally, we propose an on-the-fly weight computation strategy during index construction. By reusing the computed distances, it obtains the edge weights with almost no additional overheads.

The main contributions are summarized as follows:

- We offer a novel perspective on the graph layout optimization problem, and design a monotonic path-aware objective function based on rigorous theoretical analysis.
- We prove GLO problem is NP-hard, and introduce a greedy algorithm as the solution. To enhance efficiency, we propose two stage decoupling tailored to the nature of the graph index.
- We introduce a weight computation strategy that obtains edge weights on-the-fly during index construction. It avoids computationally expensive distance computations by completely reusing the already computed distances.
- We conduct extensive experiments in 4 real-world datasets, thoroughly verifying the accuracy and efficiency of MARGO.

2 PRELIMINARIES

2.1 Problem Setting

We focus on the k nearest neighbor (kNN) search problem for high dimensional vectors, which is formally defined as follows.

Definition 1 (k nearest neighbor (kNN) search). *Given a set of d-dimensional vectors $D = \{p_1, \ldots, p_n\} \subseteq \mathbb{R}^d$, a query vector q, and a positive integer k, the kNN search aims to find a set of k vectors $S_k \subseteq D$ such that for $\forall p_i \in S_k$ and $\forall p_j \in D \setminus S_k$, $dist(p_i, q) \leq dist(p_j, q)$, where $dist(\cdot, \cdot)$ denotes the distance function.*

However, solving the exact kNN search problem in high dimensional space is time-consuming. Therefore, recent studies trade off between accuracy and efficiency, seeking an approximate solution instead. This is formally referred to as k approximate nearest neighbor (kANN) search. The most widely adopted metric to evaluate the accuracy of kANN search is *Recall@k*, which is computed as the ratio of true nearest neighbors in the returned set over k. Specifically, given the set of returned k ANNs $\hat{S}_k$, $Recall@k = |\hat{S}_k \cap S_k|/k$, where S_k is the set of true nearest neighbors.

2.2 ANN Search in Vector Databases

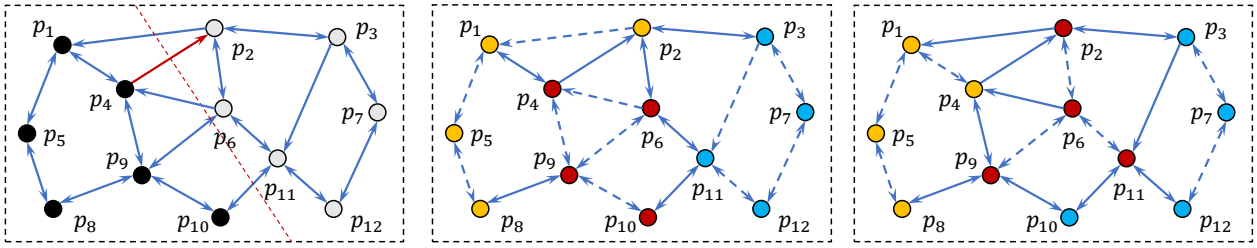
In real-world applications, vector collections can exceed billions in scale. Managing massive vectors within a single index proves impractical due to severe memory bottlenecks and lack of essential system features [17, 42]. For instance, building an entire in-memory Vamana graph for billion-scale datasets consumes 1100GB memory and takes over five days [37], severely degrading system availability. Moreover, a single index fails to deliver critical capabilities such as scalability, load balancing, and fault tolerance [17].

To overcome this, leading vector databases [17, 42] partition vectors into segments distributed across search nodes. Each node hosts multiple segments, each managing millions of vectors under strict resource constraints—typically less than 2GB of memory and 10GB of disk space [17, 42]. Upon receiving a query, a query coordinator determines the relevant segments to activate for ANN search, whose results are merged to produce the final output.

Existing ANN search methods struggle in the segment-based scenario, each facing distinct limitations. Among in-memory methods, graph-based methods such as HNSW [30] and NSG [14] become prohibitively costly at very large vector collections [8, 37]. Quantization-based methods like Faiss [11] suffer from a substantial drop in accuracy. For example, the *recall@1* is only about 0.5 when memory is far smaller than the vector collections [37]. Among disk-based methods, SPANN [8] duplicates vectors extensively, incurring excessive disk usage and violating the resource constraints of segments. DiskANN [37] and Starling [43] yield unsatisfactory search efficiency due to insufficient data locality. Therefore, it is crucial to design a disk-based ANN search method that balances search efficiency and disk usage, while also enhancing data locality.

2.3 Graph-based Index

Proximity graph (PG). The graph index is regarded as the most promising approach due to its high accuracy, efficiency, and low disk



(a) The short edge priority rule of SNG (b) Plan A for the graph layout (c) Plan B for the graph layout

Figure 1: A toy example

overhead [37, 43]. The core idea of the graph index is to construct a proximity graph $G = (V, E)$, which can be either directed or undirected, to represent the vectors. V is the vertex set, where there is a one-to-one correspondence between the vertices and the vectors. E is the edge set that is generated by a delicately designed rule. Each edge represents the neighborhood relationship between the two connected vertices.

When the query arrives, the graph index performs a greedy search strategy to find the approximate nearest neighbor. Specifically, the search starts from an entry vertex, which can be either fixed or selected by some mechanisms. Then, during each hop, it evaluates the out-neighbors of the current vertex, and moves to the one with the smallest distance to the query. If the current vertex is closer to the query than any of its out-neighbors, it is returned as the search result, and the search terminates.

Monotonic search network (MSNET). MSNET [10] is a subset of PG. Given a proximity graph $G = (V, E)$, it is an MSNET iff. the following condition is satisfied. For $\forall p_i, p_j \in V$, there exists a path $[v_1, \ldots, v_l]$ such that $dist(v_i, v_l) > dist(v_{i+1}, v_l)$ holds for $\forall i = 1, \ldots, l-1$, where $v_1 = p_i, v_l = p_j$. Such a path is called a monotonic path from p_i to p_j, and is denoted as $MP(p_i, p_j)$.

This property ensures that the greedy search starting from an arbitrary entry vertex can find the nearest neighbor without backtracking. Hence, it provides theoretical support for accuracy of the MSNET-based graph index.

Sparse neighborhood graph (SNG). However, to achieve the above property, MSNET often requires excessive edges to form enough monotonic paths, which results in low search efficiency. SNG [2] is a well-adopted directed graph index with MSNET property. The core idea is the short edge priority rule, where short edges occlude long edges. When selecting a vertex p's out-neighbors during index construction, the other vertices are evaluated in the ascending order of their distances to p. Given that p^* is already selected as p's out-neighbor, i.e., the edge $(p, p^*) \in E$, for another vertex p', if $dist(p', p^*) < dist(p', p)$, then (p, p') cannot exist. In this case, we say that (p, p^*) occludes (p, p'), or alternatively, (p, p^*) occludes p' for p. With this rule, redundant long edges are excluded, so SNG retains the property of MSNET with fewer edges.

Example 1. *Fig. 1(a) illustrates an SNG in 2D space with 12 vertices, i.e., $\{p_1, \ldots, p_{12}\}$. When selecting the out-neighbors of p_4, p_1 and p_9 are selected first, because they are the top-2 closest vertices to p_4. Then, p_2 is selected and the edge (p_4, p_2) (in red) is connected, whose perpendicular bisector is denoted by a dashed red line. Since the vertices to the right of this line (in gray) are closer to p_2 than to p_4, e.g., $dist(p_6, p_2) < dist(p_6, p_4)$, (p_4, p_2) occludes these vertices for p_4, except the already connected p_2. Therefore, p_4 has no out-edges towards these vertices.*

2.4 Graph Layout Optimization

Graph layout on disk. For the disk-based graph index, the storage format of a vertex on disk includes its corresponding vector, along with the number and IDs of its out-neighbors. Given a graph index with the maximum out-degree of R, which is often set to a small constant in practice, the length of each vertex's out-neighbor ID list is padded to R for alignment purposes. As a result, each vertex occupies the same amount of disk space, denoted as s_v bytes. Since a page is the smallest disk I/O unit, a vertex is not split across multiple pages. Given that the page size is s_p bytes, a page can accommodate up to $n_v = \lfloor s_p / s_v \rfloor$ vertices. Based on the disk graph layout format, we give a general definition of the graph layout optimization problem.

Definition 2 (Graph Layout Optimization (GLO)). *Given a proximity graph $G = (V, E)$, GLO aims to assign the n vertices to n_p pages, where each page contains at most n_v vertices. The assignment seeks to maximize an objective function $\mathcal{F}$ subject to constraints C.*

We intentionally omit the specific form of $\mathcal{F}$ and C here to allow for flexible problem settings, which is detailed in the next section.

3 MONOTONIC PATH AWARE OBJECTIVE FUNCTION

We proceed to introduce a monotonic path-aware objective function. We first analyze the essence of GLO. Then, we depict the objective function design in detail.

3.1 The Essence of GLO

Before analyzing the essence of GLO, we explain how it benefits the disk-based graph index. For a disk-based graph index, the bottleneck in search efficiency is the time-consuming random disk I/O, which occurs when the search hops from the current vertex to its out-neighbor located in a different page. On the contrary, if the search hops to an out-neighbor located in the same page as the current vertex, the need for a random disk I/O is avoided.

Table 1: The search paths and numbers of random disk I/Os of different queries

Query	Search path	# of I/Os		Query	Search path	# of I/Os		Query	Search path	# of I/Os	
		Plan A	Plan B			Plan A	Plan B			Plan A	Plan B
p_1	$[p_6, p_4, p_1]$	2	2	p_5	$[p_6, p_9, p_8, p_5]$	2	2	p_9	$[p_6, p_9]$	1	1
p_2	$[p_6, p_2]$	2	**1**	p_6	$[p_6]$	1	1	p_{10}	$[p_6, p_{11}, p_{10}]$	2	2
p_3	$[p_6, p_2, p_3]$	3	**2**	p_7	$[p_6, p_{11}, p_{12}, p_7]$	2	2	p_{11}	$[p_6, p_{11}]$	2	**1**
p_4	$[p_6, p_4]$	**1**	2	p_8	$[p_6, p_9, p_8]$	2	2	p_{12}	$[p_6, p_{11}, p_{12}]$	2	2

Example 2. *Fig. 1(b) illustrates a graph layout plan of the SNG in Fig. 1(a), where the 12 vertices are assigned to 3 pages, with each page containing 4 vertices. Vertices in the same color belong to the same page. Assume that the current vertex is p_6. If the search hops to p_2, a random disk I/O is triggered to load the page $\{p_1, p_2, p_5, p_8\}$. However, if it hops to p_4, there is no need to load the page $\{p_4, p_6, p_9, p_{10}\}$, as it is already loaded when p_6 is accessed.*

Therefore, GLO benefits the disk-based graph index by making hops happen within a page as much as possible. Since a hop is equivalent to passing an edge in PG, we contend that **the essence of GLO is to select edges, so that the number of random disk I/Os is minimized when both endpoints of each selected edge are assigned to the same page**.

However, Starling [43] presents this problem from a vertex-wise perspective. It aims to assign each vertex and its out-neighbors to the same page. The objective function of Starling is designed as the average ratio of each vertex's (say p) out-neighbors in the same page as p over all vertices in this page except p itself. That is:

$$\mathcal{F}_{Starling} = \frac{1}{|V|} \sum_{p \in V} \frac{|B(p) \cap N(p)|}{|B(p)| - 1}, \tag{1}$$

where $N(p)$ is the set of p's out-neighbors, and $B(p)$ represents the page of p. The constraints $C_{Starling}$ are as follows. First, for all $B(p_i), |B(p_i)| = n_v$, which means that all pages are full. Second, each vertex is assigned to exactly one page. However, we prove that such a vertex-wise optimization objective is equivalent to one selecting edges with each edge having the same importance.

Lemma 1. *The optimization objective of Starling is essentially equivalent to maximizing the number of edges whose both endpoints are assigned to a same page.*

Proof. For all $p_i, p_j, |B(p_i)| = |B(p_j)| = n_v$, hence

$$\begin{aligned} \arg\max_{B(\cdot)} \mathcal{F}_{Starling} &\Leftrightarrow \arg\max_{B(\cdot)} \sum_{p \in V} |B(p) \cap N(p)| \\ &= \arg\max_{B(\cdot)} \sum_{p \in V} \sum_{p^* \in N(p)} \mathbb{I}_{B(p)=B(p^*)}(p, p^*) \\ &= \arg\max_{B(\cdot)} \sum_{(p,p^*) \in E} \mathbb{I}_{B(p)=B(p^*)}(p, p^*), \end{aligned} \tag{2}$$

where $\mathbb{I}_{B(p)=B(p^*)}(p, p^*)$ is an indicator function. When p and p^* are in the same page, its value is 1. Otherwise, it is 0. □

Given $p_i^*, p_j^* \in N(p)$, either selecting (p, p_i^*) or selecting (p, p_j^*) equally contributes to the objective function by 1, which corresponds to assigning p_i^* or p_j^* to the same page as p, respectively. Therefore, Starling selects edges under the assumption that all edges are equally important. However, we argue that the edges should not be considered as equally important, which is elaborated in the next section.

3.2 Objective Function Design

Intuitively, the importance of edges varies. Specifically, some edges are frequently passed by search paths, while others are rarely visited. Compared to rarely visited edges, it would improve greatly the search efficiency by selecting frequently passed edges and assigning both endpoints to the same page.

Intuition of MARGO. Fig. 1(b) and 1(c) present two different graph layout plans. Dashed edges represent those whose both endpoints are assigned to the same page. Solid edges represent those whose both endpoints are assigned to different pages. In plans A and B, the numbers of dashed edges are 19 and 16, respectively. Since SNG is directed, forward and backward edges are considered distinct (e.g., (p_2, p_6) and (p_6, p_2) are different edges). Since $19 > 16$, if evaluated by $\mathcal{F}_{Starling}$ (Eq. 2), plan A outperforms plan B.

However, if we take p_6 as the entry vertex, and $\{p_1, \ldots, p_{12}\}$ as queries, plan B requires fewer random disk I/Os in total than plan A. The search path and number of random disk I/Os for each query are shown in Table 1. Compared to plan A, plan B increases the number of random disk I/Os by 1 for p_4, but reduces the number of random disk I/Os by 1 for p_2, p_3, p_{11}. This is because the edge selected in plan B are passed by search paths more frequently. For example, (p_6, p_{11}) is passed in four queries, i.e., $p_7, p_{10}, p_{11}, p_{12}$, while (p_9, p_{10}) is passed in no queries. Therefore, (p_6, p_{11}) is more important than (p_9, p_{10}), and is able to reduce more random disk I/Os when selected.

Based on this intuition, we contend that the edges should be weighed and selected based on the frequencies with which they are passed by search paths. The challenge lies in how to quantify such frequency for an arbitrary edge (p, p^*). To address this, we first analyze a special case, and then extend the analysis to general cases. The following analysis is under the assumption that the queries and indexed vectors follow the same distribution.

Search from p. Specially, we consider the case that p is the entry vertex. To facilitate the analysis, we begin with the definition of Monotonically Reach.

Definition 3 (Monotonically Reach). *Given an edge (p, p^*) and a vertex p', if there exists a monotonic path $MP(p, p')$ that passes (p, p^*), we say that (p, p^*) can monotonically reach p'.*

According to the greedy search strategy, the search path definitely follows a monotonic path. Therefore, given an edge (p, p^*), we use the number of vertices that it can monotonically reach, denoted as $m(p, p^*)$, to quantify the frequency with which it is

passed by search paths in the special case. The more vertices that it can monotonically reach, it is more likely to be passed during the search. Next, we derive the number of vertices that (p, p^*) can monotonically reach in an SNG by Lemmas 2 and 3.

Lemma 2. *In an SNG, a monotonic path of the form* $[p, p^*, \ldots, p']$ *exists iff.* (p, p^*) *occludes* (p, p').

Proof. **Necessity.** If the path $[p, p^*, \ldots, p']$ is a monotonic path, then we have $dist(p^*, p') < dist(p, p')$. According to the short edge priority rule of SNG, (p, p^*) occludes (p, p'). **Sufficiency.** If (p, p^*) occludes (p, p'), we have $dist(p, p') > dist(p^*, p')$. The property of SNG guarantees that there exists a monotonic path from p^* to p', denoted as $[v_1, \ldots, v_l]$, where $v_1 = p^*$, and $v_l = p'$. It holds that $dist(p, v_l) > dist(v_1, v_l) > \cdots > dist(v_{l-1}, v_l)$. Therefore, $[p, v_1, \ldots, v_l] = [p, p^*, \ldots, p']$ is a monotonic path. □

Lemma 3. *In an SNG, for an arbitrary edge* (p, p^*)*, the number of vertices that it can monotonically reach equals to the number of edges it occludes, plus one.*

Proof. We address the cases separately depending on the length of the monotonic path. **Case 1.** For a monotonic path of length 1, there is only one such path, namely $[p, p^*]$. **Case 2.** For a monotonic path of length greater than 1, which takes the form $[p, p^*, \ldots, p']$, Lemma 2 implies that each such p' corresponds to an edge occluded by (p, p^*). To sum up, the number of vertices that (p, p^*) can monotonically reach equals to the number of edges it occludes, plus one. □

Search to p. It does not always hold that p is the entry vertex. Therefore, we consider the process before p is reached by the search. Once the search reaches p, the problem reduces to the special case discussed above. Given that the number of vertices that (p, p^*) can monotonically reach remains unchanged, the more likely p is to be reached by the search, the more frequently that (p, p^*) is passed. To quantify this likelihood, we employ the number of edges that can monotonically reach p, denoted as $m(p)$. This is because when p is the query, once the search passes one of these edges, it is guaranteed to reach p. We derive this number by Lemma 4.

Lemma 4. *In an SNG, the number of edges that can monotonically reach* p *is equal to the number of edges that occlude* p *plus* p*'s in-degree.*

Proof (Sketch). As in Lemma 3, we consider the cases separately. The number of edges that can monotonically reach p with a monotonic path of length 1 is equal to the in-degree of p. The number of edges that can monotonically reach p with a monotonic path of length greater than 1 is equal to the number of edges that occlude p. □

Inspired by the multiplication rule of conditional probability, the frequency with which (p, p^*) is passed by search paths is estimated by the product of $m(p, p^*)$ and $m(p)$. That is:

$$w(p, p^*) = m(p, p^*) \cdot m(p). \tag{3}$$

To prioritize edges that are frequently passed by search paths, the GLO objective function of MARGO is designed as:

$$\mathcal{F}_{\text{MARGO}} = \sum_{(p,p^*)\in E} w(p, p^*) \cdot \mathbb{I}_{B(p)=B(p^*)}(p, p^*), \tag{4}$$

so that an edge with a larger $w(\cdot)$ can contribute more to the objective function, when it is selected. MARGO shares the same constrains with Starling, i.e., $C_{\text{MARGO}} = C_{Starling}$.

It is worth noting that the objective function of Starling (Eq. 1) is a special case of Eq. 4, where the weights $w(\cdot)$ are set to 1. This is because Starling only considers the effect of an edge in one hop, and neglects its impact in an entire monotonic path.

Discussion on approximate SNG. Due to the high construction complexity of SNG, existing graph indexes turn to approximate SNG, where the short edge priority rule is applied only within a vertex's neighborhood. Specifically, for a vertex p, its neighborhood $\Delta(p)$ is first identified using ANN search, and then edges are added between p and $\Delta(p)$ according to the short edge priority rule. Note that $\Delta(p)$ differs from the out-neighbor set $N(p)$, as only a portion of vertices from $\Delta(p)$ follow the rule and constitute $N(p)$. In this case, the induced subgraph of the neighborhood can be regarded as an SNG. Based on recent observations [35, 41], most of the hops during ANN search in the graph index occur within the query's neighborhood. Therefore, we contend that it is still reasonable to employ the conclusions derived in Lemma 3 and 4 in the context of an approximate SNG.

4 THE MARGO SOLUTION TO GLO

We proceed to introduce the graph layout optimization under the monotonic path-aware objective function. First, we prove that this problem is NP-hard. Next, a greedy algorithm is proposed. At last, two stage decoupling is introduced to improve efficiency. We assume that both the graph index and the edge weights are already obtained, the details of which are covered in Sec. 5.

4.1 Problem Reduction

Substitute our objective function (Eq. 4) and constraints into the general definition of GLO (Def. 2), we have the following definition.

Definition 4 (GLO of MARGO). *Given a PG with* n *vertices, denoted as* $G = (V, E)$*, positive integers* n_p *and* n_v*, and the edge weights* $w(\cdot) : E \to \mathbb{N}^+$*, GLO of MARGO aims to assign the* n *vertices to* n_p *pages, where each page contains exactly* n_v *vertices, and each vertex is assigned to exactly one page. The optimization objective of the assignment is:*

$$\arg\max_{B(\cdot)} \sum_{(p,p^*)\in E} w(p, p^*) \cdot \mathbb{I}_{B(p)=B(p^*)}(p, p^*), \tag{5}$$

where $B(\cdot)$ *maps each vertex to its assigned page.*

Theorem 1. *The problem in Def. 4 is NP-hard.*

Proof. Construct an undirected graph with positive weights, denoted as $G^u = (V^u, E^u, w^u(\cdot))$, which satisfies the following conditions:

- For the vertex set, it is the same as the vertex set in the proximity graph $G = (V, E)$, i.e., $V^u = V$.
- For the edge set E^u, an edge $(p, p^*)^u \in E^u$ iff. $(p, p^*) \in E$ or $(p^*, p) \in E$. Here we use a superscript to distinguish between directed and undirected edges.
- The edge weights $w^u(\cdot) : E^u \to \mathbb{N}^+$ are defined as follows. For $(p, p^*)^u \in E^u$, if both (p, p^*) and (p^*, p) belong to E, then

$w^u(p, p^*) = w(p, p^*) + w(p^*, p)$. Otherwise, $w^u(p, p^*)$ is equal to either $w(p, p^*)$ or $w(p^*, p)$, depending on which edge exists.

The problem in Def. 4 is equivalent to the following problem. Given an undirected graph with positive weights and a positive integer n_p, partition the vertices into n_p subsets, each containing exactly n_v vertices. The goal is to maximize the weight sum of edges whose both endpoints belong to the same subset.

Since the total weight sum of all edges in E^u is constant, maximizing the weight sum of edges within subsets corresponds to minimizing the weight sum of edges cut by the partition. Thus, the problem is equivalent to the minimum n_p-cut problem with equal-size subsets, which is known to be NP-hard [15]. □

This problem can be solved by Semidefinite programming (SDP). However, SDP is not applicable to large-scale datasets. Even without considering the equal-size constraints on subsets, the exact solution to this problem has a time complexity of $O(|V|^{(1.981+o(1)n_p)})$ [18]. Although there exists polynomial-time approximation algorithms with $2(1 - 1/n_p)$-approximation ratio [32, 36], these methods are still insufficient for solving the GLO problem of MARGO. On one hand, their time complexities are too high for a graph index that contains millions of vertices, with the number of pages on a similar order of magnitude. On the other hand, they cannot enforce the equal-size constraints on subsets. To the best of our knowledge, no existing solution currently addresses the GLO of MARGO.

4.2 Greedy Algorithm

We propose a greedy algorithm to solve the GLO of MARGO. First, an undirected graph is constructed as described in the proof of Theorem 4.1. Then, we assign the vertices to pages, with higher-weight edges given priority.

As the construction of the undirected graph is straightforward, we omit the details due to the space limitation. Next, to prioritize edges with higher weights, we sort the edges in descending order on their weights. When assigning vertices to an empty page, say B_i, we greedily select the edge with maximum weight whose both endpoints are not yet assigned, and assign these endpoints to B_i.

While B_i is not full, we have two strategies for determining the next vertex to assign. (1) Select a vertex that is not a neighbor (in the undirected graph there is no distinction between in- and out-neighbors) of the already-assigned vertices in B_i. In this case, we select the edge with the maximum weight, whose both endpoints are unassigned, and assign them to B_i, similar to how we handle an empty page. (2) Select the vertex from the neighbors. We scan all unassigned neighbors, compute the contribution of each neighbor to the objective function if it is assigned to B_i, and select the vertex with the highest contribution. The contribution is computed as the weight sum of edges between the vertex to assign and the vertices already in B_i.

However, we find that assigning non-neighbor vertices often leads to unsatisfactory results, even if they may contribute more to the objective function at the moment. This is because it may negatively affect the contributions of subsequent pages. For example, suppose that B_i has room for two more vertices. If we assign non-neighbor vertices, such as the endpoints of the edge $(p, p^*)^u$, the contribution is just $w^u(p, p^*)$. However, if we assign p, p^* along with other vertices in $N(p)$ and $N(p^*)$ to a new page, the contribution could exceed $w^u(p, p^*)$. Therefore, we prioritize the second strategy that selects vertices from the neighbors for assignment, until B_i is full, or there is no more unassigned neighbors. At last, any remaining unassigned vertex after all pages are processed are assigned to pages not yet full.

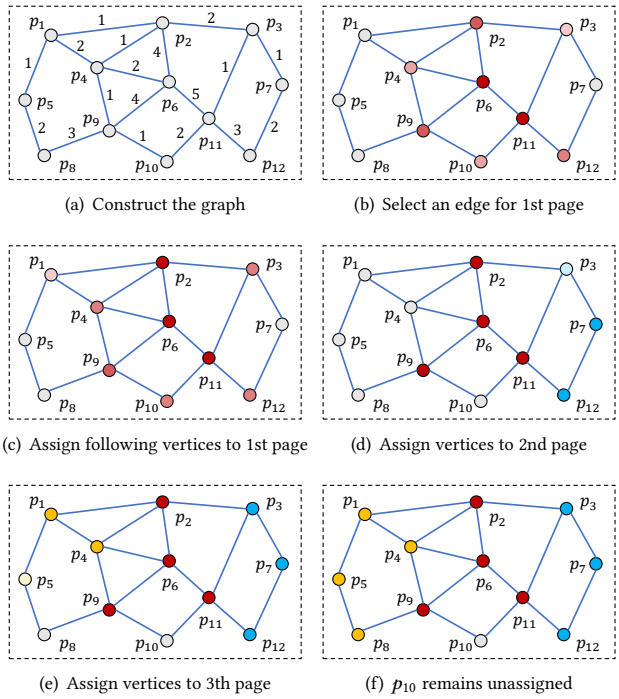


Figure 2: A running example of the greedy algorithm

Example 3. *Fig. 2 illustrates the graph layout procedure in Fig. 1(c) using the greedy algorithm. First, the undirected graph corresponding to the SNG in Fig. 1(a) is constructed, as shown in Fig. 2(a), with edge weights annotated besides the respective edges.*

The first page begins with the edge having the maximum weight, i.e., $(p_6, p_{11})^u$. After assigning p_6 and p_{11} to the first page (denoted by red in Fig. 2(b)), their neighbors become the candidates for the vertex to assign next, i.e., $p_2, p_3, p_4, p_9, p_{10}, p_{12}$. The respective contributions of these candidates are 4, 1, 2, 4, 2, 3, *with larger contributions represented by darker colors in Fig. 2(b). Both p_2 and p_9 have the maximum contribution, so we choose p_2 which has the smaller ID , and assign it to the first page. Afterward, the candidates become $p_1, p_3, p_4, p_9, p_{10}, p_{12}$ (Fig. 2(c)), with contributions of* 1, 3, 3, 4, 2, 3, *respectively. Therefore, the next vertex is p_9. At this point, the first page is full.*

The second page starts with $(p_7, p_{12})^u$, the edge with the maximum weight among the remaining edges, as shown in Fig. 2(d). Following a similar process, the second page includes p_3, p_7, p_{12}, and the third page includes p_1, p_4, p_5, p_8 (Fig. 2(f)). Since p_{10} remains unassigned and the second page is not yet full, p_{10} is assigned to the second page.

Time complexity analysis. The time complexities of sorting all edges and assigning vertices to pages are $O(|E^u| \log |E^u|)$ and $O(|E^u| + n_p \hat{R} n_v^2)$, respectively. Here, $\hat{R}$ denotes the average degree

of G^u, which is comparable to the maximum out-degree R of the graph index, e.g., 64. n_v is the number of vertices per page, typically around 5 for a common 4KB page. n_p is the number of pages, which is computed as $\lceil n_p = n/n_v \rceil$. In the term $O(|E^u| + n_p\hat{R}n_v^2)$, the first term corresponds to the complexity of scanning the edges and selecting the first two vertices for each page. Since selecting subsequent vertices for a page is $O(\hat{R}n_v^2)$, the total complexity for n_p pages is $O(n_p\hat{R}n_v^2)$. Hence, the overall complexity of the greedy algorithm is $O(|E^u|\log|E^u| + n_p\hat{R}n_v^2)$. Given that $n = n_v n_p$ and $|E^u| = \hat{R}n$, the complexity simplifies to $O(\hat{R}n\log\hat{R}n + n_v\hat{R}n)$. If n_v and $\hat{R}$ are regarded as constants, the complexity becomes $O(n\log n)$.

4.3 Two Stage Decoupling

The greedy algorithm faces two critical issues when applied in practice. First, it requires sorting a large number of edges, which typically outnumbers the vertices by orders of magnitude. Second, both edge sorting and vertices assignment are executed serially: each page is processed only after the assignments for the previous one are completed. These issues result in low efficiency of the greedy algorithm. To address these issues, we propose two stage decoupling that leverages the idea of divide-and-conquer and significantly improves efficiency. It divides the vertices into smaller subsets, processes each subset using the greedy algorithm, and then combines the results of each subset to generate a final graph layout.

Divide with clustering. In a proximity graph, edges represent the neighborhood relationships between vertices. Although the out-neighbors of a vertex may not necessarily be its nearest neighbors due to the short edge priority rule of SNG, vertices that are far apart are less likely to be connected by edges. Based on this intuition, we divide the vertices into *nlist* clusters, denoted as $\{C_1, \ldots, C_{nlist}\}$, and process each cluster independently. This division reduces the original problem into smaller sub-problems, with minimal edges cutting. In this way, instead of sorting all edges at once, we only sort smaller sets of edges decoupled into each cluster. In addition, the processes of clusters can be parallelized without synchronization, allowing us to fully exploit the parallel capabilities of multi-core processors. In practice, we only sample a small portion of vertices for clustering, making it computationally efficient.

Conquer with greedy algorithm. For each cluster C_i, we construct the corresponding induced subgraph of G^u. The induced subgraph is denoted as $G_i^u = (V_i^u, E_i^u, w_i^u(\cdot))$, where $V_i^u = \{p | p \in C_i\}$, $E_i^u = \{(p, p^*)^u | p, p^* \in V_i^u \wedge (p, p^*)^u \in E^u\}$, and $w_i^u(p, p^*) = w^u(p, p^*)$. Then, the greedy algorithm is applied to each cluster based on its induced subgraph. It is worth noting that the assignments of the vertices that remains unassigned are not handled within each cluster's process. Instead, these remaining vertices, along with those assigned to pages that are not yet full, are marked for post-processing in the subsequent combination stage.

Combine with post-process. Finally, we combine the results of all clusters to produce the final graph layout. For the pages that are full within each cluster, we simply merge them into the final result. For the vertices that remain unassigned or those assigned to pages that are not yet full, we apply a post-process step.

The motivation behind the post-process is as follows. If we directly assign the remaining vertices to the pages not yet full within their respective cluster, these vertices would contribute nothing to the objective function. This is because, within any cluster, an unassigned vertex cannot be a neighbor of any vertex in a page that is not yet full. Otherwise, it contradicts the policy of the greedy algorithm. Essentially, the divide phase decouples the edges into intra-cluster edges and inter-cluster edges. Intra-cluster edges refer to edges whose both endpoints belong to the same cluster, and inter-cluster edges are those whose both endpoints are in different clusters. The conquer phase only handles the intra-cluster edges that are further decoupled into different clusters in parallel, while neglects the inter-cluster edges.

To address this, we employ a post-process to select inter-cluster edges. This step helps ensure that the remaining vertices make positive contributions to the objective function. For example, consider an inter-cluster edge whose two endpoints are unassigned and located at the boundaries of different clusters. While these vertices cannot form a page with neighbors from the same cluster, selecting this edge allows them to be assigned to the same page with neighbors from a different cluster. In practice, we treat the vertices marked for post-processing as a special cluster, C_{post}. This cluster is handled in the same way as the other clusters, and in the final step, the remaining unassigned vertices are assigned to the pages that are not yet full. At this point, all vertices are assigned, and the result from C_{post} is merged into the final graph layout.

5 WEIGHT COMPUTATION

We proceed to introduce how MARGO computes the edge weights during construction and performs ANN search. We employ Vamana [37] as the underlying algorithm of MARGO. As an SNG-based proximity graph, Vamana is the most prominent algorithm for disk-based graph index, and exhibits outstanding performance in ANN search. However, Vamana constructs an unweighted graph, lacking the edge weights discussed in Sec. 3 that are crucial for edge selection in the GLO of MARGO. To address this, a straight forward approach is to perform ANN search for each vertex on the built graph, apply the short edge priority rule to the obtained neighborhood, and compute edge weights based on the occlusions. However, both ANN search and the application of the short edge priority rule involve extensive, time-consuming distance computations of high dimensional vectors, making this approach impractical.

Therefore, we propose MARGO that computes the edge weights on-the-fly during index construction. By reusing the distances already computed for the construction, this approach incurs almost no additional overhead. MARGO initializes with an empty graph, with the vertex closest to the medoid of all vectors as the entry vertex. Then, the vertices are incrementally inserted into the graph. The two counting variables in Eq. 3 are dynamically maintained during insertion. The construction is completed after all vertices are inserted, and the edge weights are computed by Eq. 3. Next, we describe the vertex insertion and counting variable maintenance.

Before a vertex p is inserted, the number of edges that can monotonically reach p is initially set to 0, i.e., $m(p) = 0$. To insert p into the graph, MARGO first performs ANN search to obtain its neighborhood $\Delta(p)$ that consists of L vertices, where L is a user defined parameter. Then, the short edge priority rule is applied to generate edges between p and $\Delta(p)$. Specifically, MARGO scans $\Delta(p)$ in ascending order based on distances to p. Denote that $p^* \in \Delta(p)$

is the vertex closest to p, MARGO attempts to add both forward and backward edges between p and p^*. At the same time, the two counting variables of affected edges and vertices are updated. The details of adding forward and backward edges are as follows.

Add forward edge. The addition of (p, p^*) is straight forward. To account for the monotonic path of length 1, the number of vertices that (p, p^*) can monotonically reach is initialized to 1, i.e., $m(p, p^*) = 1$. Then, MARGO removes p^* from $\Delta(p)$. In addition, any $p' \in \Delta(p)$ such that $dist(p', p^*) < dist(p', p)$ is occluded and also removed from $\Delta(p)$, in compliance with the short edge priority rule. According to Lemmas 3 and 4, $m(p, p^*)$ and $m(p')$ are both incremented by 1.

Add backward edge. Adding the backward edge (p^*, p) is more complex, as it could cause the out-degree of p^* to exceed the maximum out-degree limit R. MARGO first attempts to add (p^*, p), where $m(p, p^*)$ is initialized to 1. If, after adding (p^*, p), the out-degree of p^* remains below R, the addition is complete. However, if the out-degree exceeds R, the out-edges of p^* must be pruned. This pruning occludes at least one out-edge, possibly including the recently added (p^*, p). Specifically, during the pruning, the original out-neighbors of p^* and p are treated as the neighborhood of p^*, i.e., $\Delta(p^*) = N(p^*) \cup \{p\}$. The short edge priority rule is then applied to p^* and $\Delta(p^*)$. Since edges between p^* and $\Delta(p^*)$ are already added, MARGO only occludes existing edges rather than adding new ones. Given $p^+, p' \in \Delta(p^*)$, if (p^*, p^+) occludes (p^*, p'), both $m(p^*, p^+)$ and $m(p')$ are incremented by 1. Finally, for each occluded p', the edge (p^*, p') is deleted.

The connection of p terminates when its out-degree reaches R, or when $\Delta(p)$ becomes empty. Similar to Vamana, MARGO employs two iterations of insertion to improve the graph quality. In the second iteration, a parameter $\alpha \geq 1$ is introduced to relax the occlusion, so that long edges are retained to serve as shortcuts. Specifically, given an edge (p, p^*) and a vertex p', (p, p^*) occludes p' iff. $\alpha \cdot dist(p', p^*) < dist(p', p)$. MARGO only computes the counting variables in the second iteration, for two reasons. First, the graph yields relatively lower quality in the first iteration, resulting in inaccurate ANN search and counting variables. Second, a portion of vertices and edges may be counted multiple times if MARGO computes the counting variables in both iterations. After the second iteration, for each vertex p, $m(p)$ is incremented by p's in-degree. Then, the weight of each edge is computed according to Eq. 3.

Time complexity analysis. Compared to Vamana, MARGO introduces additional overhead to compute the two counting variables, i.e., $m(p, p^*)$ and $m(p)$ for each $(p, p^*) \in E$ and each vertex $p \in V$. However, the update is finished in $O(1)$ each time. Therefore, the overall time complexity of index construction remains $O(n \log n)$. Since the counting is relatively efficient compared to the distance computations of high dimensional vectors, MARGO demonstrates similar empirical construction performance to Vamana (Fig. 8).

Space cost analysis. To maintain the two counting variables, MARGO requires additional space of $O(|E| + |V|)$. The total space cost is $O((d+1)|V| + 2|E|)$, which we consider acceptable. In prominent databases [17, 42], the processes of index construction and search are decoupled onto independent nodes for high elasticity. Although the memory available on segments is limited, index nodes typically have sufficient memory. Notably, MARGO incurs no additional memory cost during the search procedure, as the counting variables are only used during the GLO, and do not need to be loaded onto segments.

Search procedure. MARGO employs two search techniques from existing disk-based graph indexes [37, 43]. First, it employs the Product Quantization [22] and uses approximate distances to guide the search path. Second, it expands the search space based on multiple vertices located in the same page with the current vertex. Interested readers can refer to the original papers for more details. It is worth noting that, under the same search strategy and search parameters, MARGO is able to achieve a higher recall than Starling (Table 3). This is because the delicately selected edges, which are useful for monotonic paths, help prevent the search path from getting stuck in local optima.

6 EXPERIMENTS

6.1 Experimental Setup

We implement MARGO in C++. All experiments are conducted on a Linux machine with two Intel Xeon Gold 6330 CPU @ 2.00GHz processors (28 cores), with 125GB DDR4 RAM, and six Samsung PM883 1.92TB SSDs in RAID-0 configuration. For index construction and GLO, we do not impose resource constraints, simulating a resource-rich index node. For ANN search, we limit memory usage to under 2GB and disk usage to under 10GB, to ensure all methods operate within the resource constraints typical of segments in leading vector databases [42].

Datasets. We conduct the experiments in four datasets [9] with varying dimensionalities under different distance functions. The details of the datasets are shown in Table 2, where L2 and IP refer to Euclidean distance and inner product, respectively. We follow the settings in [43] and limit the vectors to slightly under 4GB, which is a reasonable scale for a segment in vector databases.

Baselines. We compare MARGO with DiskANN [37], Starling [43], and SPANN [8]. All methods use the O_DIRECT option for disk I/O to avoid the influence of operating system-level caching. Graph partitioning methods such as KMETIS [23] and KGGGP [34] are excluded from the experiments, because GLO is not identical to graph partitioning. First, these methods are designed for real-world graphs, whose structure and properties differ from graph indexes, leading to unsatisfactory performance in GLO [43]. Second, they aim to divide vertices into roughly equal-sized subsets, which does not strictly satisfy the constraints imposed by C_{MARGO}.

Metrics. For the search procedure, we evaluate the accuracy and efficiency. The accuracy is evaluated by *Recall@k*, which measures the recall at k NNs. The efficiency is evaluated in terms of average number of I/Os, query latency, and QPS (queries per second). For the index construction, we evaluate the execution time required to build the graph index and optimize the graph layout. In addition, we measure the memory footprint of the index construction.

Parameter settings. The page size is set to 4KB, which is the most common configuration. The out-degree limit R is set to 64 for DEEP and SIFT, 127 for Tiny, and 128 for Text2image. The number L of neighbors to apply the short edge priority rule during construction is 125 for DEEP and SIFT, and 250 for Tiny and Text2image. The parameter α is set to 1.2. In all datasets, the number of clusters *nlist* is 256 for MARGO, and the number of iterations is 16 for Starling. The pruning ratio of the page search is set to 1.0. The duplication

Table 2: Dataset Statistics

Datasets	# Vectors	d	$dist(\cdot,\cdot)$	# Queries
DEEP	11M	96	L2	10K
SIFT	8M	128	L2	10K
Text2image	5M	200	IP	100K
Tiny	2.7M	384	L2	10K

count of SPANN is set to 2 to meet the resource constrains of segments. Queries are processed in batch mode for all methods with 56 threads, each handling one query at a time.

6.2 Search Performance Evaluation

We employ recall, average number of I/Os, query latency, and QPS to evaluate the search performance of MARGO. First, we show the trade-offs between accuracy and efficiency by comparing the average number of I/Os, query latency, and QPS under different recall. Then, we show the search performance of MARGO and the baseline methods with identical search parameter. Finally, we present the search performance under varying k.

Performance under different recall. We present the average number of I/Os, query latency, and QPS under different recall in Fig. 3, 4, and 5, respectively. Since SPANN performs both random and sequential disk accesses, its I/O count does not provide a meaningful comparison. Therefore, SPANN is excluded from Fig. 3. Among all methods, MARGO consistently achieves the best overall performance. Notably, MARGO shows the most significant improvements in Tiny and Text2image. This is because, in theses two datasets, the storage of a vertex needs more space due to higher dimensionality of vectors, resulting in fewer vertices per page. In this case, selecting edges carefully becomes even more critical. Specifically, at the recall of 94.8%, MARGO outperforms Starling by 21.8% in query latency and 26.6% in QPS for Tiny. In terms of the number of random disk I/Os, MARGO requires 13.1% fewer I/Os than Starling at the recall of 97.9% for Text2image. SPANN significantly lags behind the graph-based methods in most datasets, because it cannot duplicate vectors extensively due to the resource limitation of segments. However, it achieves comparable performance in Text2image, which we attribute to large quantization errors that hinder navigation of graph-based methods in this dataset.

Performance with identical parameter. Table 3 shows the *recall*@10, average number of I/Os, query latency, and QPS when L_s, a parameter that controls the trade-off between accuracy and efficiency, is set to the same value for MARGO and graph-based baseline methods. With the same search parameter, MARGO not only achieves the fewest I/Os, the shortest query latency and the highest QPS, but also yields the highest recall compared to the baseline methods. This indicates that the delicately selected edges during GLO not only reduce random disk I/Os, but also prevent ANN search from stuck in local optima. Hence, both efficiency and accuracy are improved, even with identical search parameter.

Performance under varying k. Fig. 6 compares the performance of MARGO and baseline methods under varying values of k. Due to space limitations, we only report the trade-off between recall and the average number of random disk I/Os in Tiny. As shown in Fig. 3 and 6, MARGO consistently outperforms Starling and DiskANN at k = 1, 10, and 50. In addition, the performance trends remain similar across different values of k, providing evidence for the effectiveness and robustness of MARGO.

6.3 Construction Performance Evaluation

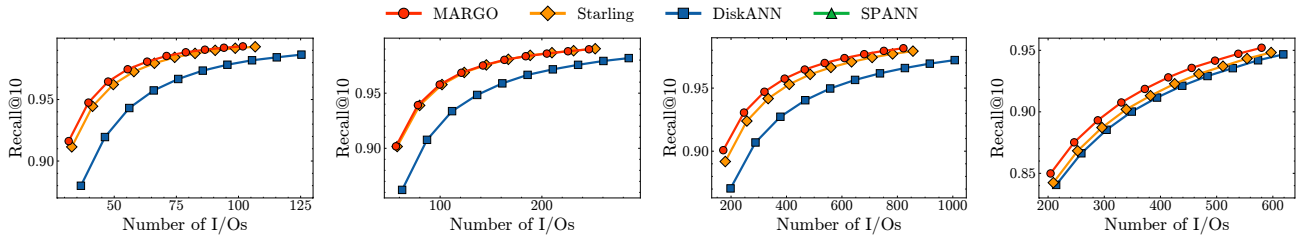
We first evaluate the graph layout, including the objective function values and execution time of the GLO procedure. Then, we compare the time and space cost for building the graph index.

Objective function values. In the graph layout evaluation, we exclude DiskANN, as it stores the vertices on disk sequentially based on their IDs without employing GLO. We compute the values of both $\mathcal{F}_{starling}$ (Eq. 1) and $\mathcal{F}_{\text{MARGO}}$ (Eq. 4) for Starling and MARGO in Table 4. Although MARGO has a different optimization objective from Starling, they achieve similar values of $\mathcal{F}_{starling}$. This is consistent with Lemma 1 that Starling is essentially selecting edges. For $\mathcal{F}_{\text{MARGO}}$, MARGO exceeds Starling by over an order of magnitude. It is worth noting that even in the case where MARGO achieves a smaller $\mathcal{F}_{starling}$ than Starling, e.g., in DEEP and SIFT, MARGO requires fewer random disk I/Os (Fig. 3). This indicates that it does not necessarily reduce random disk I/Os with more edges whose both endpoints belong to the same page. Instead, selecting fewer edges that are important for monotonic paths achieves better results. Therefore, the optimization objective of MARGO is more reasonable.

Comparison with optimal graph layout. In addition to comparisons with baseline methods, it is also valuable to assess how closely the greedy algorithm approximates the optimal graph layout. To this end, we conduct experiments on small graphs where computing the optimal graph layout remains tractable. Specifically, we sample a small subset for each dataset, construct the index with an out-degree limit of 8, and perform ANN search with k = 1. As shown in Table 5, the greedy algorithm closely approximates the optimal graph layout, with only a slight gap in both the objective function value and the average number of I/Os.

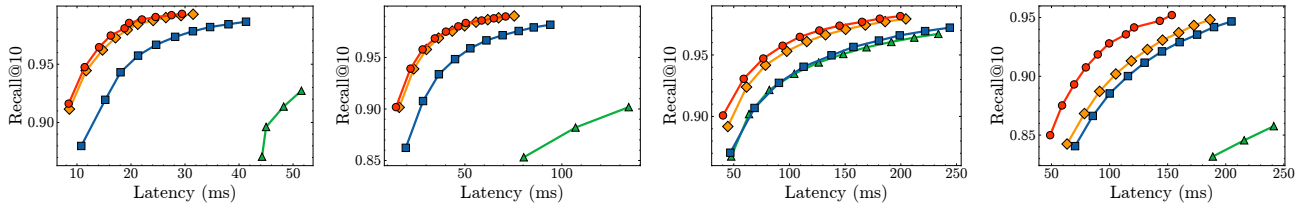
Execution time of GLO. Fig. 7 compares the GLO execution time of MARGO and Starling. MARGO demonstrates significantly higher efficiency than Starling. The enhancement in efficiency is attributed to two factors. First, Starling needs to preprocess the entire graph, which is time-consuming and takes up about 30% to 40% of the execution time. In contrast, according to the two stage decoupling, MARGO only processes the induced subgraphs in parallel without synchronization. Second, GLO of Starling is performed in an iterative manner. A large number of iterations leads to low efficiency. However, MARGO completes GLO in nearly one pass, where each vertex is assigned only once, except for a small fraction of vertices undergoing post-processing. Overall, MARGO achieves a 4.0× to 5.5× improvement in the GLO efficiency.

Time cost of index construction. Fig. 8 compares the index construction time across different methods. Starling and DiskANN share identical index time, as Starling directly uses a pre-built DiskANN index. While MARGO incurs additional overhead due to the computation of edge weights, it exhibits inconsistent performance across different datasets compared to other graph-based methods. In DEEP, and SIFT, MARGO slightly outperforms Starling and DiskANN. However, an opposite trend is observed in



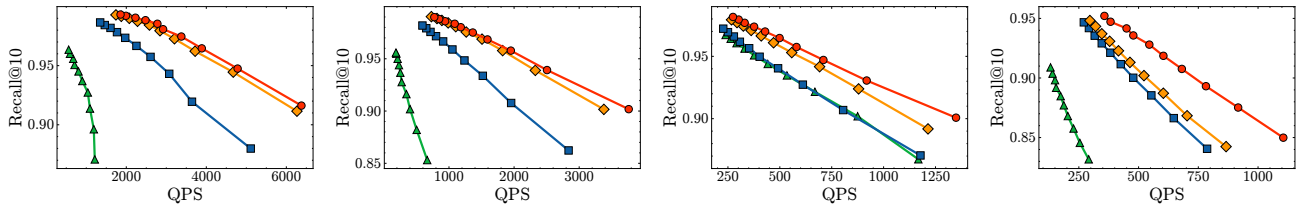
(a) DEEP (b) SIFT (c) Text2image (d) Tiny

Figure 3: The average number of I/Os under different recall



(a) DEEP (b) SIFT (c) Text2image (d) Tiny

Figure 4: The query latency under different recall



(a) DEEP (b) SIFT (c) Text2image (d) Tiny

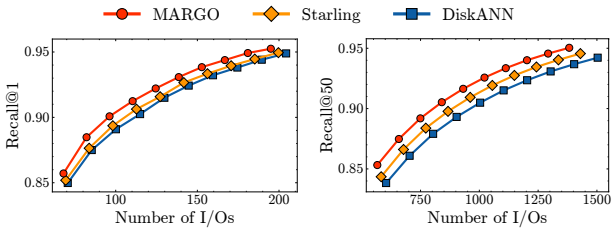
Figure 5: The throughput under different recall

Table 3: Search performance with the same parameter

Datasets	L_s	*Recall*@10 (%)			Average # of I/Os			Query latency (ms)			QPS		
		DiskANN	Starling	MARGO	DiskANN	Starling	MARGO	DiskANN	Starling	MARGO	DiskANN	Starling	MARGO
DEEP	50	94.31	96.22	**96.46**	56.06	49.62	**47.54**	18.1	14.7	**14.1**	3074	3720	**3887**
SIFT	105	93.37	**95.78**	**95.78**	111.72	101.83	**99.58**	36.6	30.2	**28.2**	1520	1825	**1950**
Text2image	370	92.73	94.17	**94.71**	378.01	333.86	**320.46**	90.5	78.5	**76.5**	611	692	**710**
Tiny	435	92.12	92.28	**92.8**	438.81	425.31	**413.89**	145.1	119.1	**99.6**	425	583	**604**

Text2image and Tiny. We attribute these slight differences to potential implementation issues or fluctuations in server performance. Nevertheless, there is no evidence suggesting that the index time is significantly impacted by the weight computation. We notice that the index construction time of SPANN is highly sensitive to data scale, incurring substantially higher costs than graph-based methods in DEEP and SIFT. This is because it requires constructing a high-quality graph index on a large number of centroids and preforming costly ANN search to accurately assign each vector to its nearest clusters. In contrast, graph-based methods are more influenced by vector dimensionality, taking longer in SIFT than DEEP. Additionally, they spend more time in Text2image and Tiny, where higher out-degree limits result in denser graphs.

Space cost of index construction. The space costs across different methods are presented in Fig. 9. MARGO consistently requires more memory than Starling and DiskANN across all datasets. This is because, in addition to storing vectors and neighbor IDs of each vertex, MARGO also stores edge weights in memory. Out of the same reason, the difference in space costs becomes smaller in datasets with higher vector dimensionality and fewer edges, e.g.,



(b) $k = 1$ in Tiny (c) $k = 50$ in Tiny

Figure 6: Performance under varying k

Table 4: The objective function values

Dataset	Objective	Starling	MARGO
DEEP	$\mathcal{F}_{Starling}$	2.8×10^7	2.2×10^7
	$\mathcal{F}_{\text{MARGO}}$	5.8×10^{10}	2.5×10^{12}
SIFT	$\mathcal{F}_{Starling}$	1.8×10^7	1.5×10^7
	$\mathcal{F}_{\text{MARGO}}$	4.8×10^{10}	8.9×10^{11}
Text2image	$\mathcal{F}_{Starling}$	7.9×10^6	8.1×10^6
	$\mathcal{F}_{\text{MARGO}}$	4.5×10^{10}	1.1×10^{12}
Tiny	$\mathcal{F}_{Starling}$	2.2×10^6	2.6×10^6
	$\mathcal{F}_{\text{MARGO}}$	6.9×10^9	8.5×10^{10}

Table 5: Comparison with optimal graph layout

Dataset	Scale	Objective value		Average # of I/Os	
		Greedy	Optimal	Greedy	Optimal
DEEP	54	16921	18158	5.26	5
SIFT	56	15777	18234	4.18	4.05
Text2image	36	8893	9841	5.57	5.54
Tiny	36	7517	7604	6.35	6.2

Tiny. Compared to graph-based methods, SPANN exhibits more consistent space costs across datasets. This stability stems from the fact that SPANN mainly stores inverted lists, whose size closely correlates with the size of the original vectors (about 3.8GB in the four datasets). In contrast, the memory usage of graph-based methods varies more significantly due to differences in graph properties such as the vertex count and average out-degree.

6.4 Parameter Study

Finally, we study the impact of the key parameter *nlist* on the performance of MARGO. The graph index layout is optimized under varying *nlist* from 64 to 1024. Fig. 10 shows the average number of I/Os and *recall*@10 for different *nlist* settings. As the *nlist* increases, the number of random disk I/Os also rises. This is because a larger *nlist* results in more inter-cluster edges, some of which may be potentially important but are cut in the divide stage and fail to be recovered in the post-process. In contrast, the execution time of GLO decreases at first, and then increases again. With a small *nlist*, each cluster contains more edges and vertices, which leads to higher costs in edge sorting and vertex assignments. However, when *nlist* becomes excessively large, the time saved by the process of each cluster no longer compensates for the increased overhead due to clustering the vertices into a large number of clusters. It is worth noting that the number of I/Os remains robust with respect to changes in *nlist*, varying by only around 2%. Although the GLO time fluctuates more noticeably, this variation (about 10 seconds) is acceptable in practice compared to the cost of index construction. Based on the empirical results, we set *nlist* to 256 for all datasets, though further tuning for each dataset may yield even better results.

6.5 Ablation Study

We conduct an ablation study of the proposed MARGO to evaluate the improvements provided by different components. Specifically, we compare the following variants of MARGO:

- Greedy: Employing the greedy algorithm during GLO.
- wo-weight: Setting all edge weights to 1.

The execution time of GLO and the number of random disk I/Os across four datasets are shown in Fig. 11.

Improvements of two stage decoupling. In Fig. 11(a), we observe a significant increase in GLO execution time when the greedy algorithm is adopted. This is due to the fact that the greedy algorithm sorts a large number of edges and executes serially. Regarding the number of random disk I/Os (Fig. 11(b)), the two stage decoupling lags slightly behind the greedy algorithm, because it may cut potentially important inter-cluster edges in the divide stage. However, such marginal difference is acceptable when compared to the substantial improvements in GLO execution time.

Improvements of edge weights. As Fig. 11(a) shows, the execution time of GLO with and without weight computation only show slight differences, because no matter what weights the edges are assigned, the two stage decoupling goes through a similar process. However, the number of I/Os increases when the edge weight computation is disabled in Fig. 11(b). For example, the required number of I/Os at the recall of 93.11% increases from 795.88 to 902.07 in text2image. This indicates that the edge weights successfully capture the importance of edges.

7 RELATED WORK

7.1 In-Memory Graph index

Graph indexes [7, 13, 14, 19, 26, 28–30, 33, 45] are considered as the most promising solutions to ANN search, offering both high accuracy and efficiency [4, 25, 26, 38].

NSW [29] approximates the Delaunay Graph by incrementally inserting vertices into the graph. Long edges formed in the early stages of insertion serve as shortcuts, improving search efficiency. HNSW [30] limits the out-degree and constructs hierarchical navigation graphs. It addresses the hubness issue in NSW that the out-degrees are extremely skewed to some vertices, and shortens the search path. NSG [14] proposes the monotonic Relative Neighborhood Graph (RNG). It relaxes the edge generation rule of RNG to provide sufficient edges that form monotonic paths. NSSG [13] further explores the potential of monotonic paths in graph indexes. It generates edges based on both distance and angle information to ensure that a vertex can guide the search path towards all directions.

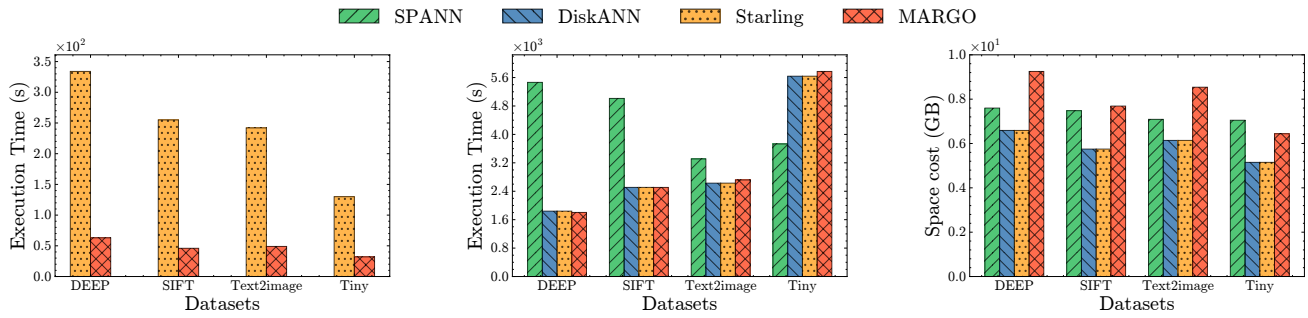
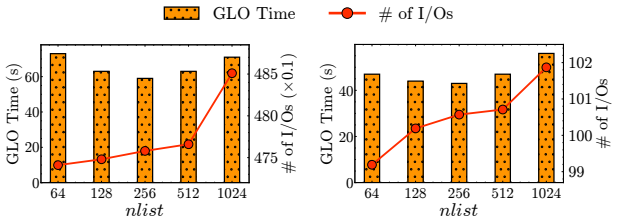


Figure 7: The execution time of GLO

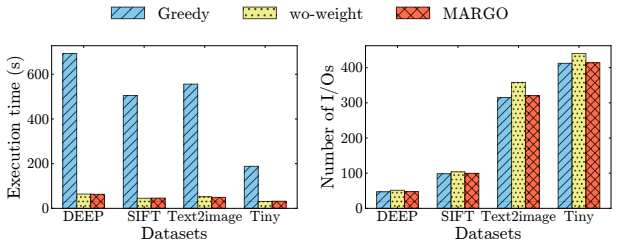
Figure 8: Index construction time

Figure 9: Index construction space cost



(a) DEEP (b) SIFT

Figure 10: Effects of *nlist*



(a) GLO execution time (b) Number of I/Os

Figure 11: Ablation study

τ-MNG [33] relaxes the edge generation rule of monotonic RNG, removing constraints on short edges and loosening restrictions on long edges. It enables the search path to approach the query faster and avoid missing neighbors after arriving at the query's neighborhood. LSH-APG [45] incorporates Locality Sensitive Hashing (LSH) to dynamically select the entry vertex during insertion, mitigating the high construction complexity issue.

Despite their excellent performance, graph indexes suffer from high memory footprint, which limits their scalability.

7.2 Disk-based ANN Search

Disk-based ANN search methods include graph index based methods [37, 43] and inverted file index based methods [8]. Among graph index based methods, DiskANN [37] proposes Vamana, an SNG-based graph index with a relaxed edge generation rule. The vertices are stored on disk sequentially according to IDs. The search path is guided by approximate distances computed using Product Quantization (PQ) [22], thereby avoiding disk I/Os for accessing raw vectors. Starling [43] is the first to optimize the graph layout. The optimization is performed in an iterative manner, where each vertex is assigned to the same page with its neighbors as much as possible. Starling proposes a page search strategy, which expands the search space with not only the target vertex determined based on the PQ distance, but also the vertices within the same page. For inverted file index based methods, SPANN [8] partitions the vectors using a fine-grained balanced clustering algorithm, limiting the number of vectors that need to be read from disk when a cluster is visited. Each vector is assigned to multiple clusters, which prevents missing neighbors located on the boundaries of clusters.

Among these methods, DiskANN and SPANN are designed for single machines. In contrast, Starling is tailored for distributed cloud-based vector databases, where vectors are distributed across segments with limited resources [17, 42]. In this scenario, SPANN cannot afford to duplicate vectors extensively, and therefore yields unsatisfactory performance. Compared to inverted file index based methods, graph index based methods are able to achieve better trade-offs between search performance and disk overhead.

8 CONCLUSION

We propose MARGO as a GLO method for disk-based ANN search. First, MARGO employs a monotonic path-aware objective function that weighs the edges based on their importance in monotonic paths. Second, we propose a greedy algorithm that prioritizes high-weight edges to optimize the graph layout on disk. To improve efficiency, two stage decoupling is put forward, which processes intra-cluster edges in parallel first, followed by inter-cluster edges. Third, MARGO leverages an on-the-fly weight computation strategy during index construction to avoid incurring additional overhead. Extensive experiments demonstrate that compared to the SOTA method Starling, MARGO achieves up to 26.6% improvement in search efficiency while maintaining the same accuracy, and up to 5.5× acceleration in graph layout optimization.

ACKNOWLEDGMENTS

This work was supported in part by NSFC Grant No. 62372194 and by 2025 Open Research Program of the MIIT Key Laboratory for Software Integrated Application and Testing & Verification.

REFERENCES

[1] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. 2018. HD-Index: Pushing the Scalability-Accuracy Boundary for Approximate kNN Search in High-Dimensional Spaces. *PVLDB*. 11, 8 (2018), 906–919.

[2] Sunil Arya and David M. Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *SODA*. 271–280.

[3] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. 2023. Retrieval-based Language Models and Applications. In *ACL*. 41–46.

[4] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.* 87 (2020).

[5] 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 *SIGMOD Conference*. 322–331.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

[7] Meng Chen, Kai Zhang, Zhenying He, Yinan Jing, and X. Sean Wang. 2024. RoarGraph: A Projected Bipartite Graph for Efficient Cross-Modal Approximate Nearest Neighbor Search. *PVLDB*. 17, 11 (2024), 2735–2749.

[8] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *NeurIPS*. 5199–5212.

[9] Billion-Scale Approximate Nearest Neighbor Search Challenge: NeurIPS'21 competition track. 2021. https://big-annbenchmarks.com/

[10] DW 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. IEEE, 548–553.

[11] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]

[12] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules Against Data Series Similarity Search. *PVLDB*. 15, 10 (2022), 2005–2018.

[13] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *TPAMI*. 44, 8 (2022), 4139–4150.

[14] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *PVLDB*. 12, 5 (2019), 461–474.

[15] M. R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*.

[16] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. 2946–2953.

[17] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: A Cloud Native Vector Database Management System. *PVLDB*. 15, 12 (2022), 3548–3561.

[18] Anupam Gupta, Euiwoong Lee, and Jason Li. 2019. The number of minimum k-cuts: improving the Karger-Stein bound. In *STOC*. 229–240.

[19] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *CVPR*. 5713–5722.

[20] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based Retrieval in Facebook Search. In *KDD*. 2553–2561.

[21] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. 604–613.

[22] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *TPAMI*. 33, 1 (2011), 117–128.

[23] George Karypis and Vipin Kumar. 1998. Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Parallel Distributed Comput.* 48, 1 (1998), 96–129.

[24] The ChatGPT Retrieval Plugin lets you easily search and find personal or work documents by asking questionsin everyday language. 2023. https://github.com/openai/chatgpt-retrieval-plugin

[25] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. 2020. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *SIGMOD Conference*. 2539–2554.

[26] 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. *TKDE*. 32, 8 (2020), 1475–1488.

[27] Defu Lian, Xing Xie, Enhong Chen, and Hui Xiong. 2021. Product Quantized Collaborative Filtering. *TKDE*. 33, 9 (2021), 3284–3296.

[28] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2021. HVS: Hierarchical Graph Structure Based on Voronoi Diagrams for Solving Approximate Nearest Neighbor Search. *PVLDB*. 15, 2 (2021), 246–258.

[29] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.* 45 (2014), 61–68.

[30] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *TPAMI*. 42, 4 (2020), 824–836.

[31] Stanislav Morozov and Artem Babenko. 2019. Unsupervised Neural Quantization for Compressed-Domain Similarity Search. In *ICCV*. 3036–3045.

[32] Joseph Naor and Yuval Rabani. 2001. Tree packing and approximating k-cuts. In *SODA*. 26–27.

[33] 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), 54:1–54:27.

[34] Maria Predari and Aurélien Esnard. 2016. A k-Way Greedy Graph Partitioning with Initial Fixed Vertices for Parallel Applications. In *PDP*. 280–287.

[35] Liudmila Prokhorenkova and Aleksandr Shekhovtsov. 2020. Graph-based Nearest Neighbor Search: From Practice to Theory. In *ICML*, Vol. 119. 7803–7813.

[36] Huzur Saran and Vijay V. Vazirani. 1995. Finding k Cuts within Twice the Optimal. *SIAM J. Comput.* 24, 1 (1995), 101–108.

[37] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NeurIPS*. 13748–13758.

[38] Yao Tian, Ziyang Yue, Ruiyuan Zhang, Xi Zhao, Bolong Zheng, and Xiaofang Zhou. 2023. Approximate Nearest Neighbor Search in High Dimensional Vector Databases: Current Research and Future Directions. *IEEE Data Eng. Bull.* 46, 3 (2023), 39–54.

[39] Yao Tian, Xi Zhao, and Xiaofang Zhou. 2022. DB-LSH: Locality-Sensitive Hashing with Query-based Dynamic Bucketing. In *ICDE*. 2250–2262.

[40] Michael Tschannen, Manoj Kumar, Andreas Steiner, Xiaohua Zhai, Neil Houlsby, and Lucas Beyer. 2023. Image Captioners Are Scalable Vision Learners Too. In *NeurIPS*.

[41] Hongya Wang, Zhizheng Wang, Wei Wang, Yingyuan Xiao, Zeng Zhao, and Kaixiang Yang. 2020. A note on graph-based nearest neighbor search. *arXiv preprint arXiv:2012.11083* (2020).

[42] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD Conference*. 2614–2627.

[43] 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), V2mod014:1–V2mod014:27.

[44] Xi Zhao, Zhonghan Chen, Kai Huang, Ruiyuan Zhang, Bolong Zheng, and Xiaofang Zhou. 2024. Efficient Approximate Maximum Inner Product Search Over Sparse Vectors. In *ICDE*. 3961–3974.

[45] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards Efficient Index Construction and Approximate Nearest Neighbor Search in High-Dimensional Spaces. *PVLDB*. 16, 8 (2023), 1979–1991.

[46] Xi Zhao, Bolong Zheng, Xiaomeng Yi, Xiaofan Luan, Charles Xie, Xiaofang Zhou, and Christian S. Jensen. 2023. FARGO: Fast Maximum Inner Product Search via Global Multi-Probing. *PVLDB*. 16, 5 (2023), 1100–1112.

[47] Bolong Zheng, Ziyang Yue, Qi Hu, Xiaomeng Yi, Xiaofan Luan, Charles Xie, Xiaofang Zhou, and Christian S. Jensen. 2023. Learned Probing Cardinality Estimation for High-Dimensional Approximate NN Search. In *ICDE*. 3209–3221.

[48] Bolong Zheng, Xi Zhao, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S. Jensen. 2020. PM-LSH: A Fast and Accurate LSH Framework for High-Dimensional Approximate NN Search. *PVLDB*. 13, 5 (2020), 643–655.

[49] Bolong Zheng, Xi Zhao, Lianggui Weng, Quoc Viet Hung Nguyen, Hang Liu, and Christian S. Jensen. 2022. PM-LSH: a fast and accurate in-memory framework for high-dimensional approximate NN and closest pair search. *VLDB J.* 31, 6 (2022), 1339–1363.