Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment

MENGZHAO WANG*†, Zhejiang University, China
WEIZHI XU†, Zilliz, USA
XIAOMENG YI, Zhejiang Lab, China
SONGLIN WU*, Tongji University, China
ZHANGYANG PENG, Hangzhou Dianzi University, China
XIANGYU KE, Zhejiang University, China
YUNJUN GAO, Zhejiang University, China
XIAOLIANG XU, Hangzhou Dianzi University, China
RENTONG GUO, Zilliz, USA
CHARLES XIE, Zilliz, USA

High-dimensional vector similarity search (HVSS) is gaining prominence as a powerful tool for various data science and AI applications. As vector data scales up, in-memory indexes pose a significant challenge due to the substantial increase in main memory requirements. A potential solution involves leveraging disk-based implementation, which stores and searches vector data on high-performance devices like NVMe SSDs. However, implementing HVSS for data segments proves to be intricate in vector databases where a single machine comprises multiple segments for system scalability. In this context, each segment operates with limited memory and disk space, necessitating a delicate balance between accuracy, efficiency, and space cost. Existing disk-based methods fall short as they do not holistically address all these requirements simultaneously.

In this paper, we present Starling, an I/O-efficient disk-resident graph index framework that optimizes data layout and search strategy within the segment. It has two primary components: **(1)** a data layout incorporating an in-memory navigation graph and a reordered disk-based graph with enhanced locality, reducing the search path length and minimizing disk bandwidth wastage; and **(2)** a block search strategy designed to minimize costly disk I/O operations during vector query execution. Through extensive experiments, we validate the effectiveness, efficiency, and scalability of Starling. On a data segment with 2GB memory and 10GB disk capacity, Starling can accommodate up to 33 million vectors in 128 dimensions, offering HVSS with over 0.9 average precision and top-10 recall rate, and latency under 1 millisecond. The results showcase Starling's superior performance, exhibiting 43.9× higher throughput with 98% lower query latency compared to state-of-the-art methods while maintaining the same level of accuracy.

[*] Work done while working with Zilliz.
[†] Both authors contributed equally to this research.

Authors' addresses: Mengzhao Wang, Zhejiang University, China, wmzssy@zju.edu.cn; Weizhi Xu, Zilliz, USA, Weizhi.Xu@zilliz.com; Xiaomeng Yi, Zhejiang Lab, China, xiaomeng.yi@zhejianglab.com; Songlin Wu, Tongji University, China, wusonglin@tongji.edu.cn; Zhangyang Peng, Hangzhou Dianzi University, China, pengzhangyang@hdu.edu.cn; Xiangyu Ke, Zhejiang University, China, xiangyu.ke@zju.edu.cn; Yunjun Gao, Zhejiang University, China, gaoyj@zju.edu.cn; Xiaoliang Xu, Hangzhou Dianzi University, China, xxl@hdu.edu.cn; Rentong Guo, Zilliz, USA, Rentong.Guo@zilliz.com; Charles Xie, Zilliz, USA, charles.xie@zilliz.com.

ACM 2836-6573/2024/2-ART14
https://doi.org/10.1145/3639269

CCS Concepts: • **Information systems → Record and block layout**; **Query optimization**; **Retrieval efficiency**.

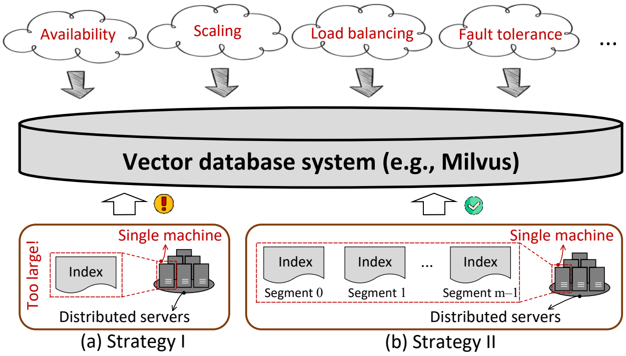
Additional Key Words and Phrases: high-dimensional vector, approximate nearest neighbor search, range search, disk-based graph index, block shuffling

ACM Reference Format:
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 (SIGMOD), Article 14 (February 2024), 27 pages. https://doi.org/10.1145/3639269

1 INTRODUCTION

The management of unstructured data, including video, image, and text, has become an urgent requirement [62]. A notable advancement in this domain has been the widespread adoption of learning-based embedding models, which leverage high-dimensional vector representations to enable effective and efficient analysis and search of unstructured data [35, 57]. High-dimensional Vector Similarity Search (HVSS) is a critical challenge in many domains, such as databases [24, 65], information retrieval [26, 30], recommendation systems [18, 50], scientific computing [48, 74], and large language models (LLMs) [7, 12, 41]. The computational complexity associated with exact query answering in HVSS has spurred recent research efforts toward developing approximate search methods [24, 46, 65]. While various compact index structures and intelligent search algorithms have been proposed to achieve a balance between efficiency and accuracy [39, 42, 65], the majority of these approaches assume that both the vector dataset and its corresponding index can be accommodated in the main memory (DRAM). However, in practical scenarios, the sheer volume of vector data often surpasses the capacity of the main memory, necessitating substantial expansion of main memory resources [4, 15, 37] to support state-of-the-art in-memory HVSS algorithms like NSG [24] and HNSW [46]. To illustrate, constructing an HNSW index based on one billion floating-point vectors in 96 dimensions would require more than 350GB of memory, resulting in out-of-memory failures [56] when executed on a typical single server [3]. Consequently, there is a growing demand for disk-based methods that leverage solid-state disks (SSDs) as a storage and retrieval medium for vector data, thereby circumventing the constraints imposed by main memory [15, 33, 70]. This paradigm shift allows for handling massive-scale vector datasets on a single machine, accommodating billions of vectors with improved efficiency and scalability.

However, maintaining a single large vector index on a solitary machine (Fig. 1(a)) proves impractical for vector databases, as it constrains many essential system features for industrial applications [27]. For example, constructing a sole DiskANN index for a dataset at a billion-scale may demand over five days and peak memory usage of 1,100GB [33], significantly impacting system availability. Additionally, a large index exhibits poor migration capabilities, essential for scaling, load balancing, and fault tolerance [27]. Consequently, **vector databases consistently partition large-scale data into *multiple* segments and assign an appropriate number of segments to a single machine** (Fig. 1(b)) [27, 38, 62]. Recent trends show that data fragmentation has become commonplace in mainstream vector databases [22, 27, 38]. Each segment operates with limited storage capacity and computing resources [4, 27], and an independent medium-sized index is constructed on each segment for autonomous searching. Leveraging specific data fragmentation strategies and a query coordinator allows us to search only *a few* segments of a machine during vector query execution [20, 27, 71]. This poses an open problem of efficient and accurate HVSS within a data segment in vector databases [6, 27]. In general, we may manage tens of millions of vectors on a typical segment with only several gigabytes of space [4, 27]. The space left for index storage is very



(a) Strategy I

(b) Strategy II

Fig. 1. Two indexing strategies on a single machine for vector database system. In a distributed setting, different machines share the same strategy [27].

limited. Therefore, HVSS within the segment necessitates a meticulous equilibrium between *search performance* (encompassing accuracy and efficiency) and *space cost*.

No existing work has achieved effective HVSS on the data segment, considering both search performance and space cost. An immediate attempt is to use existing disk-based methods on the segment to handle large-scale vector data. However, as reported in the literature [15, 56] and also empirically verified in our experiments (**§6.2**), these techniques are in a quandary in this scenario. For example, two state-of-the-art disk-based schemes—SPANN [15] and DiskANN [33]—lie in either of the two following extremes. At one extreme, SPANN [15] requires huge disk space to support efficient vector queries in the segment since each vector is copied multiple times (up to eight times [15]). Moreover, in vector databases, a segment may have multiple replicas that are distributed on different machines for fault tolerance [27]. This will further increase the disk space overhead for SPANN. At the other extreme, the vector index of DiskANN [33] can fit the segment's disk capacity but DiskANN suffers from high latency due to the large number of random disk I/Os during the search process [15, 33]. Therefore, it is particularly challenging to achieve a balance between search performance and space cost for HVSS on data segments.

We find that current disk-based methods for HVSS still follow the paradigm of memory-based solutions in their data layout and search strategy. This leads to numerous disk I/Os, which limit search performance under a given space cost. We illustrate this with DiskANN [33], the state-of-the-art disk-resident graph index algorithm[1] (cf. **§2.2**). First, each hop along the search path corresponds to a disk I/O request. The search procedure on the graph index executes a sequence of disk I/O requests, which results in high search latency. Second, the disk bandwidth is underutilized due to poor data locality. For each disk I/O in the search procedure, only one vertex's information (a vector and its neighbor IDs) is *relevant*, which typically takes a few hundred bytes. However, the smallest disk I/O unit is a block, which is usually 4KB in size. Therefore, most of the data read from the disk is wasted. According to our evaluation on the BIGANN dataset [3], DiskANN's disk I/O operation takes up to 92.5% of the time in the search procedure, and up to 94% of the vertices (they are *irrelevant*) in each loaded block are wasted (i.e., the vertex utilization ratio is low). Thus, there is much potential to achieve an I/O-efficient HVSS using graph index on the data segment.

In this paper, we introduce an I/O-efficient di**s**k-residen**t** gr**a**ph index f**r**amework tailored for high-dimensiona**l** vector s**i**milarity search o**n** a data se**g**ment, called `Starling`. `Starling` optimizes data layout and search strategy to improve the search performance without additional space cost. The data layout consists of a sampled in-memory navigation graph and a reordered disk-based

[1] We exclude SPANN as it duplicates each vector up to eight times, which far exceeds the capacity of the data segment when serving tens of millions of vectors.

graph, which help find query-aware dynamic entry points and improve data locality, respectively. Based on this data layout, we design a search strategy that reduces disk I/Os by shortening the search path and increasing the vertex utilization ratio. Specifically, `Starling` builds a navigation graph in memory by sampling a small portion of vectors. This allows the search procedure to quickly find some vertices that are close to the query vector without disk I/O, and then start searching on the disk-based graph from these vertices. To improve data locality on disk, `Starling` reorganizes the graph index layout by shuffling the vertices and storing them with their neighbors in the same block (i.e., *block shuffling*). This way, one disk I/O can load information from multiple relevant vertices in the search path. Based on this, we design a block search strategy that checks all the data from disk I/O and adds the ones that are potentially useful to the search sequence. We further improve the strategy by three computation-specific optimizations. Finally, we also propose efficient algorithms for approximate nearest neighbor search (ANNS) and range search (RS) queries based on the optimized search procedure.

To the best of our knowledge, this is the first work that tackles the fundamental data locality problem of disk-based graph index for the data segment by effective block shuffling algorithms. Our evaluation shows that `Starling` achieves 43.9× higher throughput with 98% lower query latency than state-of-the-art methods under the same accuracy. The main contributions of this work are:

- We present `Starling`, an I/O-efficient disk-resident graph index framework with excellent universality that enhances the search efficiency of various graph index algorithms (e.g., Vamana [33], NSG [24], and HNSW [46]) on a data segment (**§3**, **§6.7**).
- We design a data layout for the data segment that consists of two components: an in-memory navigation graph (**§4.2**) and a reordered disk-based graph (**§4.1**). This data layout reduces the search path length and increases the vertex utilization ratio of each loaded block when searching on the disk-based graph index.
- We prove that the block shuffling problem is NP-hard and has no polynomial time approximation algorithm with a finite approximation factor unless P=NP (**§4.1**). Hence, we devise three effective heuristic shuffling algorithms that improve the locality of the disk-based graph index (**§4.1**).
- We propose a block search strategy that exploits the data locality of the reorganized index layout to reduce disk I/Os (**§5.1**). Based on this strategy, we also provide three computation-specific optimizations (**§5.1**) and develop search algorithms for two major types of queries in vector databases (**§5.2** and **§5.3**).
- We implement `Starling` and evaluate it on four real-world datasets to verify its effectiveness, efficiency, and scalability (**§6**). We show that search efficiency can be improved significantly by simply adjusting the index layout on the disk.

2 PRELIMINARIES

In this section, we formulate two critical queries in vector databases. Then we explain why current methods are inefficient on the data segment and state our optimization goal in this paper.

2.1 Two Query Types for HVSS

K Nearest Neighbor Search (KNNS). Given a vector dataset $X = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^D$, a query vector $q \in \mathbb{R}^D$, a distance function $dist(\mathbb{R}^D, \mathbb{R}^D) \rightarrow \mathbb{R}$, and a positive integer k $(0 < k < n)$, the KNNS problem finds a set R_{knn} of k vectors from X that are closest to q such that for any $x_i \in R_{knn}$ and $x_j \in X \setminus R_{knn}$,

$$dist(x_i, q) \leq dist(x_j, q) \quad . \tag{1}$$

The distance function can be Euclidean distance or others.

The KNNS problem requires scanning all the vectors in the dataset, which is impractical for large vector datasets. Hence, most studies try to find approximate results that are close to the exact KNNS results. This is the Approximate Nearest Neighbor Search (ANNS) problem [1, 23, 24, 46]. ANNS uses $Recall \in [0, 1]$ to measure the accuracy of approximate results. Let R'_{knn} be the approximate results, then $Recall$ is

$$Recall = \frac{|R_{knn} \cap R'_{knn}|}{k} \quad . \tag{2}$$

A higher recall means a more accurate result. Many studies have designed different index structures for vector data to improve the trade-off between search efficiency and accuracy.

Range Search (RS). Unlike the KNNS problem that returns a fixed number of results, the RS problem retrieves all the vectors that are within a given distance r from the query vector q. Average precision $(AP) \in [0, 1]$ is a metric to measure the accuracy of approximate RS results. Given the exact result R_{range} and an approximate solution that ensures all the returned results R'_{range} are within r, then AP can be computed as follows:

$$AP = \frac{|R'_{range}|}{|R_{range}|} \tag{3}$$

2.2 HVSS on Data Segment.

Vector databases divide large-scale data into multiple segments and distribute them across servers [22, 27]. Each server may manage numerous segments, processing vector queries either in parallel or serially through a query coordinator. Typically, each segment operates within strict memory and disk space constraints [4, 27], often possessing less than 2GB of memory and under 10GB of disk capacity. Data segment facilitates system scalability [27], yet it also poses challenges for HVSS within a data segment. Within this context, our objective is to achieve high search accuracy and efficiency within these space limitations. This necessitates a delicate equilibrium between search performance and the space cost for HVSS. For instance, on the BIGANN dataset [3], each segment might accommodate 33 million vectors, requiring 4GB of storage. Consequently, the remaining space for index storage becomes exceedingly limited. Note that we consistently construct a substantial index to achieve a better accuracy and efficiency trade-off. In the following, we delve into the reasons why current HVSS advancements struggle within the context of data segments.

Mainstream in-memory algorithms such as HNSW [46], NSG [24], and HVS [43] encounter challenges when handling tens of millions of vectors on a segment due to their approach of loading all vectors and index into memory, exceeding memory limitations. While some vector compression methods like LSH [31] and PQ [34] store compressed vectors in memory, the compression process introduces significant errors that notably degrade search accuracy. For instance, the top-1 recall rate of the leading compression method seldom surpasses 0.5 [33]. Disk-based solutions present more promise as they require less memory and achieve high search accuracy. Traditional disk-based methods based on trees are excluded due to their susceptibility to the "curse of dimensionality" when addressing HVSS [42]. Instead, the focus is directed towards two state-of-the-art disk-based methods for HVSS: SPANN [15] and DiskANN [33]. SPANN achieves high search accuracy and efficiency but at the cost of substantial storage space. It may replicate each vector up to eight times, leading to extensive disk overhead that far surpasses the capacity of the data segment when managing tens of millions of vectors. On the other hand, DiskANN follows the graph index algorithm and achieves superior search performance with less disk capacity. It has served as a baseline for HVSS on disk [3]. However, its data layout and search strategy still align with the paradigm of memory-based solutions, necessitating disk I/O for each vertex access. In summary, devising an efficient and

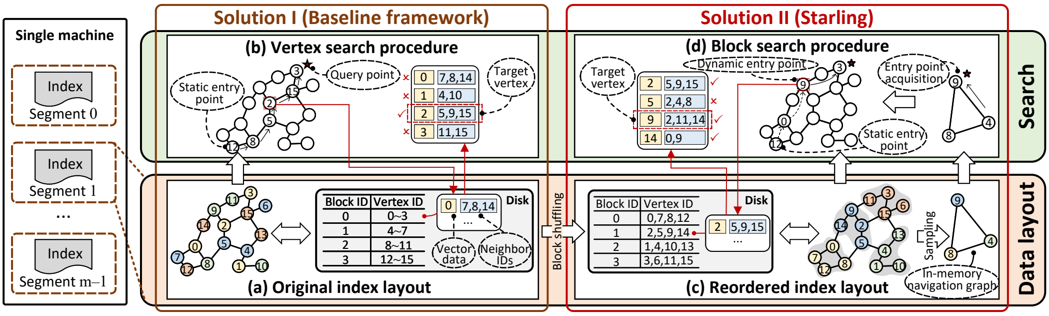


Fig. 2. Illustration of the data layouts and search strategies for the baseline and Starling, respectively.

accurate solution for HVSS on a data segment remains challenging, involving the management of tens of millions of high-dimensional vectors within limited space.

2.3 Our Optimization Objective

We draw two conclusions from our observations. First, the disk-resident graph index proves to be the optimal choice for HVSS on the data segment, offering improved search performance with minimal space overhead [56]. Second, the disk-based graph index can be further optimized for I/O-efficiency in HVSS. Consequently, the HVSS problem on the data segment can be framed as enhancing the I/O-efficiency of disk-resident graph index. Let $T_{I/O}$, T_{comp}, and T_{other} represent the I/O time, computation time, and other times (such as data structure maintenance) when conducting searches on the disk-based graph, respectively. The total search time can be denoted as

$$T_{total} = T_{I/O} + T_{comp} + T_{other}. \tag{4}$$

In Eq. 4, $T_{I/O}$ emerges as the dominant factor influencing the search efficiency. For instance, upon visualizing the search time costs of DiskANN on BIGANN (Fig. 11(d)), it becomes evident that $T_{I/O}$ constitutes up to 92.5% of T_{total}, while T_{comp} and T_{other} collectively occupy less than 7.5%. This highlights the efficiency bottleneck as the cost of disk I/Os for HVSS on the disk-resident graph index. Therefore, our objective is to improve search efficiency by minimizing $T_{I/O}$. Note that we must not introduce additional space overhead (including memory and disk), compared to the original graph index, in order to adhere to the space capacity limitation of the data segment.

3 DESIGN PHILOSOPHY

We analyze two key factors that affect I/O time for existing graph index methods. Then we give an overview of Starling, illustrating its data layout and search strategy on the data segment.

3.1 I/O-efficiency Analysis

When conducting searches on the disk-based graph, I/O-efficiency depends on two primary factors: the vertex utilization ratio in each disk I/O and the length of the search path. The former indicates the extent to which useful vertices are loaded in a given block, while the latter denotes the number of hops required from the entry point to the result point. Both of these factors influence the number of disk I/Os and thus determine the I/O time. A higher vertex utilization ratio signifies that more relevant information is loaded in each disk I/O, leading to more effective utilization of disk bandwidth and requiring fewer disk I/Os for each query. A longer search path implies an increased number of disk I/Os during the search process. We delve into an analysis of the vertex utilization ratio and the search path length within the *baseline framework*[2] and reveal two underlying issues.

[2] Unless specifically stated, we refer to DiskANN as the baseline framework.

Problem 1: Poor data locality. Fig. 2(a) shows that the baseline assigns ID-consecutive vertices (a vertex contains the vector data and neighbor IDs) to the same blocks. For example, block **0** stores vertices **0**~**3**. Since a block is the smallest disk I/O unit, reading vertex **2** also reads vertices **0**, **1**, and **3**. This wastes disk bandwidth as the other three vertices are irrelevant to vertex **2**. A naive way to avoid the wastage is to check all the vertices in a block. However, a segment has up to $10^7\times$ more vertices than a block in a real-world scenario. The probability of finding a near vertex (w.r.t the target vertex) among the non-target vertices[3] in the block is too low to improve efficiency effectively. Therefore, the baseline only checks the target vertex and discards the rest. This means a low vertex utilization ratio in each disk I/O. According to our evaluation on BIGANN (Tab. 2), up to 94% of data read from the disk is wasted. Obviously, ID-consecutive vertices in a block do not imply spatial proximity (e.g., vertices of the same color are scattered on the graph topology in Fig. 2(a)). Therefore, the baseline has poor data locality.

Problem 2: Long search path. The I/O complexity of searching on a disk-based graph index is proportional to the search path length. The baseline framework uses a fixed or random vertex as the entry point for the search. However, a segment may have tens of millions of vertices, so the entry point may be far from the query. In this case, only the last few vertices in the path are near the query and likely to be in the final result. However, we need to read each vertex along the path from the disk. Fig. 2(b) shows an extreme example where the entry point is the farthest vertex from the query point. It takes five hops to search from **12** to **3**. If we start from **9**, we can reduce the hops to two. For a dataset of tens of millions of vectors, even searching for only the top-10 nearest neighbors may generate a path of hundreds of hops to achieve high accuracy. Therefore, the baseline has a long search path.

3.2 Framework Overview

We present Starling, a framework designed to enhance the I/O-efficiency of disk-based graph index for HVSS on a data segment. While a single machine may encompass multiple segments, they all share the same index layout and search strategy. Therefore, our primary focus is on one specific segment. As depicted in Fig. 2(c), Starling preserves the graph's topology while optimizing the data layout to augment the vertex utilization ratio and diminish the search path length. Moreover, it employs a block search strategy aimed at reducing disk I/Os. Below, we provide an overview of the data layout and search strategy employed by Starling:

Data layout on disk. Starling improves data locality by shuffling the data blocks in accordance with the graph topology (Fig. 2(c) left). This aligns the data layout with the search procedure, which tends to visit neighboring vertices [65]. Specifically, Starling endeavors to store a vertex and its neighbors in the same block, enabling a single disk read to fetch not only the target vertex but also other likely candidates. This approach increases the vertex utilization ratio, as each disk I/O operation brings multiple relevant vertices. Furthermore, the search procedure can more readily jump to a closer vertex from the query, potentially reducing the number of required disk I/Os.

Data layout in memory. Starling identifies query-aware entry points for the disk-based graph using an in-memory navigation graph (Fig. 2(c) right). This approach incorporates the concept of multi-layered graphs [43, 46, 56] and involves sampling a small fraction of vectors from the disk-based graph to construct the navigation graph. During this process, any in-memory graph algorithm [23, 24] can be utilized. Given a query, Starling initially explores the in-memory navigation graph to obtain query-close vertices as entry points. Subsequently, it initiates the disk-based graph search from these identified points. This methodology effectively reduces the search path length on the disk-based graph.

[3] We load a block according to a target vertex. In this example, the target vertex is **2**.

Search strategy. Starling uses a block search strategy to exploit data locality (Fig. 2(d)). Unlike the baseline, which explores data on a vertex basis, this strategy processes data by blocks. This way, it benefits from the optimized data layout. For each loaded block, Starling computes the distance to the query for all vertices. Then, it selects the vertices that are close to the query and checks their neighbors for new search candidates. This strategy lowers the disk operation cost by exploring more relevant data per block but increases the computation cost. We further optimize the performance with three computation-specific optimizations (**§5.1**).

Example 1 shows how Starling reduces disk I/Os.

Example 1. Fig. 2(b) shows a vertex search strategy that starts from vertex **12** and reaches the result vertex **3** in five hops for a given query point. This strategy needs at least six disk I/Os to access the vertices in the path and their neighbors. This is inefficient, especially for large-scale scenarios, because the probability of finding a vertex close to the query among the non-target vertices in a loaded block is very low. Therefore, exploring data on a block basis with the original index layout is more harmful than helpful. In contrast, Starling reduces the disk I/Os to three with a reordered index layout, as shown in Fig. 2(d). The block search procedure works as follows: (1) It loads block **0** and visits vertices **12**, **0**, **7**, and **8** in the block. Then it computes their distance to the query and selects **0** to visit its neighbors. (2) It loads block **1** with vertices {**2**, **5**, **9**, **14**} and chooses **9** as the next hop. Vertex **3** as the search result will be found when Starling loads the next block according to vertex **11**. With a navigation graph built on the vectors of {**4**, **8**, **9**} in memory, it obtains **9** as the entry point for disk-based graph search. Since **9** is close to the query, it only takes two disk I/Os (one to visit the entry point and one to visit its neighbors) to get the query result.

4 DATA LAYOUT

Fig. 2(c) shows how Starling organizes data on a segment. First, it builds a disk-based graph index on the full dataset and rearranges it by block shuffling to improve data locality. We can use different methods to construct Starling's disk-based graph, such as NSG [24], HNSW [46], and Vamana [33]. For more details on these methods, please refer to the original papers. We do not focus on developing a specific graph index algorithm since existing ones are well-studied but not suitable for disk deployment. Instead, we address the block shuffling problem on the disk-based graph (**§4.1**). Second, Starling samples some data points from the full dataset and builds an in-memory navigation graph (**§4.2**). This structure allows searching on the disk-based graph to begin from some query-aware entry points, which reduces the search path length.

4.1 Block Shuffling on the Disk

Notations. Let $G = (V, E)$ represent a disk-based graph index, where V and E denote the sets of vertices and edges, respectively. The Edges are directed and are stored as adjacency lists of vertices. Each vertex necessitates γ KB of storage to house its vector data, neighbor count λ, and a list of neighbor IDs (with a maximum length of Λ)[4]. The size of a disk block is η KB. Since we do not split the data of a vertex into two blocks, each block can accommodate at most $\varepsilon = \lfloor \eta/\gamma \rfloor$ vertices. Therefore, $\rho = \lceil |V|/\varepsilon \rceil$ blocks are required to store the graph index.

Next, we give the definition of block-level graph layout:

Definition 1. ***Block-Level Graph Layout.*** *The block-level graph layout of a graph index is a scheme that assigns $|V|$ vertices to ρ blocks.*

[4] Each vertex is allocated Λ ID spaces for alignment, and padding is added when $\lambda < \Lambda$. For simplicity, the neighbor count λ is omitted in Fig. 2 and 3.

Example 2. Fig. 2(a) shows a graph index $G = (V, E)$ with 16 vertices ($|V| = 16$), four vertices per block ($\varepsilon = 4$), and four blocks in total ($\rho = 4$). For DiskANN [5] on 33 million BIGANN dataset ($|V| = 3.3 \times 10^7$) [3], each vector is 128-dimensional and one byte per value. If Λ is 31 and η is 4 KB, then $\gamma = (128 + 4 + 31 \times 4)/1,024$ KB (ID is unsigned integer type), $\varepsilon = 16$, and $\rho = 2,062,500$. Thus, DiskANN puts 16 ID-contiguous vertices in a block, such as $0 \sim 15$.

Fig. 2(a) and (c) show two different ways of organizing the graph index on disk block level. As discussed in **§3.1** (Problem 1), the graph layout affects disk I/Os during searching. Specifically, a layout without data locality lowers the vertex utilization ratio and increases random disk I/Os. We use the overlap ratio $OR(G)$ of the graph index G to measure the locality of the graph layout. For any vertex $u \in V$, $OR(G)$ is the average of $OR(u)$ over V, where $OR(u)$ is the proportion of vertices that are u's neighbors among all the vertices except u in the block. We can compute $OR(u)$ by

$$OR(u) = \begin{cases} \frac{|B(u) \cap N(u))|}{|B(u)|-1} & |B(u)| > 1 \\ 0 & |B(u)| \leq 1 \end{cases} , \tag{5}$$

where $B(u)$ is the set of vertices in the block that contains u and $N(u)$ is the set of u's neighbors ($|N(u)| \leq \Lambda$). In real-world datasets, the maximal vertex count ε is about 10 in each block and there is at most one block with a different ε (i.e., $|V|$ is not divisible by ε) in graph layout. We use $OR(B(u)) = \sum_{v \in B(u)}(OR(v)/|B(u)|)$ to denote the overlap ratio of the block $B(u)$. $OR(G)$ is calculated as $\sum_{u \in V}(OR(u)/|V|)$. A higher $OR(G)$ means better data locality and a higher vertex utilization ratio in a loaded block.

Example 3. A graph layout with optimal data locality has $OR(G) = 1$, meaning that every vertex in a block is a neighbor of any other vertex in the block. However, we get $OR(G)$ close to 0 for the DiskANN [33] graph index on the 33 million BIGANN (Fig. 9(a)). This indicates the poor data locality of the DiskANN graph layout.

To enhance data locality, we use *block shuffling* to adjust the graph layout, which is defined as:

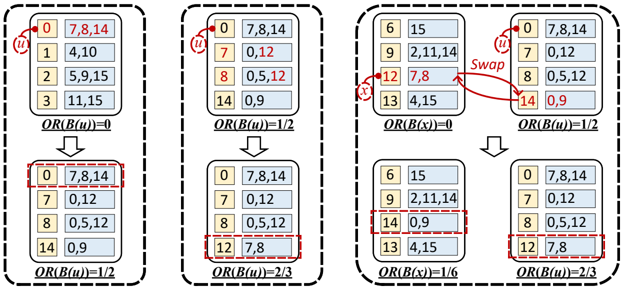
Definition 2. ***Block Shuffling.*** *Given a graph layout for a disk-based graph index G, the block shuffling aims to get a new layout that maximizes the $OR(G)$ while satisfying Def. 1.*

In Def. 2, we aim to get a graph layout where every vertex in a block is a neighbor of other vertices within the same block. However, the graph index built on high-dimensional vectors is complex, as the neighborhood relationship encompasses both navigation and similarity aspects [23, 42]. A vertex may have neighbors that belong to different clusters [46], and all vertices exhibit a constant out-degree. Hence, it is challenging, if not impossible, to ensure that any two vertices in a block are mutual neighbors. We prove that the block shuffling problem is NP-hard in Theorem 4.1. Furthermore, it lacks a polynomial time approximation algorithm with a finite approximation factor. This motivates our block shuffling research, as the baseline framework merely populates a block with ID-contiguous vertices.

Theorem 4.1. *The block shuffling problem is NP-hard and does not have a polynomial time approximation algorithm with a finite approximation factor unless P=NP.*

Proof. (Sketch.) We reduce the block shuffling problem to the triple shuffling problem, which is strongly NP-complete [10, 55]. The triple shuffling problem is defined as follows: given $t = 3 \cdot \rho$ integers $\alpha_0, \alpha_1, \cdots, \alpha_{t-1}$ and a threshold Ω such that $\Omega/4 < \alpha_i < \Omega/2$ and

$$\sum_{i=0}^{t-1} \alpha_i = \rho \cdot \Omega \quad , \tag{6}$$



(a) Algorithm I: BNP (b) Algorithm II: BNF (c) Algorithm III: BNS

Fig. 3. Block shuffling. Refer to Fig. 2 for graph topology.

the task is to partition the numbers into ρ triples and DECIDE if these triples can be shuffled by swapping numbers between triples so that each triple sums up to Ω. We construct a graph G with cliques of size α_i for each integer α_i. Let Ω and ρ be the number of vertices in a block and the number of blocks in the graph layout of G, respectively. We demonstrate that the block shuffling problem on G has a solution iff the triple shuffling problem can be solved. Consequently, the block shuffling problem is NP-hard. Assuming the existence of a polynomial time approximation algorithm with a finite approximation factor for the block shuffling problem, we could use it to solve the triple shuffling problem. However, we establish that the triple shuffling problem cannot be solved by such an approximation algorithm with a finite approximation factor. This contradicts the assumption. (The detailed proof is available in our technical report [64]) □

We design three heuristic algorithms to handle the shuffling problem. We first describe a straightforward solution (*Algorithm I*), and then present two optimized algorithms (*Algorithms II–III*).

Algorithm I: Block Neighbor Padding (BNP). This algorithm fills disk blocks in a one-by-one fashion. To fill blocks, it checks vertices in ascending order of IDs. If a vertex u is not assigned to any block yet, then the algorithm tries to assign the vertex and its neighbors to the current block. Once a block is full the algorithm opens a new block and assigns vertices to it. BNP has a time complexity of $O(|V|)$ and improves the overlap ratio $OR(G)$ by assigning vertices and their neighbors to the same block. However, the improvement is limited as some neighbors of u (denoted as $z, v \in N(u)$) may not be adjacent to each other, which lowers $OR(z)$ or $OR(v)$. Also, some neighbors of u may have been assigned to other blocks earlier, such as z also being a neighbor of o with a smaller ID than u. It cannot be stored with u as we store each vertex only once.

Example 4. In Fig. 3(a), we have $B(u) = \{$ **0**, **1**, **2**, **3** $\}$ and $OR(B(u)) = 0$ for the original layout (the ID of u is **0**). BNP puts **0** and its neighbors **7**, **8**, **14** in $B(u)$, so that $B(u) = \{$ **0**, **7**, **8**, **14** $\}$ and $OR(B(u)) = 1/2$. But vertex **12** cannot be in the same block with its neighbors **7** and **8**, that is $OR(v) = 0$ for vertex v with ID **12**.

Algorithm II: Block Neighbor Frequency (BNF). To further optimize $OR(G)$, we propose BNF, which aims to assign a vertex to the block that holds most of its neighbors, i.e., the block with the highest neighbor frequency. The BNF takes the result of BNP as an initial layout and optimizes $OR(G)$ iteratively. As illustrated in Algorithm 1: (1) It stores the current mapping of vertex IDs to block IDs and clears all the blocks of G (lines 3–5). (2) For each u in V, BNF tries to assign it to the blocks that hold its neighbors in the previous iteration (lines 6–14). The algorithm checks blocks in descending order of neighbor count and assigns u to the first block that is not full yet. If all the blocks are full, then BNF puts u in an empty block in $\mathcal{B}$ (lines 13–14). (3) It repeats this process until it reaches the iteration limit β or the $OR(G)$ gain between two iterations falls below a

Algorithm 1: Block Shuffling by BNF

```
   Input: block-level graph layout of $G = (V, E)$ from BNP, maximum iterations $\beta$, $OR(G)$ gain
          threshold $\tau$
   Output: new block-level graph layout of $G$
1  $\mathcal{B}$={$B_0, \cdots, B_{\rho-1}$} ⟵ all blocks                         ▷ $\rho$ is the number of blocks
2  while $iterations \leq \beta$ do
3      $D$ ⟵ mapping of vertex IDs to block IDs;
4      forall $B_i \in \mathcal{B}$ do
5          $B_i \longleftarrow \emptyset$;                                          ▷ clear all blocks
6      forall $u \in V$ do
7          $H \longleftarrow \bigcup_{a \in N(u)} \{D(a)\}$;                          ▷ all neighbors' block IDs
8          while $H \neq \emptyset$ do
9              $x$ ⟵ block ID with the most neighbors in $H$;
10             if $B_x$ is not full then
11                 $B_x \longleftarrow B_x \cup \{u\}$; break;
12             $H = H \setminus \{x\}$;                                            ▷ remove the block that is full
13         if $H = \emptyset$ then
14             add $u$ to a empty block in $\mathcal{B}$;
15     if $OR(G)$ gain $< \tau$ then
16         break;
17 return new layout of $G$
```

given threshold (lines 15–16). Then a new layout of G is returned. BNF has a time complexity of $O(\beta \cdot o \cdot |V|)$, where β is the number of iterations, o is the average out-degree, and $|V|$ is the number of vertices. This is because BNF needs to access the neighbors of all vertices in each iteration.

Example 5. As shown in Fig. 3(b), BNF replaces **14** with **12** in $B(u)$, since **12** has two neighbors, **7** and **8**, in the block. Thus, $B(u)$ becomes { **0**, **7**, **8**, **12** } and $OR(B(u)) = 2/3$.

Algorithm III: Block Neighbor Swap (BNS). The design of BNS is inspired by the NN-Descent method [21] in the literature of graph construction. BNS refines $OR(G)$ from an initial layout, which could be the result of BNP or BNF. In BNS, for a pair of vertices a and e that are neighbors of vertex u and belong to different blocks $B(a)$ and $B(e)$, BNS swaps vertices with the lowest overlap ratio (OR) in $B(a)$ and $B(e)$ to increase the sum of $OR(B(a))$ and $OR(B(e))$. Let o be the average out-degree, then the number of swaps is o^2 for each vertex in V. In each swap, the time complexity of computing one block's OR is $O(o \cdot \varepsilon)$, where ε is the number of vertices in a block. BNS operates in an iterative manner, where each iteration checks vertex pairs among the neighbor set of all vertices. Therefore, the time complexity of BNS is $O(\beta \cdot o^3 \cdot \varepsilon \cdot |V|)$, considering the number of iterations β. We prove that $OR(G)$ does not decrease with the number of iterations in Lemma 4.2.

Example 6. In Fig. 3(c), BNS identifies the blocks $B(u)$ and $B(x)$ that contain vertices **0** and **12**, respectively, from the neighbors of vertex **7**. Then, it swaps vertices **12** and **14**, which have the lowest OR in their blocks and can increase $(OR(B(x)) + OR(B(u)))$ by swapping. Finally, we get $OR(B(x)) = 1/6$ and $OR(B(u)) = 2/3$.

Lemma 4.2. *In BNS, the $OR(G)$ is a monotonically non-decreasing function of the number of iterations β.*

Proof. In each iteration, an update is local and affects only two blocks' vertices. For any two neighbors a and e of vertex u ($B(a) \neq B(e)$), we prove that swapping vertices does not lower the sum of $OR(B(a))$ and $OR(B(e))$. Let $OR(B(a))_i$, $OR(B(e))_i$ and $OR(B(a))_j$, $OR(B(e))_j$ be the values of $OR(B(a))$, $OR(B(e))$ before and after an iteration, respectively. We swap two vertices in $B(a)$ and $B(e)$ only if $OR(B(a))_j + OR(B(e))_j > OR(B(a))_i + OR(B(e))_i$. Therefore, the sum of $OR(B(a))$ and $OR(B(e))$ is non-decreasing. □

Analysis. Among our three block shuffling algorithms, BNP emerges as the fastest since it scans all vertices just once. On the other hand, BNF's efficiency is contingent on the number of iterations and does not ensure the convergence of $OR(G)$. However, BNF demonstrates proficiency, both in terms of efficiency and effectiveness, particularly on numerous real-world datasets. Meanwhile, BNS, although possessing the highest time complexity, guarantees that $OR(G)$ does not decrease with iterations. In our implementation, we have parallelized BNF and BNS to enhance their speed. All three algorithms notably improve $OR(G)$ in contrast to the original graph layout, with BNS exhibiting the most significant improvement, followed by BNF and then BNP. In our evaluation (**§6.5**), higher $OR(G)$ improves vertex utilization ratio and search performance. In practice, the choice of algorithm can be tailored to specific requirements. Generally, we recommend BNF due to its adept balance between efficiency and effectiveness.

Time cost. We analyze the extra time cost caused by block shuffling. A recent study of current graph algorithms [65] shows that the graph index construction time complexity is always $O(|V|\log(|V|))$. The index construction involves high-dimensional vector distance calculation, which is very time-consuming. However, block shuffling only scans vertices and performs simple statistics, without any vector calculation. According to our evaluation, the block shuffling procedure introduces a relatively low additional time cost compared to the graph index construction process. For example, BNF only occupies 3%∼10% of the graph index construction cost (see Fig. 8(a)).

Space cost. The disk-based graph index is stored as adjacency lists of vertices. Each vertex contains its vector data, neighbor count, and a list of neighbor IDs. Let Λ and D represent the maximum number of neighbor IDs and the vector dimensionality, respectively. The space complexity of the disk-based graph index is $O(|V| \cdot (D + \Lambda))$. In our experiments, we adjust Λ to conform to the disk capacity constraint of a segment. Note that the space cost of the disk-based graph index remains unchanged before and after block shuffling. This is because we only adjust the order of vertices and do not add any extra information.

Remarks. (1) Our block shuffling methods can work with any block size, not just the default 4KB for general disk. For example, we can extend to 8KB or 16KB blocks by modifying the block size. (2) Block shuffling is similar to but not identical to graph partitioning. Graph partitioning has been extensively studied for real-world graphs (e.g., Social Networks [51, 68]) in graph engines [45, 49, 59, 73], where the out-degree follows a power-law distribution (neighbors tend to cluster together) [8]. However, our graph index is based on high-dimensional vectors, where neighbors exhibit similarity and navigation traits (with about 50% long links [65]) and the out-degree distribution is uniform (neighbors may scatter across clusters) [65], making locality more challenging. Moreover, current graph partitioning methods thrive on real-world graphs with clustering properties but may falter in graph index for vectors. We evaluated some advanced graph partitioning methods for our block shuffling task, but they only gave limited improvement. For example, BNF shows 40% higher $OR(G)$ than an advanced graph partitioning method—KGGGP [53] for graph index built on the SSNPP dataset. Our block shuffling strategies are tailored for graph index with long navigation links [46] and prove well-suited under our problem setting. (3) Some recent works use some graph partitioning methods to reorder graph indexes [17, 32], but they only achieve limited improvement.

In contrast, we design the block shuffling algorithms based on block-level graph layout on the disk, which leads to better disk-based HVSS performance.

4.2 In-Memory Navigation Graph

To reduce the search path length (ℓ), many in-memory graph-based algorithms use an additional structure, such as multi-level Voronoi diagrams [43]. The disk I/O cost depends directly on ℓ for searching on the disk-based graph index, so a shorter ℓ is even more crucial.

Our in-memory navigation graph reduces ℓ by finding better entry points for the disk-based graph search. To obtain such a graph, we employ two steps: (1) *sample data points* and (2) *build a graph index*. First, we randomly sample some data points from all the data in a segment, based on the memory limit of a segment. Second, we use the same algorithm as the disk-based graph (such as HNSW [46], NSG [24]) to build a navigation graph on the sampled data. The navigation graph is memory-resident and can quickly return the dynamic entry points that are close to a query.

Time cost. Starling builds an in-memory graph only on a small data subset V', lowering the time complexity to $O(|V'|\log(|V'|))$, where $|V'|$ is less than 10% of the total number of vertices in a segment. This process is notably faster than constructing the graph on the entire dataset. In our evaluation, the in-memory graph construction demonstrates a low time cost (e.g., only 5.5% of the total index processing time in Fig. 8(a)).

Space cost. The in-memory graph and the disk-based graph share the same storage format. Let Λ' and D denote the maximum number of neighbor IDs and the vector dimensionality, respectively. The space complexity of the in-memory graph is $O(|V'| \cdot (D + \Lambda'))$. In our implementation, we adjust $|V'|$ and Λ' to adhere to the memory limitation of a segment.

5 SEARCH STRATEGY

The search strategy of Starling consists of two components: (1) *vertex search on the in-memory navigation graph* and (2) *block search on the disk-resident graph*. The first component is designed to quickly navigate to the query's neighborhood without requiring disk access. It employs the same vertex search strategy as existing graph algorithms. The vertex search results serve as the entry points for the second component. To exploit the improved data locality, we utilize block search to explore not only the target vertex but also other vertices in the same block for each disk I/O. Next, we will outline the fundamental block search strategy and its optimizations. Following that, we will show how we build the ANNS and RS algorithms based on the block search.

5.1 Block Search

Starling offline assigns vertices and their neighbors to the same block to improve data locality. This way, one online block read from disk provides multiple relevant vertices. The block search strategy efficiently explores all the useful data in a block. It updates the current search results by calculating the distance from each vertex in the block to the query. It also checks the neighbor IDs of the closer vertices for new search candidates. Starling further enhances this strategy with three computation-specific optimizations.

Block pruning. We aim for a graph layout with $OR(G) = 1$, meaning every vertex accessed by the block search is necessary. However, this ideal layout is very hard or impossible to achieve for a graph index built on high-dimensional vectors. Usually, our block shuffling results in $OR(G)$ ranging from 0.3 to 0.6 on many real-world datasets (Fig. 9(a)). This implies that some vertices in a block are irrelevant. To avoid exploring irrelevant data, Starling prunes each loaded block. Specifically, it sorts the vertices in a block by their distance to the query vector in ascending order. Then, it only checks the neighbor IDs of the top-$((\varepsilon - 1) \cdot \sigma)$ vertices for new search candidates, where $(\varepsilon - 1)$ is the number of vertices excluding the target vertex in a block and σ is pruning ratio $(0 < \sigma \leq 1)$. This way, the neighbor IDs of distant vertices are discarded early. In our evaluation, $\sigma = 0.3$ is always the optimal value for better performance.

I/O and computation pipeline. In the block search stage, disk read (DR) and distance computation (DC) are two main operations. They occupy about 80% of the block search time (see Fig. 11(d)). A naive implementation is to execute DR and DC serially—that is—load a block by DR and then select the next hop by DC (as done in DiskANN [33]). However, this schedule is inefficient because DR and DC are idle alternately. This wastes disk bandwidth and CPU. To avoid this, `Starling` uses an I/O and computation pipeline that performs DR and DC concurrently (cf. Algorithm 2). Specifically, `Starling` first executes DC for the target vertex u in the current loaded block (lines 6–7). Then, it immediately conducts the next DR, while also performing DC of other vertices in the block that contains u (line 11). Although each current DR decision does not consider the non-target vertices from the previous DR, it is acceptable as `Starling` fully utilizes the disk bandwidth and CPU. We will show in §6.5, that I/O and computation pipeline reduces query latency significantly.

PQ-based approximate distance. Recall that we need to load the full-precision vectors of all neighbors of the visited vertex to determine the next hop. Block shuffling mitigates such disk access by filling a block with many neighboring vertices. However, the number of neighbor IDs of each vertex is usually several times the number of vertices in a block, so many neighbors still require extra disk I/Os. To address this issue, we use approximate distance instead of exact distance with full-precision vectors. This is based on the observation that the next hop decision can be made by approximate computation with little accuracy damage [14]. Specifically, `Starling` preprocesses the full dataset by PQ [34] (like DiskANN [33]), a popular compression method. It encodes the full-precision vectors into short codes that reside in the main memory. Indeed, `Starling` efficiently obtains the approximate distances between the neighbors (whose full-precision vectors are not in memory) and the query by the short codes without disk I/Os to make the next disk read decision.

Time cost analysis. Let ξ be the vertex utilization ratio and ε be the number of vertices in a disk block. Then $\xi \cdot \varepsilon$ (≥ 1) represents the number of vertices accessed in each disk I/O. The vertex access complexity of the graph index is $O(o \cdot \ell)$ [65], where ℓ is the search path length and o is the average out-degree of visited vertices. Therefore, the I/O time complexity of searching on the disk-based graph is $O((o \cdot \ell)/(\xi \cdot \varepsilon))$. In addition, the computation-specific block search optimizations further enhance search efficiency.

5.2 Approximate Nearest Neighbor Search

The ANNS strategy in `Starling` uses two ordered lists to store the candidates and search results: the candidate set and the result set. It starts the search by adding the entry points of the disk-based graph to the candidate set. These entry points are obtained on the in-memory navigation graph. Then it selects the unvisited vertex that is closest to the query from the candidate set and performs a block search. It updates both sets with the block search outcome. The procedure ends when all the vertices in the candidate set are visited. To improve efficiency, the strategy limits the size of the candidate set to prevent too many candidate vertices. The result set has no size limit because it is sorted only when the search terminates.

`Starling` performs ANNS for a query q as follows (cf. Algorithm 2). (1) It finds the entry points S of q on the in-memory navigation graph G_m (line 1). (2) It initializes a fixed-size candidate set C and a result set R with S (lines 2–3). It sorts C by approximate distance using PQ short codes. (3) It picks the top-1 unvisited vertex u in C as the target vertex and reads the block B containing u from the disk (lines 4–5). (4) It adds u's neighbor IDs to C and u to R (lines 6–7). (5) It prunes some vertices that are far from q from B, adding the closer vertices into B' (line 8). (6) It executes (3) for

Algorithm 2: ANNS

```
   Input: in-memory navigation graph $G_m$, PQ short codes, disk-based graph $G_d$, query $q$,
          candidate set $C$ with fixed size, result set $R$, pruning ratio $\sigma$
   Output: top-$k$ results of $q$
1  $S \leftarrow$ entry points of $q$ by a vertex search strategy on $G_m$;
2  $C \leftarrow S$;                                              ▷ sort by PQ distance to $q$
3  $R \leftarrow S$;                                              ▷ compute exact distance to $q$
4  $u \leftarrow$ top-1 unvisited vertex in $C$;                  ▷ target vertex
5  $B \leftarrow$ load the block including $u$ from $G_d$;        ▷ DR
6  update $C$ according to $u$'s neighbor IDs;                    ▷ DC
7  add $u$ to $R$ based on $u$'s full-precision vector;           ▷ DC
8  $B' \leftarrow$ top-$((\varepsilon - 1) \cdot \sigma)$ vertices in $B \setminus \{u\}$;   ▷ block pruning
9  while $C$ has unvisited vertex do
10     $v \leftarrow$ top-1 unvisited vertex in $C$;              ▷ PQ-based routing
11     conduct line 5 for $v$, and lines 6–7 for the vertices in $B'$ in parallel;
12                                                                ▷ I/O and computation pipeline
13     conduct lines 6–8 for $v$ and the block including it;
14 return top-$k$ vertices in $R$                                 ▷ sort by exact distance to $q$
```

the next top-1 unvisited vertex v and (4) for the vertices in B' in parallel, then it executes (4)–(5) for v (lines 10–12). (7) It terminates when C has no unvisited vertices. Finally, it sorts R by exact distance using full-precision vectors and returns top-k results in R.

5.3 Range Search

A range search (RS) query returns all the vectors within a search radius r. The result length depends on the vector distribution and can vary a lot among queries. For a dataset with tens of millions of vectors, the same r may give zero to thousands of results. So, the RS algorithm should handle different result lengths. A simple RS strategy is to do ANNS repeatedly with different k values to check the result length. However, this is inefficient because it will revisit the same vertices multiple times, causing extra computation and disk I/Os.

Starling performs RS for a query q based on block search. It uses a candidate set C, a result set R, and a kicked set P to store candidate vertices, results, and vertices kicked out from C, respectively. Starling changes the length limit of C dynamically to handle different result lengths. The steps of RS are as follows. (1) It obtains the entry points from the in-memory navigation graph and initializes C and R with them. (2) It iteratively explores C and updates C and R (like ANNS). It also adds the unvisited vertices kicked out from C to P. (3) When all the vertices in C are visited, it calculates the ratio of R's length to C's length. Given a ratio threshold φ, if

$$\frac{|R|}{|C|} \geq \varphi \quad , \tag{7}$$

it doubles the size of C and restarts the search. This is because a high ratio means that most candidates are results. In this case, exploring more vertices may find new results. (4) In the next search, Starling adds the closer vertices (w.r.t q) in P to C and repeats (2)–(3) for unvisited candidates in C. (5) The search stops when Eq. 7 is not met. After changing the length of C, it resumes the search with the previous results in R, candidates in C, and some closer vertices (w.r.t q) in P. This avoids extra computation and disk access. Our evaluation shows that $\varphi = 0.5$ is optimal.

6 EXPERIMENTS

We evaluate the following aspects of Starling: (1) search performance (**§6.2**), (2) I/O-efficiency (**§6.3**), (3) index cost (**§6.4**), (4) ablation study (**§6.5**), (5) parameter sensitivity (**§6.6**), (6) scalability (**§6.7**), (7) query distribution (**§6.8**), (8) segment setup (**§6.9**), (9) large-scale search results (**§6.10**), and (10) billion-scale data (**§6.11**). Our source code, datasets, and additional evaluations [64] are available at: https://github.com/zilliztech/starling.

6.1 Experimental Setting

Table 1. Statistics of experimental datasets on a segment.

Dataset	Data type	Dimensions	Distance function	# Base vector per segment	# Query	Query type
BIGANN	uint8	128	L2	33M	10^4	ANNS/RS
DEEP	float	96	L2	11M	10^4	ANNS/RS
SSNPP	uint8	256	L2	16M	10^5	RS
Text2image	float	200	IP	5M	10^5	ANNS

Datasets. We use four public real-world datasets [3] that vary in data type, dimensions, distance, and query type. Tab. 1 shows their details. Unless specified, we limit the raw vectors per segment to under 4GB and adjust the data scale accordingly for each dataset. We randomly select vectors from standard datasets for a segment. We perform a brute-force search on the selected vectors to get the ground truth. The query sets are the same for a given dataset.

Query workloads. The query workloads' details are provided in Tab. 1. By default, all query sets are derived from real-world scenarios and are not-in-database, meaning they do not have any intersection with the base data. The queries are executed in a random order, and a batch of queries is served using a pool of threads. Each thread is assigned to handle one query at a time.

Compared methods. We compare Starling with two state-of-the-art disk-based HVSS methods that are able to process up to 4GB vector data within the 2GB memory constraint. We do not include in-memory methods (e.g., HNSW [46] and IVFPQ [34]) in the evaluation, because they either run out of memory or have very low accuracy, as reported in previous works [15, 33, 56].

- **DiskANN** [33] is the state-of-the-art disk-based graph index method, which we use as the baseline framework.
- **SPANN** [15] is a disk-based inverted index method that achieves better performance than DiskANN but requires more disk space.
- **Starling** is our framework, which implements all the optimizations proposed in this paper. Unless otherwise specified, we use the Vamana algorithms as the default option in Starling, denoted as Starling-Vamana or simply Starling.

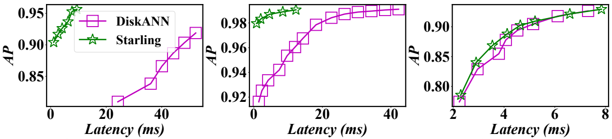
Evaluation protocol. We use *queries per second (QPS)*, *mean latency*, and *mean I/Os* to evaluate the search performance of all competitors. By default, we use eight threads on the server to serve queries and report the *QPS* based on the wall clock time from the query input to the result output. For the latency and I/Os, we record the response time and number of disk I/Os for each query and compute the *mean latency* and *mean I/Os* over all queries. For ANNS, we measure its accuracy by *Recall* (Eq. 2). Unless otherwise stated, we set $k = 10$ in *Recall.* For RS, *AP* (Eq. 3) is a more suitable accuracy metric. We fix a search radius r for each dataset following [60]. We evaluate two popular similarity metrics, $L2$ and inner product (*IP*).

Segment configuration. We follow the configuration of a mainstream open-source vector database, Milvus [6, 27], with at most 2GB memory space and 10GB disk capacity for each segment by default.

Setup. We execute the C++ codes of all methods using different instances for both index building and search to align with the protocols of current vector databases [27] and related research [24, 65]. For index building, we employ a n2-standard-64 instance (ubuntu-2004-focal-v20221121, 2 TB SSD

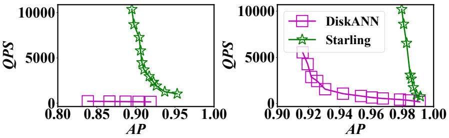
Persistent Disk) with 64 vCPUs—this facilitates fast construction and segment sharing. In terms of search functionality, vector databases typically leverage multiple smaller instances to improve query performance and promote fine-grained load balancing and scaling. In our experiments, all segments share a n2-standard-8 instance (ubuntu-2004-focal-v20221121, 375 GB NVMe Local SSD Scratch Disk) with 8 vCPUs. We use the o_direct option to read data from the disk for all competitors, circumventing operating system caching. We optimize all approaches' hyper-parameters for the segment's configuration. We report the average results of three trials on the optimum configuration.

6.2 Search Performance



(a) BIGANN (b) DEEP (a) SSNPP

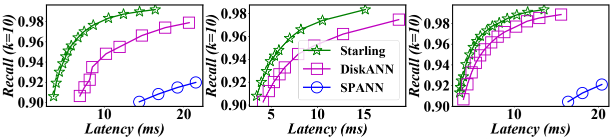
Fig. 4. RS performance (*AP* vs *Latency*).



(a) BIGANN (b) DEEP

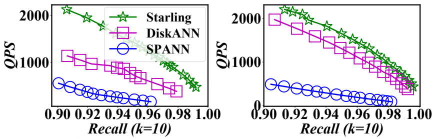
Fig. 5. RS performance (*QPS* vs *AP*).

RS performance. Current disk-based HVSS methods largely ignore RS, except for a baseline in a competition [3]. RS support is provided by calling ANNS iteratively on DiskANN (referred to as DiskANN) [5]. Fig. 4 and 5 compare the RS performance of DiskANN and Starling. Under the same *AP*, Starling reduces query latency by up to 98% compared to DiskANN. On SSNPP, Starling exhibits a slight performance boost. We observed that most query results are near the centroid for that dataset. DiskANN's cache policy loads the data near the centroid, covering most query results. However, Starling still achieves better RS performance. Fig. 5 shows that Starling has a significantly higher throughput than DiskANN under the same *AP*. For example, it is 43.9× faster than DiskANN when $AP = 0.9$ on BIGANN. Further investigation reveals that DiskANN requires numerous disk I/Os for some queries with long RS results (e.g., 1,000). In that case, Starling achieves greater gains as it avoids more redundant disk I/Os.



(a) BIGANN (b) Text2image (a) DEEP

Fig. 6. ANNS performance (*Recall* vs *Latency*).



(a) BIGANN (b) DEEP

Fig. 7. ANNS performance (*QPS* vs *Recall*).

ANNS performance. Fig. 6 contrasts the *Recall* and *Latency* of assorted methods. SPANN on Text2image is omitted, as its latency exceeds 20ms when $Recall > 0.9$. Starling consistently delivers lower latency compared to two rivals under similar *Recall* conditions. For example, at $Recall = 0.95$ on BIGANN, Starling (5ms) achieves over 2× more speed than DiskANN (10ms) and 10× more than SPANN (50ms). Fig. 7 delineates the *QPS* and *Recall* of various methods. Starling continually exhibits superior performance over its competitors. SPANN presents lackluster results, worse than anticipated. The reason is that SPANN relies heavily on data duplication (up to eight-fold the base data size [15]). Given the segment configuration, the number of data replications is restrained, leading to a performance dip (for additional analysis, see §6.9).

6.3 I/O-efficiency

Vertex utilization ratio (ξ). We calculate ξ for each block loaded in the queries and average these to obtain the overall ξ for the loaded blocks. As displayed in Tab. 2, Starling consistently outperforms DiskANN across all datasets. This superior performance is credited to Starling's

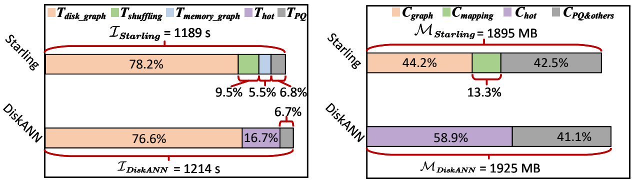
Table 2. Vertex utilization ratio (ξ) and search path length (ℓ).

Framework↓	Metric↓	BIGANN	DEEP	SSNPP	Text2image
DiskANN	ξ	0.0625	0.1429	0.1111	0.2500
Starling		**0.3438**	**0.4429**	**0.4111**	**0.8760**
DiskANN	ℓ	362	341	127	269
Starling		**182**	**240**	**100**	**167**

data locality enhancement through block shuffling, which permits a single block from the disk to encompass multiple relevant vertices. This approach heightens ξ, thereby mitigating disk bandwidth wastage. Hence, Starling exhibits higher I/O-efficiency. Nonetheless, ξ is not the exclusive factor that influences query performance–other factors such as data distribution and optimizations like the in-memory navigation graph also impact overall performance, potentially causing a non-linear relationship between ξ and query performance metrics such as latency.

Search path length (ℓ). We undertake an evaluation of ℓ for all queries, subsequently reporting the average ℓ under specified search accuracy. As denoted in Tab. 2, Starling exhibits a shorter ℓ compared to DiskANN. Within Starling, the in-memory navigation graph yields query-aware dynamic entry points that are in closer proximity to the query, thus reducing ℓ. Consequently, the IO-efficiency of Starling sees further improvement.

6.4 Index Cost



(a) Index processing time (b) Memory cost

Fig. 8. Index cost of two different frameworks on BIGANN.

Index processing time. The offline index process for Starling encompasses four components: disk-based graph construction time T_{disk_graph}, block shuffling time $T_{shuffling}$, in-memory graph construction time T_{memory_graph}, and preprocessing time for PQ short codes T_{PQ} (cf. §5.1) Consequently, the aggregate offline index procession time for Starling can be calculated as:

$$\mathcal{I}_{Starling} = T_{disk\_graph} + T_{shuffling} + T_{memory\_graph} + T_{PQ} \quad . \tag{8}$$

The offline index process for DiskANN comprises three parts: disk-based graph construction time T_{disk_graph}, hot vertices acquisition time T_{hot}, and preprocessing time for PQ short codes T_{PQ}. As per this structure, the total offline index processing for DiskANN is:

$$\mathcal{I}_{DiskANN} = T_{disk\_graph} + T_{hot} + T_{PQ} \quad . \tag{9}$$

Fig. 8(a) exposes the breakdown of index processing time on BIGANN. Despite the additional shuffling and construction time, Starling is substantially more efficient compared to DiskANN since the summation of $T_{shuffling}$ and T_{memory_graph} in Starling is less than T_{hot} in DiskANN. DiskANN necessitates the generation of hot vertices which involves the sampling of a large pool of queries and executing a slow disk-based graph search to tally vertex visit frequency. This procedure can be extremely time-consuming without hot vertices in memory. In contrast, Starling's approach of building an in-memory navigation graph proves to be faster and more effective. It is also worth noting that both Starling and DiskANN have an equal T_{disk_graph} and T_{PQ}.

Memory cost. In the main memory, Starling manages an in-memory navigation graph (C_{graph}), the mapping of vertex IDs to block IDs ($C_{mapping}$), and PQ short codes along with other data structures ($C_{PQ\&others}$). Hence, the total memory cost of Starling is

$$\mathcal{M}_{Starling} = C_{graph} + C_{mapping} + C_{PQ\&others} \quad . \tag{10}$$

DiskANN, on the other hand, houses a set of hot vertices (C_{hot}) and the PQ short codes along with other data structures ($C_{PQ\&others}$). The memory cost is hence:

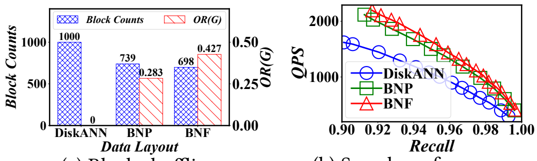
$$\mathcal{M}_{DiskANN} = C_{hot} + C_{PQ\&others} \quad . \tag{11}$$

DiskANN does not maintain $C_{mapping}$ as a group of ID-contiguous vertices are allocated into a block (it locates a block by vertex ID).

Fig. 8(b) offers a comparison of the memory costs of both frameworks when handling the same set of queries with the same accuracy on BIGANN. The results reveal Starling has lower memory overhead compared to DiskANN. The reason for this is as follows. Starling requires an extra mapping of vertex IDs to block IDs post block shuffling as vertex IDs within a block are non-consecutive. In DiskANN, each hot vertex comprises vector data and the neighbor IDs of the disk-based index. Starling's in-memory graph also includes vector data and neighbor IDs but it contains 20% fewer neighbor IDs than the disk-based graph since it possesses 10% less vector data. Resultingly, $C_{graph} + C_{mapping}$ in Starling is less than C_{hot} in DiskANN.

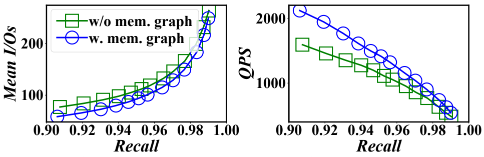
Disk cost. Starling and DiskANN undergo the same disk cost as they utilize the same disk-based graph, albeit in different layouts.

6.5 Ablation Study



(a) Block shuffling (b) Search performance

Fig. 9. Effect of block shuffling.

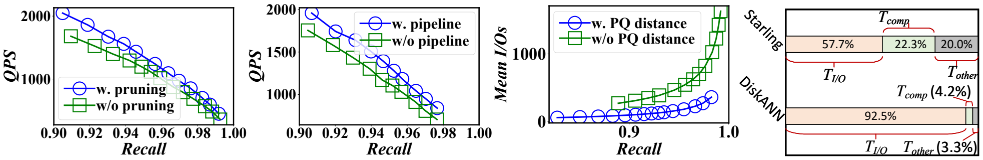


(a) Disk I/Os (b) Throughput

Fig. 10. Effect of in-memory graph.

Block shuffling. Fig. 9(a) reports the average number of blocks containing the top-1,000 nearest neighbors for each query within a given set (blue bar), as well as the overlap ratio $OR(G)$ (red bar) on DEEP. We limit our discussion to BNP and BNF, as BNS has slow processing times on a full-size segment. For BNF, we set the iteration parameter β to eight for an optimal balance between efficiency and efficacy. DiskANN's $OR(G)$ is found to be nearly zero, as the majority of vertices within a block are irrelevant. Both BNP and BNF consistently register higher $OR(G)$ than DiskANN, indicating the positive impact of our shuffling algorithms on data locality. BNF further amplifies $OR(G)$ on BNP, incurring a minor additional time cost, which is negligible about the total index processing time (~9.5%). Due to DiskANN's poor locality, almost all of the top-1,000 nearest neighbors are stored across disparate blocks, necessitating the checking of a minimum of 1,000 blocks for the retrieval of top-1,000 results. In comparison, BNP and BNF cut down the number of blocks by over 30%, enabling Starling to search through fewer blocks. Fig. 9(b) evidences BNP and BNF's exceptional performance in search tasks over DiskANN, with BNF outdoing BNP owing to its superior locality. By default, BNF is utilized for block shuffling in Starling.

In-memory navigation graph. To ascertain the impact of the in-memory navigation graph, we turn it on/off within Starling, ensuring all other settings remain identical. As depicted in Fig. 10, this analysis highlights the changes in disk I/Os and throughput on BIGANN. The activation of the in-memory graph leads to a reduction in disk I/Os by approximately 20% for the same *Recall*. This outcome stems from initiating our search on the disk-based graph from entry points positioned closer to the query, which effectively shortens the search path length (refer to **§3.1** for further details). Fig. 10(b) reveals that the in-memory graph notably elevates throughput. Importantly, the in-memory graph implementation does not alter the vertex utilization ratio.



(a) Block pruning (b) I/O-computation pipeline (c) PQ distance (d) Search profile

Fig. 11. Effect of the block search optimizations.

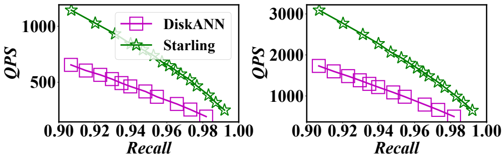
Block pruning. We assess the influence of block pruning on BIGANN by comparing Starling both with and without the implementation of pruning. The trade-off between *QPS* and *Recall* is presented for both scenarios. Fig. 11(a) demonstrates that Starling, when equipped with block pruning, exhibits superior performance. This can be attributed to the negation of unnecessary computations by pruning vertices that are distant from the query. Note that achieving an ideal graph layout with a perfect overlap ratio $OR(G) = 1$ is a challenging task (refer to **§4.1** for details). Some vertices located within a loaded block are not neighbors of the target vertex. Block pruning aids in filtering out these non-neighbor vertices, effectively circumventing unnecessary visits.

I/O and computation pipeline. Fig. 11(d) delineates the breakdown of search time for two frameworks with an identical *Recall* on BIGANN. For DiskANN, disk I/O is accountable for as much as 92.5% of the total search time, thus establishing itself as the bottleneck for HVSS on the disk-based graph. On the contrary, Starling enhances the vertex utilization ratio by leveraging superior data locality, thereby diminishing the I/O time ($T_{I/O}$) ratio to 57.7%. However, Starling also escalates the distance computation time T_{comp} since more vertices are examined in each loaded block. As a result, both I/O and computation are dominant and need to be executed in parallel for optimal performance. Fig. 11(b) outlines the impact of the I/O and computation pipeline, illustrating that the pipeline boosts *QPS* for the same *Recall*. This implementation may result in additional computations, as some vertex visitations within the current loaded block are deferred (see **§5.1** for more information). Despite this, the trade-off proves to be beneficial, enabling more efficient utilization of both the disk bandwidth and CPU, thereby improving the overall search performance.

PQ-based approximate distance. Fig. 11(c) depicts the average disk I/Os before and after the implementation of the PQ-based approximate distance optimization on BIGANN for a given batch of queries. It is clear that the number of disk I/Os is significantly reduced by applying this form of optimization, while maintaining the same level of accuracy. This optimization ranks the candidate set by approximate distance, which is calculated using PQ short codes in the main memory, thereby circumventing the need for expensive disk I/O operations. Although employing the exact distance might theoretically result in enhanced upper bound accuracy, the increased disk I/O cost can be prohibitively high. With the employment of this optimization, it is possible to maintain high levels of accuracy by adjusting other parameters (such as the size of the candidate set) while limiting the amount of disk I/Os, thus making it more suitable for practical applications.

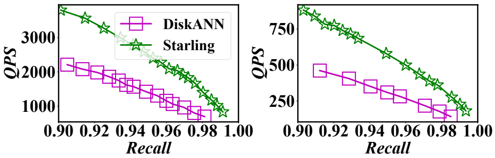
6.6 Parameter Sensitivity

Number of threads. We carry out an evaluation of the parallelism of both Starling and DiskANN, employing various thread settings ranging from 4 to 16 on BIGANN. Fig.12 highlights the *QPS* vs *Recall* relationship. In both frameworks, the *Recall* remains constant with thread variations as the number of threads does not influence the search path. Notably, irrespective of the number of threads used, the *QPS* of Starling consistently proves to be twice as fast as DiskANN.



(a) Threads = 4 (b) Threads = 16

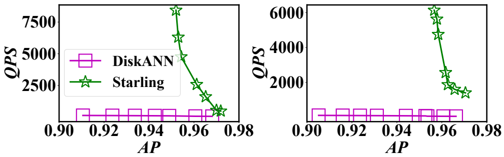
Fig. 12. Effect of different threads.



(a) k = 1 (b) k = 50

Fig. 13. Effect of different k in ANNS.

Number of search results. We conduct an examination on the effect of varying k from 1 to 50 on BIGANN. In Fig.13, it is clear that `Starling` exhibits a higher *QPS* than DiskANN for different values of k. This observation showcases the robustness of `Starling` in managing ANNS problems.



(a) r = 3,400 (b) r = 3,920

Fig. 14. Effect of different r in RS.

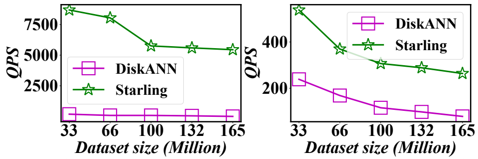
Search radius. Fig. 14 presents a comparison of the RS performance with different radii r on BIGANN. The results unequivocally demonstrate that `Starling` surpasses DiskANN across different r. This stark contrast highlights the superior performance of `Starling` in handling various r.

6.7 Scalability

Table 3. *QPS* of different number of segments on BIGANN.

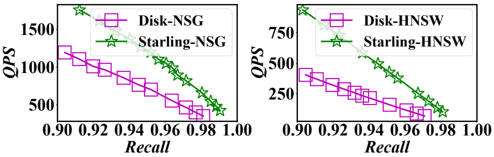
Query →	**RS** (*AP* = 0.90)		**ANNS** (*Recall* = 0.99)	
# Seg. ↓	**DiskANN**	**Starling**	**DiskANN**	**Starling**
1	181	**8,690 (48.0×)**	239	**538 (2.3×)**
2	42	**1,040 (24.8×)**	161	**321 (2.0×)**
3	31	**687 (22.2×)**	131	**257 (2.0×)**
4	23	**472 (20.5×)**	98	**193 (2.0×)**
5	19	**204 (10.7 ×)**	79	**163 (2.1×)**

Number of segments. We set a fixed segment size and manipulate the number of segments needed to process a batch of queries. Tab. 3 displays the scalability of `Starling` on a single machine. We report the *QPS* of both frameworks maintaining equivalent accuracy for ANNS and RS. The findings illustrate that `Starling` persistently surpasses DiskANN in performance for both ANNS and RS, testifying to its superior scalability in relation to the number of segments.



(a) RS (*AP* = 0.90) (b) ANNS (*Recall* = 0.99)

Fig. 15. Different data segment sizes.

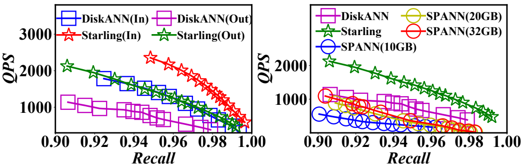


(a) NSG (b) HNSW

Fig. 16. Different graph algorithms.

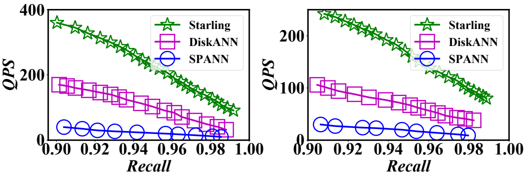
Segment size. For other experiments, we establish a fixed dataset size of 4GB within a segment. We test `Starling`'s performance on varying segment sizes via distinct dataset volumes. Remember, each segment's memory and disk space increase proportionately. Fig. 15 demonstrates the impact of data segment sizes. `Starling` sustains a higher *QPS* than DiskANN for both RS and ANNS across varying data volumes. This emphasizes that `Starling` scales efficiently with segment size.

Graph algorithms. By default, `Starling` employs the Vamana algorithm [33] to construct the disk-based graph index. Here, we assess search performance on disk with two other graph indexes, NSG [24] and HNSW [46]. We follow DiskANN to implement NSG and HNSW, but substitute NSG and HNSW (layer-0) for Vamana, resulting in `Disk`-NSG and `Disk`-HNSW. We also adapt `Starling` to use NSG and HNSW, obtaining `Starling`-NSG and `Starling`-HNSW. Notably, `Starling`-HNSW has a multi-layered in-memory navigation graph with HNSW's upper-layered graphs. Fig. 16 shows the *QPS* vs *Recall* comparison. We find that the two graph indexes on `Starling` perform better than the baseline framework. For example, `Starling`-HNSW's QPS is over 2× higher than `Disk`-HNSW. This shows `Starling`'s generality to support other graph algorithms.



(a) Query distribution (b) Segment setup

Fig. 17. Different queries and segments.



(a) 8GB (b) 16GB

Fig. 18. Different data sizes.

6.8 In- and Not-in-Database Queries

We compare different methods on in-database and not-in-database queries on BIGANN. In-database queries are vectors that exist in the graph index, while not-in-database queries are not. We randomly sample 10,000 vectors from the base data as in-database queries and execute the search on two frameworks. Fig. 17(a) shows that `Starling` consistently outperforms DiskANN for both types of queries. Moreover, in-database queries achieve higher throughput than not-in-database queries for both frameworks. This is because in-database queries can leverage some better query-aware entry points, which may be the query points themselves. However, the in-memory navigation graph can also find some entry points close to the not-in-database queries to shorten their search path. Notably, both types of query workloads benefit from data locality through offline block shuffling.

6.9 Evaluation on Other Segment Setups

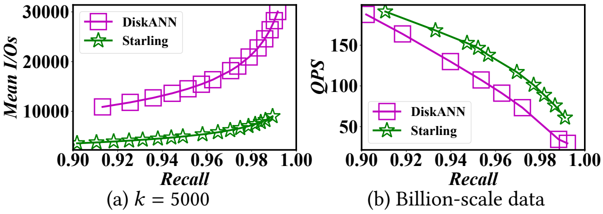
Effect of disk capacity. We fix the dataset size at 4GB and memory size at 2GB, and test different segment setups with the disk capacity from 10GB to 32GB. Fig. 17(b) shows the search performance on BIGANN. DiskANN and `Starling` exhibit the best *QPS-Recall* trade-off with an index size of less than 10GB, so we only plot one curve for each of them. SPANN improves its performance as the disk space increases, because it can duplicate more boundary data points, thereby reducing disk I/Os. However, `Starling` still performs much better than SPANN, with a smaller disk setup.

Effect of dataset size. We employ a fixed segment space with a memory of 2GB and a disk capacity of 32GB to test diverse dataset sizes of 4GB, 8GB, and 16GB. Fig. 18 shows the search performance of different methods on BIGANN (see Fig. 17(b) for the 4GB dataset). We tune the parameters of all methods to get the best *QPS-Recall* trade-off under the segment space constraint. The results conclusively depict the superior performance of `Starling` over rival methods across all dataset sizes. Importantly, the performance gap increases as the dataset size magnifies. SPANN suffers from a larger dataset size because it cannot replicate enough data to minimize disk I/Os.

6.10 Large-scale Search Results

We set the number of search results (k) to less than 50 for other experiments. In some cases, we may need much more results, such as thousands. For example, in recommendation systems, a large number of candidates are first recalled and then filtered to get the final recommendations [16, 63].

Fig. 19(a) shows the search performance with $k = 5,000$ on BIGANN. We can see that Starling has a much lower *Mean I/Os* than DiskANN. For instance, with a *Recall* of 0.99, Starling saves more than 20,000 disk I/Os per query than DiskANN. This shows that Starling is more efficient and effective in scenarios with large-scale search results.



(a) k = 5000 (b) Billion-scale data

Fig. 19. Large-scale search results and Billion-scale data.

6.11 Evaluation on Billion-scale Data

We evaluate our method on the one billion BIGANN dataset. We split the dataset into 31 segments, each with 2GB memory and 10GB disk. One query node has only 32GB of memory in our experiments, so we assigned 31 segments to two query nodes. We merge candidates from each segment to get the final results. We use the same setting for Starling and DiskANN to ensure fairness. Fig. 19(b) shows that Starling is more than 2× faster than DiskANN in the high recall regime (e.g., *Recall* > 0.96). Note that current vector databases use some data segmentation strategies and a query coordinator, which can avoid scanning all segments for a query [27]. Our method can work well with these strategies. For more details on data segmentation strategies, please refer to [22, 71].

7 DISCUSSION

Memory-based related work. Memory-based HVSS often involves preprocessing data to strike a balance between efficiency and accuracy. Existing algorithms fall into four categories: tree- [19, 44, 47]; quantization- [9, 28, 34]; hashing- [25, 31, 40]; and graph-based [23, 24, 46]. Tree-based and hashing-based methods are not prevalent in HVSS due to the "curse of dimensionality" and low accuracy [42]. Quantization-based methods (e.g., IVFPQ) prove efficient and memory-saving but tend to suffer from a poor recall rate[33, 56]. Graph-based methods (e.g., HNSW) exhibit leading efficiency-accuracy trade-offs. However, they necessitate both raw vector data and the graph index to be in the main memory, elevating memory consumption and impeding scalability for large-scale vectors [15]. Starling is a universal and I/O-efficient framework capable of integrating different graph algorithms (e.g., HNSW) into the disk-index component. For example, Starling accommodates HNSW seamlessly without sacrificing functionality (cf. Fig. 16(b)).

Comparison analysis with SPANN. While both Starling and SPANN leverage in-memory graphs and locality-centric disk indexes, they address varying challenges and utilize unique strategies. SPANN, a clustering-based disk index, partitions data utilizing k-means, thereby achieving an inherent locality allowing clustered data to be stored and accessed synchronously. Furthermore, it builds an in-memory graph for fast cluster retrieval. In contrast, Starling addresses disk-based graph search by optimizing data layout and search strategy. When residing in memory, graph-based methods excel in terms of accuracy and efficiency. However, once placed on disk, they incur many I/Os due to long search path and poor data locality. Starling mitigates these issues by building an in-memory graph to shorten the search path and using block shuffling to enhance locality. Note that the graph index neighborhood exhibits both similarity and navigation traits [65]. This implies that a vertex may have neighbors from different clusters [46], posing a major challenge for the locality of the graph index. We compare block shuffling with a naive strategy that assigns vertices to blocks by k-means on SSNPP. The results show that block shuffling achieves a 12× higher overlap ratio.

In-memory graph. Our in-memory graph serves as an index directing query routes, instead of acting as a cache for frequently accessed data. We initially used memory mapping (mmap) instead of direct I/O (o_direct) but found more disk I/Os linked to a memory-mapped graph. This lowers efficiency compared to the hot points strategy in the baseline framework. Our evaluation showed that the in-memory graph outperforms the hot points strategy in search performance and memory overhead. In Starling, we have the flexibility to utilize any graph algorithm like HNSW [46] or NSG [24] for the in-memory graph. We typically use the same algorithm for both in-memory and disk-based graphs. In HNSW, the upper-layer graphs are a subset of the layer-0 graph. Thus, we can keep the higher layers in memory as a multi-layered in-memory graph and store the layer-0 graph on disk. This makes HNSW easy to implement in Starling (see Fig. 16(b)).

Central assumption. With modern SSD (high IOPS), recent methods like DiskANN can fetch multiple blocks simultaneously in each disk round-trip. This is because fetching a small number of random blocks from a disk takes almost the same time as one block [33]. We keep this feature in Starling while increasing the vertex utilization ratio in each loaded block by enhancing locality. This reduces the total I/Os for a query by minimizing round-trips.

Data update. Starling, primarily emphasizing query optimization for static indexes, can handle incremental updates at the database level [27]. Some vector databases (e.g., ADBV [67]) segregate dynamic and static indexes to facilitate updates. The dynamic index, residing in memory, is built incrementally and uses a bitset to monitor deleted data. As incoming data continues to grow, the dynamic index expands correspondingly. Consequently, an asynchronous merging process transfers the burgeoning dynamic index to the disk-based static index, necessitating a comprehensive index reconstruction. Then, the block shuffling and in-memory graph techniques come into play.

Applications of range search (RS). RS is widely used in vector analytics, such as face recognition [58], near-duplicate detection [2], and various applications on general embeddings [54, 61, 66]. In some retrieval systems [13, 52], ANNS is followed by RS with seed vertices and similarity thresholds [29] to obtain a similar cluster and offer more comprehensive results [36]. For example, users can choose a seed vertex to get all related results. Several vector databases (e.g., PASE [69], VBase [72], Milvus [62]) and memory-based works [11, 43] support both ANNS and RS on the same dataset.

8 CONCLUSION

We conduct a study on HVSS for the data segment, which is an essential component in vector databases. HVSS for the data segment needs to meet strict requirements in terms of accuracy, efficiency, memory usage, and disk capacity. However, existing methods only address some of these aspects. Our framework, called Starling, adopts a disk-based graph index approach and considers all these requirements simultaneously. It optimizes the data layout and search strategy to minimize costly disk I/O operations. Experimental results demonstrate that Starling achieves significant performance improvements compared to state-of-the-art methods while maintaining a small space overhead. In the future, we plan to apply our methods to cache and GPU optimizations. We will also integrate Starling into Milvus for distributed optimization.

ACKNOWLEDGMENTS

This work was supported in part by the NSFC under Grants No. (62025206, U23A20296), the Primary R&D Plan of Zhejiang (2023C03198 and 2021C03156). Xiaomeng Yi and Xiaoliang Xu are the corresponding authors of the work.

REFERENCES

[1] 2018. A Library for Efficient Similarity Search and Clustering of Dense Vectors. https://github.com/facebookresearch/faiss.

[2] 2020. Using AI to detect COVID-19 misinformation and exploitative content. https://ai.meta.com/blog/using-ai-to-detect-covid-19-misinformation-and-exploitative-content/.

[3] 2021. Billion-Scale Approximate Nearest Neighbor Search Challenge: NeurIPS'21 competition track. https://big-ann-benchmarks.com/.

[4] 2021. Milvus Was Built for Massive-Scale (Think Trillion) Vector Similarity Search. https://milvus.io/blog/Milvus-Was-Built-for-Massive-Scale-Think-Trillion-Vector-Similarity-Search.md.

[5] 2021. Scalable graph based indices for approximate nearest neighbor search. https://github.com/microsoft/DiskANN.

[6] 2022. Building a Vector Database for Scalable Similarity Search. https://milvus.io/blog/deep-dive-1-milvus-architecture-overview.md.

[7] 2023. The ChatGPT Retrieval Plugin lets you easily search and find personal or work documents by asking questions in everyday language. https://github.com/openai/chatgpt-retrieval-plugin.

[8] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming graph partitioning: an experimental study. *PVLDB* 11, 11 (2018), 1590–1603.

[9] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache locality is not enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan. *PVLDB* 9, 4 (2015), 288–299.

[10] Konstantin Andreev and Harald Räcke. 2004. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. 120–124.

[11] Kazuo Aoyama, Kazumi Saito, Hiroshi Sawada, and Naonori Ueda. 2011. Fast approximate similarity search based on degree-reduced neighborhood graphs. In *SIGKDD*. 1055–1063.

[12] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. 2023. ACL 2023 Tutorial: Retrieval-based LMs and Applications. *ACL* (2023).

[13] Kai Uwe Barthel, Nico Hezel, Konstantin Schall, and Klaus Jung. 2019. Real-time visual navigation in huge image sets using similarity graphs. In *ACM MM*. 2202–2204.

[14] Patrick H Chen, Chang Wei-cheng, Yu Hsiang-fu, Inderjit S Dhillon, and Hsieh Cho-jui. 2022. FINGER: Fast Inference for Graph-based Approximate Nearest Neighbor Search. *arXiv:2206.11408* (2022).

[15] 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 2021*. 5199–5212.

[16] Rihan Chen, Bin Liu, Han Zhu, Yaoxuan Wang, Qi Li, Buting Ma, Qingbo Hua, Jun Jiang, Yunlong Xu, Hongbo Deng, and Bo Zheng. 2022. Approximate Nearest Neighbor Search under Neural Similarity Metric for Large-Scale Recommendation. In *CIKM*. 3013–3022.

[17] Benjamin Coleman, Santiago Segarra, Anshumali Shrivastava, and Alex Smola. 2021. Graph Reordering for Cache-Efficient Near Neighbor Search. *arXiv:2104.03221* (2021).

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

[19] Sanjoy Dasgupta and Yoav Freund. 2008. Random Projection Trees and Low Dimensional Manifolds. In *SOTC*. 537–546.

[20] Shiyuan Deng, Xiao Yan, KW Ng Kelvin, Chenyu Jiang, and James Cheng. 2019. Pyramid: A general framework for distributed similarity search on large-scale datasets. In *IEEE International Conference on Big Data*. 1066–1071.

[21] Wei Dong, Moses Charikar, and Kai Li. 2011. Efficient K-nearest Neighbor Graph Construction for Generic Similarity Measures. In *WWW*. 577–586.

[22] Ishita Doshi, Dhritiman Das, Ashish Bhutani, Rajeev Kumar, Rushi Bhatt, and Niranjan Balasubramanian. 2022. LANNS: A Web-Scale Approximate Nearest Neighbor Lookup System. *PVLDB* 15, 4 (2022).

[23] Cong Fu, Changxu Wang, and Deng Cai. 2021. High Dimensional Similarity Search with Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *TPAMI* (2021).

[24] 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.

[25] Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. 2020. iDEC: Indexable Distance Estimating Codes for Approximate Nearest Neighbor Search. *PVLDB* 13, 9 (2020), 1483–1497.

[26] Mihajlo Grbovic and Haibin Cheng. 2018. Real-time personalization using embeddings for search ranking at airbnb. In *SIGKDD*. 311–320.

[27] 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.

[28] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *ICML*. 3887–3896.

[29] Nico Hezel, Kai Uwe Barthel, Konstantin Schall, and Klaus Jung. 2023. Fast Approximate Nearest Neighbor Search with a Dynamic Exploration Graph using Continuous Refinement. *arXiv:2307.10479* (2023).

[30] 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 *SIGKDD*. 2553–2561.

[31] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *PVLDB* 9, 1 (2015), 1–12.

[32] Shikhar Jaiswal, Ravishankar Krishnaswamy, Ankit Garg, Harsha Vardhan Simhadri, and Sheshansh Agrawal. 2022. OOD-DiskANN: Efficient and Scalable Graph ANNS for Out-of-Distribution Queries. *arXiv:2211.12850* (2022).

[33] 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. In *NeurIPS*, Vol. 32.

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

[35] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *ICML*. 1188–1196.

[36] Hao Li, Xiaojie Liu, Tao Li, and Rundong Gan. 2020. A novel density-based clustering algorithm using nearest neighbor graph. *PR* 102 (2020), 107206.

[37] Hongzheng Li, Yingxia Shao, Junping Du, Bin Cui, and Lei Chen. 2022. An I/O-Efficient Disk-based Graph System for Scalable Second-Order Random Walk of Large Graphs. *PVLDB* 15, 8 (2022), 1619–1631.

[38] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. 2018. The Design and Implementation of a Real Time Visual Search System on JD E-commerce Platform. In *Proceedings of the 19th International Middleware Conference*. 9–16.

[39] Mingjie Li, Yuan-Gen Wang, Peng Zhang, Hanpin Wang, Lisheng Fan, Enxia Li, and Wei Wang. 2022. Deep Learning for Approximate Nearest Neighbour Search: A Survey and Future Directions. *IEEE Transactions on Knowledge and Data Engineering* (2022).

[40] Mingjie Li, Ying Zhang, Yifang Sun, Wei Wang, Ivor W. Tsang, and Xuemin Lin. 2020. I/O Efficient Approximate Nearest Neighbour Search based on Learned Functions. In *ICDE*. 289–300.

[41] Nan Li, Bo Kang, and Tijl De Bie. 2023. SkillGPT: a RESTful API service for skill extraction and standardization using a Large Language Model. *arXiv:2304.11060* (2023).

[42] 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.

[43] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2022. HVS: hierarchical graph structure based on voronoi diagrams for solving approximate nearest neighbor search. *PVLDB* 15, 2 (2022), 246–258.

[44] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. 2020. VHP: Approximate Nearest Neighbor Search via Virtual Hypersphere Partitioning. *PVLDB* 13, 9 (2020), 1443–1455.

[45] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. 135–146.

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

[47] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *TPAMI* 36, 11 (2014), 2227–2240.

[48] Ramzi Nasr, Daniel S Hirschberg, and Pierre Baldi. 2010. Hashing algorithms and data structures for rapid searches of fingerprint vectors. *Journal of chemical information and modeling* 50, 8 (2010), 1358–1368.

[49] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. 456–471.

[50] Shumpei Okura, Yukihiro Tagami, Shingo Ono, and Akira Tajima. 2017. Embedding-based news recommendation for millions of users. In *SIGKDD*. 1933–1942.

[51] Anil Pacaci and M Tamer Özsu. 2019. Experimental analysis of streaming algorithms for graph partitioning. In *SIGMOD*. 1375–1392.

[52] Youngki Park, Sungchan Park, Woosung Jung, and Sang-goo Lee. 2015. Reversed CF: A fast collaborative filtering algorithm using a k-nearest neighbor graph. *Expert Systems with Applications* 42, 8 (2015), 4022–4028.

[53] Maria Predari and Aurélien Esnard. 2016. A k-way greedy graph partitioning with initial fixed vertices for parallel applications. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. 280–287.

[54] Jianbin Qin, Yaoshu Wang, Chuan Xiao, Wei Wang, Xuemin Lin, and Yoshiharu Ishikawa. 2018. GPH: Similarity search in hamming space. In *ICDE*. 29–40.

[55] Michael r. garey and david s. johnson. 1980. *Computers and Intractability: A Guide to the Theory of NP-Completeness*.

[56] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *NeurIPS*.

[57] Amaia Salvador, Nicholas Hynes, Yusuf Aytar, Javier Marin, Ferda Ofli, Ingmar Weber, and Antonio Torralba. 2017. Learning cross-modal embeddings for cooking recipes and food images. In *CVPR*. 3020–3028.

[58] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *CVPR*. 815–823.

[59] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*. 135–146.

[60] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. 2021. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. In *NeurIPS*, Vol. 176. 177–189.

[61] Yang Song, Yu Gu, Rui Zhang, and Ge Yu. 2021. ProMIPS: Efficient high-dimensional C-approximate maximum inner product search with a lightweight index. In *ICDE*. 1619–1630.

[62] 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*. 2614–2627.

[63] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2023. An Efficient and Robust Framework for Approximate Nearest Neighbor Search with Attribute Constraint. In *NeurIPS*.

[64] 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. *arXiv:2401.02116* (2024).

[65] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *PVLDB* 14, 11 (2021), 1964–1978.

[66] Yifan Wang, Haodi Ma, and Daisy Zhe Wang. 2022. LIDER: An Efficient High-dimensional Learned Index for Large-scale Dense Passage Retrieval. *PVLDB* 16, 2 (2022), 154–166.

[67] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *PVLDB* 13, 12 (2020), 3152–3165.

[68] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *SIGMOD*. 1813–1828.

[69] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *SIGMOD*. 2241–2253.

[70] Minjia Zhang and Yuxiong He. 2019. Grip: Multi-store capacity-optimized high-performance nearest neighbor search for vector search engine. In *CIKM*. 1673–1682.

[71] Pengcheng Zhang, Bin Yao, Chao Gao, Bin Wu, Xiao He, Feifei Li, Yuanfei Lu, Chaoqun Zhan, and Feilong Tang. 2022. Learning-based query optimization for multi-probe approximate nearest neighbor search. *VLDBJ* (2022), 1–23.

[72] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, et al. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *OSDI*.

[73] Da Zheng, Disa Mhembere, Randal C. Burns, Joshua T. Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *FAST*. 45–58.

[74] Chun Jiang Zhu, Minghu Song, Qinqing Liu, Chloé Becquey, and Jinbo Bi. 2020. Benchmark on indexing algorithms for accelerating molecular similarity search. *Journal of Chemical Information and Modeling* 60, 12 (2020), 6167–6184.

Received July 2023; revised October 2023; accepted November 2023