GPS: Revisiting the Data Layout for Disk-based High-Dimensional Vector Search

PEIQI YIN, The Chinese University of Hong Kong, China
XIAO YAN*, Institute for Math and AI Wuhan, Wuhan University, China
QIHUI ZHOU, The Chinese University of Hong Kong, China
HUI LI, The Chinese University of Hong Kong, China
XIAOLU LI, Huazhong University of Science and Technology, China
MEILING WANG, Huawei, China
LIN ZHANG, Huawei, China
XIN YAO, Huawei, China
JAMES CHENG, The Chinese University of Hong Kong, China

Similarity-based vector search underpins many important applications, but a key challenge is processing massive vector datasets (e.g., in TBs). To reduce costs, some systems utilize SSDs as the primary data storage. They employ a *proximity graph*, which connects similar vectors to form a graph and is the state-of-the-art index for vector search. However, these systems are hindered by sub-optimal data layouts that fail to effectively utilize valuable memory space to reduce disk access and suffer from poor locality for accessing disk-resident data. Through extensive profiling and analysis, we found that *the structure of the proximity graph index* is accessed more frequently than the *vectors* themselves, yet existing systems do not distinguish between the two. To address this problem, we design the *GPS* system with the principle of prioritizing graph structure over vectors. Specifically, GPS features a memory cache that keeps the adjacency lists of graph nodes to improve cache hits and a disk block format that explicitly stores neighbor's adjacency lists along with a vector to enhance data locality. Experimental results show that GPS consistently outperforms three state-of-the-art disk-based systems for vector search, boosting average query throughput by 52% and reducing query latency by 32% over the best baseline PipeANN.

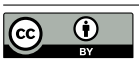
CCS Concepts: • **Information systems → Information retrieval query processing**.

Additional Key Words and Phrases: Vector Search, Disk-based System

ACM Reference Format:
Peiqi Yin, Xiao Yan, Qihui Zhou, Hui Li, Xiaolu Li, Meiling Wang, Lin Zhang, Xin Yao, and James Cheng. 2026. GPS: Revisiting the Data Layout for Disk-based High-Dimensional Vector Search. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 192 (June 2026), 29 pages. https://doi.org/10.1145/3802069

*Dr. Xiao Yan is the corresponding author.

Authors' Contact Information: Peiqi Yin, pqyin22@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China; Xiao Yan, yanxiaosunny@whu.edu.cn, Institute for Math and AI Wuhan, Wuhan University, Wuhan, China; Qihui Zhou, qhzhou@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China; Hui Li, hli@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China; Xiaolu Li, lixl666@hust.edu.cn, Huazhong University of Science and Technology, Wuhan, China; Meiling Wang, wangmeiling17@huawei.com, Huawei, Shenzhen, China; Lin Zhang, zhang.lin4@huawei.com, Huawei, Hong Kong, China; Xin Yao, yao.xin1@huawei.com, Huawei, Hong Kong, China; James Cheng, jcheng@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China.



ACM 2836-6573/2026/6-ART192
https://doi.org/10.1145/3802069

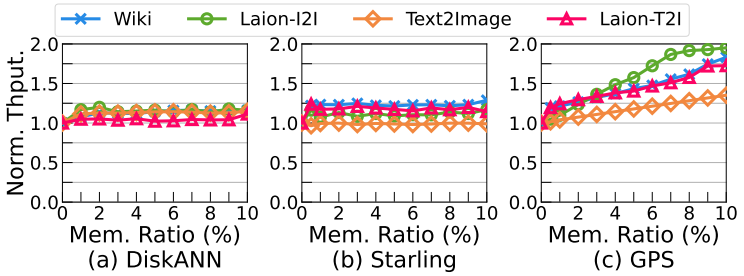


Fig. 1. Query throughput of (a) DiskANN, (b) Starling, and (c) GPS when using memory cache, normalized by the case without memory cache. Cache size is measured by the ratio of dataset size.

1 Introduction

To manage data with complex semantics, e.g., texts, images, and videos, a common practice is to map such data to high-dimensional vector embeddings using machine learning models [8, 24, 47]. A fundamental operation on these vectors is nearest neighbor search (NNS), which retrieves the top-k similar vectors to a given query vector (e.g., as measured by Euclidean distance) from a vector dataset. Since exact NNS requires a linear scan in high-dimensional space [74], approximate NNS (ANNS) is commonly employed to return most, instead of all, of the top-k similar vectors, trading some accuracy for efficiency. ANNS is crucial for many applications, including recommendation [27, 92], content search (e.g., for images and videos) [39, 81], retrieval-augmented generation (RAG) for large language models (LLMs) [4, 43, 56], and bio-informatics [63, 64]. These applications require ANNS to achieve a high recall for the search results[1] while ensuring low latency and high throughput for query processing.

Real-world vector datasets can be massive, containing billions of vectors and occupying TBs of space [59, 73]. Storing all vectors in memory requires a huge DRAM and is expensive. To reduce costs, several systems explore disk-based ANNS [10, 67, 78], utilizing SSDs as the primary data storage and using a small memory (e.g., 10%-20% of the dataset size) as the cache. DiskANN [67] and Starling [78] are state-of-the-art systems for disk-based ANNS. They employ a proximity graph index [48, 75], where each vector acts as a graph node connected to similar vectors. ANNS is conducted by traversing the graph towards vectors that are more similar to the input query. When visiting a node[2], the search process retrieves its neighbors and computes their distances to the query. DiskANN and Starling use memory to store a compressed version of all vectors [18, 19, 31], which allows to compute approximate distances to the query, while SSD holds the original vectors and their adjacency lists (i.e., the proximity graph index). When a node is visited, these systems fetch its original vector (i.e., to compute exact distance and rank the node in the search results) and adjacency list (i.e., to specify its neighbors for approximate distance computation) from disk.

Like many disk-based systems [36, 41, 44], to achieve good performance, disk-based ANNS must carefully design data layouts for both memory and disk to reduce the number of disk IOs and mitigate the challenges posed by disk access (e.g., long latency, low bandwidth, and 4KB block access). However, we identify two significant issues in the data layouts of DiskANN and Starling that impede efficiency.

- *Ineffective memory cache.* After storing the compressed vectors, DiskANN and Starling use the remaining memory as a cache (referred to as *memory cache*) to keep some original vectors alongside their adjacency lists. As shown in Figure 1, when using a larger memory cache, their

[1]Recall measures the quality of ANNS results. If k' of the returned vectors belong to the ground-truth top-k similar vectors for a query, the recall is k'/k.

[2]We use node and vector interchangeably in this paper.

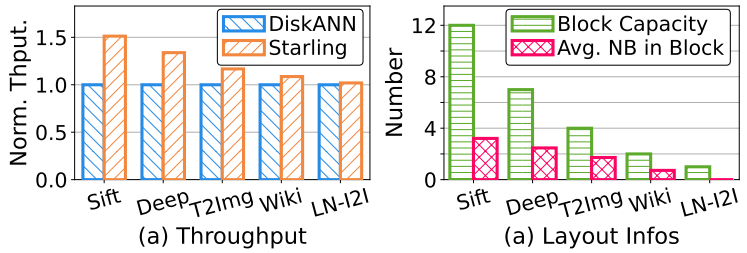


Fig. 2. (a) The performance gain of Starling's disk data layout over DiskANN, query throughput is normalized by DiskANN. The vector dimension increases from left (128) to right (768). (b) The number of vectors that fit in a 4KB disk block for different datasets, and a node's average number of neighbors (NB) in its disk block for Starling's disk layout.

query throughput quickly stabilizes after an initial improvement. This is undesirable, as memory is expensive and performance should have obvious improvements with increased memory allocation. The issue arises because each individual vector is large due to high dimensionality, limiting the number of exact vectors that can be held in memory and thus leading to limited reductions in disk IOs.

- *Poor data locality on disk.* While DiskANN simply stores nodes on disk in their ID order, Starling improves the disk data layout through graph reordering [7], which increases the likelihood that a node co-locates in the same 4KB disk block as its neighbors. When reading a vector via a disk block, these co-located neighbors are also fetched, allowing some disk IOs to be avoided if these neighbors are accessed later. However, Figure 2(a) shows that the benefit of Starling's disk layout diminishes with increasing vector dimensions. This is because a vector becomes larger at higher dimensions, and thus a 4KB disk block holds fewer vectors, reducing the number of co-located neighbors and thereby diminishing IO savings, as indicated by Figure 2(b). This issue is concerning because there is a trend to increase vector dimensions to enhance expressiveness [12].

To address these issues, we conducted extensive profiling and analysis to understand the impact of data layout designs on the performance of disk-based ANNS (§ 3). Our key observation is that adjacency lists are accessed more frequently than the exact vectors with memory-resident compressed vectors, yet DiskANN and Starling treat them equally for both the memory cache and disk block storage. Specifically, each hop of the graph traversal requires to fetch the adjacency list of the current node to identify the vectors for approximate distance computation. The graph traversal visits many nodes, but only a subset of their exact vectors is needed. This is because the compressed vectors provide approximate distances, and to achieve a high recall for the search results, it suffices to compute the exact distances for the top-ranking vectors in approximate distances (this process is called *re-ranking*). A secondary observation is that, capped by a maximum node degree (e.g., 64), the size of an adjacency list is generally much smaller than that of an exact vector and does not increase with vector dimensionality.

Given these insights, we propose the Graph Prioritized Storage layout for disk-based ANNS (i.e., GPS), featuring two key designs that prioritize adjacency lists over exact vectors.

- *Graph prioritized memory cache.* Instead of keeping each adjacency list alongside its exact vector in memory cache, we retain only the adjacency lists. Since adjacency lists are significantly smaller than vectors, this allows us to cache more adjacency lists in memory, resulting in greater reductions in disk IO. We show the impact of GPS's memory cache optimization in Figure 1(c).
- *Graph replicated disk block.* In the disk block of each node, we explicitly include the adjacency lists of its neighbors, which we refer to as *neighbor packing*. Given that adjacency lists are small,

we can store many neighbor adjacency lists in a disk block. This design mitigates disk I/O when these neighbors are subsequently traversed. Although this may consume more disk space because an adjacency list could be replicated in the disk blocks of its neighbors, the space overhead remains moderate (e.g., below 2x) for high-dimensional vectors by constraining the neighbor packing to a 4KB block and given the low cost of SSD.

To make GPS work, we introduce an additional algorithm and several system designs (§ 4). First, we modify the graph traversal algorithm for query processing to take two stages: *search* and *refinement*. The search stage bypasses disk access for exact vectors if the required adjacency lists are in memory. In the refinement stage, we re-rank the search results by reading the exact vectors from disk for the top-ranking candidates based on their approximate distances. Second, we design a complete procedure to generate the data layout for GPS offline, which includes determining the compression ratio for compressed vectors, loading the memory cache, and writing the disk block for each vector under a disk space budget. Finally, we propose an asynchronous disk block read pipeline that prefetches the required disk blocks for future computations to hide disk access latency.

To evaluate GPS, we conduct experiments using 4 real datasets and compare three state-of-the-art disk-based ANNS systems (§ 5). The results show that GPS consistently outperforms the baselines across different datasets and recall targets, achieving higher query throughput and lower query latency. Specifically, when targeting the same recall levels (e.g., 90% or 95%), GPS improves query throughput over the best baseline PipeANN by 52% when averaged over the datasets, while also reducing the query latency by 32%. Moreover, GPS effectively improves performance with increased memory cache, and ablation studies confirm the effectiveness of our designs compared to alternatives.

To summarize, we make the following contributions.

- We identify that the data layouts of existing disk-based ANNS systems hinder their performance due to ineffective memory utilization and poor disk data locality.
- We conduct extensive profiling and analysis to understand how data layout influences the performance of disk-based ANNS, establishing the key design principle that adjacency lists should be prioritized over exact vectors.
- Guided by this principle, we design the GPS system, which features a memory cache that retains adjacency lists and a disk block that includes the adjacency lists of neighbors.
- We implement GPS with additional algorithm and system designs and comprehensively evaluate its performance.

2 Background for Disk-based ANNS

Vector search requires indexes to pinpoint a small number of vectors for each query for similarity computation. Among the indexes for high-dimensional vectors [10, 15, 48, 61, 67, 73, 75, 78, 84], proximity graph provides the best performance (i.e., in the number of similarity computations to achieve a recall target) and has many variants including HNSW [48], NSG [15], and Vamana [67]. Both DiskANN [67] and Starling [78] adopt the proximity graph index, and an illustration is provided in Figure 3(c). Each vector is a graph node and is connected to similar vectors, and there is usually a degree limit (e.g., 32 and 64) for each node to control the number of neighbors. Vector search is processed via a BFS-like traversal on the proximity graph. When visiting a node (called *candidate*), the traversal computes the distances between its neighbors and the query and marks the node as *visited*; and the nearest but unvisited node becomes the next candidate. The traversal stops when no closer nodes can be found, and the top-k nearest nodes encountered in this process are returned as the search results.

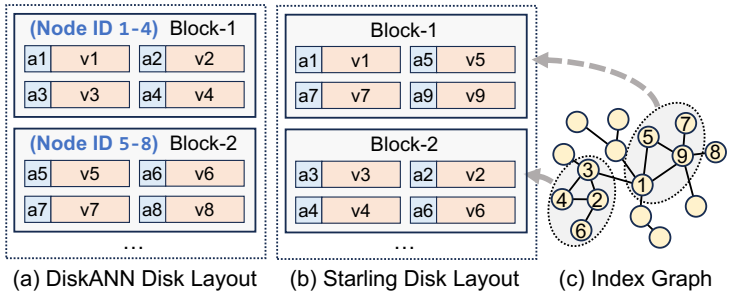


Fig. 3. The disk data layout of DiskANN and Starling. (c) is a proximity graph, and 'ax' and 'vx' are the adjacency list and vector of node x, respectively. We assume that a disk block can hold 4 vectors, and as neighbors of node 1, nodes 5 and 9 reside in the same block as node 1 in (b) due to reordering.

Algorithm 1: The search procedure of DiskANN.

$\mathcal{L}_{ext}$: Nearest node list (sorted by exact distances).
$\mathcal{L}_{appr}$: Nearest node list (sorted by approximate distances).
$\mathcal{D}$: The maximum length for $\mathcal{L}_{appr}$.

```
def ANNS(q, k):
    L_appr append {entry node e, ApproxDist(e, q)}
    while L_appr have unvisited nodes do
        u ← the nearest unvisited node in L_appr
        if node u not in node cache NC then
            Load exact vector and adj. for u from disk
        L_ext append {u, ExactDist(u, q)}
        for node v in u's adj. list do
            L_appr append {v, ApproxDist(v, q)}
        Sort L_appr and keep the top-D nearest nodes
    Sort L_ext and keep the top-k nearest nodes
    return L_ext
```

DiskANN stores in memory a compressed version of all vectors to compute approximate distances with the queries, and the specific vector compression technique is product quantization (PQ) [31]. The remaining memory serves as a *node cache*, which stores the exact vectors and adjacency lists of some graph nodes. In particular, DiskANN adopts the Vamana index [67], which uses the centroid of the dataset as the starting node for graph traversal (called *entry node*). The node cache keeps the nodes that are within a few hops (e.g., 2-3) from the entry node since they are visited frequently. Figure 3(a) depicts the disk data layout of DiskANN, in which the exact vector and adjacency list of a node are placed together, and each disk block contains several nodes.

Algorithm 1 illustrates the search procedure of DiskANN. Two nearest node lists are maintained, i.e., $\mathcal{L}_{appr}$ is ordered by approximate distances computed using the compressed vectors, and $\mathcal{L}_{ext}$ is ordered by exact distances computed using the exact vectors. In each iteration, the nearest unvisited node u in $\mathcal{L}_{appr}$ is selected as the candidate. If u is not in the node cache NC, the disk block containing u's exact vector and adjacency list will be read from the disk. Then, the approximate distances of u's neighbors are calculated and used to update $\mathcal{L}_{appr}$. The exact distance between u and the query is added to $\mathcal{L}_{ext}$. After each iteration, $\mathcal{L}_{appr}$ is adjusted to retain only the top-$\mathcal{D}$

Table 1. Statistics of the datasets. **Tot.N** is the number of vectors, **Dim.** is vector dimension, **Metric** is the measure for vector similarity, and **Tar.** is the target recall@10 for search.

Name	Tot. N	Dim.	Type	Metric	Size (GB)	Tar.
Sift [53]	100M	128	uint8	L2	11.9	95%
Deep [58]	100M	96	float32	L2	35.8	95%
Wiki [14]	100M	384	float32	L2	143.3	95%
Text2Image [58]	100M	200	float32	IP	74.5	90%
Laion-T2I [60]	100M	512	float32	Cosine	191.7	90%
Laion-I2I [59]	100M	768	float32	Cosine	286.4	95%

nearest nodes, where $\mathcal{D}$ is called the *queue size* and usually needs to be much larger than k to reach a high recall. The search terminates when all the nodes in $\mathcal{L}_{appr}$ are visited, and top-k nodes in $\mathcal{L}_{ext}$ are returned as the search results.

Starling also keeps the compressed vectors in memory but introduces two optimizations for the data layout of DiskANN. First, for the memory cache, Starling samples a portion (e.g., 10%) of the vectors and builds a proximity graph for these vectors (called *navigation index*). A query first searches the memory resident navigation index, and the results are used as the starting points for searching the proximity graph of the entire dataset. Second, Starling conducts graph reordering and tries to place each node and its neighbors in the same disk block [7], as illustrated in Figure 3(b). In particular, graph reordering assigns new IDs to the graph nodes such that nodes with similar neighbors have adjacent IDs, and Starling stores the nodes according to their new ID order. During vector search, when a disk block is loaded, Starling computes exact distances for all vectors in the block, for a portion of these vectors (e.g., 30% with the smallest distances), neighbors are checked by computing approximate distance (called *block search*). In summary, the navigation index finds good start points while the block search may access multiple required nodes via a 4KB disk block.

SPANN [10] adopts the IVF index [6] for disk-based vector search. The vectors are grouped into clusters and stored on disk, while the cluster centers are kept in memory. To conduct vector search for a query, SPANN first finds the nearest cluster centers in memory and then loads the corresponding clusters from the disk for exact distance computation. Recent research [78] shows that Starling outperforms SPANN because the IVF index requires to access many more vectors than proximity graph to reach the same recall.

3 Observations and Insights

In this part, we conduct extensive profiling and analysis to understand the influence of data layout on the performance of disk-based ANNS. Table 1 reports the statistics of the utilized datasets, and the detailed experiment settings can be found in § 5.1. Note that the query throughput and disk IOs are measured when reaching the target recalls in Table 1. To our knowledge, this is the first comprehensive profiling for data layout decisions, and we also draw insights to guide data layout designs and conduct in-depth analysis.

3.1 Accuracy of the Compressed Vectors

The first data layout decision is the compression ratio of the compressed vectors that are kept in memory. Although there are many vector quantization techniques that work in different ways [3, 18, 19, 31], a higher compression ratio always makes the compressed vectors smaller, while the approximate distance becomes cheaper to compute but less accurate.

As shown in Figure 4(b), the disk IOs for each query are stable initially but increase quickly after the compression ratio goes beyond a threshold. This is because at low compression ratios, the

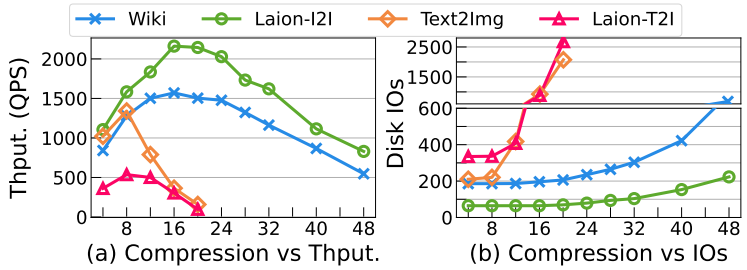


Fig. 4. DiskANN's query throughput and per query disk IOs when using different compression ratios for the memory-resident compressed vectors. Memory cache is disabled.

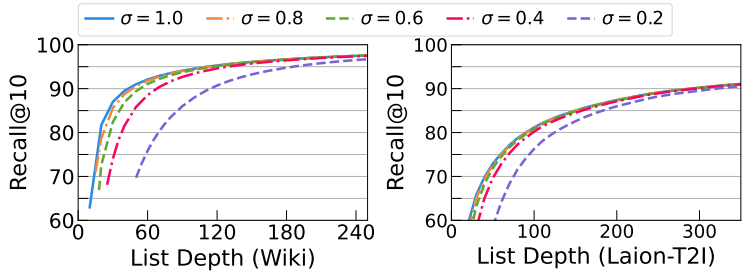


Fig. 5. Recall@10 for different refinement ratio σ and queue size $\mathcal{D}$. Tested on the Wiki and Laion-T2I datasets. We disable the memory cache for this experiment.

approximate distances are accurate and can identify good candidates (i.e., those with small exact distances to the query) for the graph traversal. As such, to reach the same recall, the length of the search path (i.e., the queue size $\mathcal{D}$) and hence the number of disk accesses resemble using uncompressed exact vectors. At high compression ratios, the approximate distances become inaccurate and lead the graph traversal to visit vectors that are dissimilar to the query, which prolongs the search path and blows up disk IOs. Figure 4(a) shows that query throughput, the main performance metric to optimize, first increases and then decreases with the compression ratio, making one compression ratio the optimal. This is because at smaller compression ratios, approximate distance computation becomes more expensive, while more disk IOs are required at higher compression ratios. In fact, the optimal is the highest compression ratio that does not blow up disk IOs.

Insight 1. *The compression ratio of approximate vectors has an optimal, and smaller or larger compression ratio degrades performance.*

Another observation is that the optimal compression ratios are different for the datasets, and cross-modal datasets (e.g., Text2Image and Laion-T2I) have smaller optimal than single-modal datasets (e.g., Wiki and Laion-I2I). In particular, cross-modal datasets use queries from one modality to search objects from another modality, for instance, Text2Image uses text embeddings to search image embeddings; they require more accurate compressed vectors as the distance gap between the vectors similar and dissimilar to a query is smaller compared with the single-modal datasets [9].

3.2 Demand for the Exact Vectors

Since the compressed vectors provide approximate distances with reasonable accuracy, we wonder whether it is necessary to compute exact distances for all the candidate vectors as in Algorithm 1 (i.e., for $\mathcal{L}_{ext}$). Instead, we may only compute exact distances for the top-ranking vectors in $\mathcal{L}_{appr}$ (see Algorithm 1) because $\mathcal{L}_{appr}$ records the approximate distances of the encountered vectors, and the real top-k neighbors of a query should have a good probability to rank high in $\mathcal{L}_{appr}$. To

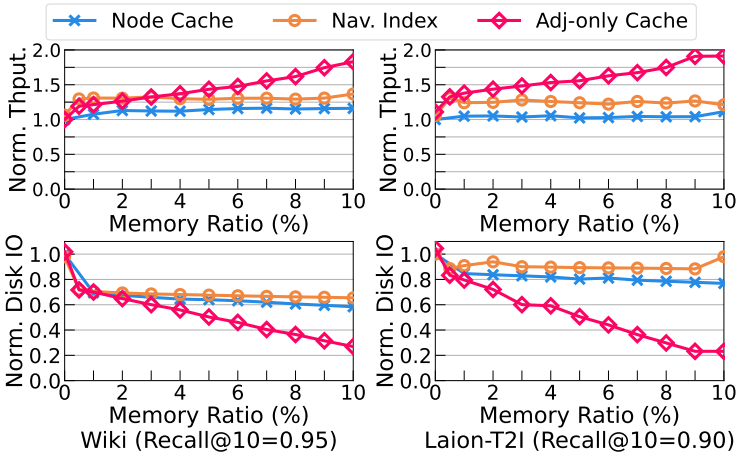


Fig. 6. The throughput and disk IOs for different memory caches when varying the size of the memory cache. Normalized by the case that does not use memory cache.

implement this idea, we define a *refinement ratio* σ and compute exact distances for $\mathcal{D}_r = \sigma\mathcal{D}$ vectors, where $\mathcal{D}$ is the size of $\mathcal{L}_{appr}$ (i.e., *queue size*). We also modify DiskANN to compute exact distances after the graph traversal terminates.

Figure 5 shows the impact of the refinement ratio σ and queue size $\mathcal{D}$ on the quality of the search results. We observe that to achieve a high recall (e.g., 95% for Wiki and 90% for Laion-T2I), $\mathcal{D}$ needs to be 10-20× of the target top-k; and at large $\mathcal{D}$, using a moderate σ (e.g., 0.5) does not harm result quality. Since the number of candidate vectors visited by the graph traversal is no smaller than $\mathcal{D}$[3], this allows to reduce a considerable portion of the exact distance computations.

Insight 2. *Re-ranking the vectors that rank top in approximate distance suffices for a high recall.*

3.3 Contents of the Memory Cache

The insight above implies that the adjacency lists are more important than the exact vectors. That is, when the graph traversal checks each candidate vector, its adjacency list is required to identify the neighbors for approximate distance computation, while its exact vector is not required if its approximate distance cannot rank top-$\mathcal{D}_r$ in the final $\mathcal{L}_{appr}$. This observation suggests that the memory cache should store only the adjacency lists rather than both exact vectors and adjacency lists as in DiskANN and Starling. To make the idea work, we modify Algorithm 1 to skip disk access if the required adjacency list is in memory and load the exact vectors for re-ranking after the graph traversal terminates (see the detailed algorithm in § 4.2).

Figure 6 compares the query throughput and disk IOs of DiskANN's node cache, Starling's navigation index, and our adjacency-only cache. The results echo Figure 1, i.e., DiskANN and Starling only observe marginal IO reduction and thus throughput improvement when increasing the memory size beyond a small threshold. This is because after including a small number of frequently visited vectors (e.g., those close to the entry node), the remaining vectors are accessed more uniformly; the cache size is small compared to the entire dataset, and thus the IO reduction is not obvious. In contrast, our adjacency-only cache achieves significant query throughput improvement and disk IO reduction when increasing the memory size. This validates our conjecture, and the following analysis explains why the adjacency-only cache works.

[3]Algorithm 1 terminates when $\mathcal{L}_{appr}$ has no unvisited candidates, and the size of $\mathcal{L}_{appr}$ is $\mathcal{D}$. As such, the graph traversal visits at least $\mathcal{D}$ candidates.

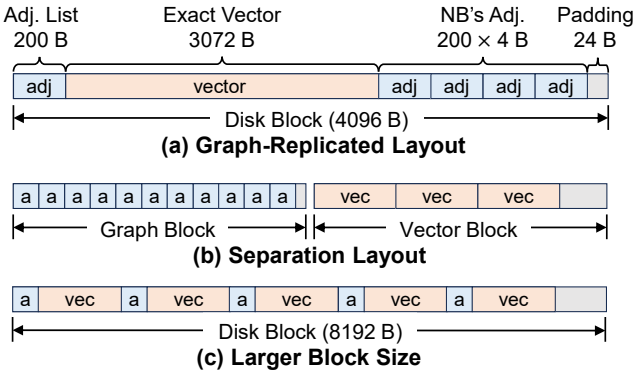


Fig. 7. Disk data layouts. (a) an 4KB illustration of our graph-replicated layout on the Laion-I2I dataset. (b) separation layout, using separate graph and vector blocks; (c) using larger blocks (8KB) to place more nodes in a block.

Analysis. Holding the adjacency lists in the memory cache outperforms holding both adjacency lists and vectors when

$$\frac{C}{N(S_v + S_a)} < \frac{C(1-\sigma)}{NS_a} \Rightarrow S_a < \frac{1-\sigma}{\sigma} S_v, \tag{1}$$

where C is the size of the memory cache, N is the number of vectors in the dataset, S_v and S_a are the sizes of an exact vector and an adjacency list, respectively, and $\sigma \approx 0.5$ is the refinement ratio defined previously. Assume that the graph traversal visits all nodes with an equal probability[4], the adjacency-vector coupled cache holds $C/(S_v + S_a)$ vectors, and thus its ratio of IO reduction is $C/N(S_v + S_a)$. The adjacency-only cache holds C/S_a adjacency lists and reduces the disk accesses for adjacency lists by a ratio of $\beta = C/(NS_a)$; without the memory cache, the ANNS search will conduct $\mathcal{D}$ disk accesses; with the cache, the total disk accesses $\mathcal{T}_a$ and the access reduction ratio $\mathcal{A}_r$ becomes

$$\mathcal{T}_a = \mathcal{D}(1-\beta) + \sigma\mathcal{D}\beta, \ \mathcal{A}_r = \frac{\mathcal{D} - \mathcal{T}_a}{\mathcal{D}} = \beta(1-\sigma). \tag{2}$$

Note that in Eq (2), we compute the disk IO reduction w.r.t. the $\mathcal{D}$ accesses for the adjacency-vector cache to align both cases. As such, Eq (1) compares the IO reduction ratios of the two cache strategies, and a larger reduction is better.

Eq (1) easily holds because an adjacency list records tens of neighbor IDs, while a vector can have hundreds or even thousands of dimensions. Take the Wiki dataset for an example, we have $S_v = 1536$B and $S_a = 200$B. When using a memory cache that is 10% of the dataset size, the adjacency-only cache holds the adjacency lists for 88% of the nodes, almost avoiding disk read during graph traversal, while the adjacency-vector coupled cache holds only 10% of the nodes.

Insight 3. *Caching the adjacency lists in memory is more effective than the exact vectors.*

3.4 Format of the Disk Block

Inspired by the importance of the adjacency lists, we introduce a disk block format that explicitly stores the adjacency lists of a node's neighbors along with the node. An illustrative example is provided in Figure 7(a) by assuming 4KB disk block size, and after keeping the exact vector and adjacency list of a vector, the adjacency lists of some neighbors are included to fill a disk block.

[4]If the visit probabilities are different for the nodes, we can select the popular nodes for both the adjacency-only cache and adjacency-vector cache.

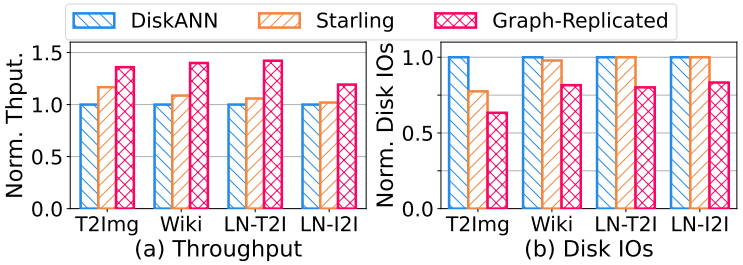


Fig. 8. Normalized throughput and disk IOs for DiskANN's, Starling's, and our graph-replicated disk layout.

We call it the *graph-replicated layout* since an adjacency list may be stored multiple times in the disk blocks of its neighbors. It consumes more disk space but we can control the space blow up as will be described in § 4.1, and disk capacity is cheap. The layout allows to read the neighbor adjacency lists along with the vector, and when these neighbors become the candidates, disk IOs can be skipped for their adjacency lists. Note that the disk block size is not constrained to 4KB, and GPS can use larger block sizes (e.g., 8KB and 16KB) for vectors with very high dimension (e.g., 1536 or 3072) like DiskANN. In these cases, the neighbor adjacency lists can also fill in the remaining space like for 4KB disk block.

Figure 8 compares the performance of our graph-replicated disk layout, Starling's disk layout, and DiskANN's disk layout with the other optimizations disabled (e.g., node cache and navigation index). The result shows that the graph-replicated layout consistently outperforms the other layouts for all 4 datasets. Figure 8(b) also explains why Starling has no improvement over DiskANN for the Laion-I2I dataset in Figure 2. That is, each vector in Laion-I2I takes around 3KB, and thus Starling can only store 1 node in the same 4KB disk page, yielding no IO reduction and throughput improvement. In contrast, since each adjacency list is small, our graph-replicated disk layout can still keep several neighbor adjacency lists along with the vector in a 4KB disk page.

Analysis. Our graph-replicated disk block outperforms Starling's disk block when

$$\theta \frac{B'}{S_v + S_a} < (1-\sigma)\theta \frac{B'}{S_a} \;\Rightarrow\; S_a < \frac{1-\sigma}{\sigma} S_v, \tag{3}$$

which easily holds since $\sigma \approx 0.5$, making Eq (3) essentially $S_a < S_v$. We consider the number of future disk accesses that can be avoided after reading a disk block in the two different formats. Let $B' = B - S_v - S_a$, where B is the block size, $B'/(S_v + S_a)$ is the maximum number of neighbors Starling can hold in a disk block. Assuming each of these neighbors has a probability of θ to become the candidate in the future, Starling reduces at most $\theta B'/(S_v + S_a)$ disk accesses. For the graph-replicated disk block, it can store the adjacency lists of B'/S_a neighbors and thus avoids $\theta B'/S_a$ disk accesses for the adjacency lists during graph traversal; for re-ranking, it needs $\sigma\theta B'/S_a$ additional disk accesses for the exact vectors of these skipped adjacency lists; and thus the total disk access reduction is $(1-\sigma)\theta B'/S_a$.[5]

Insight 4. *For high-dimensional vector datasets, packing the adjacency lists of a node's neighbors in its disk block improves performance.*

Alternatives. Besides our graph-replicated disk block, three alternative methods may also help for improve disk utilization: (i) separate the disk storage of graph and vector, (ii) using larger block sizes, and (iii) using larger graph degrees. The first method uses two types of disk blocks as shown

[5]We ignore integer rounding in the analysis for simplicity, e.g., the graph-replicated disk block actually holds $\lfloor B'/S_a \rfloor$ neighbor adjacency lists. We overestimate the number of co-located neighbors for Starling because its graph reordering does not ensure that a disk block only holds the neighbors of a vector, which is also illustrated in Figure 2(b).

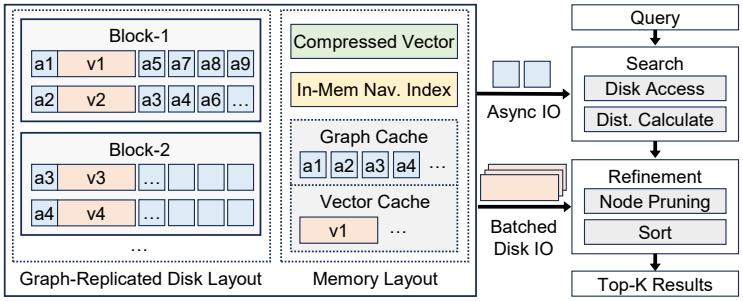


Fig. 9. The storage layout and workflow of GPS.

in Figure 7(b), i.e., *graph blocks* that contain only adjacency lists and are accessed during graph traversal, and *vector blocks* that contain only vectors and are accessed during re-ranking [35, 38]. This format allows one graph block to hold more neighbor adjacency lists, which increases the chance of reducing the disk IOs during traversal. However, a considerable portion of the nodes (i.e., $\sigma = 0.5$) need one disk access for their exact vectors during re-ranking, which may outweigh the disk IO savings for the adjacency lists. Next, uses a larger block size (e.g., 8KB) can include more nodes in a block, as shown in Figure 7(c). Although this format allows fetching more nodes via a single disk block, it wastes disk bandwidth because each disk block is larger. It has been reported that using a 4KB block size can achieve the full bandwidth of modern NVMe SSDs [23]. As such, the disk bandwidth should be used more judiciously by making access decisions individually for each 4KB block. These two formats does not change the graph structure and search pattern, and we compare our method with them and show that they yield inferior performance in §5.3.

Construct the index graph with a large graph degree (e.g., 200) can reduce the search iteration thus reduce disk IOs. However, an excessively large degree will introduce many low quality neighbors that are far from the node and thus do not contribute to search progress. Checking these low quality neighbors blows up the computation during searching. As we shown in §5.2, DiskANN already spends more than 50% of the query time on computation. Computation will become the bottleneck if we increase the graph degree.

4 The GPS System

GPS is a general system for disk-based ANNS on a single machine. Like Starling and DiskANN, GPS runs with a small memory (e.g., 10% of the dataset size) and uses disk as the primary data storage. GPS can plug in different proximity graph indexes [15, 67] and vector quantization algorithms [18, 19, 31] as all proximity graphs adopt the same graph traversal procedure for query processing, and vector quantization is used as a black box to compress the vectors and compute approximate distances. Figure 9 shows the workflow and storage layout of GPS. When GPS receives a query, it searches for its approximate top-k nearest vectors using the data stored in both memory and disk, and each query is handled by one thread. In this part, we put together the insights from § 3 and present the key designs of GPS, which includes the graph-vector decoupled storage layout (§ 4.1), a two-stage search algorithm that fits the decoupled storage layout (§ 4.2), and asynchronous block prefetch to hide the latency of disk access (§ 4.3).

4.1 Graph-vector Decoupled Storage Layout

DiskANN and Starling always store the vector and adjacency list of a node together for both memory and disk. Observing that the adjacency lists are more important than the vectors, GPS decouples the storage decisions for vectors and adjacency lists and prioritizes the adjacency lists.

Graph prioritized memory cache. After storing the compressed vectors, the remaining memory space can serve as a cache. We mainly use the cache to store the adjacency lists of the proximity graph index as analyzed in §3.3 (called *graph cache*). However, Figure 6 shows that keeping a small navigation index in memory also improves performance, and thus we also sample a small portion of the vectors (e.g., 0.5%) to build the navigation index. To select the nodes (whose adjacency lists are kept) in the graph cache, we order the nodes by their minimum distances to the nodes in the navigation index and cache the nodes with the smallest distances. The rationale is that nodes of the navigation index serve as entry nodes for graph traversal, and nodes close to these entry nodes have higher probabilities of being visited. This strategy does not require prior knowledge of the query vector distribution. Moreover, GPS can readily incorporate dynamic caching algorithms such as GoVector [96] to further improve memory cache hit ratio and performance, especially when the cache space is limited and valuable.

Graph replicated disk block. GPS uses the graph-replicated disk block format in Figure 7(a) to improve data locality. To control the disk space blow up caused by replicating the adjacency lists, we allow users to specify the number of neighbor adjacency lists that are packed for each node (denoted as R). Compared to Starling, the disk space blow up of GPS is $((1+R)S_a + S_v)/(S_a + S_v)$[6], since each node requires an extra storage of RS_a bytes for the packed adjacency lists. As discussed in §3.4, we constrain that the disk block of each vector takes up at most one 4KB disk page.

Given a proximity graph index and R, for each node u, our graph-replicated disk layout selects the R neighbors with the smallest exact distances to u as the neighbors whose adjacency lists are packed with u. This is because the close neighbors of a node are more likely to be visited after the node. To balance the replication of different adjacency lists, we constrain that each adjacency list can be replicated at most $R+1$ times. This reduces the chance that different nodes pack the same adjacency list, which wastes disk bandwidth during query processing. To implement the constraint, we conduct neighbor packing for the nodes one by one and avoid packing nodes whose adjacency lists have been packed for $R+1$ times. We also store the IDs of the packed neighbors on disk such that their exact vectors can be fetched during re-ranking. After generating the disk blocks for the nodes, we store the disk blocks in the order of their node IDs.

With our graph replicated format, a disk page may still contain more than one node (i.e., when the vector dimension and R are small). In this case, Starling's graph reordering can place neighbors on the same disk page for better locality. We do not adopt this optimization because its gain is marginal with our graph replicated format, and additional memory is required to store the mapping from node IDs to disk pages.

Memory cache planning. GPS determines the data structures to store in the memory and their sizes as follows:

❶ *Determine the compression ratio.* For the compressed vector, GPS searches the optimal compression ratio (i.e., as in § 3.1) that fits in memory using a small sampled dataset $\mathcal{S}'$ (e.g., 1% of the complete dataset) and query set $\mathcal{Q}'$. GPS uses PQ [31] by default for vector compression and tries different compression ratios on $\mathcal{S}'$ by testing the performance of $\mathcal{Q}'$. We observe that the optimal compression ratios are the same for the complete and sampled datasets, and a small sampled dataset allows to enumerate the compression ratios much faster due to faster compression. This is because the accuracy of the compressed vectors is more related to the relative error than the dataset size [18].

[6]We ignore integer rounding in the calculation for simplicity, e.g., the block capacity of Starling and GPS should be $\lfloor B/(S_a+S_v) \rfloor$ and $\lfloor B/((1+R)S_a+S_v) \rfloor$, respectively. B is the block size.

❷ *Determine the navigation index.* GPS randomly samples a small portion of the vectors in the dataset and uses them to construct the in-memory navigation index. This is because a larger navigation index does not improve performance as shown in Figure 1(b). The reason is that with typical graph degrees (e.g., 64, 128), the ground-truth neighbors of queries are already within 1-2 hops of the entry points identified by a small navigation index (e.g., 0.5%). We profile whether the navigation index improves performance by running the sampled query set Q'. If it does not improve performance, the navigation index is disabled. One such example is the Text2Image dataset in Figure 1(b).

❸ *Determine the graph cache and node cache.* For the rest of the memory space, GPS stores the adjacency lists. If there is still remaining space after storing the adjacency lists of all nodes, we store exact vectors and follow the same rule as the graph cache to select the vectors to store.

As shown in the middle plot of Figure 9, GPS may hold compressed vectors, navigation index, graph cache, and node cache in memory. Memory cache planning is fast for GPS and takes less than 5% of the overall index construction time, which is dominated by building the proximity graph.

Parameter setting. Our memory cache planning relies on query-based parameter tuning. Query set Q' can be easily obtained via the same embedding model that generates the base vectors, e.g., recommendation models produce both user and product embeddings, and CLIP produce both text and image embeddings, where user and text embeddings are used as queries. As such, Q' does not need to be real user queries, and synthetic queries generated by the same embedding model suffice.

Handling updates. Several algorithms support streaming updates (i.e., insert or delete vectors) to the proximity graph index [42, 62, 82]. GPS can utilize one of them since their core operation is running vector search by using the affected vectors as queries. GPS implements an update method similar to IP-DiskANN [82]. For node insertion, GPS searches the index graph to find the top-M nearest neighbors of the new node (M is the maximum node degree) and establishes bidirectional edges. The neighbor selection maintains only the M closest neighbors based on distance. For node deletion, GPS removes the deleted node from its neighbors' adjacency lists and establishes connections among the remaining neighbors to preserve graph connectivity.

Updates are directly applied to the adjacency lists stored in the memory cache. For the graph-replicated disk layout, we implement a lazy strategy to commit the updates. Specifically, we maintain a buffer map to record the graph update status, and the buffer is small as it does not store complete adjacency lists. When a sufficient portion of the adjacency lists are modified (e.g., 5%), we install the deltas by reading all disk-resident data via a linear scan. This is efficient since it conducts large sequential disk IO and can be conducted in the background when the query workload for normal vector search is low. Since modern SSDs have a good bandwidth (e.g., at 7GB/s), the write-back overhead is lightweight, e.g., it takes less than 10 minutes to scan the disk-resident data for the Wiki-100M dataset, while a full index rebuild takes over 6 hours. During searching, when reading an adjacency list from the disk and its deltas are not committed, GPS merges the deltas from the buffer to the adjacency list (without write back) such that the latest version of the proximity graph index is used. The overhead of using the delta buffer is negligible for search. More delicate designs are possible (e.g., install update write back when the data is fetched during query processing) but we leave them for future work.

4.2 Two-stage Search Algorithm

To adapt to the decoupled storage layout, GPS utilizes Algorithm 2 to conduct ANNS, which consists of a *search stage* (lines 10-20) and a *refinement stage* (lines 21-26). The search stage mainly conducts graph traversal by computing approximate distances: if the candidate node u is in the memory resident graph cache, disk access is bypassed, and GPS computes the approximate distances

Algorithm 2: Two-stage ANNS algorithm using the decoupled storage layout.

1 $\mathcal{L}_{ext}$: Nearest node list (sorted by exact distances).
2 $\mathcal{L}_{appr}$: Nearest node list (sorted by approx. distances).
3 $\mathcal{D}$: The maximum length for $\mathcal{L}_{appr}$.
4 $\mathcal{D}_r$: The number of vectors to re-rank, $\mathcal{D}_r = \sigma\mathcal{D}$.
5 **def** `Expand(`q`, `u`)`**:**
6 | **for** node v in u's adj. list **do**
7 | | $\mathcal{L}_{appr}$ append {v, `ApproxDist(`v`, `q`)`}
8 | Sort $\mathcal{L}_{appr}$, keep the top-$\mathcal{D}$ nodes
9 **def** `ANNS(`q`, `k`)`**:**
10 | $\mathcal{L}_{appr}$ append {entry e, `ApproxDist(`e`, `q`)`}
11 | **while** $\mathcal{L}_{appr}$ have unvisited nodes **do**
12 | | $u \leftarrow$ the nearest unvisited node in $\mathcal{L}_{appr}$
13 | | **if** u in graph cache GC **then**
14 | | | `Expand(`q`, `u`)`
15 | | **else**
16 | | | Load exact vector and adj. list for u
17 | | | $\mathcal{L}_{ext}$ append {u, `ExactDist(`u`, `q`)`}
18 | | | `Expand(`q`, `u`)`
19 | | | **for** node v in u's block **and** v in $\mathcal{L}_{appr}$ **do**
20 | | | | `Expand(`q`, `v`)`
21 | Keep the top-$\mathcal{D}_r$ nodes in $\mathcal{L}_{appr}$
22 | **for** node u in $\mathcal{L}_{appr}$ **and** u not in $\mathcal{L}_{ext}$ **do**
23 | | **if** u not in vector cache VC **then**
24 | | | Load exact vector for u
25 | | $\mathcal{L}_{ext}$ append {u, `ExactDist(`u`, `q`)`}
26 | Sort $\mathcal{L}_{ext}$, keep the top-k nodes
27 | **return** $\mathcal{L}_{ext}$

between its neighbors and the query by reading u's adjacency list from the graph cache (lines 13-14). Otherwise, GPS loads the block of u from disk, which includes u's exact vector, adjacency list, and the adjacency lists of some neighbors of u. GPS computes exact distance for u (line 17) and approximate distances for u's neighbors (line 18). A subtle point is that GPS also checks for each neighbor of u whose adjacency list is fetched (denoted as v), whether it belongs to $\mathcal{L}_{appr}$ (line 19). If this holds, v has a small approximate distance to the query and may be visited later; as such, GPS computes approximate distances for v's neighbors using its adjacency list (line 20).

After the search stage, $\mathcal{L}_{appr}$ records the top-ranking vectors in approximate distance, and $\mathcal{L}_{ext}$ keeps the exact distances computed using the loaded disk blocks. The refinement stage computes exact distances for the top-$\mathcal{D}_r$ nodes in $\mathcal{L}_{appr}$ to determine the final research results. This is because the real top-k neighbors of the query have a high probability to rank high in $\mathcal{L}_{appr}$ since the approximate distances have reasonable accuracy. By default, we set the refinement ratio $\sigma = 0.5$ and $\mathcal{D}_r = \sigma\mathcal{D}$, which provides good accuracy as shown in Figure 5. To compute exact distance for each of the top-$\mathcal{D}_r$ nodes, GPS first check if it is already in $\mathcal{L}_{ext}$, then checks if it is in the vector

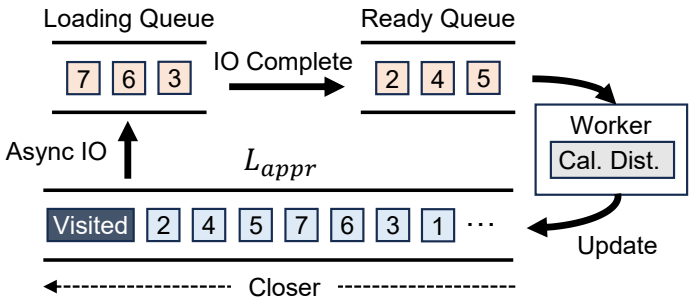


Fig. 10. The workflow of asynchronous block prefetch.

cache, and loads the vector from disk otherwise (lines 22-25). Finally, GPS returns the top-k nodes in exact distance as the ANNS results (line 27).

4.3 Asynchronous Block Prefetch

DiskANN uses synchronous IO to read the disk blocks, and computation is halted until the IO finishes. Starling improves DiskANN by checking the non-candidate nodes in the loaded disk blocks while waiting for the IOs of new disk blocks to complete. Since each disk block contains only a few nodes (i.e., 1-3) for high-dimensional vectors and the computation is lightweight for checking non-candidate nodes, the query processing thread may still be idle waiting for disk access, which prolongs query latency.

To avoid stalling computation and reduce query latency, GPS adopts the asynchronous disk block prefetch pipeline in Figure 10. In particular, we use two queues for the IO requests, i.e., *loading queue* for the disk blocks whose IOs are in-flight, and *ready queue* for the disk blocks that have already been fetched to memory. In each iteration, the worker thread first checks whether any IO request in the IO loading queue have been completed and move the completed request to the ready queue. If the loading queue is empty, it proactively select a batch of top-ranking but unvisited candidate nodes in $\mathcal{L}_{appr}$ for additional disk blocks. The actual disk IOs are executed by AIO system calls. Next, the worker will fetch a batch of nodes in ready queue for distance calculation and update $\mathcal{L}_{appr}$. The number of nodes processed per iteration is referred to as the beam width, typically set as 4 or 8. A larger beam width increases the overlap between computation and disk access, but may result in higher mis-prefetch penalties due to suboptimal candidate selection.

Our prefetching strategy bypasses the OS cache and is implemented using the asynchronous interface of `libaio` [28]. A recent paper, PipeANN [22], employs a similar idea to the hide latency of disk access. It utilizes the `io_uring` polling mode, which requires an additional thread for polling disk IOs, whereas GPS does not require the additional thread.

Other optimizations. In the refinement stage, GPS accesses a set of nodes and loads their exact vectors from disk. Since these disk accesses are independent, they can be managed in batches to fully utilize multiple disk channels and reduce latency. The batching is performed within each query rather than across queries. Disk IO libraries (e.g., `libaio` [28] and `io_uring` [2]) provide event queues that enable asynchronous disk IOs. Leveraging this mechanism, GPS calculates the exact distance for each vector as soon as its corresponding disk block is loaded, instead of waiting for the entire batch to complete, thereby overlapping computation with disk IO. By default, we use a batch size of 64 disk blocks, and the system performance remains stable across different refinement batch sizes.

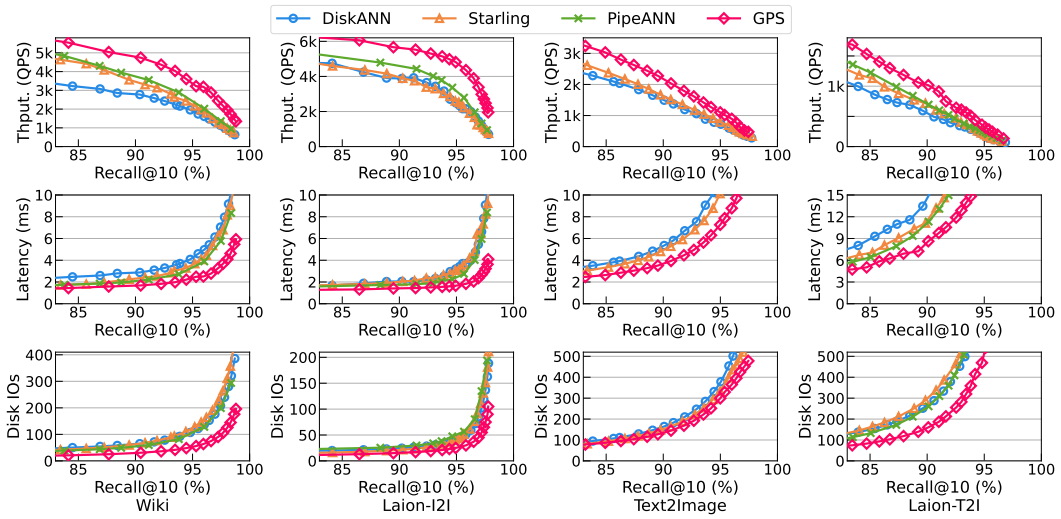


Fig. 11. The query throughput, and per query disk IOs of *GPS* and the baselines when using the same memory budget (20%). We don't run PipeANN on the Text2Image dataset as PipeANN doesn't support the inner product metric.

5 Experimental Evaluation

We extensively evaluate GPS and compare with state-of-the-art systems for disk-based ANNS. The key findings include:

- GPS consistently outperforms the baselines, achieving significantly higher query throughput and lower query latency.
- Our two key designs, graph prioritized memory cache and graph replicated disk block, are effective in reducing the disk IOs and outperform alternative designs.
- Other designs of GPS (e.g., asynchronous block prefetch) also enhance system performance.

5.1 Experiment Settings

Datasets. We conducted experiments on the 4 high-dimensional vector datasets in Table 1. Following PipeRAG [33], we use the BGE embedding model [8] to create a 384-dimensional dataset called *Wiki*, with the 100M base vectors generated from the Wikipedia text data [14] and the 100K query vectors generated from the MMLU text dataset [25]. The *Text2Image* dataset [58] is a cross-modal dataset with the base vectors derived from images and query vectors derived from texts. Laion [59, 60] includes two datasets, with dimensions of 512 and 768, respectively, and we use the first 100M image vectors in Laion for both base datasets. For the query vectors, we sample 10K text vectors for the 512-dimensional dataset, and sample 10K image vectors for the 768-dimensional dataset, following RoarGraph [9] and ANN benchmarks [5]. We donate these two datasets as Laion-T2I and Laion-I2I, respectively. Existing studies (e.g., [75]) report similar performance trends for 100M and billion-scale datasets. The high-dimensional datasets in our experiments, like Laion-I2I, even consume more space than some billion-scale datasets (SIFT-1B).

Baseline systems. We compare *GPS* with 3 state-of-the-art systems for disk-based ANNS, i.e., *DiskANN* [67], *Starling* [78], and *PipeANN* [22]. In particular, *DiskANN* stores the compressed vectors and caches both adjacency lists and vectors in memory. Several vector database systems, e.g., Milvus [76] and OpenGauss [37], use DiskANN for disk-based vector search. *Starling* builds upon *DiskANN*, conducts graph reordering for improved disk block locality, and stores a small in-memory

Table 2. The query throughput and latency of *GPS* and baseline systems when achieving the target recalls in Table 1. *Comparison* is (i) the throughput improvement factor, and (2) the ratio of *GPS*'s latency, over the best-performing baseline.

	Throughput (QPS)				Latency (ms)			
Datasets	**Wiki**	**L-I2I**	**T2Img**	**L-T2I**	**Wiki**	**L-I2I**	**T2Img**	**L-T2I**
DiskANN	1,820	2,337	1,492	589	4.39	3.42	5.36	13.56
Starling	2,134	2,529	1,514	691	3.74	3.16	5.28	11.56
PipeANN	2,249	3,027	-	721	3.62	2.51	-	11.22
GPS	3,490	4,825	2,088	1,016	2.29	1.65	3.83	8.62
Comparison	1.55x	1.59x	1.38x	1.41x	63%	66%	73%	77%

index for fast navigation. *PipeANN* use *DiskANN*'s disk layout and use the same in-memory index as *Starling*. It proposes a pipeline to overlap the disk access and execution and use `io_uring` library for disk access. We also create a baseline called *GPS-GR* by replacing the graph-replicated disk layout of *GPS* with Starling's disk layout to evaluate the trade-off of disk space and efficiency.

For all systems, following DiskANN, we adopt PQ [31] as the vector compression algorithm, and Vamana [67] as the proximity graph index. By default, the compression ratios of PQ are 1/8 for LaionT2I and Text2Image, and 1/16 for LaionI2I and Wiki, which are tuned for best performance. The search beam widths are also tuned for each dataset (e.g., 4 for LaionI2I, 8 for the others). The Vamana construction complexity cef is fixed to 128, and the graph degree is set to 48.

We do not compare with SPANN [10] as it was outperformed by Starling and DiskANN in [78]. Systems that target different settings are also excluded, e.g., AiSAQ [69] and LM-DiskANN [54] store all data on disk and do not use memory at all; SPFresh [84] and FreshDiskANN [62] focus on the streaming updates for the vector index; and FusionANNS [73] uses GPU to accelerate computation.

Testbed. We conducted all experiments on a server that hosts an Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz CPU that contains 72 cores and 504GB DRAM. The server is equipped with eight 1.9TB NVMe SSDs, and we applied RAID-0 [1] on all its SSDs and obtained a 14TB disk space with an exaggerated bandwidth of 4.0 GB/s. The operating system is Ubuntu 22.04.4 with Linux kernel version 6.8.1. By default, we use a memory that is 20% of the original dataset, following Starling [78]. The experiments were executed with 8 threads, with each query processed by a single thread. *GPS* adopts the σ value of 0.5.

Performance metrics. Our primary metrics are *query throughput*, measured by the average number of queries processed per second (QPS), *query latency*, measured by the average query processing time, and the *average disk IOs*, measured by the average number of disk accesses per query. Following previous works [10, 67, 78], we use *recall@10* to measure the quality of the search results and report the average recall@10 over the queries.

5.2 Main Results

Figure 11 reports the query throughput, query latency, and average disk IOs of *GPS* and the baseline systems. We observe that *GPS* consistently outperforms the baseline systems across all datasets at recall targets. When achieving the same recall, *GPS* improves the query throughput of *PipeANN* by 52% when averaged over datasets, reduces the query latency by 32%, and reduces the average disk IOs by 39%. The detailed results are provided in Table 2. Comparing the three baselines, We observe that *PipeANN* outperforms *DiskANN* and *Starling* by overlapping computation with disk IO, hiding part of the access latency. The gains are larger on cross-modal datasets, where longer search paths amortize pipeline overhead. *Starling* only slightly surpasses *DiskANN*. On the Laion-T2I dataset,

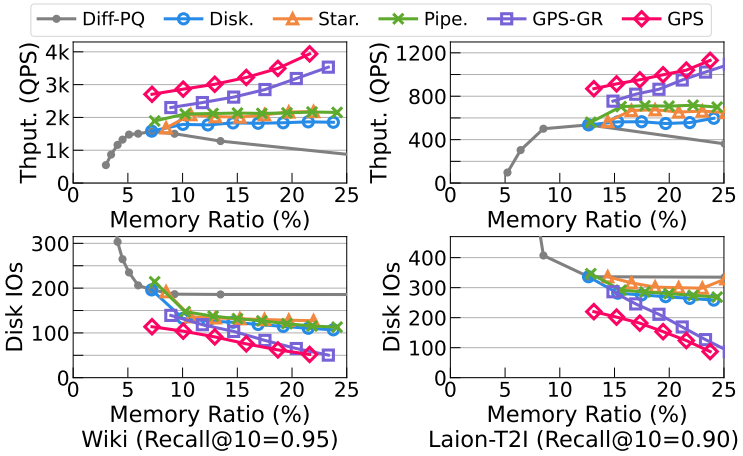


Fig. 12. The throughput and Disk IOs of *GPS* and the baselines when varying the memory consumption.

Starling performs almost the same as *DiskANN*. This is because the vector dimension is high for Laion-T2I (i.e., 768), and a 4KB disk block can only store 1 vector, which makes *Starling*'s designs fail. In contrast, *GPS* introduces the graph-replicated disk layout, where a disk block can hold several neighbors' adjacency lists, leading to better disk block utilization and fewer disk accesses.

We also observe that the improvements of *GPS* over the baselines are more pronounced for the single-modal datasets (i.e., Wiki and Laion-I2I) than the cross-modal datasets. This is because the cross-modal datasets require more accurate but larger compressed vectors than the single-modal datasets. As such, given the same memory budget, the cross-modal datasets leave less space for the graph cache of *GPS*, which limits its performance gains. Additionally, *GPS* achieves greater improvements on datasets with higher dimensionality. For example, a 59% throughput improvement for Laion-I2I (the highest-dimensional dataset), compared to a 38% improvement for Text2Image (the lowest-dimensional dataset). This is because Text2Image is a cross-modal dataset with a low dimension, less space is reserved for caching graphs compared to other datasets. With a 20% memory budget, only 25% of nodes can be stored in the graph cache for Text2Image, whereas more than 60% of nodes can be cached for the other datasets. Specifically, Laion-I2I can cache the whole index graph in its memory under a 20% memory budget.

Varying memory consumption. Figure 12 reports the query throughput and disk IOs when achieving the recall targets in Table 1 but changing the memory they can utilize. We implement *Diff-PQ* over DiskANN, which adjusts the compression ratios of PQ to use up all memory and does not utilize the memory cache. For *Diff-PQ*, we observe that search performance first improves but then decreases with the increase in memory size. This echoes our observation in § 3.1. For the other baselines, we adopt the compression ratio that achieves the highest throughput for each dataset. For *DiskANN*, *Starling* and *PipeANN*, we observe that large memory only slightly reduces their disk access, and the throughput improvement is negligible. This indicates that both the navigation index and node cache are ineffective under a high memory budget. For *GPS-GR*, we find that it outperforms the baselines even with the smallest memory budget. This is because it adopts the asynchronous block prefetch and hides the disk access latency. For higher memory budgets, *GPS-GR* has higher throughput because more nodes can be found in the graph cache in the search stage, and disk IOs are saved by checking only the top-ranking nodes in the refinement stage. Finally, by cooperating with the graph-replicated disk layout, *GPS* achieves higher throughput compared

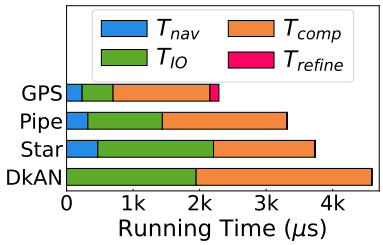


Fig. 13. Query time decomposition on the Wiki dataset under the 20% memory budget and 95% recall.

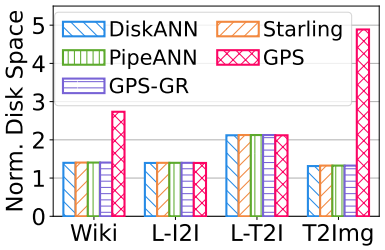


Fig. 14. The disk space used by the systems, normalized by the size of the raw *original* vector dataset.

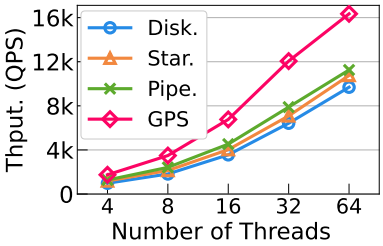


Fig. 15. The throughput of *GPS* and baselines when changing the number of threads for query processing.

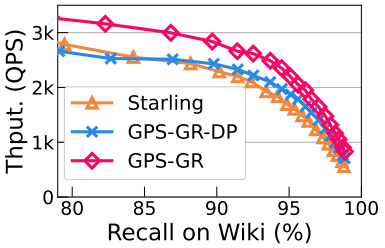


Fig. 16. The effect of *GPS*' asynchronous block prefetch. The navigation index and data caches are disabled.

with *GPS-GR*, particularly under lower memory budgets. This is because the layout caches the neighbors' adjacency lists on the disk block, thereby reducing disk access in the search stage.

Time decomposition. To understand the speedup of *GPS*, we decompose the request latency of it and the baselines in Figure 13. The result is obtained under same parameter setting as Figure 11 (i.e., 8 threads and $\sigma = 0.5$). T_{nav}, T_{IO}, T_{comp}, and T_{refine} are the time for in-memory index navigation, CPU idling waiting for disk IOs, running the search stage, and refining the final top-k, respectively. *GPS*'s T_{IO} includes the waiting time in both the search and refinement stages. We observe that all baselines suffer from long disk access time. *Starling* and *PipeANN* has relatively shorter computation and disk IO time due to the navigation index and execution optimizations. With nearly the same computation time, *GPS* significantly reduces the time for disk IOs using the two-stage search algorithm and asynchronous block prefetch, which reduce the amount of disk IOs and hide the disk IO latency, respectively. Finally, we observe that the execution time for the refinement stage is much shorter than the search stage. This is because the refinement stage computes exact distances only for the top-ranking nodes in $\mathcal{L}appr$, which is significantly fewer than the total number of compressed vector computations required for all nodes in $\mathcal{L}appr$.

Disk space overhead. Figure 14 reports the disk space consumption of the systems. *DiskANN*, *Starling*, *PipeANN*, and *GPS-GR* use the same disk space because they have no additional disk consumption. In contrast, *GPS* introduces a graph-replicated disk layout, which trades disk space for efficiency. For high-dimensional datasets, such as Laion-I2I, one 4KB block can only contain one node such that the disk space for *GPS* is the same as the other systems. However, for datasets with relatively lower dimensions, the replicated layout of *GPS* brings larger disk space amplification.

Varying the number of threads. Figure 15 displays the throughput of *GPS* and the baselines when launching varying numbers of threads to process ANNS queries. Other parameters remain the same as the main results. We observe that *GPS* consistently outperforms the baselines by a large margin when using different numbers of threads. Specifically, *GPS* achieves an average throughput

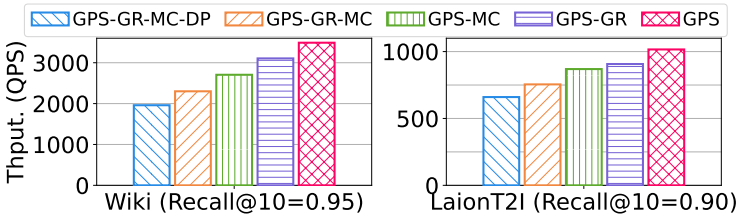


Fig. 17. Performance breakdown for *GPS*.

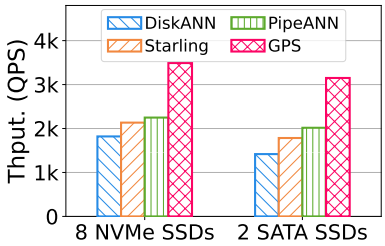


Fig. 18. Evaluation with weaker SSD setting. Report the throughput of Wiki under 95% recall.

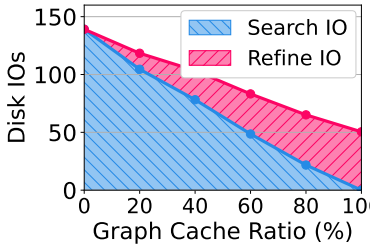


Fig. 19. Breakdown for search-IO and refine-IO on Wiki under 95% recall for GPS-GR in Figure 12.

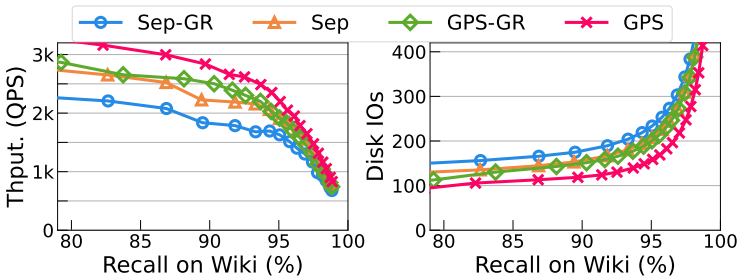


Fig. 20. Comparing the GPS' graph replicated disk layout with the vector-graph separation layout.

improvement of 83% over *DiskANN*, 60% over *Starling*, and 46% over *PipeANN*, respectively. The reason is that *GPS* requires less disk access for each query, rendering its performance less dependent on the disk bandwidth when using many threads.

5.3 Micro Experiments

Performance breakdown. To evaluate the contribution of each module in *GPS*, we conduct a performance breakdown experiment, as shown in Figure 17. Specifically, *GPS-MC* disables the memory cache, *GPS-GR-MC* disables both the memory cache and the graph-replicated layout, and *GPS-GR-MC-DP* further disables block prefetching. The results show that each technique incrementally improves throughput. For example, on the Wiki dataset, the overall performance increases by 78% when all techniques are enabled compared to the baseline configuration.

Asynchronous block prefetch. To evaluate the gain of our asynchronous block prefetch, we create a baseline *GPS-GR-DP*, which disables the block prefetch. Figure 16 compares the query throughput. Compared with *GPS-GR-DP*, *GPS-GR* achieves a throughput improvement of 18% for the Wiki dataset under 95% recall. This indicates that the block prefetch does not affect query accuracy, and it effectively hides the latency of disk access, thus improving the system performance.

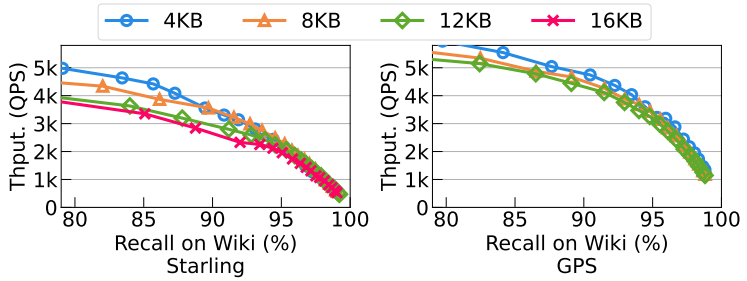


Fig. 21. The throughput of Starling and *GPS* when using 20% memory budget but varying the block size.

Performance on weaker SSD. Figure 18 presents the results under weaker SSD settings. In this experiment, two SATA SSDs were configured in RAID-0 [1], providing a total bandwidth of 680MB/s. As expected, all systems exhibit lower throughput with weaker SSDs as the disk access time increased. *DiskANN* shows a more pronounced drop, with throughput decreasing by 22.1%. In contrast, *PipeANN* and *GPS* are less affected, with throughput reductions of 10.3% and 9.7%, respectively.

Breakdown seach and refine IO. We break down the search IO and refine IO of *GPS* in Figure 19. As the graph cache ratio increases, the proportion of search IO steadily decreases, while refine IO correspondingly increases. Specifically, the total IO reduces from 139.1 when the memory cache is disabled (ratio is 0%) to 50.5 when the whole graph is stored in memory (ratio is 100%).

Alternatives for the disk layout. For each disk block, our graph-replicated disk layout places a node's adjacency list, exact vector, and its neighbors' adjacency lists to increase the disk block data locality. Two alternative layouts that may also improve disk data locality are described in § 3.4. The first is the vector-graph separation layout (e.g., BAMG [38]), which classified the blocks into graph blocks and vector blocks, with graph blocks containing only adjacency lists and vector blocks containing only vectors(i.e., Figure 7(b)). We created two baselines based on it, i.e., *Sep-GR* and *Sep*. *Sep-GR* applies *Starling*'s re-layout strategy to both the graph blocks and vector blocks; while *Sep* trades disk space for efficiency, replicating a node's neighbors' adjacency lists in the graph blocks. Compared with *DiskANN*'s disk consumption, on the Wiki dataset, *Sep-GR* requires an additional 10% disk space for block padding, while *Sep* consumes an extra 200% disk space for graph replication.

Figure 20 compares the throughput and disk IO between the separation layout and *GPS*'s layout, using the same two-stage search algorithm. The results suggest that the graph-vector separation layout requires more disk IOs and results in lower throughput compared to *GPS*'s layout. The underlying reason is that, in the refinement stage of the separation layout, all top-ranking vectors need to be loaded from the disk and each of them requires one disk access, as none of them are accessed during the search stage. This introduces an extra disk IO, and two disk accesses are needed for some nodes. As a result, the separation layout leads to poorer performance due to the additional I/O operations. Even when employing the graph-replicated layout where each graph block can store the adjacency lists of 20 neighbors (*Sep*), the performance remains inferior to that of *GPS*.

The second alternative is using larger block sizes (i.e., Figure 7(c)). Figure 21 reports the throughput of *Starling* and *GPS* when changing the block sizes. With a larger block size, a single block can accommodate more nodes in *Starling* or replicate more neighbors in *GPS*, while the cost of reading a block also increases. Note that a 12KB block size is sufficient to store all neighbors within a graph-replicated block in *GPS*. We observe that system throughput slightly decreases as the block

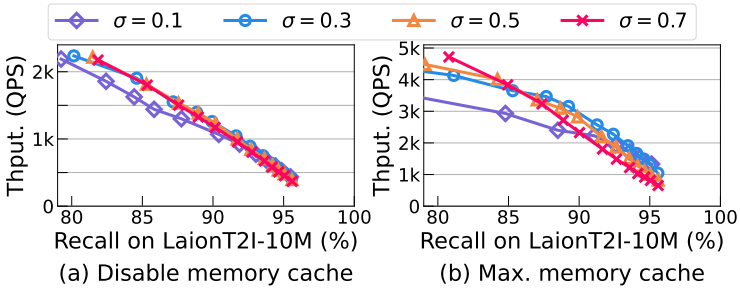


Fig. 22. The query throughput of GPS at different reranking ratio σ, (a) disables the memory cache while (b) stores the entire graph index in memory.

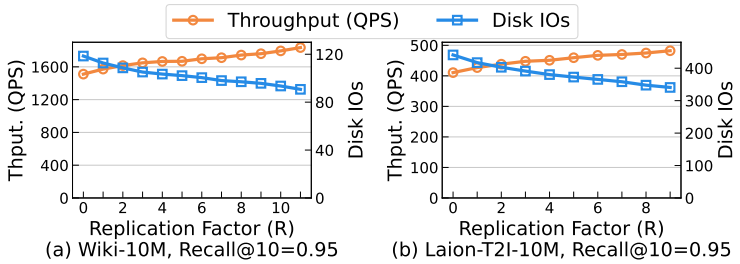


Fig. 23. Changing GPS's replication factor R on two 10M datasets. The memory cache is disabled.

size increases for both *Starling* and *GPS*. This is because a larger block size consumes more disk bandwidth for each disk access.

Changing beam widths. Figure 24 reports the throughput of *GPS* and the baselines when using different beam widths. In particular, beam width controls the number of candidate nodes checked in each iteration, and a larger beam width allows a larger disk read batch but may also introduce extra disk access, especially when the queue size $\mathcal{D}$ is small. The results show that *GPS* outperforms *DiskANN* and *Starling* across all beam widths. Notably, *GPS* exhibits consistent performance across different beam widths, while *Starling* achieves better throughput with larger beam widths.

Changing σ value. We check the influence of σ in Figure 22. Two settings are experimented, i.e., disabling the memory cache for adjacency list and caching all adjacency lists in memory. The results show that $\sigma = 0.3$ and $\sigma = 0.5$ perform consistently well across different datasets and memory caching conditions.

Changing replication factor R. Figure 23 report that as R increases, disk IO consistently decreases and thus query throughput keeps increasing. We adopt a degree limit of 48, and thus each adjacency list takes up about 200 bytes. For the Laion-T2I dataset with 512 dimension, a 4 KB disk page can store only a single node (i.e., a vector and its adjacency list), leaving the remaining space idle for padding, and thus increasing R from 0 to 9 does not enlarge the disk overhead by using the idle space. In particular, with $R = 0$, a node occupies 2248 bytes ($512 \times 4 + 200 = 2248$), leaving 1848 bytes idle of padding; with $R = 9$, the size of a node becomes 4048 bytes ($512 \times 4 + 200 + 200 \times 9$), leaving only 48 bytes of padding. In contrast, for the Wiki dataset with 384 dimension, the disk space blowup increases from 1x to 2x when R increases from 1 to 2 because a 4 KB disk page can keep 2 nodes with $R = 1$ but only 1 node with $R = 2$. As such, if disk space is a concern, users can properly configure R for GPS to ensure that the disk space blowup is 1x.

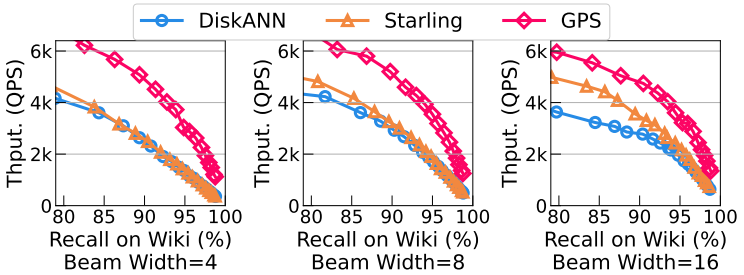


Fig. 24. The throughput of *GPS* and baselines when using 20% memory budget but changing the beam width.

6 Related Work

ANNS indexes. There are two main types of indexes for ANNS, i.e., IVF [6] and proximity graph [48]. IVF clusters the vectors and only checks a few nearest clusters for each query. Many IVF-based index structures are proposed with different clustering methods and search algorithms [6, 10, 46, 55, 73, 75]. Proximity graph connects similar vectors to form a graph and processes queries by graph traversal. Proximity graph provides the state-of-the-art performance for ANNS, achieving the same recall with much fewer distance computations than IVF [48]. As such, both DiskANN [67] and Starling [78] adopt proximity graphs. There are many variants for proximity graph [9, 15, 29, 48, 49, 51, 68, 77, 91, 93], they differ in the graph construction and edge connection rules but adopt the same graph traversal procedure for query processing. Thus, their data access patterns are the same for disk-based ANNS, and GPS can be generalized across them. The vector database in ANNS indexes could be embedded from language models [8, 92], image models [39, 81], recommendation models [50, 86], or graph models [85, 94, 95]. Vector quantization is an important technique for ANNS that compresses each vector for smaller storage and faster distance computation. There are many vector quantization methods, e.g., PQ [31], OPQ [19], LVQ [3], deltaPQ [79], and RabitQ [16, 18], that work in different ways. DADE [13], GleanVec [71], LeanVec [70], and ADSampling [17] utilize projection methods like PCA to reduce the vector dimension for acceleration. GPS generalizes across the quantization methods because they are used as a black box to compute approximate distance. Some researches consider more complex vector queries than top-k nearest neighbors, e.g., ANNS with range or attribute filters [20, 32, 40, 83, 97]. However, proximity graph is also popular for these queries, and thus GPS can be extended by incorporating the attributes.

Disk-based ANNS. To handle large vector datasets at a low cost, several systems use disk for data storage. DiskANN [67], Starling [78], and PipeANN [22] run with a small memory (e.g., 10-20% of the dataset size) and a large disk. GPS targets the same scenario but outperforms them by a large margin by carefully designing the data layout to reduce disk IOs. MARGO [89] enhances Starling's re-layout algorithm by assigning weights to the edges such that important edges make higher impact during the graph reordering. PageANN [34] constructs a block-aligned graph, and computes distances for all nodes when a disk block is loaded. They suffer from the same problem as Starling, i.e., a disk block holds a few vectors when the vector dimension increases, which decreases the gains of their optimizations. GoVector [96] keeps in memory a dynamic node cache, where the vector and adjacency list of each node are still kept together. We summarized these existing disk-based ANNS systems in Table 3. To the best of our knowledge, GPS is the only work that identifies the non-equivalence between graph traversal and vector access in this setting, and thus decouples their memory and disk placement to reduce disk IOs.

SPANN [10] and FusionANNS [73] adopt the IVF index for disk-based ANNS. SPANN stores the cluster centers in memory to identify the nearest centers for a query and fetch the required clusters

Table 3. Summarizing the designs of disk-based ANNS systems. "-" represents no optimization for an aspect.

System	Execution	Memory Cache	Disk Layout
DiskANN [67]	Beam Search	Node Cache	-
Starling [78]	Pipelined Block Search	Nav. Index	Reordering
PipeANN [22]	Async.	Nav. Index	-
MARGO [89]	-	-	Path-aware Graph Construction
PageANN [34]	Pipelined Block Search	Block Cache	Block-aligned Graph Construction
GoVector [96]	-	LFU Dynamic Node Cache	Reordering
BAMG [38]	Block Search	Multi-Layer Nav. Index	Node-Vector Separation
GPS (Ours)	Delayed Ranking, Async.	Small Nav. Index, Adj. Lists	Graph-replicated

(of vectors) from the disk on demand. FusionANNS keeps the cluster centers on CPU memory, compressed vectors on GPU memory, and exact vectors on disk. A query first identifies its nearest clusters on the CPU, then uses GPU to compute approximate distances for these clusters, and finally reads the top-ranking vectors from disk for re-ranking with exact distance. Since the IVF index computes more distances than proximity graph index, SPANN is outperformed by Starling [78], and small GPU memory limits the accuracy of the compressed vectors for FusionANNS, which may require reading many exact vectors for re-ranking. LM-DiskANN [54] and AiSAQ [69] target a more extreme scenario that uses only disk but no memory.

Other ANNS systems. Some studies handle large vector datasets using emerging hardware, such as HM-ANN [57] for persistent memory, CXL-ANNS [30] for CXL memory, MemANNS [11] for UPMEM PIM, and SmartANNS [72] for SmartSSD. These systems are tailored to fit the performance characteristics of their target hardware. To improve search efficiency, some works employ accelerators such as GPUs [21, 52, 73, 75, 87] and FPGAs [65, 88, 90] to accelerate the distance computations. Several studies scale ANNS beyond a single server. For example, Milvus [76], AnalyticDB [80], Hakes [26], and d-HNSW [45] can run ANNS over distributed machines, while Vexless [66] utilizes serverless cloud functions.

7 Conclusion

In this paper, we revisit the storage layout for disk-based graph indexes. Through extensive profiling and analysis, we observe that for proximity graph, which is the state-of-the-art index for vector search, the adjacency lists are more important than the vectors. Guided by this insight, we prioritize the adjacency lists over the vectors and propose two key storage layout designs, i.e., graph prioritized memory cache and graph replicated disk block. Based on the new storage layout, we built an efficient and general disk-based ANNS system GPS, and GPS significantly improves the performance of state-of-the-art systems for disk-based vector search.

References

[1] 2003. Linux RAID Array. https://docs.kernel.org/admin-guide/md.html.

[2] 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[3] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore L. Willke. 2023. Similarity search in the blink of an eye with compressed indices. *Proc. VLDB Endow.* 16, 11 (2023), 3433–3446. doi:10.14778/3611479.3611537

[4] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. 2023. Retrieval-based Language Models and Applications. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts, ACL 2023, Toronto, Canada, July 9-14, 2023*. Association for Computational Linguistics, 41–46. doi:10.18653/V1/2023.ACL-TUTORIALS.6

[5] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.* 87 (2020). doi:10.1016/J.IS.2019.02.006

[6] Artem Babenko and Victor S. Lempitsky. 2012. The inverted multi-index. In *2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, June 16-21, 2012*. IEEE Computer Society, 3069–3076. doi:10.1109/CVPR.2012.6248038

[7] Wing-Man Chan and Alan George. 1980. A linear time implementation of the reverse Cuthill-McKee algorithm. *BIT Numerical Mathematics* 20, 1 (1980), 8–14.

[8] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation. *CoRR* abs/2402.03216 (2024). arXiv:2402.03216 doi:10.48550/ARXIV.2402.03216

[9] Meng Chen, Kai Zhang, Zhenying He, Yinan Jing, and X. Sean Wang. 2024. RoarGraph: A Projected Bipartite Graph for Efficient Cross-Modal Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 17, 11 (2024), 2735–2749. doi:10.14778/3681954.3681959

[10] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 5199–5212. https://proceedings.neurips.cc/paper/2021/hash/299dc35e747eb77177d9cea10a802da2-Abstract.html

[11] Sitian Chen, Amelie Chi Zhou, Yucheng Shi, Yusen Li, and Xin Yao. 2024. MemANNS: Enhancing Billion-Scale ANNS Efficiency with Practical PIM Hardware. *CoRR* abs/2410.23805 (2024). arXiv:2410.23805 doi:10.48550/ARXIV.2410.23805

[12] Nadav Cohen, Or Sharir, and Amnon Shashua. 2016. On the Expressive Power of Deep Learning: A Tensor Analysis. In *Proceedings of the 29th Conference on Learning Theory, COLT 2016, New York, USA, June 23-26, 2016 (JMLR Workshop and Conference Proceedings, Vol. 49)*. JMLR.org, 698–728. http://proceedings.mlr.press/v49/cohen16.html

[13] Liwei Deng, Penghao Chen, Ximu Zeng, Tianfu Wang, Yan Zhao, and Kai Zheng. 2024. Efficient Data-aware Distance Comparison Operations for High-Dimensional Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 18, 3 (2024), 812–821. https://www.vldb.org/pvldb/vol18/p812-zheng.pdf

[14] Wikimedia Foundation. 2007. *Wikimedia Downloads*. https://dumps.wikimedia.org

[15] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474. doi:10.14778/3303753.3303754

[16] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2024. Practical and Asymptotically Optimal Quantization of High-Dimensional Vectors in Euclidean Space for Approximate Nearest Neighbor Search. *CoRR* abs/2409.09913 (2024). arXiv:2409.09913 doi:10.48550/ARXIV.2409.09913

[17] Jianyang Gao and Cheng Long. 2023. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. *Proc. ACM Manag. Data* 1, 2 (2023), 137:1–137:27. doi:10.1145/3589282

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

[19] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 4 (2014), 744–755. doi:10.1109/TPAMI.2013.240

[20] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *Proceedings of the ACM Web Conference 2023, WWW 2023, Austin, TX, USA, 30 April 2023 - 4 May 2023*. ACM, 3406–3416. doi:10.1145/3543507.3583552

[21] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. 2023. GGNN: Graph-Based GPU Nearest Neighbor Search. *IEEE Trans. Big Data* 9, 1 (2023), 267–279. doi:10.1109/TBDATA.2022.3161156

[22] Hao Guo and Youyou Lu. 2025. Achieving Low-Latency Graph-Based Vector Search via Aligning Best-First Search Algorithm with SSD. In *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*. USENIX Association, 171–186. https://www.usenix.org/conference/osdi25/presentation/guo

[23] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 9 (2023), 2090–2102. doi:10.14778/3598581.3598584

[24] Tengda Han, Weidi Xie, and Andrew Zisserman. 2019. Video Representation Learning by Dense Predictive Coding. In *2019 IEEE/CVF International Conference on Computer Vision Workshops, ICCV Workshops 2019, Seoul, Korea (South), October 27-28, 2019*. IEEE, 1483–1492. doi:10.1109/ICCVW.2019.00186

[25] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=d7KBjmI3GmQ

[26] Guoyu Hu, Shaofeng Cai, Tien Tuan Anh Dinh, Zhongle Xie, Cong Yue, Gang Chen, and Beng Chin Ooi. 2025. HAKES: Scalable Vector Database for Embedding Search Service. *arXiv preprint arXiv:2505.12524* (2025).

[27] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based Retrieval in Facebook Search. In *KDD '20: The 26th ACM SIGKDD*

Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020. ACM, 2553–2561. doi:10.1145/3394486.3403305

[28] The Linux Programming Interface. 2008. POSIX asynchronous I/O (AIO). https://man7.org/linux/man-pages/man7/aio.7.html.

[29] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. *CoRR* abs/1810.07355 (2018). arXiv:1810.07355 http://arxiv.org/abs/1810.07355

[30] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 585–600. https://www.usenix.org/conference/atc23/presentation/jang

[31] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128. doi:10.1109/TPAMI.2010.57

[32] MENGXU JIANG, ZHI YANG, FANGYUAN ZHANG, GUANHAO HOU, JIEMING SHI, WENCHAO ZHOU, FEIFEI LI, and SIBO WANG. 2025. DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter. (2025).

[33] Wenqi Jiang, Shuai Zhang, Boran Han, Jie Wang, Bernie Wang, and Tim Kraska. 2024. PipeRAG: Fast Retrieval-Augmented Generation via Algorithm-System Co-design. *CoRR* abs/2403.05676 (2024). arXiv:2403.05676 doi:10.48550/ARXIV.2403.05676

[34] Dingyi Kang, Dongming Jiang, Hanshen Yang, Hang Liu, and Bingzhe Li. 2025. Scalable Disk-Based Approximate Nearest Neighbor Search with Page-Aligned Graph. *arXiv preprint arXiv:2509.25487* (2025).

[35] Ji-Hoon Kim, Yeo-Reum Park, Jaeyoung Do, Soo-Young Ji, and Joo-Young Kim. 2023. Accelerating Large-Scale Graph-Based Nearest Neighbor Search on a Computational Storage Platform. *IEEE Trans. Computers* 72, 1 (2023), 278–290. doi:10.1109/TC.2022.3155956

[36] Cangyuan Li, Ying Wang, Cheng Liu, Shengwen Liang, Huawei Li, and Xiaowei Li. 2021. GLIST: Towards In-Storage Graph Learning. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. USENIX Association, 225–238. https://www.usenix.org/conference/atc21/presentation/li-cangyuan

[37] Guoliang Li, Jiang Wang, and Guo Chen. 2024. openGauss: An Enterprise-Grade Open-Source Database System. *J. Comput. Sci. Technol.* 39, 5 (2024), 1007–1028. doi:10.1007/S11390-024-4302-2

[38] Huiling Li and Jianliang Xu. 2025. BAMG: A Block-Aware Monotonic Graph Index for Disk-Based Approximate Nearest Neighbor Search. *CoRR* abs/2509.03226 (2025). arXiv:2509.03226 doi:10.48550/ARXIV.2509.03226

[39] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. 2021. Embedding-based Product Retrieval in Taobao Search. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*. ACM, 3181–3189. doi:10.1145/3447548.3467101

[40] Anqi Liang, Pengcheng Zhang, Bin Yao, Zhongpu Chen, Yitong Song, and Guangxu Cheng. 2024. UNIFY: Unified Index for Range Filtered Approximate Nearest Neighbors Search. *CoRR* abs/2412.02448 (2024). arXiv:2412.02448 doi:10.48550/ARXIV.2412.02448

[41] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. 2019. Cognitive SSD: A Deep Learning Engine for In-Storage Data Retrieval. In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 395–410. https://www.usenix.org/conference/atc19/presentation/liang

[42] Dawei Liu, Bolong Zheng, Ziyang Yue, Fuhao Ruan, Xiaofang Zhou, and Christian S Jensen. 2025. Wolverine: Highly Efficient Monotonic Search Path Repair for Graph-Based ANN Index Updates. *Proceedings of the VLDB Endowment* 18, 7 (2025), 2268–2280.

[43] Haotian Liu, Kilho Son, Jianwei Yang, Ce Liu, Jianfeng Gao, Yong Jae Lee, and Chunyuan Li. 2023. Learning Customized Visual Models with Retrieval-Augmented Knowledge. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*. IEEE, 15148–15158. doi:10.1109/CVPR52729.2023.01454

[44] Renjie Liu, Yichuan Wang, Xiao Yan, Zhenkun Cai, Minjie Wang, Haitian Jiang, Bo Tang, and Jinyang Li. 2024. DiskGNN: Bridging I/O Efficiency and Model Accuracy for Out-of-Core GNN Training. *CoRR* abs/2405.05231 (2024). arXiv:2405.05231 doi:10.48550/ARXIV.2405.05231

[45] Yi Liu, Fei Fang, and Chen Qian. 2025. Efficient Vector Search on Disaggregated Memory with d-HNSW. *arXiv preprint arXiv:2505.11783* (2025).

[46] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2024. JUNO: Optimizing High-Dimensional Approximate Nearest Neighbour Search with Sparsity-Aware Algorithm and Ray-Tracing Core Mapping. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*. ACM, 549–565. doi:10.1145/3620665.3640360

[47] Aravindh Mahendran and Andrea Vedaldi. 2015. Understanding deep image representations by inverting them. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 5188–5196. doi:10.1109/CVPR.2015.7299155

[48] Yury A. Malkov and Dmitry A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *CoRR* abs/1603.09320 (2016). arXiv:1603.09320 http://arxiv.org/abs/1603.09320

[49] Magdalen Dobson Manohar, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2024, Edinburgh, United Kingdom, March 2-6, 2024*. ACM, 270–285. doi:10.1145/3627535.3638475

[50] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). arXiv:1906.00091 http://arxiv.org/abs/1906.00091

[51] Naoki Ono and Yusuke Matsui. 2023. Relative NN-Descent: A Fast Index Construction for Graph-Based Approximate Nearest Neighbor Search. In *Proceedings of the 31st ACM International Conference on Multimedia, MM 2023, Ottawa, ON, Canada, 29 October 2023- 3 November 2023*. ACM, 1659–1667. doi:10.1145/3581783.3612290

[52] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2024. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 4236–4247. doi:10.1109/ICDE60146.2024.00323

[53] Big ANN Organizers. 2021. *Big ANN Benchmark*. https://big-ann-benchmarks.com/neurips21.html

[54] Yu Pan, Jianxin Sun, and Hongfeng Yu. 2023. LM-DiskANN: Low Memory Footprint in Disk-Native Dynamic Graph-Based ANN Indexing. In *IEEE International Conference on Big Data, BigData 2023, Sorrento, Italy, December 15-18, 2023*. IEEE, 5987–5996. doi:10.1109/BIGDATA59044.2023.10386517

[55] Jeffrey Pound, Floris Chabert, Arjun Bhushan, Ankur Goswami, Anil Pacaci, and Shihabur Rahman Chowdhury. 2025. MicroNN: An On-device Disk-resident Updatable Vector Database. *arXiv preprint arXiv:2504.05573* (2025).

[56] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2023. In-Context Retrieval-Augmented Language Models. *Trans. Assoc. Comput. Linguistics* 11 (2023), 1316–1331. doi:10.1162/TACL_A_00605

[57] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. https://proceedings.neurips.cc/paper/2020/hash/788d986905533aba051261497ecffcbb-Abstract.html

[58] Yandex Research. 2021. *Benchmarks for Billion-Scale Similarity Search*. https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search

[59] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, Patrick Schramowski, Srivatsa Kundurthy, Katherine Crowson, Ludwig Schmidt, Robert Kaczmarczyk, and Jenia Jitsev. 2022. LAION-5B: An open large-scale dataset for training next generation image-text models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. http://papers.nips.cc/paper_files/paper/2022/hash/a1859debfb3b59d094f3504d5ebb6c25-Abstract-Datasets_and_Benchmarks.html

[60] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. 2021. LAION-400M: Open Dataset of CLIP-Filtered 400 Million Image-Text Pairs. *CoRR* abs/2111.02114 (2021). arXiv:2111.02114 https://arxiv.org/abs/2111.02114

[61] Chanop Silpa-Anan and Richard I. Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2008), 24-26 June 2008, Anchorage, Alaska, USA*. IEEE Computer Society. doi:10.1109/CVPR.2008.4587638

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

[63] Fatima Zohra Smaili, Xin Gao, and Robert Hoehndorf. 2018. Onto2Vec: joint vector-based representation of biological entities and their ontology-based annotations. *Bioinform.* 34, 13 (2018), i52–i60. doi:10.1093/BIOINFORMATICS/BTY259

[64] Fatima Zohra Smaili, Xin Gao, and Robert Hoehndorf. 2019. OPA2Vec: combining formal and informal content of biomedical ontologies to improve similarity-based prediction. *Bioinform.* 35, 12 (2019), 2133–2140. doi:10.1093/BIOINFORMATICS/BTY933

[65] Yifeng Song, Chenjie Liu, Rongrong Zhang, Danyang Zhu, and Zhongfeng Wang. 2025. An Efficient FPGA Implementation of Approximate Nearest Neighbor Search. *IEEE Trans. Very Large Scale Integr. Syst.* 33, 6 (2025), 1705–1714. doi:10.1109/TVLSI.2025.3544342

[66] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. 2024. Vexless: a serverless vector data management system using cloud functions. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[67] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 13748–13758. https://proceedings.neurips.cc/paper/2019/hash/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Abstract.html

[68] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 13748–13758. https://proceedings.neurips.cc/paper/2019/hash/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Abstract.html

[69] Kento Tatsuno, Daisuke Miyashita, Taiga Ikeda, Kiyoshi Ishiyama, Kazunari Sumiyoshi, and Jun Deguchi. 2024. AiSAQ: All-in-Storage ANNS with Product Quantization for DRAM-free Information Retrieval. *CoRR* abs/2404.06004 (2024). arXiv:2404.06004 doi:10.48550/ARXIV.2404.06004

[70] Mariano Tepper, Ishwar Singh Bhati, Cecilia Aguerrebere, Mark Hildebrand, and Theodore L. Willke. 2024. LeanVec: Searching vectors faster by making them fit. *Trans. Mach. Learn. Res.* 2024 (2024). https://openreview.net/forum?id=wczqrpOrIc

[71] Mariano Tepper, Ishwar Singh Bhati, Cecilia Aguerrebere, and Ted Willke. 2024. GleanVec: Accelerating vector search with minimalist nonlinear dimensionality reduction. *CoRR* abs/2410.22347 (2024). arXiv:2410.22347 doi:10.48550/ARXIV.2410.22347

[72] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. 2024. Scalable Billion-point Approximate Nearest Neighbor Search Using SmartSSDs. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 1135–1150. https://www.usenix.org/conference/atc24/presentation/tian

[73] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Xuecang Zhang, Junhua Zhu, and Yu Zhang. 2024. FusionANNS: An Efficient CPU/GPU Cooperative Processing Architecture for Billion-scale Approximate Nearest Neighbor Search. *CoRR* abs/2409.16576 (2024). arXiv:2409.16576 doi:10.48550/ARXIV.2409.16576

[74] Nimish Ukey, Zhengyi Yang, Binghao Li, Guangjian Zhang, Yiheng Hu, and Wenjie Zhang. 2023. Survey on Exact kNN Queries over High-Dimensional Data Space. *Sensors* 23, 2 (2023), 629. doi:10.3390/S23020629

[75] Karthik V., Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. 2024. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. *CoRR* abs/2401.11324 (2024). arXiv:2401.11324 doi:10.48550/ARXIV.2401.11324

[76] 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 '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2614–2627. doi:10.1145/3448016.3457550

[77] Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Yifan Zhu, and Wenchao Zhou. 2025. Accelerating Graph Indexing for ANNS on Modern CPUs. *CoRR* abs/2502.18113 (2025). arXiv:2502.18113 doi:10.48550/ARXIV.2502.18113

[78] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proc. ACM Manag. Data* 2, 1 (2024), V2mod014:1–V2mod014:27. doi:10.1145/3639269

[79] Runhui Wang and Dong Deng. 2020. DeltaPQ: Lossless Product Quantization Code Compression for High Dimensional Similarity Search. *Proc. VLDB Endow.* 13, 13 (2020), 3603–3616. doi:10.14778/3424573.3424580

[80] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proc. VLDB Endow.* 13, 12 (2020), 3152–3165. doi:10.14778/3415478.3415541

[81] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul N. Bennett, Junaid Ahmed, and Arnold Overwijk. 2021. Approximate Nearest Neighbor Negative Contrastive Learning for Dense Text Retrieval. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=zeFrfgyZln

[82] Haike Xu, Magdalen Dobson Manohar, Philip A. Bernstein, Badrish Chandramouli, Richard Wen, and Harsha Vardhan Simhadri. 2025. In-Place Updates of a Graph Index for Streaming Approximate Nearest Neighbor Search. *CoRR* abs/2502.13826 (2025). arXiv:2502.13826 doi:10.48550/ARXIV.2502.13826

[83] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 6 (2024), 239:1–239:26. doi:10.1145/3698814

[84] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. ACM, 545–561. doi:10.1145/3600006.3613166

[85] Peiqi Yin, Xiao Yan, Jinjing Zhou, Qiang Fu, Zhenkun Cai, James Cheng, Bo Tang, and Minjie Wang. 2023. DGI: An Easy and Efficient Framework for GNN Model Evaluation. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*. ACM, 5439–5450. doi:10.1145/3580305.3599805

[86] Peiqi Yin, Qihui Zhou, Xiao Yan, Chao Wang, Eric Lo, Changji Li, Lan Lu, Hua Fan, Wenchao Zhou, Ming-Chang Yang, and James Cheng. 2025. CARINA: An Efficient CXL-Oriented Embedding Serving System for Recommendation Models. *Proc. ACM Manag. Data* 3, 3 (2025), 137:1–137:29. doi:10.1145/3725274

[87] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2022. GPU-accelerated Proximity Graph Approximate Nearest Neighbor Search and Construction. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 552–564. doi:10.1109/ICDE53745.2022.00046

[88] Wei Yuan and Xi Jin. 2025. FANNS: An FPGA-Based Approximate Nearest-Neighbor Search Accelerator. *IEEE Trans. Very Large Scale Integr. Syst.* 33, 4 (2025), 1197–1201. doi:10.1109/TVLSI.2024.3496589

[89] Ziyang Yue, Bolong Zheng, Ling Xu, Kanru Xu, Shuhao Zhang, Yajuan Du, Yunjun Gao, Xiaofang Zhou, and Christian S Jensen. 2025. Select Edges Wisely: Monotonic Path Aware Graph Layout Optimization for Disk-Based ANN Search. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4337–4349.

[90] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, and Yu Wang. 2023. DF-GAS: a Distributed FPGA-as-a-Service Architecture towards Billion-Scale Graph-based Approximate Nearest Neighbor Search. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*. ACM, 283–296. doi:10.1145/3613424.3614292

[91] Ximu Zeng, Liwei Deng, Penghao Chen, Xu Chen, Han Su, and Kai Zheng. 2025. LIRA: A Learning-based Query-aware Partition Framework for Large-scale ANN Search. In *Proceedings of the ACM on Web Conference 2025, WWW 2025, Sydney, NSW, Australia, 28 April 2025- 2 May 2025*. ACM, 2729–2741. doi:10.1145/3696410.3714633

[92] Yanhao Zhang, Pan Pan, Yun Zheng, Kang Zhao, Yingya Zhang, Xiaofeng Ren, and Rong Jin. 2018. Visual Search at Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*. ACM, 993–1001. doi:10.1145/3219819.3219820

[93] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards Efficient Index Construction and Approximate Nearest Neighbor Search in High-Dimensional Spaces. *Proc. VLDB Endow.* 16, 8 (2023), 1979–1991. doi:10.14778/3594512.3594527

[94] Chenguang Zheng, Guanxian Jiang, Xiao Yan, Peiqi Yin, Qihui Zhou, and James Cheng. 2024. GE^2: A General and Efficient Knowledge Graph Embedding Learning System. *Proc. ACM Manag. Data* 2, 3 (2024), 183. doi:10.1145/3654986

[95] Qihui Zhou, Peiqi Yin, Xiao Yan, Changji Li, Guanxian Jiang, and James Cheng. 2024. Atom: An Efficient Query Serving System for Embedding-based Knowledge Graph Reasoning with Operator-level Batching. *Proc. ACM Manag. Data* 2, 4 (2024), 193:1–193:29. doi:10.1145/3677129

[96] Yijie Zhou, Shengyuan Lin, Shufeng Gong, Song Yu, Shuhao Fan, Yanfeng Zhang, and Ge Yu. 2025. GoVector: An I/O-Efficient Caching Strategy for High-Dimensional Vector Nearest Neighbor Search. *CoRR* abs/2508.15694 (2025). arXiv:2508.15694 doi:10.48550/ARXIV.2508.15694

[97] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 1 (2024), 69:1–69:26. doi:10.1145/3639324

Received October 2025; revised January 2026; accepted February 2026