APPENDIX

A DISK-BASED GRAPH INDEX METHOD

Graph-based vector search methods are more efficient and accurate than other solutions [21, 32]. Fig. 17 shows how they build a graph index $G = (V, E)$ for the vector dataset. Each data point (i.e., vector) is a vertex in V and each edge in E connects close vertices [42, 57]. A block is the smallest unit for disk I/O operations. The disk stores the adjacency list of each vertex with its vector data, and packs data of multiple vertices into one block. This way, one disk access can read both vector data and neighbor IDs for a vertex. In the example, each block stores data of four vertices. The entire block is loaded into the main memory when accessing data of any vertex in it.

Most graph index methods [20, 42, 44] use a similar search strategy, called the greedy search strategy (we call it as the vertex search strategy in this paper) [21]. Fig. 17 shows an example of finding the nearest vertex to a query vector. The search starts from a random/fixed vertex and traverses the graph. In the example, vertex **12** is the entry point. Its neighbors are visited to measure their distance to the query vector. Then vertex **8** is chosen as the closest visited vertex to the query vector. This process repeats iteratively. In each iteration, it finds a neighbor of the current vertex that is closer to the query vector. In the example, the search visits vertices **8**, **5**, **2**, **15** in order, until it reaches vertex **3** who has no closer neighbors to the query vector than itself. In practice, a priority queue maintains candidate vertices. In each iteration, we pop one vertex from the queue, visit its neighbors, and add the neighbors that may update the current result to the queue. In the example, we set the queue length as 1 for simplicity.

The vertex search strategy has a computational cost of $O(o \cdot \ell)$, where ℓ is the search path length (i.e., the number of hops) and o is the average out-degree of visited vertices. This is because we need to calculate the distance to the query vector for all the vertices in the search path and their neighbors. Let ξ be the vertex utilization ratio and ε be the number of vertices in a disk block. Then $\xi \cdot \varepsilon$ represents the number of vertices accessed in each disk I/O. The I/O complexity is then $O((o \cdot \ell)/(\xi \cdot \varepsilon))$. The current disk-based graph index method only checks the target vertex and discards the rest for a loaded block, so $\xi \cdot \varepsilon$ is 1. The average disk I/O complexity is then $O(o \cdot \ell)$, since each vertex data access requires one disk I/O operation. A recent study [32] reduced the I/O complexity by keeping the compressed vector data of all vertices in memory. It uses compressed data to calculate the approximate distance to the query before accessing a vertex. It skips vertices that are unlikely to be closer to the query without reading data from disk. However, this strategy trades accuracy for efficiency. Many disk I/Os are still needed for high accuracy. Modern storage media have relatively high performance on latency and peak bandwidth, but they are much slower than processors. Therefore, disk I/O operation is the main cost of the disk-based vector search procedure.

B PARAMETERS OF BNF

BNF has two hyper-parameters: maximum iterations β and $OR(G)$ gain threshold τ. They affect the $OR(G)$ of the graph layout. We evaluate their effect below. We set the maximum number of neighbors Λ to 48 and block size η to 4 KB. The experimental dataset is BIGANN, where each vector is 128-dimensional and one byte per

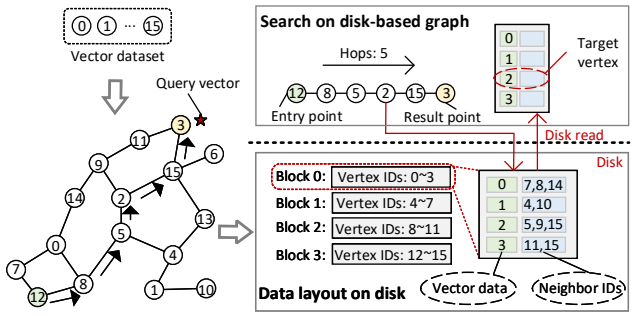


Figure 17: Illustration of a disk-based graph index: logical topology (left), physical data layout (right).

Table 4: ρ and $|E|$ on BIGANN with different volumes.

	BIGANN10K	BIGANN1M	BIGANN10M	BIGANN100M
ρ	834	83,334	833,334	8,333,334
$\lvert E\rvert$	48×10^4	48×10^6	48×10^7	48×10^8

Table 5: $OR(G)$ of graph layout generated by BNF.

$\beta \rightarrow$	4	8	16	32	64
BIGANN10K	0.4763	0.4979	0.5015	0.5048	0.5056
BIGANN1M	0.3312	0.3462	0.3542	0.3587	0.3612
BIGANN10M	0.2951	0.3103	0.3183	0.3230	0.3256
BIGANN100M	0.2640	0.2792	0.2876	-	-

Table 6: Execution time (seconds) of BNF under different β.

$\beta \rightarrow$	4	8	16	32	64
BIGANN10K	0.0158	0.0303	0.0637	0.1393	0.2819
BIGANN1M	2.508	4.850	9.759	20.15	39.31
BIGANN10M	35.68	70.64	140.7	280.2	554.8
BIGANN100M	451.0	887.2	1760.5	-	-

value. So, each vertex takes $\gamma = (128 + 4 + 48 \times 4)/1,024$ KB (ID is unsigned integer type), and a block can hold at most $\varepsilon = 12$ vertices. Tab. 4 shows the number of blocks (ρ) and the number of edges of G ($|E|$) on different datasets. We use 64 threads for block shuffling.

Results. Tab. 5 and 6 show the $OR(G)$ and execution time results on BIGANN with different data volumes. We can see that BNF reaches a stable $OR(G)$ after a few iterations and τ is always 0.01. A smaller τ gives a higher $OR(G)$ but takes more time. A larger τ gives a lower $OR(G)$. So, we use $\tau = 0.01$ by default in other experiments. Also, we observe that the $OR(G)$ increases slowly and the time increases sharply as β increases. $\beta = 8$ is enough to get a high $OR(G)$. So, we use $\beta = 8$ by default. We note that the $OR(G)$ is lower and the time is higher for a larger dataset. In vector databases, each segment has about tens of millions of vectors and BNF is efficient on a data segment. We evaluate the $OR(G)$ of graph layout on a data segment based on BNF in Appendix C.

C BNF EVALUATION ON DATA SEGMENT

In our experiment study, we use BNF for block shuffling by default. We show the $OR(G)$ and execution time of BNF with different iterations β on a data segment in Fig. 18. We use $\beta = 8$ by default. See Appendix B for parameters evaluation.

D BNS ALGORITHM

Algorithm 2 shows the detailed procedure of BNS. First, we obtain the input graph layout based on BNP or BNF. Then, for each u in V, we optimize the blocks where u's neighbors are located by swapping the vertices with the lowest OR in these blocks. This

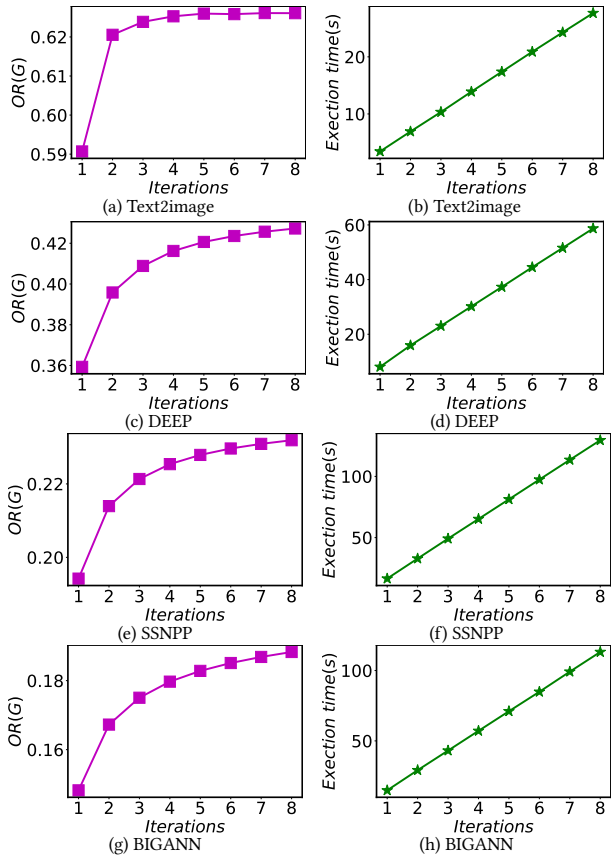


Figure 18: The effect of different iterations (β) on $OR(G)$ and execution time for a data segment on different datasets.

refinement can also be repeated iteratively like BNF (Algorithm 1), to get a better graph layout. Let o be the average out-degree. For each vertex u in V, the number of swaps is o^2. In each swap, the time complexity of computing one block's OR is $O(o \cdot \varepsilon)$, where ε is the number of vertices in a block. Therefore, the time complexity of BNS is $O(\beta \cdot o^3 \cdot \varepsilon \cdot |V|)$ for β iterations, or $O(\beta \cdot o^2 \cdot \varepsilon \cdot |E|)$ ($|E| = o \cdot |V|$).

E BNF VS. BNS

Tab. 7 shows the block shuffling performance of BNF and BNS on BIGANN1M (which contains one million vectors in 128 dimensions). We use 64 threads for block shuffling. The results show that BNS achieves better $OR(G)$ with fewer iterations, but it is much slower than BNF in each iteration. This makes BNS impractical on larger datasets, unless we can speed up BNS. We leave this as future work. In our experiments, we use BNF for block shuffling by default.

F GRAPH PARTITIONING EVALUATION

We evaluate how three existing graph partitioning methods perform on our block shuffling task for disk-based graph index. We summarize the three methods below.

- **GP1 [13].** This method partitions V by hierarchical balanced clustering. It iteratively splits the vertices in a large cluster into smaller clusters until each cluster has a limit number of vertices.

Algorithm 3: Block Shuffling by BNS

Input: block-level graph layout of $G = (V, E)$ returned by BNP or BNF, number of iterations β, $OR(G)$ gain threshold τ
Output: new block-level graph layout of G

while *iterations* $\leq \beta$ **do**
  forall $u \in V$ **do**
    forall $a, e \in N(u)$ *and* $a \neq e$ **do**
      /* vertex that has minimal OR in $B(a)$ */
      $x \leftarrow \arg\min_{x \in B(a)} OR(x)$;
      /* vertex that has minimal OR in $B(e)$ */
      $y \leftarrow \arg\min_{y \in B(e)} OR(y)$;
      /* OR before swapping */
      $OR_{old} \leftarrow OR(B(a)) + OR(B(e))$;
      /* OR after swapping */
      $OR_{new} \leftarrow$ $OR(B(a)) \setminus \{x\} \cup \{y\}) + OR(B(e)) \setminus \{y\} \cup \{x\})$;
      if $OR_{old} < OR_{new}$ **then**
        /* update $B(a)$ and $B(e)$ */
        $B(a) \leftarrow B(a) \setminus \{x\} \cup \{y\}$;
        $B(e) \leftarrow B(e) \setminus \{y\} \cup \{x\}$;
  if $OR(G)$ *gain* $< \tau$ **then**
    break;
return new layout of G

Table 7: BNF vs. BNS on BIGANN1M.

Algorithm↓	β	Execution time (s) for each iteration	$OR(G)$
BNF	64	0.627	0.361
BNS	6	1,877	0.401

Table 8: GP1 vs. BNF on BIGANN10K.

Algorithm↓	β	Total execution time (s)	$OR(G)$
GP1	-	6.8	0.3130
BNF	64	0.2819	0.5056

Table 9: GP1 vs. BNF on BIGANN1M.

Algorithm↓	β	Total execution time (s)	$OR(G)$
GP1	-	192.24	0.2060
BNF	64	39.31	0.3612

Table 10: GP1 vs. BNF on SSNPP10M.

Algorithm↓	β	Total execution time (s)	$OR(G)$
GP1	-	234.0	0.0200
BNF	16	200.3	0.2374

- **GP2 [49].** This method is based on K-way greedy graph growing algorithm (KGGGP). The main idea is a standard greedy approach for bipartitioning. It starts with two random "seed" vertices in the two parts. Then, it adds vertices alternately to the parts, choosing the vertex that minimizes a selected criterion.
- **GP3 [11].** This method is from prioritized restreaming algorithms for balanced graph partitioning. It can partition graphs into balanced subsets, which is important for many distributed computing applications. It also prioritizes the gain order of nodes.

Results. We compare the three graph partitioning methods with BNF using 64 threads to perform all methods. Tab. 8–12 show the efficiency and effectiveness results. GP1 and GP2 are not iterative

Table 11: GP2 vs. BNF on DEEP10M.

Algorithm↓	β	Total execution time (s)	$OR(G)$
GP2	-	300.0	0.3100
BNF	64	306.8	0.4290

Table 12: GP3 vs. BNF on BIGANN10M.

Algorithm↓	β	$OR(G)$
GP3	4	0.2666
GP3	8	0.2801
BNF	4	0.2951
BNF	8	0.3103

Table 13: T_{disk_graph} and $T_{shuffling}$ on four datasets.

Dataset↓	T_{disk_graph} (s)	$T_{shuffling}$ (s)
BIGANN	930	113
DEEP	668	59
SSNPP	1,055	130
Text2image	519	28

Table 14: Search performance and memory overhead under different μ values.

Sample ratio	***Recall***$(k = 10)$	***QPS***	**Memory overhead (MB)**
$\mu = 0.001$	0.9916	3,914	521
$\mu = 0.01$	0.9917	3,991	558
$\mu = 0.1$	0.9919	4,303	918

algorithms, so they do not have β data. We implement GP3 by adding the gain order [11] to BNF, so the time of GP3 is slower than BNF (it is at least equal to BNF). The results show that BNF outperforms the three graph partitioning methods in time and $OR(G)$ over different datasets. Therefore, existing graph partitioning methods are not suitable for our block shuffling task. We analyze the reasons below. Graph partitioning usually targets real-world social networks [48, 59], which have natural clustering and power-law degree distribution [8]. But block shuffling handles proximity graph index built on high-dimensional vectors, where the edge needs both navigation and similarity [20, 41] and follows a uniform degree distribution [57].

G BLOCK SHUFFLING COST

We report the disk-based graph index construction time (T_{disk_graph} in **§6**) and block shuffling time ($T_{shuffling}$ in **§6**) in Tab. 13. From the results, we can see that $T_{shuffling}$ is only 3%∼12% of T_{disk_graph}.

H PARAMETERS OF IN-MEMORY NAVIGATION GRAPH

In in-memory navigation graph, we have parameter of sample ratio μ and parameters of graph index. For graph index parameters, we follow the current proximity graph algorithms [57] and consider the memory limit of a segment. For example, given a sample ratio μ, we adjust the maximum number of neighbors in in-memory navigation graph to keep the memory overhead less than 2 GB on a segment. We analyze the effect of μ on search performance with the same graph index parameters below.

Results. Tab. 14 shows the search performance and memory overhead with different μ values. We can see that the *Recall* and *QPS* increase with μ. This is because more data in memory may provide better entry points that are closer to query vector. These entry points can reduce disk I/Os and increase the chance of finding true nearest vectors. But a larger sample ratio also increases the memory overhead. We need to balance search performance and memory overhead on a segment due to the limited memory capacity.

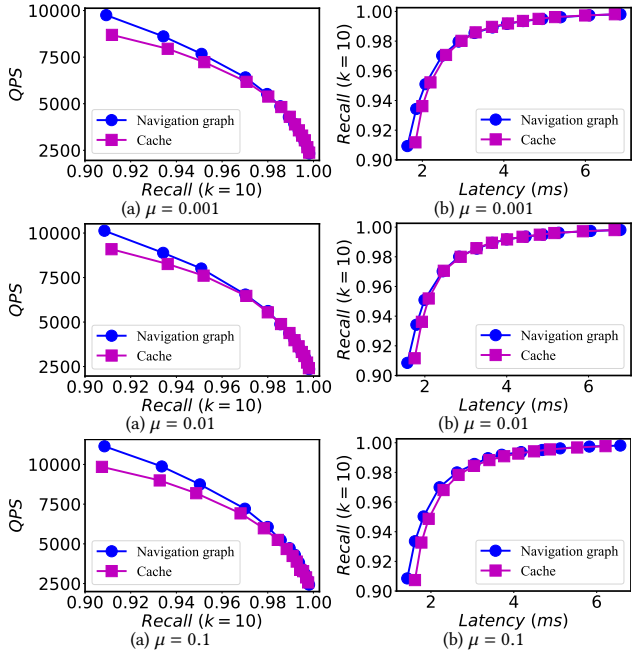


Figure 19: Search performance comparison under different μ values for two data layouts in memory.

Table 15: Memory overhead (MB) under different μ values for two data layouts in memory.

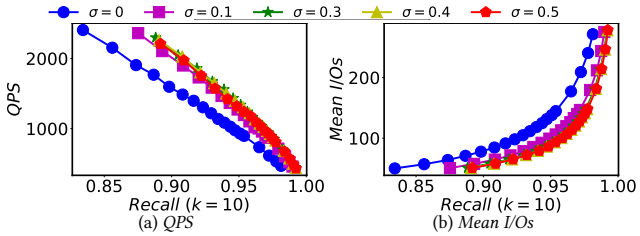
Strategy→	Navigation graph	Cache
$\mu = 0.001$	521	661
$\mu = 0.01$	558	1,039
$\mu = 0.1$	918	5,122

I IN-MEMORY NAVIGATION GRAPH VS. CACHING HOT VERTICES

DiskANN [32] (the baseline framework) caches some hot vertices and their neighbor IDs of disk-based graph in memory. We evaluate the effect of in-memory navigation graph on the BIGANN10M dataset, compared to DiskANN's cache strategy. We implement DiskANN's cache strategy on Starling by replacing the in-memory navigation graph with it. We keep the sample ratio μ of two in-memory data layouts the same. We use 64 threads for the search procedure.

Efficiency and accuracy. Fig. 19 shows the *QPS* vs. *Recall* and *Recall* vs. *Latency* comparison with different sample ratios μ on BIGANN10M. We can see that the in-memory navigation graph is always better than the cache strategy, especially when *Recall* is between 0.90 and 0.97. We also see that when *Recall* is above 0.97, two strategies are similar. The reason is that a higher *Recall* needs more disk I/Os, and the I/Os reduction with the cache data or in-memory navigation graph is small compared to the total I/Os. So, the difference between the cache data and in-memory navigation graph becomes insignificant.

Memory overhead. Tab. 15 shows the memory overheads of two in-memory data layouts with the same *Recall* (*Recall=0.99*). We can see that the in-memory navigation graph has a lower memory



(a) *QPS* (b) *Mean I/Os*

Figure 20: *QPS* and *Mean I/Os* under different pruning ratios.

Table 16: Parameter values of the disk-based graph used in our experiments.

Parameter↓	BIGANN	DEEP	SSNPP	Text2image
Λ	31	48	48	54
L	128	128	128	128
B	0.500	0.256	0.960	0.470
γ	0.250	0.566	0.441	0.996
η	4	4	4	4
ε	16	7	9	4
ρ	2,062,500	1,571,429	1,777,778	1,250,000

Table 17: Parameter values of in-memory navigation graph used in our experiments.

Parameter↓	BIGANN	DEEP	SSNPP	Text2image
Λ	20	10	48	100
L	128	128	128	128
μ	0.09	0.10	0.10	0.09

cost than the cache strategy of DiskANN. This is because our in-memory graph keeps a light-weight graph index built on sample data in memory, but the cache strategy loads the vector data of vertices and their neighbor IDs of disk-based graph (they are always larger than in-memory graph index) into memory. We note that the memory overhead gap between navigation graph and cache strategy is larger as the sample ratio increases.

J PRUNING RATIO FOR BLOCK PRUNING

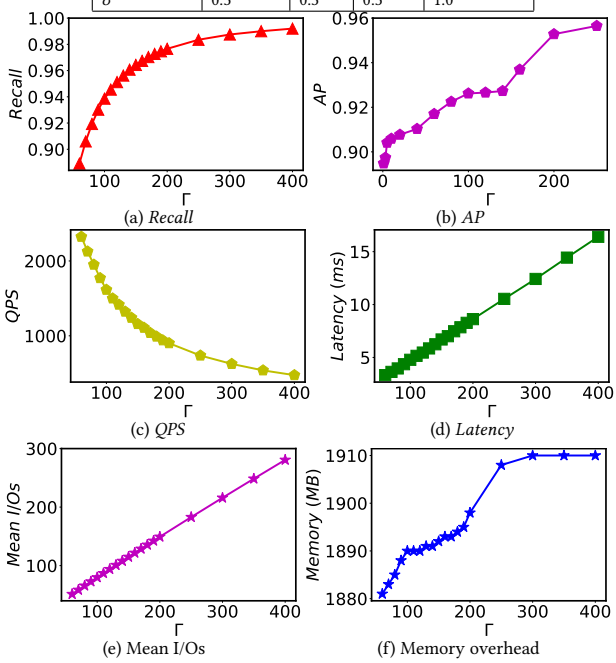
Fig. 20 shows how different pruning ratios σ affect the search performance on BIGANN10M. We use 8 threads to execute a batch of queries. The results show that increasing σ from 0 to 0.3 improves the *QPS* under the same *Recall*. This is because a higher σ allows us to check more neighbors and reach the vertices closer to the query with a high probability. This exploits the data locality and reduces the disk I/Os. When $\sigma = 0$, we only visit the target vertex, which is equivalent to the vertex search strategy of the baseline framework. This does not benefit from the data locality from block shuffling, so it has the worst performance. However, when σ exceeds a threshold (from 0.3 to 0.5), the *QPS* under the same *Recall* declines. This is because a larger σ causes redundant computation. The mean I/Os decrease as σ increases, because visiting more neighbors helps us find a vertex closer to the query and reduce disk I/Os. Therefore, there is an optimal pruning ratio that we can obtain by grid search. For example, the optimal σ for the BIGANN dataset is 0.3. Tab. 18 shows the optimal σ for other datasets.

K PARAMETERS FOR INDEX PROCESS

Please refer to the DiskANN paper [32] for details on disk-based graph index construction. We explain the main parameters for building the index. Λ is the maximum degree of the disk-based graph. A higher Λ leads to larger indexes and longer construction

Table 18: Pruning ratio used in our experiments.

Parameter↓	BIGANN	DEEP	SSNPP	Text2image
σ	0.3	0.3	0.3	1.0



(a) *Recall* (b) *AP* (c) *QPS* (d) *Latency* (e) Mean I/Os (f) Memory overhead

Figure 21: Effect of candidate set size Γ on search performance

Table 19: Memory overhead and search performance under different datasets.

Dataset↓	Method	*Recall*(k = 10)	Memory overhead (MB)	*QPS*
BIGANN	Starling	**0.9920**	**1,910**	**475**
	DiskANN	0.9827	1,925	340
DEEP	Starling	**0.9950**	**1,213**	**504**
	DiskANN	0.9927	1,244	379

time, but also better search accuracy. L is the size of candidate neighbors during index construction. A larger L takes more time to construct but produces indexes with higher *Recall* for the same search efficiency. L should be at least as large as Λ. B is the limit on the memory footprint of PQ short codes. This determines how much we compress the vector data to maintain in memory. A smaller B reduces the memory overhead for the PQ short codes. Tab. 16 shows the parameter values we used in our experiments (see §4.1 for more parameter description). In-memory navigation graph uses the same proximity graph, but it has smaller Λ and L. Tab. 17 provides the parameter values of in-memory navigation graph.

L PARAMETERS FOR SEARCHING

We modify the candidate set size Γ to obtain different *Recall* or *AP* when searching. Fig. 21 shows the effect of Γ on the BIGANN dataset. The results indicate that a larger Γ increases the accuracy but also slows down the search and consumes more memory. This parameter applies to both in-memory navigation graph and disk-based graph. Tab. 18 lists the pruning ratio on different datasets we used in our experiments.

M MEMORY COST AND SEARCH PERFORMANCE

DiskANN keeps PQ short codes, hot vertices and their neighbor IDs in memory. For Starling, we mainly store in-memory navigation graph, PQ short codes, and the mapping of vertex IDs to block IDs. Our evaluation in Appendix I shows that navigation graph uses less memory than caching hot vertices and achieves better search performance. In one segment, the memory limit is 2 GB. We adjust the memory overhead by sample ratio for both Starling and DiskANN (see Appendix H and I). Tab. 19 shows the memory overhead for similar *Recall*. Starling has lower memory overhead and higher *QPS* for higher *Recall*.

N RELATED WORK

Vector search algorithms. High-dimensional vector similarity search (HVSS) is a well-studied topic, and most works aim to find an algorithm that balances efficiency and accuracy by preprocessing data [57]. We can roughly categorize current algorithms into four common groups based on how they process the data: tree-based [16, 43, 45]; quantization-based [9, 28, 33]; hashing-based [25, 30, 39]; and graph-based [20, 21, 44]. Graph-based methods have shown to be very effective for HVSS in many empirical studies [21, 35, 41]. However, existing graph-based algorithms require both raw vector data and graph index to be stored in main memory. This leads to high memory consumption and limits the scalability to hundreds of millions of vectors [13]. Starling builds a disk-based graph index and optimizes the block-level graph layout by block shuffling. It also maintains an in-memory navigation graph based on sampled data. Thus, Starling solves the high memory overhead problem while achieving the state-of-the-art efficiency and accuracy trade-off.

Most of the research has focused on million-scale datasets in main memory. For web-scale vector search scenarios, existing vector search algorithms are limited by the main memory-served indexes [32]. Current solutions include large-scale methods and distributed vector databases.

Large-scale solutions. Large-scale solutions aim to reduce memory consumption from two perspectives: vector quantization and hybrid storage. On one hand, FAISS [1] uses vector quantization techniques (e.g., IVFPQ [33], OPQ [23]) to compress the vector data and store them in memory. However, the compression introduces quantization error, which lowers the search quality. On the other hand, some methods working on disk and heterogeneous memory (HM), such as GRIP [61], DiskANN [32], HM-ANN [51], SPANN [13], BBAnn [27], allocate vector data or index to persistent storage. They use some hardware-oriented optimizations to alleviate the I/O bottleneck and support billion-scale vectors on a single machine. However, as data volume grows, it becomes harder to scale to distributed vector database systems for system features purpose [17]. Moreover, applying current disk-based solutions to the data segment of vector databases causes huge disk overhead or high I/O complexity. Starling solves this problem by optimizing the data layout and search strategy following the state-of-the-art disk-based graph index paradigm.

Vector database systems. Vector database research has gained more interest recently [27, 37, 56]. AnalyticDB-V [58], PASE [60], and VBase [63] extend relational databases to support vector data with the one-size-fits-all strategy. However, those systems require optimization for either transaction or analytical workloads and are not specialized for vector data management. Some purpose-built vector data management systems, such as Vearch [37] and Milvus [27, 56], store and search large-scale vector data efficiently by organizing data with segments for industrial applications. There are also some distributed solutions, e.g., Pyramid [17], LANNS [19], LEQAT [62]. They focus on data segmentation strategies or query dispatching optimizations. They assign the vectors that are close to each other to the same segment and probe only a few segments when searching based on the query dispatching optimizations. However, they ignore the search optimization opportunities within the data segment of vector databases. In contrast, Starling addresses the problem of how to search efficiently in the data segment by optimizing the data layout and search strategy.