Disk-Resident Vector Similarity Search: A Survey

Yitong Song, Huiling Li, Zheng Wu, Lanjing Yi, Bojian Zhu, Xuanhe Zhou, Xin Huang, and Jianliang Xu

***Abstract*—Vector similarity search (VSS) has become ubiquitous in modern multimodal data retrieval systems, supporting a wide range of applications such as document retrieval, music recognition, image search, and code assistants. With the rapid growth of vector datasets and the prohibitive cost of main memory, VSS techniques are increasingly shifting from memory-resident to disk-resident settings. Unlike memory-resident designs, which keep both raw vectors and index structures entirely in memory, disk-resident VSS places raw vectors and fine-grained index structures on secondary storage while retaining only lightweight components in memory. Consequently, disk-resident VSS must explicitly account for data access costs (i.e., I/O) and block-level layouts, leading to index organizations and query execution strategies that are fundamentally different from those used in memory-resident systems.**

This survey provides the first systematic review of disk-resident VSS, a critical yet challenging task for modern large-scale data systems. We introduce a unified framework that categorizes existing techniques into IVF-based, graph-based, and tree-based paradigms, based on the primary index structure used for filtering. For each category, we further decompose the overall design into key technical components, including index construction, block-aware layouts, query execution strategies, and update mechanisms. Moreover, we summarize commonly used datasets to facilitate reproducible benchmarking, and identify open challenges and promising directions for future research.

***Index Terms*—High-dimensional vector, Approximate nearest neighbor search, Disk-resident search**

I. Introduction

Vector Similarity Search (VSS) retrieves the vectors most similar to a given query from large-scale, high-dimensional datasets. It has become a fundamental building block of modern data-intensive applications, including information retrieval [1], music recognition [2], image search [3], and code assistants [4]. More recently, VSS has also emerged as a core component of retrieval-augmented generation (RAG) for large language models (LLMs) [5, 6, 7], where vector databases store billions of embeddings and VSS is used to retrieve semantically relevant knowledge to enhance factuality and mitigate hallucinations during LLM generation.

Early VSS solutions are predominantly memory-resident, assuming that both high-dimensional vectors and their index structures can be fully stored in main memory. Under this assumption, a broad spectrum of in-memory index structures has been developed to accelerate VSS [8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. However, as vector datasets continue to grow and memory remains both limited and expensive, maintaining large collections of vectors together with memory-intensive indexes entirely in memory becomes increasingly impractical [18, 19, 20]. This challenge has motivated a new search paradigm, namely *Disk-Resident VSS*, in which vectors and fine-grained index components are placed on secondary storage (typically SSDs), while only lightweight structures and cached data are retained in memory to guide retrieval.

Different from purely memory-resident approaches, disk-resident VSS methods must explicitly consider hybrid memory–disk storage architectures and block (i.e., page) layouts, as well as the dominant I/O costs that are negligible in memory-only settings. To address these challenges, recent studies have proposed a wide range of techniques, such as balanced data partitioning [19, 21], block-aware index construction [22, 23, 24, 25], locality-aware disk layout [20, 26, 27], and hybrid caching strategies [28, 26, 24]. However, these methods differ substantially in their underlying index paradigms, physical layouts, query execution strategies, and update mechanisms, resulting in a highly heterogeneous design space that is difficult to navigate. This fragmentation motivates the need for a systematic survey that provides a unified view of disk-resident VSS and addresses the following key questions: (1) What types of disk-resident VSS indexes exist, and which application scenarios are they best suited for? (2) How does each category construct its index structures and organize them along with raw vectors across heterogeneous memory–disk storage? (3) Given diverse VSS query tasks (e.g., ANN, range search, and filtered ANN), which tasks does each method support, and how are they implemented in practice? and (4) How are incremental updates and index maintenance handled in disk-resident settings?

Motivated by these questions, this survey presents a systematic and comprehensive review of disk-resident VSS methods, categorized according to a unified taxonomy based on their underlying index structures. The paper is organized in accordance with this taxonomy.

Taxonomy & Organization. In contrast to purely memory-resident designs, which often rely on a single index structure, disk-resident VSS methods typically integrate multiple index structures (e.g., graph with quantization [18, 20], or tree with hashing [43, 44]) to support a two-stage filtering pipeline: one for in-memory coarse filtering and another for disk-resident fine-grained filtering. As illustrated in Fig. 1, based on the primary index structures used for filtering, we categorize existing approaches into three groups: *IVF-based methods* (Section III), *graph-based methods* (Section IV), and *tree-based methods* (Section V). Table I summarizes representative methods from each category, enabling an at-a-glance comparison both across and within the three categories. Given the distinct design goals and architectural choices across these categories, we further decompose each class into different technical components, as detailed below.

- *IVF-Based Methods.* These approaches partition vectors

Yitong Song, Huiling Li, Zheng Wu, Lanjing Yi, Bojian Zhu, Xin Huang, and Jianliang Xu are with Hong Kong Baptist University, Hong Kong SAR, China. Xuanhe Zhou is with Shanghai Jiao Tong University, Shanghai, China. E-mail:{ytsong, cshlli, cszhengwu, csljyi, csbjzhu, xinhuang, xujl}@comp.hkbu.edu.hk, zhouxuanhe@sjtu.edu.cn.

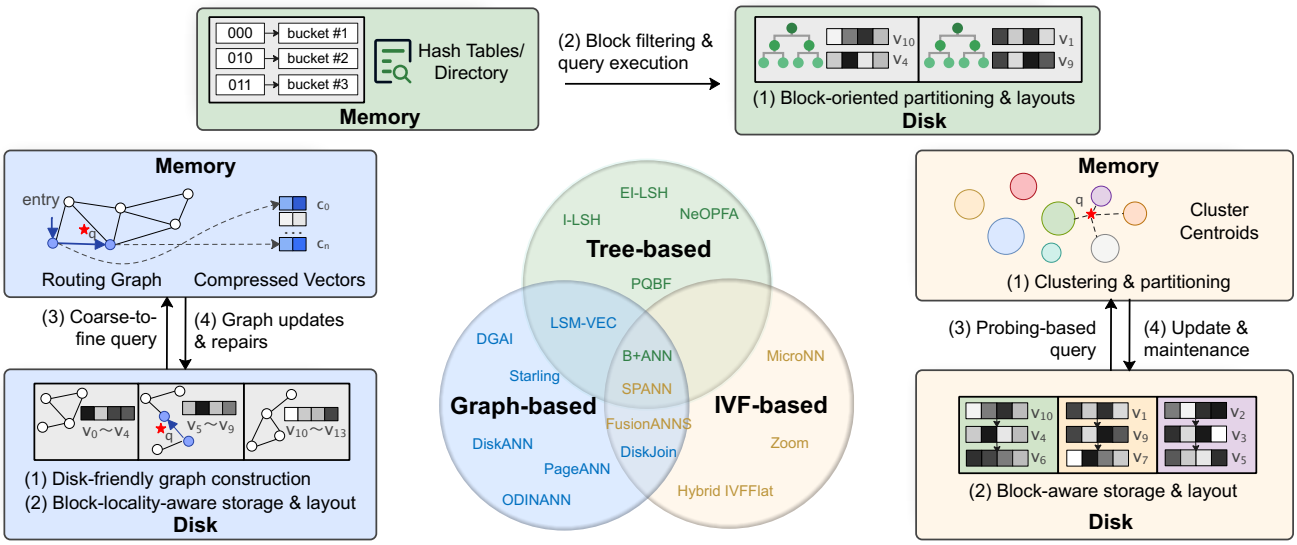


Fig. 1. Illustration of Disk-Resident VSS Methods.

into clusters and restrict query processing to probe only a small number of relevant clusters [19, 21, 29, 30, 31, 32, 48]. This partition-based design easily aligns with block-structured storage, enabling efficient block reads and supporting append-friendly updates. However, repeated insertions within the same clusters may cause cluster imbalance and degrade query performance over time. We review IVF-based methods along four aspects: (1) clustering and partitioning (Section III-A), (2) block-aware storage and layout (Section III-B, (3) probing-based query strategies (Section III-C, and (4) update and maintenance mechanisms (Section III-D).

- *Graph-Based Methods.* These approaches organize vectors into proximity graphs and answer queries by traversing these graphs [18, 33, 35, 34, 20, 25, 36, 41, 37, 42, 39, 38, 40]. However, proximity graphs inherently exhibit highly irregular access patterns, and directly placing the entire graph on disk can incur excessive random I/O during query processing. To address this, existing methods redesign graph structures to be more disk-friendly and adopt block-locality-aware layouts to reduce random accesses. They further rely on memory-resident components, such as lightweight routing graphs and compressed vectors, to perform coarse filtering before accessing fine-grained graph neighbors and raw vectors on disk. Moreover, maintaining graph quality under updates is challenging, as deletions often break connectivity and impair query performance. We survey graph-based techniques along four dimensions: (1) disk-friendly graph construction (Section IV-A), (2) block-locality-aware storage and layout (Section IV-B), (3) coarse-to-fine query processing (Section IV-C), and (4) graph updates and repairs (Section IV-D).
- *Tree-Based Methods.* These methods organize vectors into disk blocks using B$^+$-tree structures [46, 47], which are often combined with hash functions [43, 44] or learned models [45] to map high-dimensional vectors into one-dimensional keys. This block-oriented organization enables queries to efficiently locate a small set of candidate blocks via key lookups or range scans, followed by in-block filtering using distances computed from compressed or full-precision vectors. We summarize tree-based techniques along two aspects: (1) block-oriented partitioning and layouts (Section V-A), and (2) block filtering and query execution (Section V-B).

Comparison & Contributions. While several excellent surveys on approximate nearest neighbor search exist [49, 50, 51, 52, 53], they have predominantly focused on memory-resident algorithms, where I/O costs are not a primary concern. Other surveys on vector databases [54] cover system-level architecture but only briefly touch upon the nuances of disk-based indexing. To our knowledge, no prior work has provided a systematic, in-depth analysis dedicated solely to the challenges and solutions for disk-resident VSS. This survey fills that critical gap by providing a taxonomy and a component-wise analysis of methods designed explicitly for hybrid memory-disk environments. Our contributions are summarized as follows:

- *Dedicated and Systematic Coverage of Disk-Resident VSS.* We present the first survey that focuses exclusively on disk-resident VSS methods, systematically clarifying their query settings, design objectives, and the fundamental I/O challenges that distinguish them from memory-resident approaches.
- *Taxonomy-Driven Analysis of Methods and Design Components.* We classify existing disk-resident VSS methods into IVF-based, graph-based, and tree-based categories, and decompose each category into key technical components, including index construction, block-aware storage layouts, query execution strategies, and update and maintenance mechanisms.
- *Reproducible Benchmarking Resources.* We consolidate commonly used datasets from public sources to enable fair benchmarking of disk-resident VSS methods.

TABLE I
COMPARISON OF REPRESENTATIVE DISK-RESIDENT VSS METHODS.

Type	Method	Key Techniques	Storage		Query Type			Up.-Opt.	Applications
			Memory	Disk	ANN	RS	FANN		
IVF-based	SPANN [19] / SPFresh [21]	Hierarchical balanced clustering / Lightweight rebalancing	IVF centroids + routing graph	IVF lists + vectors	✓	×	×	✓	Dynamic workloads / frequent updates
	MicroNN [29]	On-device lightweight cluster reorganization	IVF centroids	IVF lists + vectors	✓	×	✓	✓	
	Zoom [30]	In-memory approximate search & SSD exact reranking	Com. vectors + IVF centroids + IVF lists	Vectors	✓	×	×	×	
	Hybrid IVFFlat [31]	IVFFlat with advanced filtering	IVF centroids + filter structures	IVF lists + vectors	✓	×	✓	×	
	FusionANNS [32]	CPU–GPU cooperative ANN processing	IVF lists + com. vectors (GPU memory) + routing graph	Vectors	✓	×	×	×	
Graph-based	DiskANN Series [18, 33, 34, 35]	Vamana graph / Query-sensitive entry / Isomorphic layout / Delta-overlay graph	Com. vectors + entry nodes	Graph + vectors	✓	×	✓	✓	Latency-critical, high-accuracy search
	Starling [20]	Segment-based graph partitioning	Com. vectors + routing graph	Vamana + vectors	✓	✓	×	×	
	ODINANN [36]	Direct insert for stable search	Com. vectors + routing graph	Graph + vectors	✓	×	×	✓	
	LSM-VEC [37]	LSM-style multi-level index	Upper layer of HNSW	Bottom layer of HNSW	✓	×	×	✓	
	PageANN [25]	Page-aligned graph	Routing graph	Graph + Com. vectors + vectors	✓	×	×	×	
	DiskJoin [38]	SSD-oriented vector similarity join processing	Bucket centroids + graph	Vectors	×	✓	✓	×	
	BatANN [39]	Distributed query-state baton passing	Com. vectors + head index	Graph + vectors	✓	×	×	×	
	PipeANN [40]	SSD-aligned graph search	Com. vectors + routing graph	Graph + vectors	✓	×	×	×	
	DGAI [41]	Decoupled graph and vector storage	Com. vectors	Graph + vectors	✓	×	×	✓	
	FlashANNS [42]	GPU-driven dependency-relaxed asynchronous I/O	Entry nodes	Graph + vectors	✓	×	×	✓	
Tree-based	I-LSH [43] / EI-LSH [44]	Incremental LSH with aggressive early termination	Hash tables	Multi-B^+-tree + vectors	✓	×	×	×	Low-to-medium dimensions
	NeOPFA [45]	Learned sorted-lists with sequential page access	Hash tables	Sorted lists (indexed by B^+-tree) + vectors	✓	×	×	×	
	PQBF [46]	PQ-based filtering with tree pruning	PQ metadata + directory	Com. vectors + PQB$^+$-tree	✓	×	×	×	
	B+ANN [47]	Blocked B^+-tree with semantic clustering and hybrid traversal	Upper-level B^+-tree nodes + block metadata	Blocked B^+-tree leaves + graph + vectors	✓	×	×	✓	

* RS, FANN, and Up.-Opt. denote range search, filtered ANN search (ANN with attribute predicates), and whether the method offers optimized update support, respectively.

- *Open Challenges and Future Research Directions.* We identify key open challenges and outline several promising research directions for advancing disk-resident VSS.

Roadmap. The remainder of this paper is organized as follows. Section II introduces the core concepts of VSS, including common query tasks and widely used underlying techniques. Section III examines IVF-based methods, highlighting how cluster partitioning, block-aligned layouts, probing strategies, and update mechanisms are designed to improve I/O efficiency. Section IV presents graph-based methods, focusing on disk-friendly graph redesigns, locality-aware layouts, coarse-to-fine searches, and update/repair mechanisms. Section V analyzes tree-based approaches, showing how directory structures and key-based partitioning enable efficient block localization and selective on-disk filtering. Section VI provides an overview of commonly used datasets and their publicly available sources. Section VII discusses open challenges and future research opportunities, and Section VIII concludes the paper.

II. PRELIMINARIES

This section first outlines the core query problems addressed by VSS methods, and then surveys the key techniques employed in disk-resident VSS, along with their respective advantages and limitations.

A. Core Query Types

Disk-resident VSS systems mainly address three fundamental query types: *Approximate Nearest Neighbor* (ANN), *Approximate Range Search* (ARS), and *Filtered Approximate Nearest Neighbor* (FANN). The formal definitions of these query tasks are presented below.

Approximate Nearest Neighbor (ANN). ANN search is the most commonly used query type in VSS systems. Before defining the ANN problem, we first introduce the definition

of k nearest neighbor (kNN) problem, which serves as its foundation. Given a finite set of vectors $\mathcal{D}$, a query vector q, and an integer k, the goal of kNN is to retrieve k vectors in $\mathcal{D}$ that are most similar to q under a similarity (or distance) measure Γ. Formally, kNN returns a subset $\mathcal{R}_{knn} \subseteq \mathcal{D}$ such that $|\mathcal{R}_{knn}| = k$ and for all $v \in \mathcal{R}_{knn}$ and $o \in \mathcal{D} \setminus \mathcal{R}_{knn}$, it holds that $\Gamma(v, q) \leq \Gamma(o, q)$. Common choices for Γ include Euclidean distance, cosine distance, angular distance, and (negative) inner product, among others.

However, in high-dimensional spaces, exact kNN search becomes extremely costly due to the curse of dimensionality [15, 16]. Consequently, recent studies [15, 16, 18, 20, 55] focus on ANN, which relaxes exactness and significantly improves query efficiency. The accuracy of an ANN query is commonly evaluated using the query recall ($Recall@k$), which measures the proportion of true nearest neighbors that are successfully retrieved. Formally,

$$Recall@k = \frac{|\mathcal{R}_{knn} \cap \mathcal{R}_{ann}|}{k},$$

where $\mathcal{R}_{knn}$ and $\mathcal{R}_{ann}$ represent the exact and approximate result sets, respectively.

Approximate Range Search (ARS). ARS is another basic query for VSS systems. Unlike kNN search, which returns a fixed number of neighbors, range search (RS) focuses on identifying all vectors within a specified similarity radius of the query. Mathematically, RS returns a subset $\mathcal{R}_{rs} \subseteq \mathcal{D}$ such that $\forall v \in \mathcal{R}_{rs}, \Gamma(v, q) \leq radius$. Similarly, exact RS is computationally expensive in high-dimensional spaces [20]. Therefore, ARS is proposed to return an approximate subset of in-range vectors, trading completeness for query efficiency. The accuracy of ARS is typically evaluated by the average precision ($AP@e\%$), which measures the proportion of true in-range vectors retrieved. The precision is defined as

$$AP@e\% = \frac{|\mathcal{R}_{ars}|}{|\mathcal{R}_{rs}|},$$

where $\mathcal{R}_{rs}$ and $\mathcal{R}_{ars}$ represent the exact and approximate result sets, respectively, and $e\%$ indicates the percentage of vectors that fall within the query range.

Filtered Approximate Nearest Neighbor (FANN). FANN search extends the ANN search by incorporating attribute-based filtering. Each data object consists of a high-dimensional vector component and an associated scalar attribute (e.g., label, timestamp, and price). The exact FANN query (denoted as FNN) aims to retrieve k objects whose attributes satisfy a given filter condition and whose vectors are most similar to the query vector. Formally, let $\mathcal{D}$ be a dataset of n objects $\{o_1, o_2, \ldots, o_n\}$, where each object o has a scalar attribute $o.a$ and a vector attribute $o.v$. Given a query $Q = (q, \mathcal{A}, k)$ consisting of a query vector q, an attribute constraint set $\mathcal{A}$, and an integer k, FNN returns a subset $\mathcal{R}_{fnn}$ of $\mathcal{D}$ satisfying: (1) $\mathcal{R}_{fnn} \subset \mathcal{R}_a$, where $\mathcal{R}_a = \{o | o \in \mathcal{D}, o.a \in \mathcal{A}\}$, and (2) $\forall o \in \mathcal{R}_{fnn}, \forall u \in \mathcal{R}_a \setminus \mathcal{R}_{fnn}, \Gamma(o.v, q) < \Gamma(u.v, q)$. Note that If $|\mathcal{R}_a| < k$, $\mathcal{R}_{fnn}$ contains all data objects in $\mathcal{R}_a$. FANN returns an approximate result set $\mathcal{R}_{fann}$ that trades query accuracy for query efficiency. Its accuracy is commonly measured using the query recall, defined as

$$Recall@k = \frac{|\mathcal{R}_{fnn} \cap \mathcal{R}_{fann}|}{\min(k, |\mathcal{R}_{fnn}|)}.$$

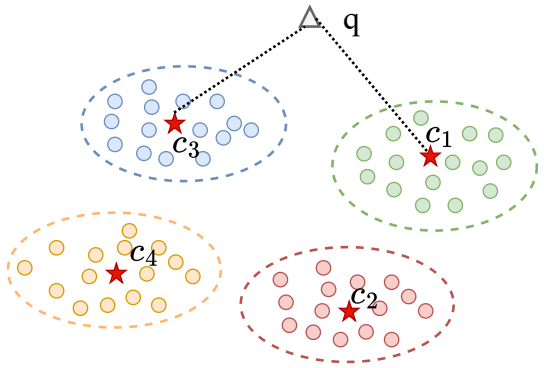


Fig. 2. The IVF Index.

B. Underlying Techniques

Disk-resident VSS methods accelerate queries through a variety of indexing techniques, most notably *Inverted File* (IVF), *Quantization*, *Proximity Graphs* (PG), and *Locality-Sensitive Hashing* (LSH). In the following, we review these techniques and discuss their respective advantages and limitations.

Inverted File (IVF). As illustrated in Fig. 2, the IVF technique partitions the entire vector dataset via clustering, and then maintains an inverted list (also known as posting list) for each cluster to record the vectors assigned to it. Common clustering methods include k-means [56] and its variants [57, 58, 59] that address issues such as partition imbalance and stability.

During ANN query processing, the query vector q is first compared to all cluster centroids to identify the $nprobe$ nearest clusters. Only the vectors within these selected clusters are then scanned to compute their distances to q. The k vectors with the smallest distances are returned as the final result. The parameter $nprobe$ controls the search granularity: a smaller value of $nprobe$ reduces the scan size but may impair query accuracy, while increasing $nprobe$ improves accuracy at the cost of query latency.

Moreover, IVF structures are often extended into *hierarchical* or *multi-level* forms, where each coarse cluster is further partitioned into multiple finer-grained sub-clusters [19, 60]. Rather than relying on a single flat clustering layer, these designs introduce a coarse-to-fine clustering hierarchy that progressively narrows the candidate set before accessing disk-resident data blocks.

Advantages and Limitations. In general, IVF provides only coarse pruning, since all vectors within the probed clusters must still be scanned, and thus its query efficiency is typically lower than other methods offering finer-grained search (e.g., graph-based methods). Nevertheless, IVF remains one of the most widely adopted index structures in disk-resident VSS methods due to its strong compatibility with disk access patterns. By grouping similar vectors into the same cluster,

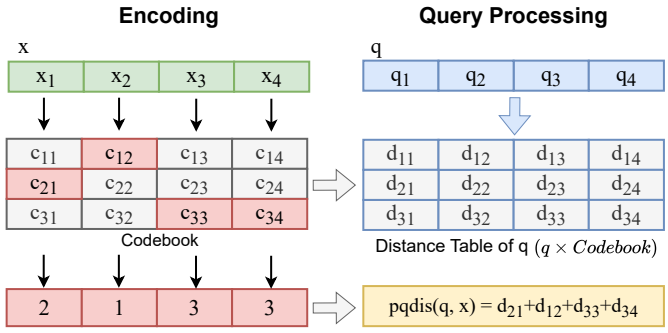


Fig. 3. Product Quantization (PQ).

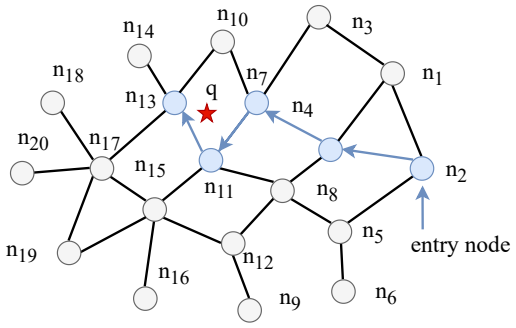


Fig. 4. Proximity Graph (PG).

IVF improves I/O locality: vectors that are likely to be examined together are stored in shared storage blocks, allowing the system to retrieve them with only a small number of block reads rather than multiple scattered lookups. In addition, the inverted list structure naturally supports *dynamic data maintenance*. Insertions can be handled by simply appending vectors to cluster lists, and deletions require only lightweight list updates, with minimal impact on query accuracy. Over time, however, repeated updates in the same cluster may cause cluster imbalance and degrade query efficiency. Consequently, recent work focuses on *adaptive partitioning* or *incremental re-clustering* strategies to maintain balanced clusters and stable query efficiency.

Quantization. Quantization is a fundamental technique for vector compression in disk-resident VSS methods. It encodes high-dimensional vectors into compact representations that can be stored in memory. Using these compact codes, the system can approximate distances to the query vector without accessing the raw vectors, allowing data vectors that are evidently dissimilar to be filtered out before any disk I/O occurs. This significantly reduces the amount of data that must be read from storage and thus lowers query latency.

Existing quantization approaches can be broadly categorized into vector quantization (VQ) [61], scalar quantization (SQ) [62, 63, 13], and product quantization (PQ) [64, 12]. Among these, PQ has become the dominant choice in disk-resident VSS, as it provides a favorable balance between compression efficiency, query accuracy, and distance computation cost. We therefore describe PQ in detail below.

As shown in Fig. 3, PQ first partitions the original d-dimensional space into m subspaces, and learns a codebook by clustering the sub-vectors in each subspace into C groups and storing their centroids. Each vector x is then encoded as an m-dimensional code, where the i-th entry (an integer from 1 to C) indicates which centroid in the i-th subspace best represents the corresponding sub-vector.

During query processing, the query vector q is similarly partitioned, and a distance lookup table is constructed by computing the distances between each sub-vector of q and the C centroids in its corresponding subspace. The distance between q and any encoded vector is then estimated by summing the m precomputed lookup values associated with its code. Finally, the k vectors with the smallest estimated distances are returned as the query results.

Advantages and Limitations. Compared to directly computing full d-dimensional distances, PQ substantially accelerates query processing by reducing costly distance computations to lightweight table lookups. However, quantization inevitably introduces approximation error, which can lead to noticeable recall degradation. In addition, quantization only speeds up distance computation and do not reduce the number of vectors examined during search. Therefore, it is commonly combined with other index structures (e.g., IVF and graph) that provide search-space pruning. Typical hybrid designs include IVFPQ [64], HNSWPQ [65, 66, 27], IVFRaBitQ [13], among others. These methods typically use IVF or graph-based traversal to first narrow the candidate set, and then apply quantization-based distance estimation to efficiently evaluate the remaining vectors.

Proximity Graph (PG). As illustrated in Fig. 4, PG organizes vectors as nodes in a graph, where each node is connected to a selected set of nearby vectors. Rather than simply linking each node to its k nearest neighbors, PG selects neighbors from diverse directions within the local neighborhood, which helps avoid overly dense cliques and improves reachability during graph traversal.

To construct a PG, a heuristic two-stage procedure is commonly used. For each node, the algorithm first identifies a larger pool of candidate neighbors (controlled by the parameter $efConstruction$), and then prunes them to retain at most m edges. The pruning strategy prefers neighbors that provide complementary directional coverage, removing those that lie in similar directions to already selected neighbors.

During query processing, the search follows a best-first traversal strategy while maintaining a dynamic candidate set $\mathcal{S}$ of size ef ($ef \geq k$). The search is initialized from an entry node and iteratively expands the closest unvisited node in $\mathcal{S}$. Upon expansion, the neighbors of the selected node are evaluated and inserted into $\mathcal{S}$, after which $\mathcal{S}$ is updated to retain only the ef closest candidates to the query q. The process continues until no closer candidate can be found. Finally, the top-k nodes in $\mathcal{S}$ are returned as the query result. The parameter ef controls the accuracy-efficiency tradeoff: larger ef generally improves recall but increases search cost.

Advantages and Limitations. Thanks to the elaborated graph structures, PG-based methods generally achieve the best trade-off between query efficiency and accuracy among all types of VSS methods [49, 51]: it can reach high query accuracy with

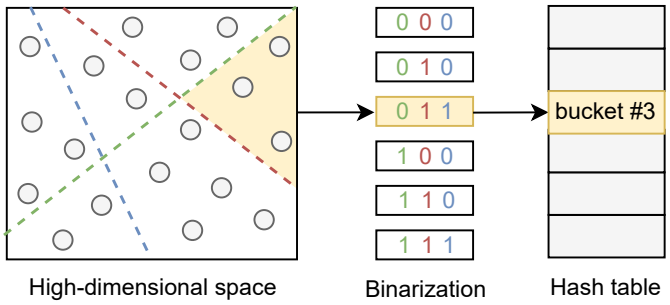


Fig. 5. Locality-Sensitive Hashing (LSH).

relatively few distance calculations. However, PG structure is memory-intensive, which limits scalability when the dataset grows. When the graph is placed on disk, its inherently random access pattern introduces a large number of random I/O operations, significantly degrading performance. In addition, graph-based indexes are inherently less friendly to dynamic updates. Insertions require costly neighbor re-evaluation, and deletions may degrade graph quality if the graph is not properly repaired, making efficient online maintenance an open challenge.

Locality-Sensitive Hashing (LSH). As illustrated in Fig. 5, LSH partitions the high-dimensional vector space into multiple sub-regions using hash functions, where each sub-region corresponds to a hash bucket identified by a hash code. Each vector is mapped to a bucket according to the region in which it falls. These hash functions are designed to preserve locality, such that vectors close in the original metric space are likely to be mapped into the same bucket. Formally, given a metric space with distance function Γ, a hash function h is said to be (r_1, r_2, p_1, p_2)-sensitive if for any two data points x and y:

- If $\Gamma(x, y) \leq r_1$, then $\Pr[h(x) = h(y)] \geq p_1$,
- If $\Gamma(x, y) \geq r_2$, then $\Pr[h(x) = h(y)] \leq p_2$.

During query processing, the same hashing procedure is applied to the query vector, and the search is restricted to buckets with hash codes close to that of the query, typically measured by Hamming distance [67].

Advantages and Limitations. LSH-series methods offer strong theoretical guarantees and can retrieve approximate nearest neighbors within a factor of $(1+\epsilon)$ of the true nearest neighbor distance with high probability. However, achieving high query accuracy typically requires long hash codes, which leads to an exponential increase in the number of buckets. This results in highly sparse bucket occupancy and many empty buckets due to the curse of dimensionality, significantly reducing search efficiency. Consequently, hash-based methods often struggle to maintain both scalability and accuracy in large-scale, high-dimensional search tasks.

III. IVF-Based Methods

This section introduces IVF-based disk-resident VSS methods, whose query processing workflow is illustrated in Fig. 6. These methods process a VSS query by first identifying a small set of nearest clusters, then retrieving the corresponding vectors from page-aligned secondary storage, and finally computing exact distances to produce the final results. We review existing techniques from four key aspects: (1) *Clustering and Partitioning* (Section III-A); (2) *Block-Aware Storage and Layout* (Section III-B); (3) *Probing-Based Query Strategies* (Section III-C); and (4) *Update and Maintenance Mechanisms* (Section III-D).

A. Clustering and Partitioning

A standard IVF construction trains cluster centroids and assigns each vector to a corresponding inverted list. For billion-scale datasets, this process suffers from several major limitations: (1) centroid training and vector assignment can be expensive under constrained memory budgets; (2) classical clustering methods often produce highly uneven list sizes, leading to imbalanced disk I/O and increased tail latency; and (3) assigning each vector to a single cluster may result in recall loss, since queries typically probe only a limited number of inverted lists. To address these limitations, we subsequently review memory-efficient clustering techniques, balanced partitioning strategies, and boundary vector replication mechanisms, corresponding to the above challenges.

Memory-Efficient Clustering. Standard k-means clustering [68] typically requires multiple passes over the dataset and assumes that all vectors can be held in memory during training. This assumption often breaks down at the billion scale, where memory capacity becomes a critical bottleneck. For example, SPFresh [21] reports substantial resource overhead for global rebuilds of IVF index structures, illustrating why full reconstructions are impractical in large-scale settings. To build under tighter memory budgets, MicroNN [29] and Hybrid IVFFlat [31] adopt *mini-batch k-means* [69], which updates centroids using small batches streamed from disk. This approach significantly reduces peak memory consumption and enables index construction on modest machines. However, mini-batch training may slightly degrade clustering quality. Therefore, systems typically compensate by employing more aggressive probing strategies or additional query-time filtering.

Balanced Partitioning. Classical k-means minimizes quantization error but does not control cluster sizes. When stored on disk, this can create a few oversized inverted lists that dominate query latency. A common solution is to add balance constraints or size penalties during assignment, e.g.,

$$\min_{C,P} \sum_{x \in X} \|x - c_{P(x)}\|^2 \; + \; \lambda \sum_{i=1}^{N_{\text{list}}} \Big(|X_i| - |X|/N_{\text{list}} \Big)^2,$$

where $|X_i|$ is the length of the inverted list, and the second term discourages overly large lists. SPANN [19] is a representative design that explicitly targets the imbalanced clusters on disk by producing many small inverted lists with tightly controlled sizes (via hierarchical balanced clustering). MicroNN [29] follows the same goal but integrates balance awareness into a lightweight clustering pipeline that works well with mini-batch training. For dynamic workloads, SPFresh [21] maintains a balanced SPANN-style partitioning under inserts/deletes by splitting oversized lists, merging underfull ones, and then locally reassigning nearby vectors to restore nearest-partition assignment. In contrast, Zoom [30] and Hybrid IVFFlat [31] use the standard k-means algorithm

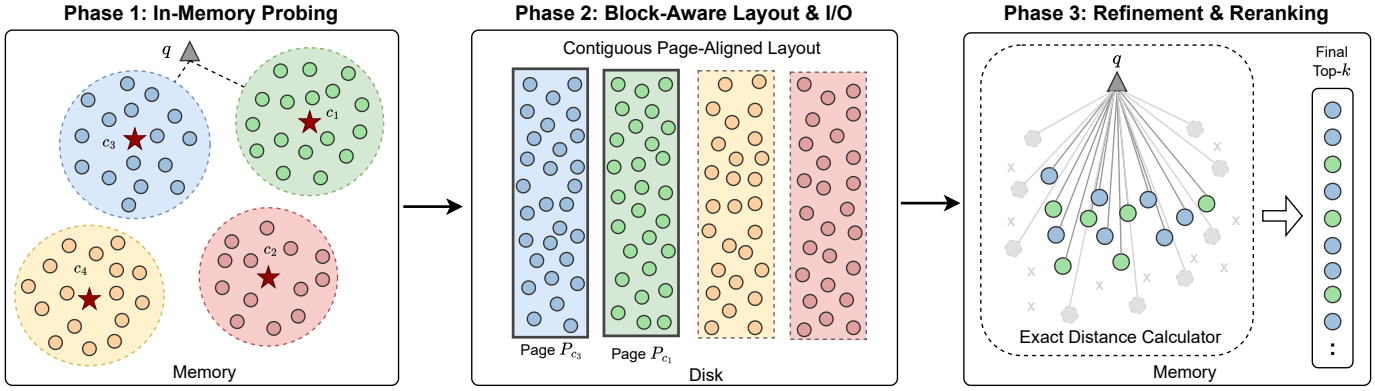


Fig. 6. Query Workflow of IVF-Based Methods.

without explicit balance constraints. In practice, they tune the number of clusters (N_{list}) and probing budgets to trade off scan cost and accuracy.

Boundary Vector Replication. When only a small number of clusters are probed, true nearest neighbors may fall just across cluster boundaries and be excluded from the search, leading to recall loss. Boundary vector replication offers an effective mechanism to mitigate this issue. SPANN [19] implements this idea via *closure assignment*, where a vector may be stored in multiple nearby inverted lists if it is nearly equidistant to several centroids. A typical replication rule is:

$$d(x, c_{i_j}) \leq (1+\varepsilon)\, d(x, c_{i_1}), \quad j = 1, \ldots, R_{\max},$$

which ensures that only boundary vectors are duplicated, subject to a small replication cap. While this strategy increases index size, it significantly improves query recall under a fixed probing budget. FusionANNS [32] adopts a similar boundary replication criterion during clustering, but subsequently modifies which data are actually placed on disk. In contrast, MicroNN [29], Zoom [30], and the Hybrid IVFFlat [31] typically rely on probing more centroids (i.e., increasing n_{probe}) to recover recall, which may increase scan cost.

B. Block-Aware Storage and Layout

An IVF-based disk-resident VSS method typically maintains three core components: cluster centroids, inverted lists, and raw vectors. Most existing approaches keep cluster centroids in memory while placing inverted lists and raw vectors on disk. A smaller body of work goes one step further by also retaining inverted lists (recording vector IDs) and other lightweight index structures (e.g., PQ codes) in memory, leaving only raw vectors on disk. To efficiently support FANN queries, prior work adopts hybrid designs for inverted-list storage and data layout. Below, we review these storage and layout strategies in detail.

Centroids in Memory, Inverted Lists and Raw Vectors on Disk. SPANN [19] keeps all cluster centroids in memory and builds a lightweight in-memory routing structure (i.e., SPTAG) over the centroids for efficient cluster identification, while storing inverted lists and raw vectors on SSD as contiguous arrays. SPFresh [21] preserves this "centroids-in-memory / lists-on-SSD" design and augments it with lightweight metadata to support efficient in-place updates. MicroNN [29] similarly retains only IVF centroids in memory, but stores vectors in a relational store (SQLite) using a clustered key (e.g., by partition ID), such that scanning an inverted list translates into a near-sequential range scan over the underlying file.

Centroids and Inverted Lists in Memory, Raw Vectors on Disk. Zoom [30] keeps all cluster centroids and inverted lists in memory, while storing raw vectors on disk. In addition to these three components, it maintains PQ codes for all vectors in memory as a compact "preview" index. During query processing, the preview stage scans PQ codes in memory to generate a small candidate set, after which only a limited number of random SSD reads are issued to fetch full vectors for exact reranking. FusionANNS [32] extends this design with a *CPU–GPU–SSD* split. It stores the centroid routing graph and per-list vector IDs in CPU memory, pins PQ-compressed vectors in GPU HBM, and places only raw vectors on SSD for final reranking. Crucially, FusionANNS avoids transferring full inverted lists across PCIe: after the CPU selects candidate lists, it sends only vector IDs to the GPU for PQ-based filtering, and retrieves a small number of raw vectors from SSD for exact distance computation.

Hybrid Lists with Filtering Attributes. Hybrid IVFFlat [31] stores both raw vectors and their associated filtering attributes as unified "hybrid vectors" within each inverted list, which are maintained on SSD. During query processing, candidate lists are first selected via centroid-based pruning, after which attribute predicates are applied while scanning the lists. This design is particularly effective for FANN queries, as it preserves the locality of both vector data and filtering attributes within the same disk blocks.

C. Probing-Based Query Strategies

IVF-based disk-resident VSS methods process queries by probing only a small subset of clusters on disk. Efficiently routing queries to relevant clusters, reducing unnecessary data accesses, and minimizing repeated I/O across queries are critical to search performance. In the following, we review techniques for cluster probing and dynamic pruning, as well as caching and batching strategies, respectively.

Cluster Probing and Dynamic Pruning. A basic IVF implementation compares the query against all cluster centroids to identify the n_{probe} nearest clusters for probing. To reduce this routing overhead, SPANN [19] builds a lightweight in-memory graph over centroids to accelerate nearest-centroid search. Zoom [30] similarly replaces exhaustive centroid scans with an in-memory HNSW-based routing layer. FusionANNS [32] also employs a centroid graph for fast list selection, but further relies on GPU-side PQ filtering to keep the subsequent SSD access lightweight.

Beyond routing acceleration, SPANN [19] introduces a *query-aware dynamic pruning* rule to reduce the amount of data read from SSD during probing. After identifying the closest K centroids, an inverted list is scanned only if its centroid is sufficiently close to the nearest one:

$$q \xrightarrow{\text{scan}} X_{ij} \iff \text{Dist}(q, c_{ij}) \le (1+\varepsilon)\,\text{Dist}(q, c_{i1}),$$
$$\text{Dist}(q, c_{i1}) \le \cdots \le \text{Dist}(q, c_{iK}).$$

This pruning criterion avoids unnecessary inverted-list reads for "easy" queries while preserving high recall. In contrast, OrchANN [70] applies traditional triangle-inequality–based pruning at the vector level to reduce the number of raw vectors read from disk.

Caching and Batching. To reduce repeated and unnecessary I/O, disk-resident IVF methods commonly adopt caching strategies and batching optimizations. MicroNN [29] largely delegates caching to the database buffer cache, making its performance sensitive to cache warmness; it also supports a batch execution mode that groups queries by partition and scans each partition once, thereby amortizing both I/O and distance computation. Hybrid IVFFlat [31] treats frequently accessed lists as an on-demand cache, thereby avoiding repeated disk accesses to hot centroids and inverted lists. Zoom [30] reduces SSD reads by shifting most candidate generation to an in-memory preview stage: PQ codes are scanned in memory to produce a small candidate set, and only these candidates are fetched from SSD for exact reranking. FusionANNS [32] follows a similar idea by combining GPU-side PQ filtering with a lightweight reranking stage on SSD. It also reduces the reranking cost through two techniques: (1) performing reranking in mini-batches with early termination once the top-k results stabilize, and (2) mitigating read amplification by deduplicating and merging I/Os targeting the same SSD pages and by exploiting the memory buffer across mini-batches.

D. Update and Maintenance Mechanisms

In real-world applications, vector datasets are highly dynamic, with frequent insertions and deletions. For IVF indexes, updates are naturally supported by simply appending or removing vectors from the cluster lists. However, as updates accumulate, inverted lists may become imbalanced and cluster centroids may drift, which can degrade both routing accuracy and I/O efficiency. Accordingly, existing methods address this challenge from two complementary aspects: efficient update support and incremental rebalancing mechanisms.

Update Support Strategies. Several disk-resident IVF methods explicitly address update efficiency at the storage layer. MicroNN [29] leverages the transactional capabilities of its underlying relational database to support atomic inserts and deletes, delegating concurrency control and crash recovery to the storage engine. SPFresh [21] enables in-place updates on SSD by maintaining versioned metadata for inverted lists, allowing list contents to be modified without rewriting the entire index.

Incremental Rebalancing. Beyond storage-level update support, SPFresh [21] further addresses structural degradation caused by skewed updates through the *Lightweight Incremental RE-balancing (LIRE)* protocol. LIRE monitors inverted list sizes and performs local reorganization when imbalance occurs: oversized lists are split, undersized lists are merged, and affected vectors are locally reassigned to restore nearest-partition consistency.

IV. Graph-Based Methods

In contrast to IVF-based approaches, whose query procedures naturally align with block-oriented access patterns and whose index structures (excluding raw vectors) are update-friendly and lightweight enough to be fully resident in memory, graph-based methods exhibit three fundamental characteristics: (1) they incur irregular, random access to individual vectors during graph traversal, resulting in poor I/O locality; (2) they impose substantially higher memory overhead due to their complex index structures, making it difficult to keep the entire index resident in memory; and (3) under dynamic workloads, maintaining the quality of graph structures is challenging, as deletions may break graph connectivity and degrade query performance.

To mitigate these limitations, existing graph-based disk-resident VSS methods typically integrate quantization techniques (e.g., PQ). As shown in Fig. 7, a lightweight routing graph together with compressed vectors (e.g., PQ codes) is maintained in memory, while raw vectors and fine-grained graph structure is stored on disk. They either redesign the graph to be more disk-friendly or preserve the topology and optimize the physical layout of graph and vectors to improve locality. Query processing generally follows a coarse-to-fine strategy: an in-memory routing graph and PQ codes drive coarse search, and only a small number of relevant disk blocks are accessed for refinement. To address update-related challenges, several graph repair and maintenance mechanisms have also been proposed.

In this section, we review graph-based disk-resident VSS methods along four dimensions: (1) *disk-friendly graph construction* (Section IV-A), (2) *block-locality-aware storage and layout* (Section IV-B), (3) *coarse-to-fine query processing* (Section IV-C), and (4) *graph updates and repair mechanisms* (Section IV-D).

A. Disk-Friendly Graph Construction

This section introduces graph structures specifically designed for disk-resident settings and discusses construction under limited memory.

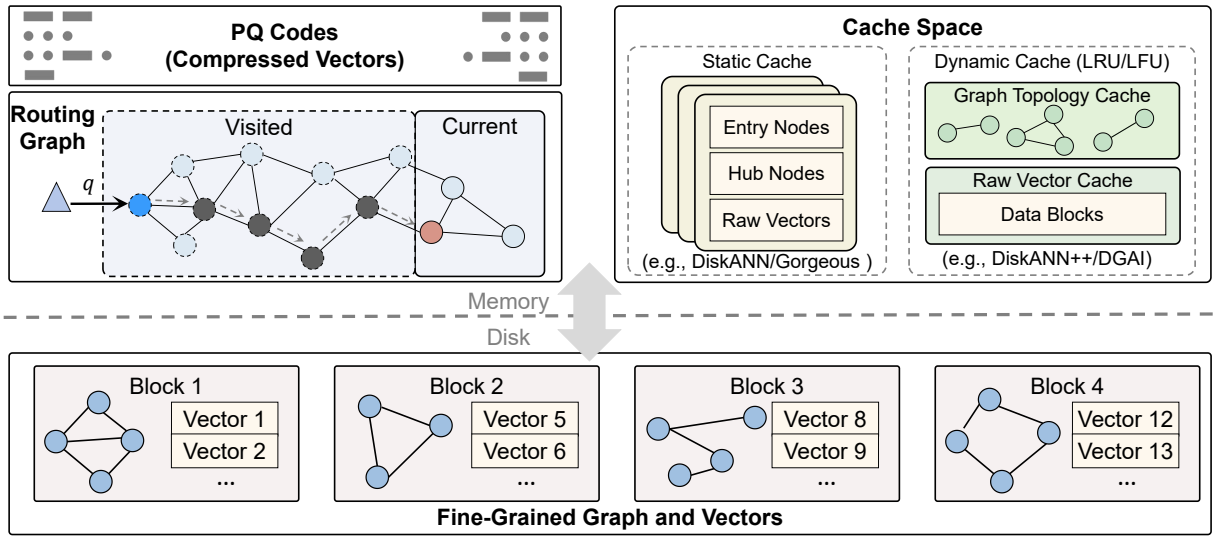


Fig. 7. Illustration of Graph-based Methods.

Disk-Based Graph Structures. We review four graph structures specifically designed for disk-resident VSS, including Vamana [18], XN-Graph [24], BAMG [22], and PageANN [25].

- *Vamana* [18] is a widely used graph index for disk-based VSS. It constructs a sparse proximity graph while using a pruning rule with a tunable parameter $\alpha > 1$ that controls the pruning process by retaining only those neighbors whose inclusion results in a sufficiently large improvement in distance to the query point. To support efficient disk access, Vamana enforces a fixed out-degree so that it can be aligned with the block size when stored on disk. Compared with in-memory graphs like NSG [16] and HNSW [15], these design choices often lead to fewer traversal hops and disk round trips, making Vamana well suited for disk-based search.
- *XN-Graph* [24] is a disk-oriented graph index inspired by NN-Descent [71]. It iteratively collects diversified intermediate neighbors from a wider coverage of the data space, rather than relying on neighbors encountered along traverse paths in prior methods [18, 16]. This design helps the graph provide multiple alternative paths during search.
- *BAMG* [22] proposes a block-aware edge selection strategy that differentiates between intra-block and inter-block connections. Unlike Vamana, BAMG focuses on optimizing the per disk access progress, preferentially retaining edges that improve local navigability without incurring additional I/O operations.
- *PageANN* [25] groups multiple similar vectors into a single unit that fits within one disk page, referred to as a *page node*. Search is then performed over these page nodes, so that each traversal step corresponds to reading a disk page rather than accessing individual vectors. For graph construction, PageANN derives links among page nodes from a high-quality graph index (e.g., Vamana), while eliminating redundant edges and removing links among nodes within the same node.

Partition-Then-Merge Construction. When the entire dataset cannot fit in memory, constructing a graph index becomes challenging, as graph construction typically requires repeated neighbor comparisons and sufficient in-memory context. A widely adopted solution is the *partition-then-merge* paradigm, which avoids holding the full dataset in memory by dividing it into smaller partitions and constructing the graph incrementally. Under this paradigm, the dataset is first partitioned into multiple subsets that can fit into memory. A graph is then constructed independently within each partition, and the resulting subgraphs are finally merged into a single global graph. DiskANN [18] is a representative example of this approach. It partitions the dataset using k-means [68] clustering and assigns each vector to multiple nearby clusters (e.g., two) to preserve cross-partition connectivity. Within each partition, a Vamana graph is constructed entirely in memory. Afterward, the subgraphs are merged by taking the union of their edges to form the final index.

The partition-then-merge paradigm enables graph construction at billion scale under limited memory budgets. However, the quality of the final graph depends on the effectiveness of both partitioning and merging, and suboptimal connections may arise near partition boundaries.

B. Block-Locality-Aware Storage and Layout

Existing methods differ significantly in their storage modalities and physical layout strategies. How graph structures and raw vectors are placed and organized directly affects access patterns, cache utilization, and I/O behavior, and ultimately determines query efficiency. In this section, we review memory–disk hybrid storage strategies and block-locality-aware layout designs adopted by existing methods.

Memory-Disk Hybrid Storage Strategies. Graph-based disk-resident VSS methods typically involve three types of data to be stored: compressed vectors (e.g., PQ codes), full graph structures, and raw vectors, while the routing graph is maintained selectively. Depending on whether available memory is

sufficient to hold PQ codes for all vectors, existing methods can be broadly categorized into two storage regimes:

- *All-in-Disk*, where PQ codes, raw vectors, and full graph structures are all stored on disk [72, 73];
- *Major-in-Disk*, where raw vectors and graph structures reside on disk with PQ codes in memory. This is the predominant design adopted by most existing methods [18, 33, 35, 34, 20, 24, 74].

The former minimizes memory consumption at the cost of increased disk I/O, whereas the latter reduces I/O overhead by trading for a larger memory footprint. In the following, we review these two types of approaches in turn.

(1) *All-in-Disk Methods.* All-in-Disk approaches reduce memory requirements by persisting all major data structures on disk, retaining in memory only minimal state for routing (e.g., lightweight routing structures) and I/O coordination (e.g., buffering). Representative methods include LM-DiskANN [73], which maintains only small routing structures and metadata in memory. Similarly, AiSAQ [72] stores only a tiny amount of fixed metadata, PQ codebooks, and optionally the PQ codes of a few entry points in memory. During search, each hop reads an SSD page containing neighbor identifiers and their PQ codes. The traversal decisions are guided by PQ-based distance estimates computed from these in-chunk codes, while exact distance evaluation is deferred until full vectors are fetched from the same chunk. PageANN [25] reduces in-memory requirements by focusing on fast routing rather than distance approximation, maintaining only a lightweight hash-based routing index in memory.

(2) *Major-in-Disk Methods.* These methods are exemplified by DiskANN [18], which strikes a balance between memory usage and query performance by keeping PQ codes in memory for fast approximate distance estimation, while storing the full graph structure and raw vectors on disk to support fine-grained traversal and final result refinement. The in-memory PQ codes are used to estimate distances to neighboring nodes and guide graph traversal toward the neighbor with the smallest estimated distance. The disk block containing the selected neighbor is then fetched to retrieve its adjacency information for continued traversal, as well as the corresponding raw vectors for exact distance evaluation. This design typically limits disk access to one block read per hop and has become the mainstream approach in graph-based disk-resident VSS [33, 35, 34, 20, 24, 74]. However, the memory footprint of this approach grows linearly with the dataset size, making it less suitable for scenarios with extremely constrained memory budgets.

While most existing systems rely on in-memory PQ codes to guide on-disk traversal and refinement, several recent works explore alternative ways to organize memory and disk responsibilities. Instead of storing per-vector PQ codes in memory, these systems retain only coarser or more compact metadata, such as shard-level centroids, bucket summaries, or page-aligned routing structures, to reduce memory usage while still enabling effective pruning and traversal. For example, SmartANNS [75] keeps shard centroids, a lightweight GBDT-based shard-pruning model, and minimal scheduling metadata in host DRAM. At query time, this information is used to select a small set of likely shards and balance load

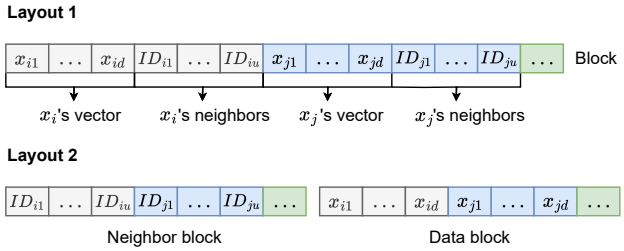


Fig. 8. Two Typical Data Layouts [27].

across SmartSSDs, after which fine-grained HNSW traversal and distance computation are offloaded to SSD-side FPGAs. DiskJoin [38], designed for vector join queries, stores bucket-level metadata in memory, such as bucket centers, radii, and ordering state, to determine which bucket pairs to verify and in what order, enabling batched, cache-aware disk reads and minimizing unnecessary I/O. Full-precision verification is performed only when required.

Finally, some systems further extend these storage strategies to heterogeneous memory environments. For instance, HM-ANN [76] exploits multi-tier memory architectures by placing hot data (frequently accessed nodes and their neighbors) in faster tiers (e.g., DRAM or NVM) and keeping cold data, including raw vectors, in slower tiers.

Bock-Locality-Aware Layouts. In graph-based disk-resident VSS, a key design challenge is how to organize graph indexes, raw vectors, and associated metadata across disk blocks to achieve efficient I/O. As illustrated in Fig. 8, existing layout strategies can be broadly categorized into two classes: *coupled* layouts [18, 20, 23, 25, 26], in which raw vectors are co-located with their corresponding graph nodes, and *decoupled* layouts [27, 77, 41, 22], in which graph adjacency lists and raw vectors are stored separately. Coupled layouts reduce per-hop I/O by ensuring that adjacency information and raw vectors required in the same traversal step are fetched together, while decoupled layouts offer greater flexibility by allowing independent optimization of graph traversal and data access. In the following, we detail each of these two layout strategies.

(1) *Coupled Layouts.* Coupled layouts co-locate graph adjacency lists with their corresponding raw vectors within the same disk blocks or pages. This design philosophy is based on the observation that during graph traversal, when a node is accessed, both its vector representation and its adjacency list are typically needed. By storing them together, a single disk I/O operation can retrieve both pieces of information, reducing the total number of I/O operations required per query. However, when adopting coupled layouts that co-locate vectors with their adjacency lists, a key challenge arises: determining which raw vectors and their adjacency lists should share a disk block so as to minimize random I/O during graph traversal. This problem has been shown to be NP-hard and is formalized as follows [20]:

***Problem** 1: Block Shuffling.* Given a graph index $G = (V, E)$ and disk blocks, each storing up to ϵ nodes, the block shuffling problem aims to assign all nodes in V to blocks so as to maximize the average overlap ratio $OR(G)$, which is

TABLE II
SUMMARY OF LAYOUT OPTIMIZATION METHODS.

Type	Method	Layout Optimization	
Coupled	DiskANN [18]	Round robin assignment	
	DiskANN++ [33]	Isomorphic mapping	
	Starling [20]	Block shuffling	
	MARGO [23]	Monotonic path-aware packing	
	PageANN [25]	Clustering-based	
Type	**Method**	**Neighbor**	**Vector**
Decoupled	GaussDB-Vector [77]	Heuristic	Sorted contiguous
	DGAI [41]	Similarity-aware	Similarity-aware
	TRIM [27]	Block shuffling	Block-aligned
	BAMG [22]	Block shuffling	Block-aligned

defined as the average proportion of nodes within a block that are neighbors across all nodes in V.

We next present existing approaches that address the block shuffling problem. As summarized in Table II, DiskANN++ [33] applies an isomorphic mapping approach for disk-based graph layout, where nodes that are close in the graph structure are packed into the same SSD page using star packing and bin packing algorithms. This increases the probability that neighbor nodes are loaded together in a single I/O, significantly reducing redundant disk reads during search and improving query performance. Starling [20] addresses the block shuffling problem by heuristically grouping each vertex with as many of its neighbors as possible into the same disk block, maximizing the overlap ratio within blocks. This block-level reordering enhances data locality and block utilization, so that each disk I/O is more likely to bring in useful graph neighbors for the searching process. Furthermore, MARGO [23] formulates block layout optimization as a monotonic path-aware edge selection problem, weighting edges by how frequently they appear in real search paths. It assigns important, frequently-traversed edges' endpoints to the same block, maximizing the likelihood that necessary nodes are co-located for typical queries and thereby reducing disk I/Os. PageANN [25] proposes a page-node graph abstraction, where similar vectors are clustered into logical pages that align precisely with SSD pages. Each page stores both vector data and their neighbor connections, ensuring that traversing the graph at the page level closely matches SSD I/O units, which reduces I/O amplification and improves search efficiency.

(2) *Decoupled Layouts.* Decoupled layouts store graph adjacency lists and raw vectors in separate storage regions or files. This separation provides greater flexibility in organizing each component according to its own access patterns. GaussDB-Vector [77] adopts a decoupled layouts at the level of database system engineering. Its indexes (whether IVF or graph-based) are maintained in dedicated files or pages, while raw vectors are sorted and stored contiguously in data blocks optimized for sequential disk access. During search, the traversal happens in the lightweight index, and only when necessary are the corresponding raw vectors fetched. TRIM [27] and BAMG [22] also recognized the benefits of decoupled layouts. In BAMG, the graph index, including neighbor lists and block metadata, is stored in compact structures designed to reduce disk I/Os, while the raw vectors, which consume the majority of storage space, are organized in separate contiguous disk regions and accessed only in batches during the reranking stage. TRIM [27] adopts a decoupled data layout that stores PQ codes and metadata separately from full-dimensional vectors, so that disk access to raw vectors is incurred only for a small set of candidates after pruning. This separation not only avoids unnecessary data loading and distance computations, but also enables independent, hardware-adaptive layout optimizations for index structures and data blocks. DGAI [41] decouples the storage of the graph topology and vectors to accelerate index updates by eliminating redundant I/O operations. To mitigate the query latency caused by this separation, it employs similarity-aware topology reordering, that clusters topologically close nodes on the same pages to improve data locality and minimize I/O overhead during graph traversal.

C. Coarse-to-Fine Query Processing

Disk-resident graph search must jointly minimize (1) how far the search traverses (i.e., the number of hops) and (2) how much data it reads per hop (i.e., the number of I/Os). A common baseline maintains a small in-memory candidate set, iteratively expands the nearest unvisited node, estimates cheap approximate distances (e.g., via PQ methods) for pruning, fetches on-disk blocks only when necessary, and finally re-ranks a small candidate set using full-precision vectors. In this section, we review the search patterns and execution strategies adopted by existing methods, including in-memory coarse search, disk-based fine search with asynchronous execution, cache management strategies, as well as query optimizations for special scenarios such as distributed search and hardware (e.g., GPUs) acceleration.

In-Memory Coarse Search. To reduce disk I/Os, existing methods employ lightweight in-memory routing structures combined with quantization-based evaluation strategies [18, 73, 76, 20, 40, 23]. Specifically, DiskANN [18] keeps static entry points and PQ codes in memory, enabling fast initial navigation and approximate distance computations. The system first uses PQ codes to quickly evaluate candidates, and only retrieves full-precision vectors for candidates that advance the search. This two-stage strategy allows the system to guide neighbor selection and candidate pruning using PQ distances during traversal without disk reads, performing precise distance calculation only in the re-ranking stage. HM-ANN [76] adopts a similar strategy in heterogeneous memory architectures. During search, the system continuously monitors access patterns and dynamically adjusts data placement across memory tiers, ensuring that frequently accessed navigation nodes reside in faster memory tiers. Starling [20], however, constructs a lightweight navigation graph in memory rather than just caching entry points. During the search, it first searches on the navigation graph to find a better starting point before performing a search on the disk-resident full graph.

Disk-Based Fine Search with Asynchronous Execution. Disk-resident VSS methods often exploit asynchronous I/O

and disk-aware layouts to optimize I/O scheduling during query processing and reduce data access costs. As shown in Fig. 9, DiskANN [18] serves as a baseline method that adopts a synchronous execution pattern, whereas Starling [20] and PipeANN [40] are two representative asynchronous search systems based on PageSearch and PipeSearch, respectively. We then review PageSearch and PipeSearch to illustrate their distinct approaches to execution optimization.

(1) *PageSearch.* Starling [20] adopts the PageSearch algorithm, which performs the block-based traversal and reduces the block reads by exploiting block-locality-aware layouts. Since disk access is performed at the block granularity, accessing a node implicitly loads the entire block containing that node into memory. When the layout within each block is well optimized, this block typically contains not only the currently accessed node but also other nodes that are likely to be accessed. PageSearch leverages this property by prioritizing the exploration of nodes within the currently loaded block before accessing nodes from other blocks, allowing multiple search expansions to be served by a single block read and thereby reducing disk I/O. To further improve I/O efficiency, PageSearch employs asynchronous prefetching based on the current candidate set, issuing non-blocking reads for blocks that are likely to be accessed next. By overlapping block loading with distance computation, the algorithm reduces idle waiting time and improves effective disk bandwidth utilization.

(2) *PipeSearch.* PipeANN [40] proposes the PipeSearch algorithm, which reduces I/O cost through pipelined execution and differs from PageSearch by explicitly aligning the search procedure with SSD I/O characteristics. The key observation is that the candidate pool already reveals which records to fetch next. Therefore, the dependency between computation and I/O is largely a pseudo-dependency that is not strict and can be pipelined. Exploiting this insight, PipeSearch proactively issues asynchronous reads for the nearest unread candidates whenever the SSD I/O pipeline has available capacity, without waiting for previous I/O operations to complete or for all neighbor expansions to finish. Meanwhile, the algorithm processes previously fetched vectors in a best-effort manner, allowing computation and I/O to progress independently. By continuously overlapping candidate processing with block fetching, PipeSearch keeps the I/O pipeline busy and effectively hides I/O latency.

Cache Management Strategies. Caching is a key component for reducing disk I/O in disk-resident VSS. By keeping frequently accessed data in memory, systems can significantly reduce disk reads and improve query latency. Existing cache management strategies can be broadly categorized into three classes: static, dynamic, and hybrid static–dynamic strategies.

(1) *Static Caching Strategy.* Static caching strategies determine cache contents prior to query execution, typically during index construction or offline preprocessing. Such strategies are effective when access patterns are predictable, or when certain data objects (e.g., entry points or hub nodes) are known to be frequently accessed regardless of query characteristics. Exemplifying this approach, DiskANN [18] caches a static set of frequently visited nodes, specifically those within a few hops of the graph's entry point. LEANN [74] also noticed

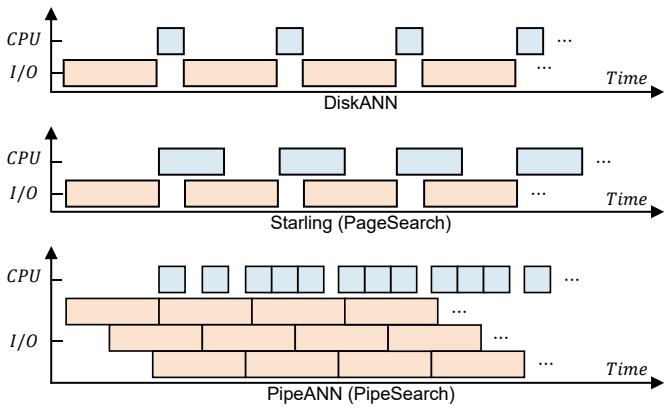


Fig. 9. Execution Timeline for DiskANN, Starling, and PipeANN.

that some nodes might be accessed frequently, which were treated as hub nodes. These hub nodes are then materialized and stored in a dedicated cache region. These static caching strategies are largely driven by historical access patterns and do not anticipate future query behavior. In contrast, DiskJoin [38] constructs a predetermined read schedule over bucket pairs to maximize locality, and combines it with a Belady-style [78] eviction policy that leverages knowledge of future accesses to minimize cache misses. Guided by this schedule, the cache retains exactly the buckets required at each step, proactively reducing disk I/O. A key insight from recent works [41, 26, 33] is that graph structure (i.e., adjacency lists) is accessed far more frequently than raw vector data during search. Building on this observation, Gorgeous [26] prioritizes caching compact adjacency lists in memory. In such design, the cache can accommodate a substantially larger fraction of the graph structure.

(2) *Dynamic Caching Strategies.* Dynamic caching strategies make cache decisions online based on query characteristics, access patterns, and system state, and are particularly effective for query-dependent workloads where optimal cache contents cannot be determined a priori. For example, on heterogeneous memory platforms, HM-ANN [76] promotes frequently accessed high-level adjacency lists to fast DRAM, while placing less frequently used graph regions in slower but higher-capacity NVRAM. As workloads evolve, HM-ANN automatically migrates data across memory tiers to adapt to changing access patterns and improve cache efficiency. DiskANN++ [33] further exploits dynamic caching by maintaining a page heap that enables asynchronous expansion of the search frontier on cached SSD pages while future I/O requests are still in flight. This design both increases cache hit rates for topology pages and hides SSD access latency through overlapping computation and prefetching. Similarly, XN-Graph [24] prioritizes expanding nodes whose neighborhoods are already resident in the cache, rather than waiting for pending SSD reads, and adopts a simple FIFO policy for cache management.

(3) *Hybrid Static-Dynamic Caching.* Hybrid static–dynamic caching mechanisms combine static and dynamic caching strategies to balance predictability and adaptivity. Such designs provide fast access to entry points and high-frequency nodes

identified from global workload statistics, while simultaneously adapting to query-dependent access patterns at runtime. For example, DGAI [41] introduces a hybrid cache management scheme tailored to a decoupled graph-based index. It reserves a small static cache region to pin entry nodes and their multi-hop neighbors that benefit all queries, ensuring efficient navigation during the initial search phase. In parallel, DGAI maintains a query-level dynamic buffer that caches only the topological pages recently traversed within each query, thereby exploiting locality along search paths. A similar principle underlies GoVector [28], which also integrates static and dynamic caching but extends it with similarity-aware adaptation. Instead of caching only entry points and recently accessed nodes, GoVector dynamically adjusts its cache based on the actual query path by leveraging vector similarity. It proactively batches the loading of the current node with nearby nodes, which are clustered on disk according to similarity, increasing the likelihood that subsequent expansions are served from memory. Combined with an LFU replacement policy, this adaptive caching strategy achieves consistently higher cache hit rates and lower I/O latency, even under unpredictable and highly query-dependent workloads.

Optimizations for Special Scenarios. Beyond core query designs, practical graph-based vector search systems must accommodate diverse deployment environments and workload characteristics. In large-scale and performance-critical settings, this often necessitates distributed deployments and hardware acceleration. Accordingly, this section reviews representative optimizations tailored to distributed execution and hardware-accelerated query processing.

(1) Distributed Search. As vector datasets grow beyond the capacity of a single server, distributed systems [79, 80, 1] become essential for achieving scalable throughput and capacity. For graph-based vector search, existing systems primarily adopt one of two distribution paradigms: (i) partitioning the dataset and building independent graph indexes for each partition, or (ii) constructing a single global graph and distributing its nodes across multiple servers. Under the first paradigm, query processing follows a scatter–gather workflow: a query is dispatched to all relevant partitions, each partition performs a local graph search over its subgraph, and a central coordinator merges the local results to produce the final answer. This design is simple and naturally parallelizable, but may sacrifice global graph connectivity and, consequently, search efficiency. In contrast, the second paradigm constructs a single global graph over the entire dataset, preserving the structural properties and search efficiency of a unified graph index. However, it introduces the challenge of efficiently traversing graph edges that span multiple servers. BatANN [39] addresses this challenge by adopting an asynchronous state-passing execution model, in which the entire query state is forwarded to the server owning the next graph region, allowing traversal to continue locally, thereby reducing cross-server communication cost.

(2) Hardware Acceleration. Graph-based VSS is often accelerated using hardware technologies, such as GPUs [81, 82, 42, 83] and SIMD [84, 85]. In disk-resident settings, hardware acceleration is commonly used, particularly for distance computations. FlashANNS [42] introduces a GPU-driven asynchronous I/O framework to address the storage-compute bottleneck. It decouples I/O and distance computation while waiting for I/O responses. The difference is that it leverages GPUs to optimize the distance computation. GustANN [81] focuses on GPU-accelerated batch processing, targeting scenarios where multiple queries need to be processed concurrently. It leverages GPU's parallel processing capabilities to handle distance computations for all queries in the batch simultaneously. This batched approach amortizes the overhead of GPU memory transfers and kernel launches across multiple queries, significantly improving overall system throughput.

D. Graph Updates and Repairs

Dynamic workloads that require continuous updates to both vector data and graph structures have become pervasive in real-world VSS applications. Supporting efficient updates while retaining high query recall is therefore crucial but challenging, particularly for graph-based VSS, where updates can significantly affect graph connectivity and neighbor quality. In this subsection, we review existing update mechanisms and graph repair strategies for graph-based disk-resident VSS methods.

Update Strategies. When handling updates, a primary bottleneck arises from the high I/O overhead incurred during index maintenance. In typical coupled storage schemes, such as those adopted by [18, 20], vector data and graph structures are co-located within the same disk pages, which often leads to redundant reads and writes. As a result, any structural modification, including node insertion or deletion, requires updating the disk pages of affected neighboring nodes, causing substantial I/O amplification. To mitigate this overhead, existing methods generally adopt one of two update strategies: in-place updates and out-of-place updates.

(1) *In-Place Updates.* In-place update strategies directly modify index structures at their existing disk locations, avoiding deferred writes and bulk reorganization. By eliminating buffering and merge-induced write amplification, these strategies reduce update-induced I/O bursts and help stabilize query performance under dynamic workloads. OdinANN [36] exemplifies this approach with a direct-insert policy, in which each new vector is immediately inserted into the on-disk graph without intermediate buffering. This design avoids costly merge operations and prevents update-intensive phases from interfering with foreground queries, resulting in smoother latency behavior. DGAI [41] further improves update efficiency through a decoupled storage layout that separates graph structure from vector data, making in-place updates less I/O-intensive. Under this design, inserting or adjusting graph connections typically involves modifying only compact graph pages rather than vector pages. Moreover, DGAI performs locality-aware in-place updates by placing new nodes near their most similar neighbors and limiting page splits to local reorganization. Together, the decoupled layout and locality-preserving placement reduce the scope of page modifications, mitigating I/O amplification during updates.

(2) *Out-of-Place Updates.* Out-of-place updates write new or modified vectors to a new location (e.g., a short-term layer)

rather than overwriting the originals, then asynchronously merge and clean up old versions to keep sequential writes and high throughput. Fresh-DiskANN [34] employs a background *StreamingMerge* process that batches incremental updates from an in-memory short-term index into a long-term SSD-resident index, amortizing costly writes and preserving system responsiveness. LSM-VEC [37] adopts a hierarchical architecture combining in-memory upper layers with a disk-resident bottom layer managed via a log-structured merge (LSM) tree, optimizing write amplification and enabling scalable ingestion of dynamic data. In addition, it stores vectors and graph structures separately, allowing LSM-VEC to update the graph without touching vectors.

Graph Repair. Maintaining high query recall under continuous updates is challenging, as naive insertions and deletions can gradually degrade graph connectivity, reduce navigability, and ultimately harm retrieval quality. As a result, effective graph repair mechanisms are essential to restore high-quality neighbor links and preserve query recall over time. Fresh-DiskANN [34] addresses this challenge by introducing *Fresh-Vamana*, a graph index that proactively maintains graph quality during updates, avoiding expensive periodic rebuilds. Specifically, it enforces a relaxed α-RNG property to stabilize recall under dynamic workloads. During insertions, it queries the existing graph to identify candidate neighbors and reconstructs local connections by applying the α-RNG rules to the newly inserted nodes. During deletions, it repairs the graph by reestablishing connections among affected neighboring nodes. Similar graph repair strategies are also adopted by LSM-VEC [37], OdinANN [36], and DGAI [41], which rebuild or adjust edge connections locally for nodes impacted by insertions and deletions, thereby mitigating quality degradation caused by dynamic updates.

V. Tree-based Methods

Tree-based methods constitute an important class of disk-resident VSS systems, particularly for low- to medium-dimensional data. As shown in Fig. 10, rather than organizing data via inverted lists or navigating proximity graphs, these approaches rely on explicit tree structures, typically B^+-tree–style indexes, to manage disk-resident vectors and control access order. Although they differ in their vector transformations, most tree-based methods follow a shared external-memory design philosophy: vectors are first mapped into orderable and low-dimensional representations and then organized within tree-managed disk blocks to enable efficient pruning and controlled traversal during search. We review their techniques from two aspects: (1) *Block-Oriented Partitioning and Layouts* (Section V-A), and (2) *Block Filtering and Query Execution* (Section V-B).

A. Block-Oriented Partitioning and Layouts

A central goal of tree-based disk-resident VSS methods is to organize vectors into disk blocks that can be accessed and pruned efficiently under external-memory constraints. Unlike inverted files or graph layouts, which emphasize inverted lists or neighbor links, tree-based methods rely primarily on data ordering to construct hierarchical index structures. This ordering determines how vectors are partitioned into blocks, how blocks are laid out on disk, and ultimately how effectively block access can be exploited during search. We classify existing methods into three classes: projection-based, quantization-based, and hierarchical-clustering-based methods.

Projection-based Methods. A large class of tree-based methods achieves block-oriented layouts by first mapping high-dimensional vectors into orderable or low-dimensional representations. Early approaches such as C2LSH [86], QALSH [9], SRS [87], I-LSH [43], and EI-LSH [44] employ sets of 2-stable hash functions to generate randomized projections, producing multiple hash values per vector. These hash values are then indexed using tree structures. More recent learning to hash (L2H) based approaches like [88] follow the same structural principle but replace random projections with data-dependent learned mappings.

From a layout perspective, both approaches result in ordered sequences of keys managed by tree leaf nodes, enabling block-level locality and range-based access on disk. Differences between randomized and learned mappings primarily affect the quality of locality within blocks, rather than the underlying partitioning or storage structure.

Quantization-Based Methods. PQBF [46] organizes disk-resident data based on PQ partitions. It builds a separate PQB^+-tree for each partition, forming a PQB^+-forest. Within each PQB^+-tree, PQ codes are arranged according to a linear order designed to approximately preserve similarity under asymmetric quantizer distance (AQD), enabling tree-based indexing over compressed representations. PQBF uses B^+-trees to manage large collections of PQ codes on disk, achieving compact storage and structured access. However, the imposed linear order does not strictly guarantee disk-page locality, and sequential access during search is not always ensured.

Hierarchical-Clustering-Based Methods. B+ANN [47] creates a highly localized disk layout by partitioning the dataset before indexing. It employs recursive k-means [56] to group vectors into semantically coherent partitions, ensuring that vectors within each partition are closely related. These final partitions are then used to construct a B^+-tree variant where the inner nodes store cluster centroids as keys for navigation, and the leaf nodes store the actual vectors. By physically co-locating the vectors of each leaf node into contiguous disk blocks, this design explicitly maps the semantic hierarchy from clustering directly onto the physical storage, optimizing the layout for efficient, sequential block access.

B. Block Filtering and Query Execution

Tree-based methods differ primarily in how they filter candidate blocks and execute queries under external-memory constraints. Based on their data partitioning and traversal strategies, we categorize existing approaches into three classes: (1) sequential traversal over projections, (2) pruning-based traversal over quantized codes, and (3) tree and graph traversal over hierarchical structures.

Sequential Traversal over Projections. In projection-based methods like C2LSH [86] and QALSH [89], query processing

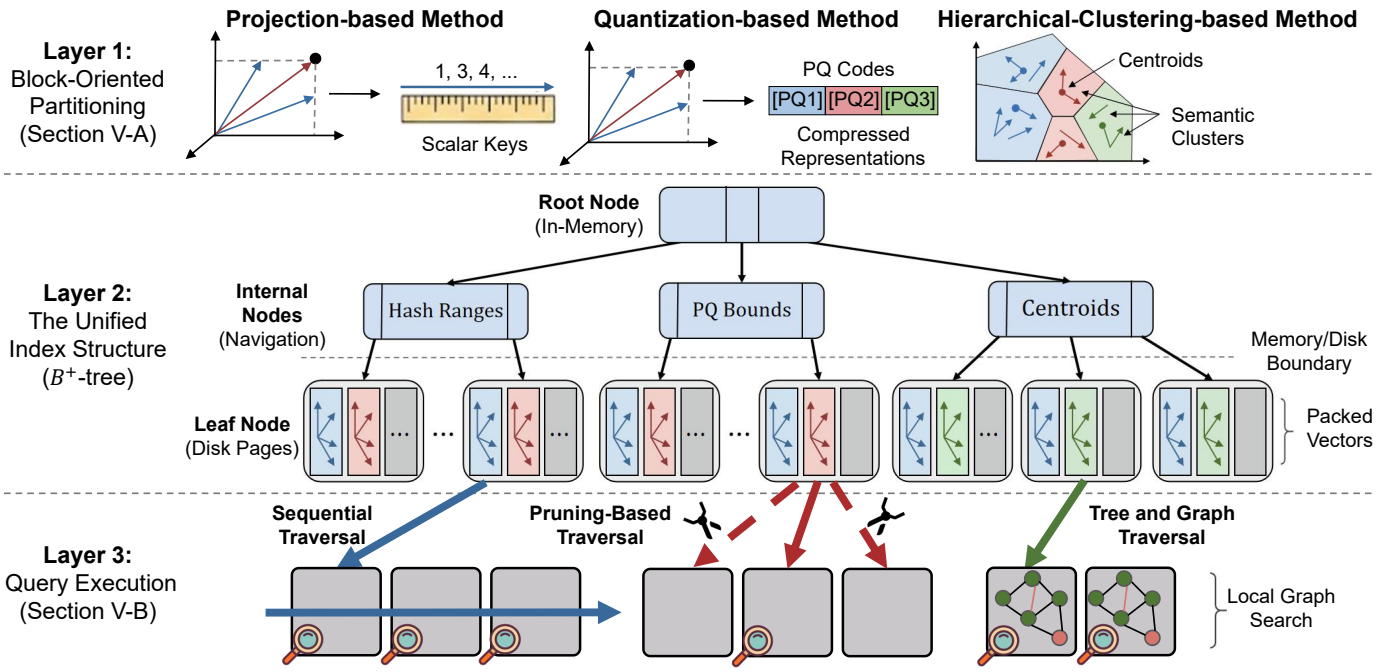


Fig. 10. Illustration of Tree-based Methods.

involves expanding search ranges over projected values to retrieve candidate objects from B^+-tree indexes. However, these earlier methods adopt a bucket exponential expansion strategy, where the search range (bucket width) is enlarged by a factor of c in each round. This approach often leads to a large number of I/Os, as all points within the newly expanded bucket must be visited to maintain theoretical guarantees, even if a suitable candidate is found early in the round. Furthermore, this fixed expansion can be inefficient, sometimes retrieving too many new objects and other times retrieving none at all.

To address these inefficiencies, methods like I-LSH [43] and EI-LSH [44] adopt an incremental search strategy. Instead of expanding buckets exponentially, they access projected objects sequentially in increasing order of their distance to the query on the projected dimension. This allows for a more fine-grained search that prioritizes the most promising candidates first, significantly increasing the proportion of sequential disk accesses. This incremental approach also enables the use of a more aggressive early termination condition, which stops the search as soon as a candidate is found with a high probability of being the correct answer, thus reducing I/O costs while still maintaining theoretical guarantees.

Li et al. [88] follow the same execution paradigm but benefit from higher-quality ordering. Since learned projections are optimized to preserve neighborhood structure, blocks accessed consecutively during range scans are more likely to contain relevant candidates. As a result, learned methods typically reduce the number of scanned blocks compared to random projections. Nevertheless, when multiple projections or lists are involved, random I/O remains unavoidable during candidate aggregation and verification.

Pruning-Based Traversal over Quantization Codes. The quantization-based methods, such as PQBF [46], rely less on long sequential scans and more on tree-based pruning. Queries traverse selected PQB$^+$-trees using AQD lower bounds to eliminate irrelevant subtrees. While this significantly reduces the number of accessed blocks, the imposed linear order over PQ codes does not guarantee disk-page adjacency. Consequently, PQBF's query execution often involves non-trivial random I/O, with efficiency gains stemming primarily from pruning rather than sequential access.

Tree and Graph Traversal over Hierarchical Structures. B+ANN [47] implements a hybrid query execution strategy that minimizes disk I/O by separating coarse-grained filtering from fine-grained search. During a query, a traversal begins from the root of the B+ANN tree, using only the inner nodes, which can be kept in memory, to navigate the index. This in-memory traversal acts as an efficient filter, quickly identifying a small set of promising leaf-node blocks on disk without reading the vectors themselves. Disk access is then confined to loading only these targeted blocks. Once in memory, a more precise, graph-like greedy search can be performed using pre-built "skip-edge" connections between vectors within these leaf nodes to refine the final candidates. This two-phase execution model converts queries into a small number of coarse-grained, largely sequential block reads and performs graph-like refinement within each loaded block, substantially reducing the impact of random I/O latency.

VI. Benchmarks

In this section, we summarize commonly used datasets for disk-resident VSS. Their key characteristics are reported in Table III, including dimensionality, dataset size, data origin, distance metric, query workload size (when available), and

TABLE III
DATASET SUMMARY.

Dataset	#Dim	#N	Origin	Distance Metric	Query Size	RC	LID	Feature
MovieLens-10M [90]	65,134	69K	Set	Jaccard	N/A	1.66	14.4	✓
Kosarak [91]	27,983	7K	Set	Jaccard	N/A	2.34	37.3	×
OpenAI-3072 [92]	3,072	1M	Text	Euclidean	N/A	1.94	18.9	✓
WIT-ResNet50 [93]	2,048	45K	Image	Euclidean	N/A	1.45	32.6	×
OpenAI-1536 [94]	1,536	1M	Text	Euclidean	N/A	2.43	15.4	✓
Enron [95]	1,369	94K	Text	Angular	200	6.42	13.6	✓
Wiki [96]	1,024	247.2M	Text	Angular	N/A	2.38	14.2	✓
MSMARCO [97]	1,024	53.2M	Text	Angular	1,677	1.69	22.6	✓
GIST [11]	960	1M	Image	Euclidean	1,000	1.94	18.9	×
MNIST [98]	784	60K	Image	Euclidean	10,000	2.43	12.7	✓
LAION-5B [99]	768	5B	Image	Angular	N/A	2.27	14.2	✓
COCO-I2I [100]	512	113K	Image	Angular	10,000	3.32	9.37	×
MSong [101]	420	1M	Audio	Euclidean	200	3.81	9.50	×
Tiny [102]	384	5M	Image	Euclidean	10,000	1.77	32.8	×
GloVe-300 [103]	300	2.2M	Text	Angular	10,000	1.96	28.1	×
Crawl [104]	300	2M	Text	Angular	10,000	2.44	15.9	×
Facebook SimSearchNet++ [105]	256	1B	Image	Euclidean	100,000	1.20	70.0	×
UQvideo [106]	256	1M	Video	Euclidean	10,000	8.39	7.20	×
NYTimes [107]	256	290K	Text	Angular	10,000	1.82	19.3	×
Yandex Text-to-Image [105]	200	1B	Text & Image	Inner-product	100,000	3.94	4.44	×
GloVe-200 [103]	200	1.2M	Text	Angular	10,000	2.13	26.3	×
Audio [108]	192	53K	Audio	Euclidean	200	2.96	16.1	×
SIFT [109]	128	1B	Image	Euclidean	10,000	3.50	9.30	×
SPACEV [105]	100	1.4B	Text	Euclidean	29,316	2.24	17.4	×
Turing [105]	100	1B	Text	Euclidean	100,000	1.56	32.6	×
GloVe-100 [103]	100	1.2M	Text	Angular	10,000	1.82	20.0	×
DEEP [110]	96	1B	Image	Euclidean	10,000	1.96	12.1	×

whether feature attributes are provided. The table also includes two dataset difficulty metrics, namely Relative Contrast (RC) and Local Intrinsic Dimensionality (LID). All datasets and the difficulty evaluation tools are publicly available in our GitHub repository,[1] which also provides brief dataset descriptions. Below, we introduce the two difficulty metrics and report their evaluation results.

Relative Contrast (RC). RC measures the separability of true nearest neighbors by comparing the typical query-to-database distance with the nearest-neighbor distance. An RC value closer to 1 indicates strong distance concentration and, consequently, a more challenging search problem, whereas larger RC values imply clearer distance separation and therefore an easier search regime. Using our developed tool, we compute the RC values for all datasets listed in Table III. The results show that Facebook SimSearchNet++ exhibits RC values closest to 1, reflecting severe global distance concentration. In contrast, datasets such as UQvideo exhibit substantially larger RC values, indicating clearer global distance separation.

Local Intrinsic Dimensionality (LID). LID characterizes the *local growth rate* of the neighborhood around a query, capturing how rapidly the number of nearby points increases as the radius expands. In practice, LID is estimated from the empirical distances to a query's nearest neighbors. Higher LID values indicate that many points lie at similar distances within the local neighborhood, making nearest-neighbor identification more ambiguous and thus more difficult. Lower LID values suggest stronger local contrast and typically an easier search regime. We evaluate the LID values for all datasets, and the results show that Facebook SimSearchNet++ exhibits particularly high LID values, suggesting complex and rapidly expanding local neighborhoods. In contrast, datasets such as UQvideo exhibit much lower LID values, reflecting simpler local geometry and well-separated neighborhood structures.

[1]https://github.com/hkbudb/disk-vss-survey

VII. FURTHER DIRECTIONS

Despite significant progress, disk-resident VSS remains a rapidly evolving research area with many open challenges. Below, we outline several promising directions that are not yet fully addressed by existing methods, but are becoming increasingly critical as vector databases move toward larger-scale deployments, support more mixed and dynamic workloads, and operate in cloud-native, multi-tenant, and privacy-constrained environments.

(1) Unified Support for Mixed and Dynamic Workloads. While most existing disk-resident VSS methods focus on optimizing a single aspect, such as ANN queries, FANN queries, or update efficiency, real-world deployments often require *joint support* for mixed query types under dynamic workloads. In practice, a single system may need to simultaneously serve high-recall ANN queries, FANN queries, and continuous insertions or deletions, each exhibiting distinct access patterns, pruning behaviors, and I/O characteristics. Naively supporting such mixed workloads using separate indexes or independently tuned execution paths can lead to excessive storage overhead, duplicated I/O, and suboptimal cache utilization. Future research could explore unified index structures

and adaptive frameworks that dynamically balance ANN and FANN processing while efficiently handling updates.

(2) Exploiting Emerging Storage Hardware and Cloud-Native Environments. Most disk-resident VSS techniques are originally designed for local SSDs with relatively stable latency and bandwidth characteristics. However, modern large-scale deployments increasingly rely on cloud-native storage infrastructures [111, 112, 113, 114, 115], including disaggregated storage, remote NVMe, and object stores, where access latency, throughput, and cost can vary significantly across storage tiers. These environments challenge traditional assumptions regarding block size, sequential access, and caching behavior, and often expose new trade-offs between performance elasticity and cost efficiency. Future research may investigate storage-aware and cloud-native VSS designs that explicitly account for heterogeneous storage properties, such as tier-aware indexing, adaptive block layouts, and elastic query execution strategies that migrate or replicate index components across memory, SSDs, and object storage. Co-designing disk-resident VSS with cloud orchestration mechanisms and resource schedulers also presents an important direction for enabling scalable and cost-efficient vector similarity search.

(3) Privacy-Preserving and Federated Disk-Resident VSS. As vector databases are increasingly deployed in privacy-sensitive, multi-tenant, and cross-organizational environments, supporting disk-resident VSS under strict data isolation and privacy constraints becomes an important open challenge. In such settings, vectors are often distributed across multiple sites and cannot be centrally aggregated, necessitating federated vector similarity search with limited information sharing, constrained communication budgets, and potentially untrusted execution environments. Existing VSS methods rarely consider the interaction between disk-efficient indexing, communication overhead, and privacy guarantees. Future research could explore federated and privacy-preserving disk-resident VSS designs that integrate secure indexing, encrypted or compressed representations, and communication-aware query processing. Developing pruning and early-termination techniques that reduce cross-site communication while preserving search accuracy is particularly important for enabling trustworthy vector search in federated and privacy-constrained deployments.

(4) Parameter Tuning and Auto-Configuration. Disk-resident VSS systems typically expose a large number of parameters, including index configurations, block sizes, memory budgets, pruning thresholds, and cache sizes, all of which critically influence query latency, accuracy, and I/O efficiency. In practice, these parameters are often manually tuned based on empirical heuristics or assumed workload characteristics, making them difficult to generalize across datasets, hardware platforms, and deployment environments. Moreover, optimal configurations may change over time as workloads evolve or system resources fluctuate. Developing systematic parameter tuning and auto-configuration mechanisms therefore remains a significant challenge. Promising directions include workload-aware and data-aware tuning strategies, online performance monitoring with adaptive reconfiguration, and learning-based approaches that automatically balance accuracy, latency, and I/O costs under dynamic conditions.

(5) Disk-Resident VSS as a System Substrate for RAG and LLM-Centric Applications. Retrieval-Augmented Generation (RAG) and large language model (LLM)–centric applications increasingly rely on vector databases as an external memory to store and retrieve long-term knowledge, conversational history, and task-specific context. Unlike traditional ANN workloads, these applications introduce new access patterns and system requirements, including multi-granularity retrieval (e.g., passages, documents, episodic memories), strict latency constraints in interactive settings, and frequent memory updates driven by continual learning or user interactions. Moreover, LLM memory workloads often require semantic freshness, temporal locality, and selective forgetting, which are not well supported by existing disk-resident VSS designs. Future research could investigate disk-resident VSS architectures that explicitly model LLM memory semantics, such as time-aware or session-aware indexing and query processing mechanisms jointly optimized for retrieval quality and generation efficiency. Bridging disk-resident VSS with RAG pipelines also raises new challenges in end-to-end optimization, including co-designing retrieval granularity, caching policies, and pruning strategies to better align vector search behavior with downstream generation objectives.

(6) Standardized Benchmarking and Reproducibility. Evaluating disk-resident VSS systems remains challenging due to the diversity of storage architectures, index layouts, caching strategies, and workload assumptions adopted across existing studies. Therefore, prior work often relies on different datasets, metrics, and experimental configurations, complicating fair and reproducible comparison. Establishing standardized benchmarks and evaluation protocols that capture realistic storage hierarchies, hybrid workloads, and cost–performance trade-offs is therefore essential. Open-source implementations, unified experimental frameworks, and reproducible end-to-end evaluation pipelines would greatly facilitate systematic comparison and accelerate progress in disk-resident VSS research.

VIII. Conclusion

In this survey, we have presented a comprehensive and structured review of disk-resident VSS, a research area that has become increasingly important as vector datasets grow beyond the capacity of main memory. We organized existing disk-resident VSS methods into three major categories, namely IVF-based, graph-based, and tree-based approaches, based on their fine-grained filtering structures on disk. For each category, we systematically analyzed their core design principles and decomposed their architectures into key technical components, including index construction, block-aware layouts, query execution pipelines, and update and maintenance mechanisms. Beyond method classification, we have reviewed the underlying techniques, summarized commonly used datasets, and discussed several open challenges and promising research directions. We hope that this survey will serve as a foundational reference to guide and inspire future research and practical system development in this rapidly evolving area.

REFERENCES

[1] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai, "Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data," *PVLDB*, vol. 13, no. 12, pp. 3152–3165, 2020.

[2] B. Agüera y Arcas, B. Gfeller, R. Guo, K. Kilgour, S. Kumar, J. Lyon, J. Odell, M. Ritter, D. Roblek, M. Sharifi, and M. Velimirović, "Now playing: Continuous low-power music recognition," 2017. [Online]. Available: https://arxiv.org/abs/1711.10958

[3] Y. Rui, T. S. Huang, and S.-F. Chang, "Image retrieval: Current techniques, promising directions, and open issues," *Journal of visual communication and image representation*, vol. 10, no. 1, pp. 39–62, 1999.

[4] P. Gao, Z. Tian, X. Meng, X. Wang, R. Hu, Y. Xiao, Y. Liu, Z. Zhang, J. Chen, C. Gao *et al.*, "Trae agent: An llm-based agent for software engineering with test-time scaling," *arXiv preprint arXiv:2507.23370*, 2025.

[5] M. Arslan, H. Ghanem, S. Munawar, and C. Cruz, "A survey on rag with llms," *Procedia computer science*, vol. 246, pp. 3781–3790, 2024.

[6] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li, "A survey on rag meeting llms: Towards retrieval-augmented large language models," *ACM SIGKDD*, pp. 6491–6501, 2024.

[7] A. Asai, Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi, "Self-rag: Learning to retrieve, generate, and critique through self-reflection," *ICLR*, 2024.

[8] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE TPAMI*, vol. 36, no. 11, pp. 2227–2240, 2014.

[9] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng, "Query-aware locality-sensitive hashing for approximate nearest neighbor search," *PVLDB*, vol. 9, no. 1, pp. 1–12, 2015.

[10] J. Li, X. Yan, J. Zhang, A. Xu, J. Cheng, J. Liu, K. K. Ng, and T.-c. Cheng, "A general and efficient querying method for learning to hash," *ACM SIGMOD*, pp. 1333–1347, 2018.

[11] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE TPAMI*, vol. 33, no. 1, pp. 117–128, 2010.

[12] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization for approximate nearest neighbor search," *IEEE TPAMI*, pp. 2946–2953, 2013.

[13] J. Gao and C. Long, "Rabitq: Quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search," *ACM SIGMOD*, vol. 2, no. 3, pp. 1–27, 2024.

[14] F. André, A.-M. Kermarrec, and N. Le Scouarnec, "Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan," *PVLDB*, vol. 9, no. 4, p. 12, 2016.

[15] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE TPAMI*, vol. 42, no. 4, pp. 824–836, 2018.

[16] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *PVLDB*, vol. 12, no. 5, pp. 416–474, 2019.

[17] Y. Peng, B. Choi, T. N. Chan, J. Yang, and J. Xu, "Efficient approximate nearest neighbor search in multi-dimensional databases," *ACM SIGMOD*, vol. 1, no. 1, pp. 1–27, 2023.

[18] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," *NeurIPS*, vol. 32, 2019.

[19] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "Spann: Highly-efficient billion-scale approximate nearest neighborhood search," *NeurIPS*, vol. 34, pp. 5199–5212, 2021.

[20] M. Wang, W. Xu, X. Yi, S. Wu, Z. Peng, X. Ke, Y. Gao, X. Xu, R. Guo, and C. Xie, "Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment," *ACM SIGMOD*, vol. 2, no. 1, pp. 1–27, 2024.

[21] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang *et al.*, "Spfresh: Incremental in-place update for billion-scale vector search," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 545–561.

[22] H. Li and J. Xu, "Bamg: A block-aware monotonic graph index for disk-based approximate nearest neighbor search," *arXiv preprint arXiv:2509.03226*, 2025.

[23] Z. Yue, B. Zheng, L. Xu, K. Xu, S. Zhang, Y. Du, Y. Gao, X. Zhou, and C. S. Jensen, "Select edges wisely: Monotonic path aware graph layout optimization for disk-based ann search," *PVLDB*, vol. 18, no. 11, pp. 4337–4349, 2025.

[24] C. Zhang, J. Wang, W.-L. Zhao, and S. Xiao, "Highly efficient disk-based nearest neighbor search on extended neighborhood graph," *ACM SIGIR*, pp. 2513–2523, 2025.

[25] D. Kang, D. Jiang, H. Yang, H. Liu, and B. Li, "Scalable disk-based approximate nearest neighbor search with page-aligned graph," *arXiv preprint arXiv:2509.25487*, 2025.

[26] P. Yin, X. Yan, Q. Zhou, H. Li, X. Li, L. Zhang, M. Wang, X. Yao, and J. Cheng, "Gorgeous: Revisiting the data layout for disk-resident high-dimensional vector search," *arXiv preprint arXiv:2508.15290*, 2025.

[27] Y. Song, P. Zhang, C. Gao, B. Yao, K. Wang, Z. Wu, and L. Qu, "Trim: Accelerating high-dimensional vector similarity search with enhanced triangle-inequality-based pruning," *ACM SIGMOD*, 2025.

[28] Y. Zhou, S. Lin, S. Gong, S. Yu, S. Fan, Y. Zhang, and G. Yu, "Govector: An i/o-efficient caching strategy for high-dimensional vector nearest neighbor search," *arXiv preprint arXiv:2508.15694*, 2025.

[29] J. Pound, F. Chabert, A. Bhushan, A. Goswami, A. Pacaci, and S. R. Chowdhury, "MicroNN: An on-device disk-resident updatable vector database," in *Companion of the 2025 International Conference on Management of Data*, 2025, pp. 608–621.

[30] M. Zhang and Y. He, "Zoom: Ssd-based vector search for optimizing accuracy, latency and memory," *arXiv preprint arXiv:1809.04067*, 2018.

[31] S. Emanuilov and A. Dimov, "Billion-scale similarity search using a hybrid indexing approach with advanced filtering," *Cybernetics and Information Technologies*, vol. 24, no. 4, pp. 45–58, 2024.

[32] B. Tian, H. Liu, Y. Tang, S. Xiao, Z. Duan, X. Liao, X. Zhang, J. Zhu, and Y. Zhang, "Fusionanns: An efficient cpu/gpu cooperative processing architecture for billion-scale approximate nearest neighbor search," *arXiv preprint arXiv:2409.16576*, 2024.

[33] J. Ni, X. Xu, Y. Wang, C. Li, J. Yao, S. Xiao, and X. Zhang, "Diskann++: Efficient page-based search over isomorphic mapped graph index using query-sensitivity entry vertex," *arXiv preprint arXiv:2310.00402*, 2023.

[34] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri, "Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search," *arXiv preprint arXiv:2105.09613*, 2021.

[35] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, S. Raz, Y. Lin, Y. Zhang, N. Mahapatro, P. Srinivasan *et al.*, "Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters," *Proceedings of the ACM Web Conference 2023*, pp. 3406–3416, 2023.

[36] H. Guo and Y. Lu, "Odinann: Direct insert for consistently stable performance in billion-scale graph-based vector search," *FAST*, 2026.

[37] S. Zhong, D. Mo, and S. Luo, "Lsm-vec: A large-scale disk-based system for dynamic vector search," *arXiv preprint arXiv:2505.17152*, 2025.

[38] Y. Chen, X. Yan, A. Meliou, and E. Lo, "Diskjoin: Large-scale vector similarity join with ssd," *ACM SIGMOD*, vol. 3, no. 6, pp. 1–27, 2025.

[39] N. A. Dang, B. Landrum, and K. Birman, "Passing the baton: High throughput distributed disk-based vector search with batann," *arXiv preprint arXiv:2512.09331*, 2025.

[40] H. Guo and Y. Lu, "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 25)*, 2025, pp. 171–186.

[41] J. Lou, Q. Yu, S. Gong, S. Yu, Y. Zhang, and G. Yu, "Dgai: Decoupled on-disk graph-based ann index for efficient updates and queries," *arXiv preprint arXiv:2510.25401*, 2025.

[42] Y. Xiao, M. Sun, Z. Song, B. Tian, J. Zhang, J. Sun, and Z. Wang, "Breaking the storage-compute bottleneck in billion-scale anns: A gpu-driven asynchronous i/o framework," *arXiv preprint arXiv:2507.10070*, 2025.

[43] W. Liu, H. Wang, Y. Zhang, W. Wang, and L. Qin, "I-lsh: I/o efficient c-approximate nearest neighbor search in high-dimensional space," *IEEE ICDE*, pp. 1670–1673, 2019.

[44] W. Liu, H. Wang, Y. Zhang, W. Wang, L. Qin, and X. Lin, "Ei-lsh: An early-termination driven i/o efficient incremental c-approximate nearest neighbor search," *VLDBJ*, vol. 30, pp. 215–235, 2021.

[45] M. Li, Y. Zhang, Y. Sun, W. Wang, I. W. Tsang, and X. Lin, "I/o efficient approximate nearest neighbour search based on learned functions," *IEEE ICDE*, pp. 289–300, 2020.

[46] Y. Liu, H. Cheng, and J. Cui, "Pqbf: i/o-efficient approximate nearest neighbor search by product quantization," *CIKM*, pp. 667–676, 2017.

[47] S. F. Tekin and R. Bordawekar, "B+ ann: A fast billion-scale disk-based nearest-neighbor index," *arXiv preprint arXiv:2511.15557*, 2025.

[48] Z. Zhang, F. Liu, G. Huang, X. Liu, and X. Jin, "Fast vector query processing for large datasets beyond gpu memory with reordered pipelining," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 23–40.

[49] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement," *IEEE TKDE*, vol. 32, no. 8, pp. 1475–1488, 2019.

[50] Y. Tian, Z. Yue, R. Zhang, X. Zhao, B. Zheng, and X. Zhou, "Approximate nearest neighbor search in high dimensional vector databases: Current research and future directions." *IEEE Data Eng. Bull.*, no. 3, pp. 39–54, 2023.

[51] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *arXiv preprint arXiv:2101.12631*, 2021.

[52] Y. Lin, K. Zhang, Z. He, Y. Jing, and X. S. Wang, "Survey of filtered approximate nearest neighbor search over the vector-scalar hybrid data," *arXiv preprint arXiv:2505.06501*, 2025.

[53] J. Gao, Y. Gou, Y. Xu, J. Shi, C. Long, R. C.-W. Wong, and T. Palpanas, "High-dimensional vector quantization: General framework, recent advances, and future directions." *IEEE Data Eng. Bull.*, vol. 48, no. 3, pp. 3–19, 2024.

[54] J. J. Pan, J. Wang, and G. Li, "Survey of vector database management systems," *VLDBJ*, vol. 33, no. 5, pp. 1591–1615, 2024.

[55] K. Lu, M. Kudo, C. Xiao, and Y. Ishikawa, "Hvs: hierarchical graph structure based on voronoi diagrams for solving approximate nearest neighbor search," *PVLDB*, vol. 15, no. 2, pp. 246–258, 2021.

[56] K. Krishna and M. N. Murty, "Genetic k-means algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 29, no. 3, pp. 433–439, 1999.

[57] M. I. Malinen and P. Fränti, "Balanced k-means for clustering," in *Joint IAPR international workshops on statistical techniques in pattern recognition (SPR) and structural and syntactic pattern recognition (SSPR)*. Springer, 2014, pp. 32–41.

[58] P. S. Bradley, K. P. Bennett, and A. Demiriz, "Constrained k-means clustering," *Microsoft Research, Redmond*, vol. 20, no. 0, p. 0, 2000.

[59] H. Liu, Z. Huang, Q. Chen, M. Li, Y. Fu, and L. Zhang, "Fast clustering with flexible balance constraints," in *2018 IEEE International Conference on Big Data (Big Data)*, 2018.

[60] J. Mohoney, D. Sarda, M. Tang, S. R. Chowdhury, A. Pacaci, I. F. Ilyas, T. Rekatsinas, and S. Venkataraman, "Quake: Adaptive indexing for vector search," *arXiv preprint arXiv:2506.03437*, 2025.

[61] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar, "Accelerating large-scale inference with anisotropic vector quantization," *ICML*, pp. 3887–3896, 2020.

[62] C. Aguerrebere, I. Bhati, M. Hildebrand, M. Tepper, and T. Willke, "Similarity search in the blink of an eye with compressed indices," *arXiv preprint arXiv:2304.04759*, 2023.

[63] C. Aguerrebere, M. Hildebrand, I. S. Bhati, T. Willke, and M. Tepper, "Locally-adaptive quantization for streaming vector search," *arXiv preprint arXiv:2402.02044*, 2024.

[64] M. AI, "Faiss," https://ai.facebook.com/tools/faiss, 2017.

[65] Y. Gou, J. Gao, Y. Xu, and C. Long, "Symphonyqg: Towards symphonious integration of quantization and graph for approximate nearest neighbor search," *ACM SIGMOD*, vol. 3, no. 1, pp. 1–26, 2025.

[66] X. Zhong, H. Li, J. Jin, M. Yang, D. Chu, X. Wang, Z. Shen, W. Jia, G. Gu, Y. Xie *et al.*, "Vsag: An optimized search framework for graph-based approximate nearest neighbor search," *arXiv preprint arXiv:2503.17911*, 2025.

[67] A. X. Liu, K. Shen, and E. Torng, "Large scale hamming distance query processing," *IEEE ICDE*, pp. 553–564, 2011.

[68] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[69] D. Sculley, "Web-scale k-means clustering," *Proceedings of the 19th international conference on World wide web*, 2010.

[70] C. Huan, L. Chen, Z. Yang, S. Ma, R. Gu, R. Yao, Z. Wang, M. Zhang, F. Xi, J. Tao *et al.*, "Orchann: A unified i/o orchestration framework for skewed out-of-core vector search," *arXiv preprint arXiv:2512.22838*, 2025.

[71] B. Bratić, M. E. Houle, V. Kurbalija, V. Oria, and M. Radovanović, "Nn-descent on high-dimensional data," *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics*, pp. 1–8, 2018.

[72] K. Tatsuno, D. Miyashita, T. Ikeda, K. Ishiyama, K. Sumiyoshi, and J. Deguchi, "Aisaq: All-in-storage anns with product quantization for dram-free information retrieval," *arXiv preprint arXiv:2404.06004*, 2024.

[73] Y. Pan, J. Sun, and H. Yu, "Lm-diskann: Low memory footprint in disk-native dynamic graph-based ann indexing," *2023 IEEE International Conference on Big Data (BigData)*, pp. 5987–5996, 2023.

[74] Y. Wang, S. Liu, Z. Li, Y. Wu, Z. Mao, Y. Zhao, X. Yan, Z. Xu, Y. Zhou, I. Stoica *et al.*, "Leann: A low-storage vector index," *arXiv preprint arXiv:2506.08276*, 2025.

[75] H. Liu, B. Tian, Z. Duan, X. Liao, and Y. Zhang, "A smartssd-based near data processing architecture for scalable billion-point approximate nearest neighbor search," *ACM Transactions on Storage*, 2025.

[76] J. Ren, M. Zhang, and D. Li, "Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory," *NeurIPS*, vol. 33, pp. 10 672–10 684, 2020.

[77] J. Sun, G. Li, J. Pan, J. Wang, Y. Xie, R. Liu, and W. Nie, "Gaussdb-vector: A large-scale persistent real-time vector database for llm applications," *PVLDB*, vol. 18, no. 12, pp. 4951–4963, 2025.

[78] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[79] R. Guo, X. Luan, L. Xiang, X. Yan, X. Yi, J. Luo, Q. Cheng, W. Xu, J. Luo, F. Liu, Z. Cao, Y. Qiao, T. Wang, B. Tang, and C. Xie, "Manu: a cloud native vector database management system," *PVLDB*, vol. 15, no. 12, p. 3548–3561, 2022.

[80] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu *et al.*, "Milvus: A purpose-built vector data management system," *ACM SIGMOD*, pp. 2614–2627, 2021.

[81] H. Jiang, H. Guo, M. Xie, J. Shu, and Y. Lu, "High-throughput, cost-effective billion-scale vector search with a single gpu," *ACM SIGMOD*, vol. 3, no. 6, 2025.

[82] K. Venkatasubba, S. Khan, S. Singh, H. V. Simhadri, and J. Vedurada, "BANG: Billion-Scale Approximate Nearest Neighbour Search Using a Single GPU," *IEEE Transactions on Big Data*, vol. 11, no. 06, pp. 3142–3157, 2025.

[83] Y. Peng, D. Yang, Z. Xie, J. Sun, L. Shou, K. Chen, and G. Chen, "Svfusion: A cpu-gpu co-processing architecture for large-scale real-time vector search," *arXiv preprint arXiv:2601.08528*, 2026.

[84] M. Wang, H. Wu, X. Ke, Y. Gao, Y. Zhu, and W. Zhou, "Accelerating graph indexing for anns on modern cpus," *ACM SIGMOD*, vol. 3, no. 3, pp. 1–29, 2025.

[85] Z. Song, B. Wang, and X. Yang, "Accelerating high-dimensional ann search via skipping redundant distance computations," *ACM SIGMOD*, vol. 3, no. 6, pp. 1–29, 2025.

[86] J. Gan, J. Feng, Q. Fang, and W. Ng, "Locality-sensitive hashing scheme based on dynamic collision counting," *ACM SIGMOD*, pp. 541–552, 2012.

[87] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin, "Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index," *PVLDB*, 2014.

[88] M. Li, Y. Zhang, Y. Sun, W. Wang, I. W. Tsang, and X. Lin, "I/o efficient approximate nearest neighbour search based on learned functions," *IEEE ICDE*, pp. 289–300, 2020.

[89] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng, "Query-aware locality-sensitive hashing for approximate nearest neighbor search," *PVLDB*, vol. 9, no. 1, pp. 1–12, 2015.

[90] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM transactions on interactive intelligent systems (tiis)*, vol. 5, no. 4, pp. 1–19, 2015.

[91] B. Goethals and M. J. Zaki, "Advances in frequent itemset mining implementations: report on fimi'03," *Acm Sigkdd Explorations Newsletter*, vol. 6, no. 1, pp. 109–117, 2004.

[92] Qdrant, "dbpedia-entities-openai3-text-embedding-3-large-3072-1m," Hugging Face Datasets, Feb. 2024. [Online]. Available: https://huggingface.co/datasets/Qdrant/dbpedia-entities-openai3-text-embedding-3-large-3072-1M

[93] K. Srinivasan, K. Raman, J. Chen, M. Bendersky, and M. Najork, "Wit: Wikipedia-based image text dataset for multimodal multilingual machine learning," in *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*, 2021, pp. 2443–2449.

[94] Qdrant, "dbpedia-entities-openai3-text-embedding-3-large-1536-1m," Hugging Face Datasets, Feb. 2024. [Online]. Available: https://huggingface.co/datasets/Qdrant/dbpedia-entities-openai3-text-embedding-3-large-1536-1M

[95] B. Klimt and Y. Yang, "The enron corpus: A new dataset for email classification research," *European conference on machine learning*, pp. 217–226, 2004.

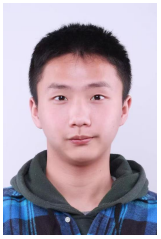
[96] W. Foundation. Wikimedia downloads. [Online]. Available: https://dumps.wikimedia.org

[97] T. Nguyen, M. Rosenberg, X. Song, J. Gao, S. Tiwary, R. Majumder, and L. Deng, "Ms marco: A human-generated machine reading comprehension dataset," *OpenReview*, 2016.
[98] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.
[99] C. Schuhmann, R. Beaumont, R. Vencu, C. Gordon, R. Wightman, M. Cherti, T. Coombes, A. Katta, C. Mullis, M. Wortsman *et al.*, "Laion-5b: An open large-scale dataset for training next generation image-text models," *NeurIPS*, vol. 35, pp. 25 278–25 294, 2022.
[100] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," *European conference on computer vision*, pp. 740–755, 2014.
[101] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," *ISMIR 2011: Proceedings of the 12th International Society for Music Information Retrieval Conference*, pp. 591–596, 2011.
[102] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *IEEE TPAMI*, vol. 30, no. 11, pp. 1958–1970, 2008.
[103] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," *EMNLP*, pp. 1532–1543, 2014. [Online]. Available: http://www.aclweb.org/anthology/D14-1162
[104] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, and A. Joulin, "Advances in pre-training distributed word representations," in *Proceedings of the eleventh international conference on language resources and evaluation (LREC 2018)*, 2018.
[105] H. V. Simhadri, G. Williams, M. Aumüller, M. Douze, A. Babenko, D. Baranchuk, Q. Chen, L. Hosseini, R. Krishnaswamny, G. Srinivasa *et al.*, "Results of the neurips'21 challenge on billion-scale approximate nearest neighbor search," *NeurIPS 2021 Competitions and Demonstrations Track*, pp. 177–189, 2022.
[106] J. Song, Y. Yang, Z. Huang, H. T. Shen, and R. Hong, "Multiple feature hashing for real-time large scale near-duplicate video retrieval," *ACM Multimedia*, pp. 423–432, 2011.
[107] D. Newman, "Bag of Words," UCI Machine Learning Repository, 2008, DOI: https://doi.org/10.24432/C5ZG6P.
[108] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," *Proceedings of the 20th international conference on World wide web*, pp. 577–586, 2011.
[109] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: re-rank with source coding," *ICASSP*, pp. 861–864, 2011.
[110] A. Babenko and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," *CVPR*, pp. 2055–2063, 2016.
[111] C. Huang. Migrating from s3 vectors to zilliz cloud: Unlocking the power of tiered storage. [Online]. Available: https://zilliz.com/blog/migrating-from-s3-vectors-to-zilliz-cloud-unlocking-the-power-of-tiered-storage
[112] C. Yun. Introducing amazon s3 vectors: First cloud storage with native vector support at scale (preview). [Online]. Available: https://aws.amazon.com/cn/blogs/aws/introducing-amazon-s3-vectors-first-cloud-storage-with-native-vector-support-at-scale/
[113] T. Inc. (2025) Turbopuffer — serverless vector & full-text search built from first principles on object storage. Company website / product page. [Online]. Available: https://turbopuffer.com/
[114] Y. Song, X. Zhou, C. S. Jensen, and J. Xu, "Vector search for the future: From memory-resident, static heterogeneous storage, to cloud-native architectures," *ACM SIGMOD Tutorial*, 2026.
[115] Z. Li, W. Ding, S. Huang, Z. Wang, Y. Lin, K. Wu, Y. Park, and J. Chen, "Cloud-native vector search: A comprehensive performance analysis," *arXiv preprint arXiv:2511.14748*, 2025.

Yitong Song received the B.Eng. degree in Computer Science from China University of Geosciences and the Ph.D. degree in Computer Science from Shanghai Jiao Tong University. She is currently a postdoctoral fellow in the Department of Computer Science at Hong Kong Baptist University. Her research interests include vector databases and AI data infrastructure.

Huiling Li received the BEng degree in software engineering from Zhengzhou University, in 2021. He is currently working toward the PhD degree in the Department of Computer Science of Hong Kong Baptist University. His research interests include multi-agent computing, vector databases and spatiotemporal data processing.

Zheng Wu received the BEng degree in software engineering from Zhejiang Normal University, in 2024. He is currently working toward the PhD degree in the Department of Computer Science of Hong Kong Baptist University. His research interests include graph algorithms and vector data management.

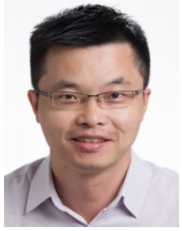


Lanjing Yi received the BEng degree in computer science and engineering from Southern University of Science and Technology, in 2023. He is currently working toward the PhD degree in the Department of Computer Science of Hong Kong Baptist University. His research interests include efficient inference of large language models and long-term memory.

Bojian Zhu received the BEng degree in computer science and technology from Xidian University, in 2024. He is currently working toward the PhD degree in the Department of Computer Science of Hong Kong Baptist University. His research interests include spatiotemporal data management, parallel computing and vector databases.

Xuanhe Zhou is an assistant professor in the Department of Computer Science, Shanghai Jiao Tong University. He received his PhD degree in Computer Science from Tsinghua University. His research interests lie in data systems. He has received the SIGMOD 2025 Jim Gray Dissertation Honorable Mention and VLDB 2023 Best Industry Paper Runner-up.

Xin Huang received the PhD degree from the Chinese University of Hong Kong in 2014. He is currently an Associate Professor at Hong Kong Baptist University. His research interests mainly focus on graph data management and mining.



Jianliang Xu received the BEng degree in computer science and engineering from Zhejiang University, China, and the PhD degree in computer science from the Hong Kong University of Science and Technology. He is a Chair Professor and Head of the Department of Computer Science, Hong Kong Baptist University. His research spans databases, blockchain, and privacy-aware data management. With an h-index of 65, he has published extensively in top-tier venues and is a Fellow of IEEE.