A Survey on Query Processing in Vector Databases

JIADONG XIE, The Chinese University of Hong Kong, China
YINGFAN LIU, Xidian University, China
JEFFREY XU YU, The Hong Kong University of Science and Technology (Guangzhou), China

High-dimensional vectors have become a fundamental data representation in modern applications, such as information retrieval and large language model systems, making vector databases and their query processing an essential research area. While approximate nearest neighbor search has long been the central primitive, modern vector workloads increasingly involve richer query types, including filtered similarity search, multi-vector similarity search, and similarity join. These developments substantially expand the design space of vector query processing and make it harder to obtain a clear and structured view of existing techniques. This survey presents a comprehensive review of query processing in vector databases. We first formalize four query types: similarity search, filtered similarity search, multi-vector similarity search, and similarity join. We then organize existing studies under a unified taxonomy. In particular, we review proximity graphs and quantizations, the two state-of-the-art approaches for similarity search, together with related directions such as distance computation, hard-query processing, and secure search. We further summarize universal and dedicated approaches for filtered similarity search, different methods for multi-vector similarity search, and both exact and approximate algorithms for similarity join. Through this survey, we provide a structured view of current approaches, highlight their connections and differences, and discuss open challenges and future directions.

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

Additional Key Words and Phrases: High-Dimensional Vector, Vector Database, Similarity Search, Similarity Join

ACM Reference Format:
Jiadong Xie, Yingfan Liu, and Jeffrey Xu Yu. 2026. A Survey on Query Processing in Vector Databases. 1, 1 (May 2026), 37 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In the era of big data and AI, high-dimensional vectors have become a fundamental representation for modern applications. A wide range of data objects, such as texts [114], images [121], audio [14], and graphs [53], are increasingly represented as dense or sparse vectors produced by embedding models. As a result, managing and querying vector data has become a core functionality in many emerging systems, including information retrieval [103], and recommendation systems [116]. More recently, the rapid development of retrieval-augmented generation (RAG) [57, 86], large language model (LLM) powered data processing [61, 126], and agentic systems [30, 193] has further strengthened this trend. These new embedding settings make vector databases no longer used only for similarity search. Beyond similarity search, they increasingly need to support a broader spectrum of query types and workloads required by these applications.

Authors' Contact Information: Jiadong Xie, The Chinese University of Hong Kong, China, jdxie@se.cuhk.edu.hk; Yingfan Liu, Xidian University, China, liuyingfan@xidian.edu.cn; Jeffrey Xu Yu, The Hong Kong University of Science and Technology (Guangzhou), China, jeffreyxuyu@hkust-gz.edu.cn.

Manuscript submitted to ACM

Among the supported queries in vector databases, similarity search has long been the central primitive. Given a query vector, it aims to retrieve the most similar vectors under metrics such as Euclidean distance, inner product, and cosine similarity. Its approximate variant, i.e., approximate nearest neighbor (ANN) search, has been extensively studied because it can substantially reduce query latency with slight accuracy loss.

Indeed, recent similarity search research has led to a rich collection of effective techniques, especially proximity graphs [41, 110, 128, 170] and quantizations [45, 47, 72], together with many advances in reducing indexing and search cost [181, 200], supporting dynamic maintenance [75, 176], accelerating distance computation [29, 143], handling hard or out-of-distribution queries [22, 63], and enabling secure search [104, 137]. However, as vector databases are increasingly adopted in real-world applications, user demands have quickly moved beyond classical ANN search. Practical workloads often require retrieval to be combined with structured predicates, multiple vector representations, or large-scale matching across datasets; such needs are also increasingly reflected in real-world datasets and benchmark workloads, especially mixed and hybrid query workloads. For example, filtered similarity search must jointly consider vector similarity and predicate satisfaction [127, 171, 195]; multi-vector similarity search must support multiple dense or sparse vector representations and richer matching semantics [26, 153, 168, 184].

These recent developments make query processing in vector databases substantially more complex.

On the one hand, the method space has expanded rapidly: even within a single query type, multiple index structures, pruning strategies, search paradigms, and system optimizations have been proposed, with related studies emerging from multiple research communities. On the other hand, the boundaries between different query types are becoming increasingly blurred in practice. For instance, recent studies have already begun to unify retrieval with and without predicates [171], dense, sparse and text hybrid search [96]. As a result, obtaining a clear and structured view of the algorithmic landscape has become increasingly difficult.

To address this issue, this paper presents a comprehensive review of query processing techniques in vector databases. We organize existing studies according to four query types that have been widely studied in the literature: similarity search, filtered similarity search, multi-vector similarity search, and similarity join. Fig. 1 provides an overview of the method landscape covered in this paper and summarizes the major categories that will be discussed in the subsequent sections. For similarity search, we review both proximity graph based and quantization based approaches, as well as related directions such as distance computation acceleration, hard-query processing, and secure similarity search. For filtered similarity search, we summarize universal and dedicated index approaches for categorical, numerical, interval, and timestamp predicates. For multi-vector similarity search, we discuss one-to-one, one-to-many, all/any, and dense-sparse combined settings. For similarity join, we cover both exact and approximate methods, including selection-based and reuse-aware approaches. Through this taxonomy, we aim to provide a unified view of a rapidly expanding research area and clarify the connections and differences among these lines of work.

Compared with prior surveys, our focus is not on vector database systems as a whole, which have mainly been reviewed from the perspectives of system architecture, and system challenges [58, 124, 125, 144], or on ANN search in isolation [93, 149, 158, 161], filtered ANN search in isolation [92, 99, 205], and the interplay between vector databases and LLM-based applications [77], but specifically on query processing on different query types in vector databases. This perspective allows us to place a broad range of recent techniques under a common framework and to analyze how different query types give rise to different indexing, search, maintenance, and optimization strategies. At the same time, it also highlights several emerging directions that are not yet fully captured by existing taxonomies, such as unified query processing frameworks, theory-aware evaluation, and new workloads brought by RAG, LLMs, and agentic

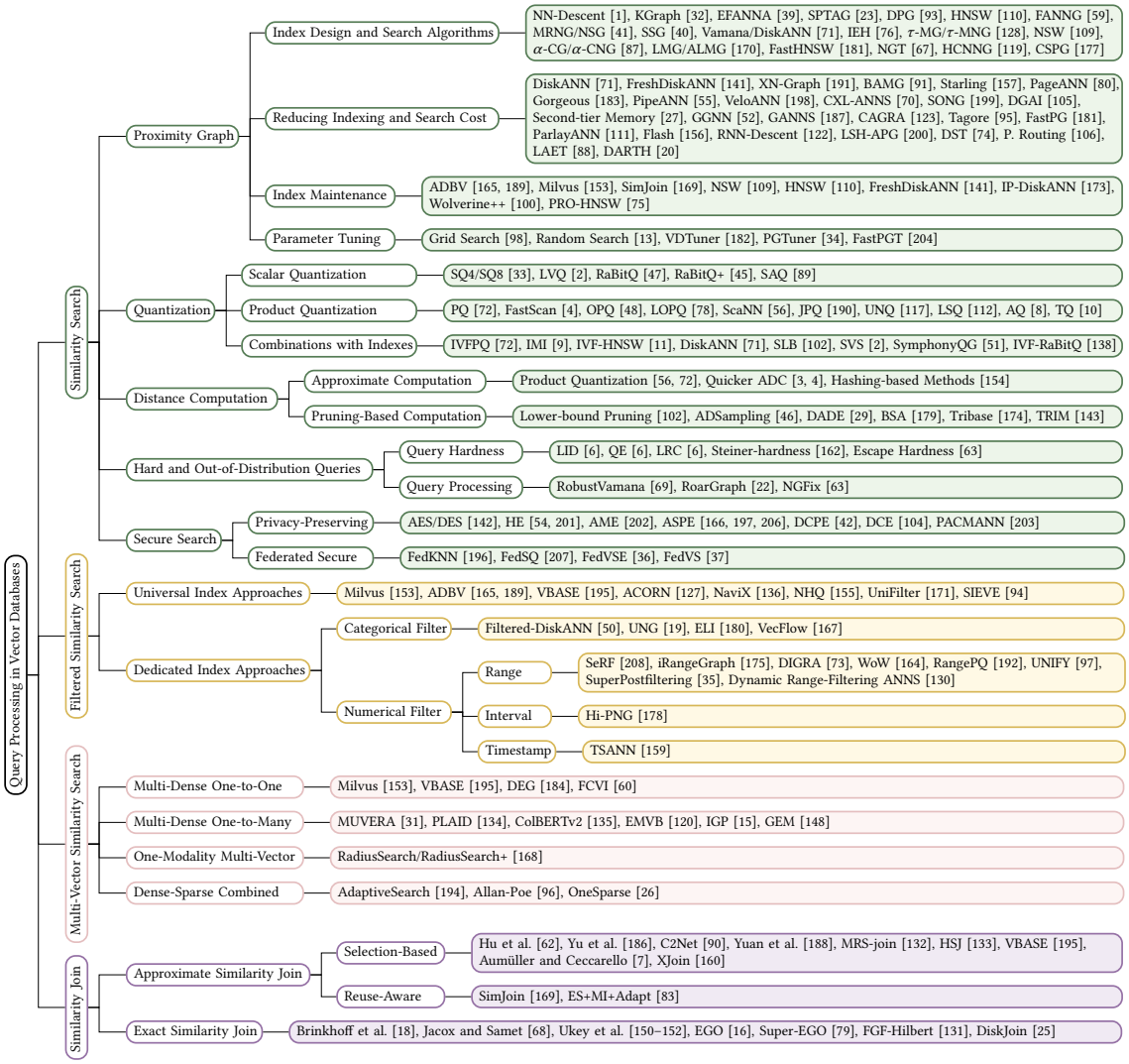


Fig. 1. A taxonomy of approaches for query processing in vector databases

systems. We aim to present this paper as a useful reference for researchers and practitioners seeking a systematic understanding of the current landscape and future directions of vector query processing.

The remainder of this paper is organized as follows. Section 2 introduces preliminaries and formalizes the four query types considered in this paper. Sections 3-6 review techniques for similarity search, filtered similarity search, multi-vector similarity search, and similarity join, respectively. Section 7 discusses future directions and open challenges, and Section 8 concludes the paper.

2 Preliminaries

In this paper, we focus on four query types that are widely studied in vector databases: similarity search, filtered similarity search, multi-vector similarity search, and similarity join. In this section, we first formalize these queries and present their definitions in the context of vector databases.

Given a size-n object set O, each object $o_i \in O$ is represented as a d-dimensional vector and stored in a vector database. We denote the collection of stored vectors as a dataset $D \subseteq \mathbb{R}^d$, where $|D| = n$.

Distance/Similarity Metrics: For any two vectors $u, v \in D$, we denote their distance (or similarity) by $\delta(u, v)$, which can be instantiated by different metrics. The most commonly used measures include Euclidean distance (L_2 norm), inner product, and cosine distance. Specifically, for two d-dimensional vectors $u = (u_1\, u_2 \cdots u_d)$ and $v = (v_1\, v_2 \cdots v_d)$: (1) the **Euclidean distance** between them is $\delta_{\ell_2}(u, v) = \|u - v\|_2 = \sqrt{\sum_{j=1}^{d}(u_j - v_j)^2}$; (2) the **inner product** between them is $\delta_{\text{ip}}(u, v) = \langle u, v\rangle = \sum_{j=1}^{d} u_j v_j$; (3) the **cosine distance** between them is $\delta_{\cos}(u, v) = 1 - \langle u, v\rangle / (\|u\|_2\, \|v\|_2)$, where $\|u\|_2 = \sqrt{\sum_{j=1}^{d} u_j^2}$. Note that the inner product is a *similarity* measure (larger is better), whereas cosine and Euclidean are *distance* measures (smaller is better). For simplicity, unless otherwise specified, we use $\delta(u, v)$ to denote the distance between two vectors u and v, and assume that smaller values indicate higher similarity.

2.1 Similarity Search

Similarity search in vector databases refers to *nearest neighbor (NN) search*.

NN/k-NN: Given a dataset $D \subseteq \mathbb{R}^d$ with $|D| = n$ vectors and a query vector $q \in \mathbb{R}^d$, NN search aims to find the vector $p^\star \in D$ closest to q, i.e., $p^\star = \arg\min_{x \in D} \delta(x, q)$. This naturally extends to *k-NN search*, which returns the k vectors in D with the smallest distances to q.

Exact NN search is often costly in high dimensions and may require scanning a large fraction of the dataset due to the curse of dimensionality [66]. Therefore, similarity search in modern vector databases typically refers to *approximate nearest neighbor (ANN) search*.

ANN/k-ANN: ANN search aims to retrieve vectors that are sufficiently close to q while substantially reducing query cost. The *k-ANN* variant returns a size-k result set $R = \{p_1, \ldots, p_k\} \subseteq D$ that approximates the exact k nearest neighbors.

For modeling and evaluation, we use *recall* to quantify the approximation quality. For a query vector q, let P denote the set of exact k nearest neighbors of q in D. Given a retrieved set $R \subseteq D$, the recall is defined as $\text{recall@}k(R) = |P \cap R|/k$.

2.2 Filtered Similarity Search

Filtered similarity search assumes that each object stored in a vector database is associated not only with a vector representation, but also with (one or more) attributes. Formally, each object $o_i \in O$ is represented as a pair (v_i, a_i), where $v_i \in \mathbb{R}^d$ is a d-dimensional vector and a_i denotes its attribute(s).

A filtered similarity search query likewise consists of two components: a query vector $v_q \in \mathbb{R}^d$ and an attribute predicate p_q. Let $O_{p_q} = \{\, o_i \in O \mid p_q(a_i) \text{ is true} \,\} \subseteq O$ denote the subset of objects whose attributes satisfy p_q. Filtered similarity search aims to perform (approximate) similarity search over the vectors of objects in O_{p_q} with respect to v_q, i.e., it aims to return the k-ANN of v_q from the set $\{v_i \mid o_i = (v_i, a_i) \in O_{p_q}\}$. We use $Sel(p_q)$ to denote the **selectivity** of p_q, i.e., the fraction of objects that satisfy the predicate p_q, such that $Sel(p_q) = |O_{p_q}|/|O|$.

In the literature, beyond works that treat p_q abstractly without committing to a specific predicate form, attributes are commonly modeled as either *numerical* or *categorical*.

Numerical Attributes: Let A_N denote a numerical domain. For each object $o_i = (v_i, a_i)$, the attribute a_i can be either (i) a single value $a_i \in A_N$ or (ii) an closed interval $a_i \subseteq A_N$, i.e., $a_i = [l_i, r_i]$, where $l_i \le r_i$ and $l_i, r_i \in A_N$.

A filtered query specifies, in addition to the query vector v_q, a *constraint* c_q over A_N, which induces a predicate $p_q(\cdot)$ on attributes. We next define several studied variants on numerical attributes.

(1) Range-Filtered ANN (RFANN) [73, 175, 208]: In RFANN, each attribute is a single value $a_i \in A_N$, while the query constraint is a numerical range $c_q \subseteq A_N$. An object $o_i = (v_i, a_i)$ satisfies the filter if a_i is included in the interval of query constraint, i.e., $p_q(a_i) = \left[a_i \in c_q\right]$.

(2) Interval-Filtered ANN (IFANN) [178]: In IFANN, each attribute is an interval $a_i \subseteq A_N$, and the query constraint is also a numerical range $c_q \subseteq A_N$. An object $o_i = (v_i, a_i)$ satisfies the filter if its interval is contained in the query range, i.e., $p_q(a_i) = \left[a_i \subseteq c_q\right]$.

(3) Timestamp-Filtered ANN (TFANN) [159]: In TFANN, each attribute is a time interval $a_i \subseteq A_N$ (where A_N represents a time interval), and the query constraint is a single timestamp $c_q \in A_N$. An object $o_i = (v_i, a_i)$ satisfies the filter if the timestamp falls within the interval, i.e., $p_q(a_i) = \left[c_q \in a_i\right]$.

Categorical Attributes: Let A_C denote a categorical domain. For each object $o_i = (v_i, a_i)$, the attribute is a set of labels $a_i = \{\ell_1, \ell_2, \ldots, \ell_{|a_i|}\}$, where $\ell_j \in A_C$. A query specifies a constraint set $c_q \subseteq A_C$ (e.g., a list of required labels), which induces the predicate $p_q(\cdot)$. The existing studies on categorical attributes assume o_i satisfies the predicate if it contains all required labels, i.e., $p_q(a_i) = \left[c_q \subseteq a_i\right]$.

2.3 Multi-Vector Similarity Search

In multi-vector similarity search, queries can be categorized into two classes depending on whether each object is associated with a *single* vector or *multiple* vectors.

Multi-Vector Objects: This setting assumes that each object $o_i \in O$ is associated with multiple vectors. Existing studies mainly consider two variants.

(1) One-to-One Vector Matching [153, 184, 195]: Each object is represented as $o_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,m})$, and the vectors $v_{i,1}, \ldots, v_{i,m}$ may lie in different embedding spaces and thus may have different dimensionalities (e.g., produced by different embedding models). A query is similarly specified as $q = (q_1, q_2, \ldots, q_m)$, and the query distance (or similarity) is defined by aggregating per-field distances between corresponding vector pairs $(v_{i,j}, q_j)$ for $j \in \{1, \ldots, m\}$, e.g., a weighted sum $\mathrm{dist}(o_i, q) = \sum_{j=1}^{m} w_j\, \delta_j(v_{i,j}, q_j)$, where $\delta_j(\cdot,\cdot)$ denotes the distance (or similarity) measure used in the j-th vector space and w_j is an optional weight. The goal is to return the top-k objects with the smallest (or largest) aggregated value[1], depending on whether distances or similarities are used.

A special case in this variant is **hybrid search** [26], where some of the vectors are a *sparse* vector (e.g., BM25-style term-weight vectors), while the others are *dense* vectors produced by embedding models. Dense vectors are effective at capturing semantic similarity, whereas sparse vectors are well-suited for lexical or keyword matching. Sparse vectors typically have very high dimensionality (often on the order of the vocabulary size) and are therefore mostly zero.

(2) One-to-Many Vector Matching [15, 31]: Each object is represented as a set of vectors $o_i = \{v_{i,1}, v_{i,2}, \ldots, v_{i,m_i}\}$, where m_i may vary across objects, and each $v_{i,j} \in \mathbb{R}^d$ is a embedding. Similarly, a query is represented as $q = \{q_1, q_2, \ldots, q_t\}$, where each $q_\ell \in \mathbb{R}^d$. Unlike one-to-one matching, the vectors in o_i and q are not pre-aligned. Instead, the relevance is computed by fine-grained interactions between the two vector sets. A common formulation is $\mathrm{score}(o_i, q) = \sum_{\ell=1}^{t} \max_{1 \le j \le m_i} s(q_\ell, v_{i,j})$, where $s(\cdot,\cdot)$ is a similarity function, e.g., inner product. That is, each query vector q_ℓ is

[1] If δ_j is a similarity (e.g., inner product), the objective becomes maximizing the aggregated score rather than minimizing $\mathrm{dist}(\cdot,\cdot)$.

matched to its best counterpart in the object, and the final score is obtained by aggregating these best-match similarities. The goal is to return the top-k objects with the largest scores.

Single-Vector Objects: In this setting, each object is represented by a single vector $v_i \in D$, but the query contains multiple vectors $q = [q_1, \ldots, q_m]$. Two representative semantics are *all* and *any*, depending on whether the result must be close to *all* query vectors or to *at least one* of them.

(1) All-k NN [168]: Given a dataset D and a query $q = [q_1, \ldots, q_m]$, define $\mathcal{R}_r^{\forall} = \bigcap_{i=1}^{m} \{ u \in D \mid \delta(u, q_i) \leq r \}$. Let $r^{\star} = \min\{ r \mid |\mathcal{R}_r^{\forall}| \geq k \}$. The all-$k$ NN result is then $\mathcal{R}_{r^{\star}}^{\forall}$.

(2) Any-k NN [168]: Given a dataset D and a query $q = [q_1, \ldots, q_m]$, define $\mathcal{R}_r^{\exists} = \bigcup_{i=1}^{m} \{ u \in D \mid \delta(u, q_i) \leq r \}$, and let $r^{\star} = \min\{ r \mid |\mathcal{R}_r^{\exists}| \geq k \}$. The any-$k$ NN result is $\mathcal{R}_{r^{\star}}^{\exists}$. Both queries can be extended to their approximate counterparts, i.e., all/any-k ANN. As in ANN evaluation, recall (e.g., recall@k) is commonly used to quantify approximation quality.

2.4 Similarity Join

There are two types of similarity join in vector databases: ϵ-similarity join and k-similarity join.

ϵ-Similarity Join: Given two sets of vectors, $X = \{x_1, x_2, \cdots, x_n\}$ and $Y = \{y_1, y_2, \cdots, y_m\}$, in $\mathbb{R}^d$, for $d > 0$, the ϵ-similarity join $X \bowtie_{\epsilon} Y$ is defined as $X \bowtie_{\epsilon} Y = \{(x_k, y_l) \in X \times Y \mid \delta(x_k, y_l) \leq \epsilon\}$, where $\delta(x_k, y_l)$ is the distance metric, e.g., Euclidean distance, between $x_k \in X$ and $y_l \in Y$, and ϵ is a user-given threshold where $\epsilon > 0$. When two given datasets are identical, we refer to the ϵ-similarity join as ϵ-similarity self-join.

k-Similarity Join: The k-similarity join $X \bowtie_k Y$ between two d-dimensional dataset X and Y is defined as $X \bowtie_k Y \triangleq \{(x_i, y_j) \in X \times Y \mid y_j \in TopK(x_i)\}$, where $TopK(x_i)$ is the set of top-k nearest neighbors of x_i in Y. When two given datasets are identical, we refer to the k-similarity join as k-similarity self-join.

3 Similarity Search

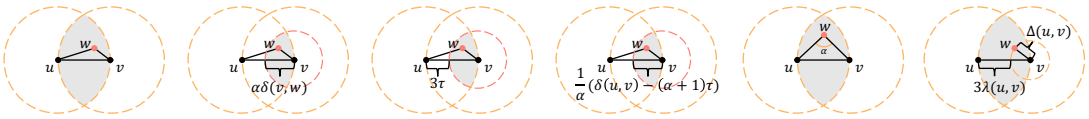
Similarity search aims to find the k-approximate nearest neighbors of a query vector q from a dataset D. According to recent studies [5, 88, 93, 158], among the various types of approaches for similarity search, such as hashing-based methods [44, 65, 101, 107, 145, 146], tree-based methods [12, 82, 118, 140], and others, the state-of-the-art methods are mainly *proximity graph-based* and *quantization-based* approaches. Therefore, in this paper, and in this section in particular, we focus on these two types of approaches.

Beyond these two categories, this section also summarizes three related directions in similarity search: (1) approaches for accelerating distance computation, which can be utilized in both proximity graph-based and quantization-based methods; (2) approaches for addressing hard or out-of-distribution queries, for which it is difficult to find high-quality k-ANN results; and (3) secure k-ANN search, which supports similarity search while preserving privacy.

3.1 Proximity Graph Based Approaches

A proximity graph (PG) $G = (V, E)$ on a dataset D is a directed graph in which each node in V uniquely corresponds to a vector, and two nodes are connected by an edge in E if their corresponding vectors satisfy a certain proximity property.

Beam Search: Given a PG, the standard search algorithm on it is beam search. Starting from one or more entry points, it maintains a candidate pool of the currently best vectors with respect to the query, repeatedly expands the closest unvisited node in the pool, and inserts its neighbors into the pool until no promising unvisited node remains. *Greedy search* is a special case of beam search with beam width 1.



(a) RNG pruning [41, 110] (b) Robust pruning [71] (c) τ-pruning [128] (d) α-CG Pruning [87] (e) α-pruning [181] (f) ALMG pruning [170]

Fig. 2. Different pruning strategies for similarity search

3.1.1 Pruning Strategies. PG-based ANNS algorithms share a common search framework, while existing studies mainly differ in proximity properties on edges, i.e., how graph edges are selected and pruned during index construction. Thus, before reviewing the individual graph indexes, we first summarize the pruning strategies adopted in existing approaches.

k-ANN Pruning Strategy: The simplest pruning strategy is to retain, for each node u, only the nearest k candidates according to their distances, i.e., a top-k nearest-neighbor selection [32]. This strategy is adopted as a basic pruning strategy in existing approaches, whereas other pruning strategies are proposed based on the k-ANN pruning.

RNG Pruning Strategy: A more widely used pruning strategy is RNG pruning, which removes a candidate edge if it is occluded by another shorter edge [41, 59, 110, 128]. Concretely, as shown in Fig. 2(a), for a node u and two candidate neighbors v and w, the edge (u, v) is pruned if $\delta(u, w) < \delta(u, v)$ and $\delta(v, w) < \delta(u, v)$. Intuitively, if there exists a closer neighbor w of u that is also close to v, then (u, v) is redundant, since the search can first move from u to w and then to v.

Parameterized RNG Pruning Strategy: Parameterized RNG pruning augments RNG pruning with extra parameters, yielding several variants.

(1) Robust Pruning [71]. Vamana [71] adopts a parameterized pruning, called *RobustPrune*. As shown in Fig. 2(b), for a node u, it repeatedly keeps the closest candidate neighbor v and prunes each remaining candidate w satisfying $\alpha \cdot \delta(v, w) \leq \delta(u, w)$.

The parameter $\alpha \geq 1$ controls pruning aggressiveness: a larger α preserves more long-range edges and thus helps reduce graph diameter.

(2) Query-Aware Pruning [128]. τ-MG [128]/τ-MNG [128] introduces a threshold parameter τ. As shown in Fig. 2(c), for a node u and a candidate neighbor v, the pruning radius is defined as $r = \delta(u, v) - 3\tau$. Then v is pruned if there exists a neighbor w lying in the intersection of the two balls centered at u and v with radius $\delta(u, v)$ and r, respectively.

(3) α-CG Pruning [87]. α-CG [87] jointly uses α and τ in the pruning radius. As shown in Fig. 2(d), for a node u and a candidate neighbor v, it defines $r = \frac{1}{\alpha}\big(\delta(u, v) - (\alpha + 1)\tau\big)$. Then v is pruned if there exists a selected neighbor w inside the corresponding pruning region induced by u, v, and r.

(4) Pruning at Index Construction [181]. FastPG [181] introduces an α-pruning strategy for neighbor selection during index construction. As shown in Fig. 2(e), for a node u and a candidate neighbor v, the edge (u, v) is pruned if there exists a selected neighbor w such that $delta(u, w) < \delta(u, v), \delta(v, w) < \delta(u, v), \angle uwv > \alpha$. Compared with RNG pruning, this rule further controls pruning by the angle threshold α: when $\alpha = 60°$, it degenerates to RNG pruning; when $\alpha > 60°$, pruning becomes weaker and more edges are preserved.

Distribution-Aware Pruning: Some approaches preserve directionally diverse neighbors.

(1) DPG Pruning [93].

For a node u, DPG selects neighbors that are not only close but also directionally diverse: after sorting candidate neighbors by $\delta(u, v)$, it incrementally keeps candidates whose directions differ sufficiently from those already selected,

Table 1. Summary of proximity graph based approaches for similarity search

Type	Method	Edge	Theoretical Guarantee	
			1-NN	k-NN
KNNG	NN-Descent [1]	Directed	✗	✗
	KGraph [32]	Directed	✗	✗
	EFANNA [39]	Directed	✗	✗
	SPTAG [23]	Directed	✗	✗
	IEH [76]	Directed	✗	✗
	DPG [93]	Undirected	✗	✗
RNG	MRNG/NSG [41]	Directed	for MRNG: $\triangle$ (exact if $\delta(q,\tilde{p}_1)=0$); for NSG: ✗	✗
	FANNG [59]	Directed	$\triangle$ (exact if $\delta(q,\tilde{p}_1)<\tau$)	✗
	SSG/NSSG [40]	Directed	✗	✗
	Vamana/DiskANN [71]	Directed	for Vamana: $\triangle$ (exact if $\delta(q,\tilde{p}_1)=0$; otherwise $\left(\frac{\alpha+1}{\alpha-1}+\epsilon\right)$-ANN); for DiskANN: ✗	✗
	τ-MG/τ-MNG [128]	Directed	for τ-MG: $\triangle$ (exact if $\delta(q,\tilde{p}_1)<\tau$); for τ-MNG:✗	✗
	α-CG/α-CNG [87]	Directed	for α-CG: $\triangle$ (exact if $\delta(q,\tilde{p}_1)<\tau$; otherwise $\left(\frac{\alpha+1}{\alpha-1}+\epsilon\right)$-ANN); for α-CNG: ✗	✗
	LMG/ALMG [170]	Directed	for LMG: ✓; for ALMG ✗	for LMG: ✓; for ALMG ✗
NSWG	NSW [109]	Undirected	✗	✗
	HNSW [110]	Directed	✗	✗
	FastHNSW [181]	Directed	✗	✗
Partition	HCNNG [119]	Directed	✗	✗
	NGT [67]	Directed	✗	✗
	CSPG [177]	Directed	✗	✗

i.e., for selected neighbors $v, w \in N(u)$, the angle $\angle vuw$ should be as large as possible. Thus, DPG approximately minimizes $\{\delta(u,v) \mid v \in N(u)\}$ while maximizing the angular spread of $\{\overrightarrow{uv} \mid v \in N(u)\}$.

(2) NSSG Pruning [40].

NSSG favors candidates whose directions are not covered by the already selected neighbors. For a node u in D and a candidate neighbor v, an edge (u,v) is kept only if there is no other vector inside the cone centered at (u,v) with angular diameter 2α and radius $\delta(u,v)$, i.e., $\text{Cone}(\overrightarrow{uv},\alpha) \cap B(u,\delta(u,v)) \cap D = \emptyset$.

Labeled Pruning: LMG [170] proposes a labeled pruning strategy. Instead of permanently removing an occluded edge, LMG keeps all edges and assigns each edge (u,v) a label $\lambda(u,v)$. Specifically, as shown in Fig. 2(f), if $(u,v) \in E_0$, then $\lambda(u,v)=0$; otherwise, $\lambda(u,v) = (\delta(u,v) - \Delta(u,v))/3, \Delta(u,v) = \min_{w \in N_<(u,v)} \delta(v,w)$, where $N_<(u,v) = \{\, w \mid (u,w) \in E_0 \wedge \delta(u,w) < \delta(u,v) \,\}$,, and E_0 is the edge set based on the RNG pruning strategy. Given a query-dependent threshold $\tau^\star$, the search is conducted on the induced subgraph $E_{\tau^\star} = \{(u,v) \in E \mid \lambda(u,v) \le \tau^\star\}$. In this way, labeled pruning transforms static edge deletion into dynamic query-dependent edge extraction.

3.1.2 Index Design and Search Algorithms. We first summarize the existing PG-based approaches for similarity search.

Taxonomy: Existing approaches for index design and search algorithms can be categorized based on their edge-selection strategies: *KNNG-based*, *RNG-based*, *NSWG-based*, and *Partition-based*. Table 1 summarizes these methods from several perspectives, including their graph index category, whether the index is directed or undirected, whether they provide a guarantee for exact 1-NN search or exact k-NN search, and the pruning strategy adopted in index construction.

(1) KNNG-Based Approaches. A k-nearest neighbor graph (KNNG) connects each node to its k nearest neighbors. Its main advantage is the bounded out-degree, but the graph may have weak global connectivity, which is unfavorable for greedy search [93, 158]. **(1-i)** NN-Descent [1] is a representative early approach for constructing an approximate KNNG, which iteratively refines the graph based on the principle that the neighbors of a node's neighbors are likely to also be its neighbors. **(1-ii)** KGraph [32] is an upgraded version of this line, which improves the update strategy and constructs a larger intermediate neighbor set before returning the final top-k neighbors. **(1-iii)** EFANNA [39] improves KGraph by using multiple KD-trees to obtain better initial candidates before running iteratively neighbor refinement. **(1-iv)** SPTAG [23] adopts a divide-and-conquer strategy: it first recursively partitions the dataset into clusters, builds a local KNNG in each partition, and then merges them into a global graph. **(1-v)** IEH [76] improves KNNG-based search from the query side by using hash-based seeds to initialize the graph traversal, thereby reducing the risk of starting from

poor entry points. **(1-vi)** DPG [93] considers diversifying the angular distribution of neighbors, so that the outgoing edges of a node are spread in different directions rather than concentrated in a small region.

(2) RNG-Based Approaches. RNG is constructed based on KNNG, it removes some edges from KNNG by different pruning strategies to limit the out-degree of each node to a small constant. Except various pruning strategies, they will further enhance their connectivity by adding extra edges between connected components.

Among the RNG methods, the majority of them do not have a theoretical guarantee for finding 1-NN, e.g., NSG [41], SSG [40]. They apply different pruning strategies to limit the out-degree of each node as discussed in Section 3.1.1.

Some existing approaches do have theoretical guarantees for 1-NN but are with constraints. Let $\bar{p}_1$ denote the 1-NN of the query vector q. MRNG [41] and Vamana [71] provide a theoretical guarantee to find 1-NN of a query vector q only if q exists in D (i.e., $\delta(q, \bar{p}_1) = 0$). FANNG [59], τ-MG [128] and α-CG [87] relax the condition but request the 1-NN, $\bar{p}_1$, of a query point q that exists near q in D within a distance less than a predefined parameter τ (denoted as $\delta(q, \bar{p}_1) < \tau$). Their PG-based index is built based on the predefined τ.

Vamana [71] and α-CG [87] further ensure the identification of the $\left(\frac{\alpha+1}{\alpha-1} + \epsilon\right)$-ANN when $\delta(q, \bar{p}_1) > 0$ and $\delta(q, \bar{p}_1) > \tau$ respectively. Here, the term $\left(\frac{\alpha+1}{\alpha-1} + \epsilon\right)$-ANN signifies that the result of algorithm, $\bar{p}'_1$, adhere to the rule that $\delta(\bar{p}'_1, q) \leq \left(\frac{\alpha+1}{\alpha-1} + \epsilon\right) \cdot \delta(\bar{p}_1, q)$. Recently, LMG [170] has been proposed to ensure the finding of both the 1-NN and k-NN without any constraints on $\delta(\bar{p}_1, q)$. Its key idea is to combine the labeled pruning strategy introduced above with a two-phase search algorithm. In the first phase, LMG dynamically activates edges according to their labels and searches the induced subgraph to find 1-NN. In the second phase, it further traverses the neighbors of the found 1-NN to guarantee the exact k-NN result set.

However, although these methods provide theoretical guarantees for exact 1-NN/k-NN search, they generally incur high construction cost, since the guarantees rely on graph properties, which need expensive edge selection. Thus, each of them further proposes a practical version for k-ANN search without theoretical guarantees, such as DiskANN [71], τ-MNG [128], α-CNG [87], and ALMG [170]. The design pattern is to first construct an approximate KNNG and then apply the corresponding pruning strategy to derive the final PG, where the pruning strategies are introduced in Section 3.1.1.

Therefore, while these practical versions inherit the overall construction philosophy of their theoretical counterparts, they trade exact guarantees for significantly lower construction cost and better practical k-ANN performance.

(3) NSWG-Based Approaches. Navigable small-world graph (NSWG) based approaches are motivated by the small-world phenomenon [115], namely that two nodes in a large graph can often be connected by a short path discoverable by greedy routing. Accordingly, they are typically built incrementally, inserting nodes one by one and linking each new node to nearby nodes in the current graph. **(3-i)** NSW [109] is the basic representative of this line. **(3-ii)** HNSW [110] further introduces a hierarchical multilayer structure so that search starts from the upper layers and progressively descends to lower layers, which contain more nodes, for greatly improving search efficiency.

(3-iii) FastHNSW [181] improves HNSW through a layer-by-layer construction. Instead of directly following HNSW's incremental insertion strategy, it first assigns layer assignments to all vectors, then constructs each layer from top to bottom. For each layer, FastHNSW builds an RNG graph index, making the graph in that layer more navigable. The lower layers are further refined by α-pruning strategy introduced before, which improves both search performance and construction efficiency.

(4) Partition-Based Approaches. This line first partitions the dataset and then builds local graphs on the partitions,

Table 2. Summary of approaches for reducing indexing and search cost in proximity graph based ANNS

Type	Method	Index	Search	Platform	Supported Graph Index
Storage	DiskANN [71]	✓	✓	SSD	RNG
	FreshDiskANN [141]	✓	✓	SSD	RNG
	XN-Graph [191]	✓	✓	SSD	XN-Graph
	BAMG [91]	✓	✓	SSD	RNG
	Starling [157]	✓	✓	SSD	No restriction
	PageANN [80]	✓	✓	SSD	Page-aligned graph
	Gorgeous [183]	✓	✓	SSD	No restriction
	PipeANN [55]	✗	✓	SSD	No restriction
	VeloANN [198]	✓	✓	SSD	No restriction
	AiSAQ [147]	✓	✓	SSD	No restriction
	DGAI [105]	✓	✓	SSD	No restriction
	CXL-ANNS [70]	✓	✓	CXL memory	No restriction
	Second-tier Memory [27]	✓	✓	Remote DRAM/NVM	No restriction
GPU	SONG [199]	✗	✓	GPU	NSWG
	GGNN [52]	✓	✓	GPU	No restriction
	GANNS [187]	✓	✓	GPU	KNNG/NSWG
	CAGRA [123]	✓	✓	GPU	No restriction
	Tagore [95]	✓	✗	GPU	RNG
CPU	ParlayANN [111]	✓	✓	CPU	No restriction
	Flash [156]	✓	✗	CPU	NSWG
Algorithm	RNN-Descent [122]	✓	✗	CPU	RNG
	LSH-APG [200]	✓	✓	CPU	No restriction
	Probabilistic Routing [106]	✗	✓	CPU	No restriction
	FastPG [181]	✓	✓	CPU	RNG/NSWG
	DST [74]	✗	✓	CPU/FPGA	No restriction
Learning	LAET [88]	✗	✓	CPU	No restriction
	DARTH [20]	✗	✓	CPU	No restriction

aiming to reduce construction cost and improve search efficiency by exploiting intra-partition locality while preserving global navigability through inter-partition connections.

(4-i) NGT [67] combines tree-based partitioning with neighbor-graph construction: it organizes nearby vectors into local regions via a tree, builds neighbor graphs on the partitions, refines them with a path adjustment to improve cross-region connectivity, and at query time uses the tree to find promising entry points and candidate regions.

(4-ii) HCNNG [119] uses multiple hierarchical clusterings for partitioning; for each cluster, it constructs an MST to obtain a sparse yet connected structure, fuses the induced graphs into a unified compact graph, and at query time uses multiple global KD-trees to generate seeds for guided search on the fused graph.

(4-iii) CSPG [177] partitions the dataset into multiple overlapping subsets, where shared overlap preserves cross-partition reachability, builds a PG on each partition separately, and searches within these local graphs so that fewer vectors are explored than in a single global graph, thereby combining partition-level locality with global connectivity.

3.1.3 Reducing Indexing and Search Cost to Enhance Scalability. A major issue of PG is the high indexing cost. In particular, for many PG indexes, the construction process is tightly coupled with search; for example, inserting a node typically requires repeatedly searching the current graph to find candidate neighbors before pruning or refinement. Thus, indexing and search should not be viewed as two isolated problems: reducing the cost of graph search helps reduce indexing overhead, while better construction pipelines can in turn improve the search efficiency. Recent studies have therefore explored how to reduce the index and search cost,

as summarized in Table 2, we indicate whether each approach accelerates indexing, search, or both, as well as the platform it targets and the type of graph index it supports.

(1) Storage-Based Scalability. One direction improves scalability by leveraging external storage or richer memory hierarchies [24]. **(1-i) Major-in-Disk SSD Indexing.** DiskANN [71] keeps compressed vectors in memory to guide traversal, while storing the full graph and raw vectors on SSD. To scale construction, it first partitions the data into

overlapping clusters, builds a Vamana graph for each cluster, and then merges their edges into a single graph, thereby preserving cross-cluster connectivity. FreshDiskANN [141] extends this design to streaming settings through localized rebuilding. XN-Graph [191] reduces SSD I/O by expanding neighbors before pruning, thereby shortening search paths. BAMG [91] further co-designs graph construction and storage layout through a block-aware monotonic graph, decoupled storage, and block-first traversal, so that each disk read yields more progress. **(1-ii) Layout-Aware and Block-Aware Search.** Starling [157] jointly optimizes disk layout and traversal. It uses an in-memory navigation graph on sampled points to provide query-aware entry points, reorders disk-resident nodes to improve locality, and introduces block search that treats blocks rather than individual vertices as the retrieval unit. PageANN [80] aligns graph search with SSD pages by grouping similar vectors into page nodes and co-designing page layout and traversal around page-level access. Gorgeous [183] instead prioritizes frequently accessed graph topology in memory and augments disk blocks with neighbors' adjacency information to improve locality during traversal. **(1-iii) Overlapping I/O and Computation.** PipeANN [55] observes that the computation-I/O dependency in beam search is often only a pseudo-dependency: the next blocks to fetch can often be inferred from the in-memory frontier before the current iteration fully completes. It therefore pipelines candidate evaluation and SSD fetching, dynamically adjusts beam width, and limits speculative exploration to avoid wasted bandwidth. VeloANN [198] further redesigns the SSD-resident execution engine with hierarchical compression, affinity-based placement, a record-level buffer pool, asynchronous prefetching, and cache-aware beam search, thereby reducing storage stalls and improving throughput under a small memory budget. **(1-iv) All-in-Disk Designs.** AiSAQ [147] advocates an all-in-storage design that offloads compressed vectors to SSD, substantially reducing DRAM usage and enabling DRAM-free search. DGAI [105] explores a decoupled on-disk architecture that separates vectors from adjacency lists, making updates more efficient than in coupled layouts; to reduce the resulting query overhead, it further introduces multi-stage filtering and incremental page-level topological reordering. **(1-v) Richer Memory Hierarchies Beyond SSD.** CXL-ANNS [70] scales graph search through CXL-based memory disaggregation by placing the full graph and vector data in a large CXL memory pool, and reducing far-memory latency via relationship-aware caching and ANNS-aware prefetching. [27] identifies a fundamental dilemma of SSD-based vector indexes: improving throughput often requires substantial index amplification because SSD block accesses are too coarse for graph search. It therefore advocates second-tier memory, such as RDMA- or CXL-connected remote DRAM/NVM, and redesigns graph and cluster indexes around fine-grained remote access.

(2) GPU-Based Acceleration. Another direction exploits the massive parallelism of GPUs. **(2-i)** SONG [199] reuses an NSWG graph and redesigns GPU search by decoupling it into candidate locating, bulk distance computation, and data-structure maintenance, together with GPU-oriented data structures and memory optimizations to reduce overhead. **(2-ii)** GGNN [52] proposes a GPU-friendly graph search structure based on nearest-neighbor graphs with information propagation, and supports both *hierarchical index construction* and query-time search on GPUs. **(2-iii)** GANNS [187] redesigns both PG search and construction for GPUs. It parallelizes search stages such as candidate locating, neighbor exploration, distance computation, and candidate update, and constructs the index by partitioning the data, building local NSWG graphs in parallel, and then progressively merging them. The same framework is also extended to KNNG construction. **(2-iv)** CAGRA [123] proposes a GPU-native PG framework that first builds an approximate high-degree KNNG and then prunes it with a GPU-friendly heuristic to obtain a sparse graph; its search is likewise tailored to GPU execution through parallel candidate evaluation and bulk distance computation. **(2-v)** Tagore [95] targets RNG indexes with a more complete GPU indexing pipeline. It proposes GNN-Descent for GPU-specific KNNG initialization, introduces a unified CFS (Collect-Filter-Store) framework for different pruning strategies, and further supports out-of-memory datasets through an asynchronous GPU-CPU-disk indexing framework with cluster-aware caching.

(3) CPU-Based Acceleration. Recent work revisits PG indexing on modern CPU architectures. **(3-i)** ParlayANN [111] parallelizes graph construction on multi-core CPUs via batch updates, where vectors in the same batch obtain neighbors on a snapshot of the graph and the generated edges are merged in parallel. This improves scalability for indexes such as DiskANN and HNSW, but large batches may hurt index quality by deviating from sequential insertion semantics. **(3-ii)** Flash [156] identifies distance computation as the main bottleneck during graph indexing on modern CPUs. It replaces full-precision distance computation with a SIMD-friendly compact-code approximation: vectors are partitioned into subspaces and encoded so that lookup distances fit into SIMD-register tables. This reduces random memory accesses and enables in-register distance lookup and accumulation, improving both cache efficiency and arithmetic throughput.
(4) Algorithmic Redesign. Several studies reduce indexing/search cost by redesigning graph construction or traversal algorithms. **(4-i)** RNN-Descent [122] accelerates RNG construction by unifying NN-Descent and RNG pruning into a direct construction process. It simultaneously adds candidate neighbors via Relative NN-Descent and removes redundant edges by RNG pruning, thus bypassing the costly KNNG construction used by methods such as NSG, substantially reducing build time while preserving search quality. **(4-ii)** LSH-APG [200] speeds up construction by introducing a lightweight LSH framework into PG indexing. LSH provides better entry points and pruning opportunities, reducing unnecessary distance computations during both insertion and search, though the approximate nature of LSH may trade some final search performance for higher construction efficiency. **(4-iii)** Probabilistic Routing [106] reduces search cost by redesigning neighbor exploration during traversal as a probabilistic routing problem. Given the current node v, a threshold δ, and an error bound ε, any neighbor u within distance δ should be explored with probability at least $1-\varepsilon$. Based on this, it proposes PEOs to identify promising neighbors for exact distance computation while pruning many unpromising ones. This reduces exact distance computations during search without modifying the graph index, and is orthogonal to graph construction. **(4-iv)** FastPG [181] reduces construction cost without sacrificing index quality of both RNG and NSWG construction. For RNG, it replaces the traditional search-before-refinement pipeline with refinement-before-search plus α-pruning to obtain higher-quality candidate neighbors more efficiently. For NSWG, it further replaces node-by-node insertion with layer-by-layer insertion to improve candidate quality during build. **(4-v)** Delayed-synchronization traversal (DST) [74] reduces synchronization overhead in PG search by transforming strictly synchronized beam search into a grouped, block-style traversal. Instead of waiting for all candidates in one beam step to finish before globally re-sorting, DST maintains up to m_g candidate groups, each with up to m_c candidates, and launches new groups non-blockingly as earlier groups complete. This keeps the pipeline full, enables concurrent fetching and evaluation of multiple candidates, improves hardware utilization and throughput.
(5) Learning-Based Early Termination. Another line of work studies whether PG search can stop earlier while still meeting a target quality level. **(5-i)** LAET [88] replaces a fixed beam width with a query-specific stopping point predicted by a GBDT model. After a short initial search, it extracts both static and runtime features from intermediate search state and predicts the minimum search effort needed for the query. **(5-ii)** DARTH [20] extends this to declarative recall. It trains a recall predictor from search-state features and periodically estimates recall during search. The search stops once the predicted recall reaches the user-specified target, with an adaptive prediction interval to reduce inference overhead.

3.1.4 Index Maintenance. As one of the basic indices in vector databases, proximity graphs need to support dynamic maintenance under insertions and deletions.
(1) Out-of-Place Updates. A common design decouples online updates from the base index and periodically rebuilds the graph. ADBV [165, 189] and Milvus [153] follow this approach: newly inserted vectors are buffered in a secondary in-memory index, while deleted vectors are only logically removed via a bitmap; buffered insertions are later merged

into the base index, and logically deleted vectors are physically removed during periodic global rebuilds. This avoids expensive per-update structural modifications, but the rebuild phase dominates the maintenance cost. To reduce this cost, SimJoin [169] leverages approximate similarity join to accelerate rebuilding. It performs a k-similarity join $X \bowtie_k Y$ between the new vectors and the base dataset using the existing graph indexes on both sides to obtain cross-partition candidate neighbors, then combines them with the old graphs before pruning. For deletions, it first marks deleted points and later repairs affected in-neighbors using a sliding-based routine to find replacement neighbors.
(2) In-Place Updates. Another line of work maintains the PG index directly without periodic global rebuilds. **(2-i)** NSWG indexes, such as HNSW [110] and NSW [109], naturally support incremental insertions, but deletions remain challenging. **(2-ii)** FreshDiskANN [141] handles deletions by locally reconnecting the neighbors of the deleted point: it treats them as candidate neighbors for one another and reapplies pruning to restore connectivity. This avoids full reconstruction, though quality may degrade after long update sequences. **(2-iii)** IP-DiskANN [173] observes that only out-neighbors are stored, so it first approximately reconstructs the deleted points' in-neighbors, and then repairs each affected out-neighbor using only a restricted nearby candidate set rather than exhaustive reconnection. This substantially reduces repair costs. **(2-iv)** Wolverine++ [100] improves deletion repair by expanding the candidate space from 1-hop to 2-hop neighbors, since replacement edges may not exist among direct neighbors alone. To control the enlarged search space, it first filters candidates, then performs distance computation and pruning. **(2-v)** PRO-HNSW [75] repairs HNSW after updates through three lightweight modules: RemoveObsoleteEdges, which removes edges to deleted nodes while preserving a minimum number of alive outgoing neighbors; RepairDisconnectedNodes, which reconnects disconnected components via limited-hop BFS and available neighbor slots; and ResolveUnidirectionalEdges, which scans level 0 for missing reciprocal edges and inserts reverse links when possible.

3.1.5 Parameter Tuning. The performance of PG-based approaches is highly sensitive to hyperparameters, such as the degree limit, construction beam width, and search beam width, since these parameters jointly determine the trade-off between index cost and search performance. A straightforward way to tune them is Grid Search or Random Search. **(1)** Grid search [98] evaluates configurations on a predefined grid, while **(2)** Random search [13] samples directly from the search space and is more efficient when only a few parameters are truly important. However, both still require repeatedly building and evaluating indexes, which is costly. To reduce this cost, **(3)** VDTuner [182] formulates tuning as a constrained optimization problem under a target recall and uses constrained Bayesian optimization to iteratively recommend promising configurations, substantially reducing the number of trials. **(4)** PGTuner [34] targets automatic and transferable tuning for PGs by combining a pre-trained query-performance predictor with a deep reinforcement learning-based recommendation model, and further supports efficient retuning through out-of-distribution detection and deep active learning. **(5)** FastPGT [204] observes that the main bottleneck lies in repeatedly constructing and evaluating PGs, rather than recommending parameters. It therefore builds multiple PGs simultaneously, shares repeated computations across configurations, and recommends multiple promising configurations for joint evaluation, thereby accelerating tuning without sacrificing quality.

3.2 Quantization-Based Approaches

Quantization-based ANN methods compress each high-dimensional vector into a short discrete code, reducing memory and accelerating distance estimation at the cost of approximation error.
Taxonomy: Existing approaches are commonly divided into scalar quantization (SQ) and product quantization (PQ).

Table 3. Summary of quantization-based ANN approaches

Category	Method	Sub-family	Codebook unit	Combination	Bits/dim	Learned?	Theory?
SQ	SQ4 / SQ8 [33]	Uniform SQ	Scalar (per dim)	Independent	4 or 8	✗	✗
	RaBitQ [47]	Non-uniform SQ	Scalar (per dim)	Independent	1	✗	✓
	RaBitQ+ [45]	Non-uniform SQ	Scalar (per dim)	Independent	b (multi-bit)	✗	✓
	LVQ [2]	Non-uniform SQ	Scalar (per dim)	Independent	4 or 8	✗	✗
	SAQ [89]	Non-uniform SQ	Scalar (per dim)	Independent	Variable	✗	✗
PQ	PQ [72]	Vanilla PQ	Sub-vector (d/m dims)	Concatenation	$\sim 8m/d$	✗	✗
	FastScan [4]	Vanilla PQ	Sub-vector (d/m dims)	Concatenation	$\sim 8m/d$	✗	✗
	OPQ [48]	Vanilla PQ	Sub-vector (d/m dims)	Concatenation	$\sim 8m/d$	✓	✗
	LOPQ [78]	Vanilla PQ	Sub-vector (d/m dims)	Concatenation	$\sim 8m/d$	✓	✗
	ScaNN [56]	Learned PQ	Sub-vector (d/m dims)	Concatenation	$\sim 8m/d$	✓	✗
	JPQ [190]	Learned PQ	Sub-vector (d/m dims)	Concatenation	$\sim 8m/d$	✓	✗
	UNQ / LSQ [112, 117]	Learned PQ	Sub-vector (d/m dims)	Concatenation	$\sim 8m/d$	✓	✗
	AQ [8]	Other PQ	Full vector (d dims)	Addition	Variable	✓	✗
	TQ [10]	Other PQ	Full vector (tree-assigned)	Addition	Variable	✓	✗
Hybrid	IVFPQ [72]	Inverted file + PQ	Sub-vector (d/m dims)	Concatenation	$\sim 8m/d$	✗	✗
	IMI [9]	Product IVF	Sub-vector (d/m dims)	Concatenation	$\sim 8m/d$	✗	✗
	IVF-HNSW [11]	Graph-assisted IVF	Sub-vector (d/m dims)	Concatenation	$\sim 8m/d$	✗	✗
	DiskANN [71]	Graph + PQ on disk	Sub-vector (d/m dims)	Concatenation	32	✗	✗
	SVS [2]	Graph + LVQ	Scalar (per dim)	Independent	4 or 8	✗	✗
	SLB [102]	Plugin lower bound	Sub-vector (d/m dims)	Concatenation	$\sim 8m/d$	✗	✗
	SymphonyQG [51]	Graph + RaBitQ	Scalar (per dim)	Independent	1 (+ refinement)	✗	✓
	IVF-RaBitQ [138]	GPU IVF + RaBitQ	Scalar (per dim)	Independent	Variable	✗	✓

SQ quantizes each dimension independently into a b-bit integer. PQ quantizes groups of dimensions jointly using multi-dimensional codewords; in the standard form, the vector is partitioned into disjoint subspaces and the approximation is formed by concatenating sub-code indices, while more general variants relax the disjoint-subspace constraint and combine multiple codewords additively.

Table 3 summarizes quantization-based ANN methods from several perspectives, including the granularity of the codebook unit, the way in which multiple codewords are combined for vector reconstruction, whether the quantizer is learned via gradient-based optimization beyond k-means, and whether the method comes with a provable distance estimation error bound.

(1) Scalar Quantization (SQ). SQ quantizes each coordinate independently to a b-bit integer, so a compressed vector is simply a short integer array. Approximate distances are then computed directly on the quantized coordinates, often with SIMD-friendly integer arithmetic. The central design question is how the floating-point range of each dimension is mapped to discrete bins.

(i) Uniform SQ. Uniform SQ partitions each dimension range into 2^b equal-width bins and stores the bin index. SQ4 and SQ8 in FAISS [33] are canonical implementations using 4-bit and 8-bit integers. Their appeal is simplicity: encoding is a linear rescaling, decoding is a multiply-add, and storage drops by about 8× for SQ4 or 4× for SQ8 relative to float32. Their weakness is that one global range per dimension must serve all vectors, which can cause severe distortion when the data distribution is skewed or contains outliers.

(ii) Non-Uniform SQ. Non-uniform SQ adapts binning or scaling to the data distribution and usually yields lower error at the same bit budget. **LVQ** [2] is designed for graph-based ANN search and modern SIMD hardware. It subtracts the global mean and then performs per-vector scaling before scalar quantization, so each vector is quantized within its own centered dynamic range. This local scaling substantially reduces the distortion caused by a single global range and can also be used in a two-level residual scheme.

RaBitQ [47] takes a different route. After normalizing vectors onto the unit hypersphere, it uses a randomly rotated binary codebook derived from the vertices of a hypercube. This yields an unbiased inner-product estimator together

with an $O(1/\sqrt{d})$ error bound, making it notable for providing a sharp theoretical guarantee. **RaBitQ+** [45] extends the idea from 1 bit to multi-bit quantization by replacing the binary grid with a shifted integer grid, again followed by normalization and random rotation. The resulting scheme offers a tunable trade-off between code length and estimation error and can reach a higher recall than 1-bit RaBitQ without relying as heavily on re-ranking.

SAQ [89] targets two practical issues of SQ: expensive encoding and uniform bit allocation across dimensions. It first generates a scalar code and then refines it by coordinate-wise code adjustment to improve directional alignment with the original vector. After PCA, it groups dimensions by variance and uses dynamic programming to allocate more bits to higher-variance segments. Its multi-stage estimator can terminate early after evaluating only the most informative segments.

(2) Product Quantization (PQ): PQ quantizes groups of dimensions jointly, which lets it capture cross-dimensional structure missed by SQ. In the common multi-codebook setting, a vector is encoded by m code indices from m codebooks, producing an effective codebook of size k^m while keeping storage and lookup cost linear in m. We group PQ methods into vanilla PQ, learned PQ, and more general additive or full-space variants.

(i) Vanilla PQ. Vanilla PQ partitions a d-dimensional vector into m disjoint subspaces of dimension d/m, trains one K-Means codebook per subspace, and encodes the vector by concatenating the m centroid indices. The key runtime primitive is asymmetric distance computation (ADC): for a query, distances to all k codewords in each subspace are precomputed into m lookup tables, and the approximate distance to any database code is obtained by summing one entry from each table in $O(m)$ time. **PQ** [72] establishes this framework, but its axis-aligned partition is often suboptimal when the data covariance is not aligned with the coordinate axes. **OPQ** [48] addresses this by learning an orthogonal rotation that better aligns the data with the subspace decomposition, thereby reducing inter-subspace correlation and balancing distortion. **LOPQ** [78] further localizes this idea by learning a separate rotation and product quantizer inside each coarse IVF cell, yielding lower distortion on residual vectors at the price of a more tightly coupled two-stage system. **FastScan** [4] keeps the PQ model unchanged but accelerates ADC by compressing lookup tables into SIMD-register-sized structures for in-register access.

(ii) Learned PQ. Learned PQ preserves the subspace structure but replaces pure K-Means training with gradient-based optimization. **ScaNN** [56] introduces anisotropic quantization loss for inner-product search, emphasizing errors that matter more for ranking. **JPQ** [190] jointly trains a dense encoder and PQ embeddings with a ranking-oriented loss so that the encoder produces representations more compatible with PQ. **UNQ** [117] learns a universal codebook with both reconstruction and task loss to obtain quantization-friendly embeddings, while **LSQ** [112] focuses on faster code assignment by casting additive quantization encoding as a least-squares problem, reducing beam-search encoding cost.

(iii) Other PQ Methods.

Some methods allow codewords to cover the full vector and combine them additively. **AQ** [8] represents a vector as the sum of m full-dimensional codewords from jointly optimized codebooks. This can reduce distortion relative to vanilla PQ at the same rate, but finding the best code combination is NP-hard and is handled approximately with beam search. **TQ** [10] places codebooks on the vertices of a coding tree and assigns dimensions to edges, producing an additive representation that interpolates between flat subspace PQ and full additive quantization.

(3) Combinations with Indexes: At a billion scale, quantization alone is rarely enough: even scanning compressed codes exhaustively is too slow, and retrieval quality remains limited by quantization distortion. Many systems therefore combine quantization with an index that narrows the candidate set before approximate distance evaluation. In these hybrids, quantization mainly serves one of two roles: it is either the primary distance-computation primitive, or it is a lightweight signal used to guide graph or list traversal.

(i) Distance Computation Primitive. In methods such as **IVFPQ** [72], **IMI** [9], and **IVF-HNSW** [11], the stored compressed code is the object on which most distance evaluations are performed. IVFPQ combines an inverted file partition with PQ-compressed residuals and searches only the top-w coarse cells. IMI replaces the flat coarse quantizer with a product coarse partition, generating K^2 implicit cells from two subspace codebooks while storing only $2K$ centroids. IVF-HNSW accelerates coarse assignment by organizing centroids in an HNSW graph rather than scanning them exhaustively; inside each selected cell, PQ residual codes are still compared via ADC.
(ii) Compressed Codes Guide Graph Traversal. In **DiskANN** [71], **SLB** [102], and **SVS** [2], compressed codes mainly help navigation, while final distances are obtained more accurately later. DiskANN stores the graph and full vectors on SSD but keeps PQ codes in memory, using them to rank candidate neighbors during traversal before fetching only a small final shortlist for exact re-ranking. SLB reinterprets ADC as a statistical lower bound on the true distance and uses it as a plugin pruning rule on top of existing indexes. SVS combines a Vamana graph with LVQ-compressed in-memory vectors; LVQ reduces traversal distortion, and Turbo LVQ improves SIMD decoding by permuting dimensions to better match AVX-512 layouts.
(iii) Tightly Co-design. A more recent direction co-designs quantization and the index together. **SymphonyQG** [51] replaces PQ with RaBitQ, exploits RaBitQ's error bounds for implicit candidate filtering, and aligns graph out-degree with FastScan's SIMD batch size to reduce wasted computation. **IVF-RaBitQ** [138] integrates IVF and RaBitQ into a GPU-oriented pipeline that explicitly targets device-memory limits, parallel execution, and memory-access efficiency.

3.3 Distance Computation Acceleration

Distance computation is the dominant source of query latency in k-ANN search. Therefore, accelerating distance computation is an important direction that can benefit both proximity graph-based and quantization-based methods. Existing approaches mainly fall into two categories [163]: (i) *approximate distance computation*, which replaces exact distance computation with a cheaper approximation; and (ii) *pruning-based computation*, which first computes a cheap estimate or lower bound and only evaluates the exact distance when necessary.
(1) Approximate Distance Computation. A common approach is to approximate distances using compressed or transformed vector representations. Quantization-based methods, such as product quantization [56, 72], reduce the cost of distance computations by representing vectors with compact codes and computing approximate distances through lookup tables or codeword comparisons. Quicker ADC [3, 4] improves the efficiency of code-based distance evaluation by optimizing asymmetric distance computation for PQ with SIMD instructions. Hashing-based methods, e.g., [154], provide another way to accelerate distance computation by mapping vectors into binary codes, so that distance estimation can be performed efficiently in the hash space.
(2) Pruning Based Computation. Exploiting lower bounds [102] studies another way to accelerate distance computation, which is by first computing a cheap estimate or lower bound on the distance between a query and a candidate vector. For each candidate, this bound is evaluated before the exact distance. If the bound is already worse than the current top-k threshold, the candidate can be pruned; otherwise, its exact distance is computed.

(2-i) ADSampling [46] proposes a randomized distance comparison algorithm based on dimension sampling. It progressively samples dimensions to estimate the distance and stops early once the estimate is sufficient for reliable comparison, enabling sublinear-time distance comparison with high probability. **(2-ii)** DADE [29] improves this idea by approximating distances in a data-aware lower-dimensional space rather than randomly sampling original dimensions. It uses an unbiased estimator together with adaptive hypothesis testing, so that only the necessary amount of distance computation is performed.

(2-iii) BSA [179] proposes replacing the random projection used in ADSampling with a data-aware orthogonal projection, so that the approximate distance can better fit the data distribution. It also decouples distance correction from distance approximation through a data-driven correction mechanism, which improves the tightness of the estimation. **(2-iv)** Tribase [174] enhances cluster-based search by refining cluster granularity and combining distance-based and angle-based triangle pruning at multiple levels, so that many candidates can be safely filtered before exact evaluation. **(2-v)** TRIM [143]

strengthens this line of pruning by optimizing landmark vectors used to form the triangles and introducing relaxed lower bounds whose tightness can be tuned according to user needs.

3.4 Hard and Out-of-Distribution Queries

An important issue in similarity search is robustness to *hard* queries, i.e., queries for which standard ANN indexes require substantially more effort to reach the same recall. Such hard queries are often related to out-of-distribution (OOD) settings, where the query distribution differs from the data distribution used to construct the index. For instance, in tasks like image-text retrieval, image and text embeddings are generated by separate encoders. In existing studies [22, 64, 69], to quantify these distribution disparities, the Wasserstein distance [81] is employed to mathematically measure the distinctions between the two distributions, and the Mahalanobis distance [108] is utilized to measure the distance from a vector to a distribution. A vector q is considered out-of-distribution if its Mahalanobis distance $D_M(q, X)$ to the base dataset X significantly differs from the distances between base vectors [22, 69], e.g., $D_M(x_i, X)$ for $x_i \in X$.

Taxonomy: Existing approaches for hard and OOD queries can be categorized into two groups: one focuses on estimating query hardness to distinguish hard queries from easy ones, while the other focuses on designing index or search algorithms to better handle such hard or OOD queries.

Query Hardness Estimation: Existing methods for query hardness estimation mainly fall into two categories: (1) *distribution-based* measures and (2) *graph-native* measures.

(1) Distribution-Based Measures. A commonly used measure is the local intrinsic dimensionality (LID) [6]. Given the distances from a query q to its k nearest neighbors, $d_1(q) \le \cdots \le d_k(q)$, LID is estimated as $\widehat{\mathrm{LID}}_k(q) = -\left(\frac{1}{k}\sum_{i=1}^{k}\log\frac{d_i(q)}{d_k(q)}\right)^{-1}$. A larger LID means neighbor distances expand faster around q, making nearby candidates less distinguishable and ANN search harder. [6] also discusses *query expansion* (QE) and *local relative contrast* (LRC). For a dataset D, query q, and $k' > k$, QE is defined as $\mathrm{QE}_{k,k'}(q) = d_{k'}(q)/d_k(q)$, where $d_k(q)$ and $d_{k'}(q)$ are the distances from q to its k-th and k'-th nearest neighbors. A larger QE indicates slower neighbor growth and usually an easier query. LRC is defined as $\mathrm{LRC}_k(q) = d_{\mathrm{mean}}(q)/d_k(q)$, where $d_{\mathrm{mean}}(q)$ is the average distance from q to all vectors in D. A larger LRC means the nearest neighbors are more distinguishable from average vectors and the query tends to be easier.

(2) Graph-Native Measures. These approaches estimate query hardness from the graph index used for ANN search. *Steiner-hardness* [162] models hardness from graph connectivity. Let N_k be the k-NN of query q, and let $d_k = D(q, n_k)$ be the distance from q to its k-th nearest neighbor. Given a recall requirement $Acc \in [0, 1]$, a probability lower bound $p \in (0, 1]$, and a threshold τ, it defines a constrained minimum-effort subgraph $Y = (V_Y, E_Y)$ such that some $N'_k \subseteq N_k$ with $|N'_k| \ge p \cdot k$ is sufficiently connected within distance $(1+\tau)d_k$. To reflect greedy-search cost, it defines the decision set $DS(v)$ and the decision cost $Cost(Y) = |\bigcup_{v\in Y} DS(v)|$, and then defines Steiner-hardness as the minimum such cost at the critical threshold τ_0: $\mathrm{SH}(q) = ME^{p}_{\tau_0@Acc\text{-exh}}(q)$. This formulation can be reduced to a Directed Steiner Tree problem for efficient computation. *Escape Hardness* [63] instead models hardness from graph reachability. Let

$G_t(q)$ be the neighboring subgraph induced by the top-t nearest neighbors of q. The Escape Hardness from u to v is $EH_q(u \to v) = \min\{t \mid u \rightsquigarrow v \text{ in } G_t(q)\}$, namely, the smallest NN-rank radius needed before v becomes reachable from u. These values form an Escape Hardness matrix for the query. The method computes them by constructing $G_t(q)$ and then applying Floyd-Warshall [38] to derive all-pairs reachability/distances.

Out-of-Distribution Query Processing: OOD query processing aims to improve ANN performance when the query distribution differs from the indexed data distribution. Existing approaches mainly fall into two categories: *OOD-aware index construction* and *search-time graph rectification*.

(1) OOD-Aware Index Construction. RobustVamana [69] improves graph-based ANN search by incorporating historical OOD queries during index construction: it inserts these queries into the graph as auxiliary navigators so that the index better reflects the query distribution. RoarGraph [22] improves this design by avoiding permanent query-node insertion. It first builds a bipartite graph between base vectors and historical OOD queries, then eliminates query nodes via a neighbor-aware projection: for each base node, it gathers candidates from the adjacent query nodes' base-side neighbors and projects back at most M neighbors using distance-aware sorting and RNG pruning, thereby preserving the neighbor relationships revealed by OOD queries.

(2) Search-Time Graph Rectification. NGFix [63] repairs defective graph regions at search time based on Escape Hardness. Its NGFix component repairs the neighboring graph around a historical query and greedily inserts edges to reduce the Escape Hardness. Its RFix component focuses on the first search stage: if greedy search from the entry point fails to reach the vicinity of a historical query, it adds edges from the nearest point found so far to nodes closer to the query.

3.5 Secure Similarity Search

Secure similarity search aims to support k-NN/ANN queries over cloud or distributed environments without revealing sensitive information about the data or queries. Existing solutions mainly fall into two paradigms: privacy-preserving ANN search, where encrypted data are hosted on a (semi-honest) cloud server, and federated secure similarity search, where data remain decentralized across multiple parties and queries are answered collaboratively.

Privacy-Preserving ANN Search (PP-ANNS): PP-ANNS methods [42, 43, 137, 166, 202, 203] typically encrypt vectors and build auxiliary indexes over ciphertexts to improve efficiency, often by adapting existing ANN indexes. Proximity graphs are used in [104, 203], while locality-sensitive hashing is adopted in [129, 137]. Depending on whether distance comparison is supported over ciphertexts, existing methods can be divided into three categories.

(1) Distance-Incomparable Encryption.

Traditional encryption schemes such as AES and DES [142] ensure strong confidentiality but do not support distance comparison over ciphertexts. Thus, the server can only return a set of encrypted candidates, and the user must decrypt them locally to compute exact distances. This paradigm incurs substantial communication overhead and limits the utilization of cloud computation.

(2) Distance-Comparable Encryption. Distance-comparable encryption enables direct distance comparison on encrypted vectors. Representative approaches include homomorphic encryption (HE) [54, 201], asymmetric matrix encryption (AME) [202], asymmetric scalar-product-preserving encryption (ASPE) [166, 197, 206], and distance-comparison-preserving encryption (DCPE) [42]. These schemes offer different trade-offs: HE and AME are computationally expensive; ASPE has $O(d)$ complexity but leaks some privacy [104]; DCPE supports only approximate comparison; and the recent distance comparison encryption (DCE) [104] supports exact comparison in $O(d)$ time.

(3) Client-Assisted Secure Search. Another line of work offloads graph traversal and distance evaluation to the client while preserving access privacy via primitives such as private information retrieval (PIR) and oblivious RAM (ORAM) [28,

Table 4. Summary of filtered similarity search approaches

Type	Method	Graph	Graph index type	Quantization	Platform
General	Milvus [153]	✓	Unrestricted	✓	CPU/GPU
	ADBV [165, 189]	✓	Unrestricted	✓	CPU
	VBASE [195]	✓	Unrestricted	✓	CPU
	ACORN [127]	✓	HNSW	✗	CPU
	NHQ [155]	✓	Unrestricted	✗	CPU
	NaviX [136]	✓	HNSW	✓	CPU/Disk
	JAG [172]	✓	Unrestricted	✗	CPU
	UniFilter [171]	✓	Unrestricted	✗	CPU
	SIEVE [94]	✓	Unrestricted	✗	CPU
Categorical	FilteredVamana/StitchedVamana [50]	✓	Vamana	✗	CPU/Disk
	UNG [19]	✓	Unrestricted	✗	CPU
	ELI [180]	✓	HNSW	✗	CPU
	VecFlow [167]	✓	Unrestricted	✗	GPU
Numerical	SeRF [208]	✓	HNSW	✗	CPU
	HSIG [97]	✓	HNSW	✗	CPU
	SuperPostfiltering [35]	✓	Unrestricted	✗	CPU
	iRangeGraph [175]	✓	HNSW	✗	CPU
	DIGRA [73]	✓	HNSW	✗	CPU
	WoW [164]	✓	Unrestricted	✗	CPU
	RangePQ [192]	✗	N/A	✓	CPU
	Dynamic Range-Filtering ANNS [130]	✓	Unrestricted	✗	CPU
	Hi-PNG [178]	✓	Unrestricted	✗	CPU
	TS-Graph [159]	✓	Unrestricted	✗	CPU

49]. For example, PACMANN [203] lets the client privately retrieve local subgraphs and compute distances locally, avoiding expensive server-side cryptographic distance computation.

Federated Secure Similarity Search: Federated approaches assume that data are distributed across multiple parties and cannot be centralized. **(i)** FedKNN [196] supports secure federated k-NN search by letting each provider compute local results and using a secure central aggregator based on trusted execution environments and oblivious primitives. It further proposes DANN for adaptive workload allocation, and its secure variant DANN* provides (ϵ, δ)-differential obliviousness.

(ii) FedSQ [207] improves the efficiency-accuracy trade-off by selectively probing only a subset of participants using a cost-aware framework, together with sampling and pruning for approximate global top-k search. **(iii)** FedVSE [36] supports both k-NN and hybrid queries by combining trusted execution environments with vector and learned indexes. It uses a two-stage pipeline in which local candidates are first generated at each party and then securely aggregated within enclaves. **(iv)** FedVS [37] further studies federated vector search with attribute filters. It adopts a TEE-based two-phase framework for filter-aware local refinement and secure global top-k aggregation, together with cost-aware scheduling and adaptive allocation to reduce load imbalance and unnecessary computation.

4 Filtered Similarity Search

Filtered similarity search (also called *filtered* vector search) augments ANN search with constraints over structured attributes. Each object $o_i \in O$ is represented as (v_i, a_i), where $v_i \in \mathbb{R}^d$ is the embedding vector and a_i denotes its attribute(s). A filtered query is specified as $q = (v_q, p_q)$, where $v_q \in \mathbb{R}^d$ is the query vector and c_q associating with p_q is an attribute constraint (e.g., a label set, a numeric range, or an interval/timestamp condition), which induces a predicate $p_q(a_i)$. Let $O_{c_q} = \{o_i = (v_i, a_i) \in O \mid p_q(a_i) \text{ is true}\}$ be the valid subset. Filtered k-ANN then returns a size-k set from $\{v_i \mid o_i \in O_{c_q}\}$ that is approximately closest to v_q. In this section, we denote k-ANN search without predicates as $c_q = \emptyset$.

Taxonomy. Existing approaches for filtered similarity search can be broadly categorized into *universal* and *dedicated* index approaches. Universal approaches aim to support arbitrary attributes and predicates using a single (or lightly modified) vector index, whereas dedicated approaches build specialized index structures tailored to a specific attribute

type and predicate family. Table 4 summarizes these approaches from several perspectives, including whether they support general filters or focus specifically on categorical or numerical attributes, whether they are based on proximity-graph indexes, quantization-based indexes, or both, whether they impose restrictions on the underlying graph index type, and the target platform they are designed for.

4.1 Universal Index Approaches

Universal index approaches support filtered k-ANN queries $q = (v_q, p_q)$ with arbitrary attributes and predicates by reusing a single ANN index and making minimal changes to query processing or the index structure. Existing methods mainly fall into four categories: (i) *search-algorithm-based* methods, which keep the index unchanged and only modify query execution; (ii) *index-augmentation-based* methods, which modify or augment a PG/HNSW-style index to improve reachability under filtering; (iii) *index-introduction-based* methods, which introduce an auxiliary predicate graph to form an enhanced PG that supports both $c_q = \emptyset$ and $c_q \neq \emptyset$ efficiently; and (iv) *workload-aware* methods, which build a collection of indexes and use an analytical model to choose the best one at query time.

(1) Search Algorithms: This category keeps the index unchanged and incorporates the predicate only at query time.
(1-i) Pre-Filtering and Post-Filtering: Milvus [153] and ADBV [165, 189] propose two strategies: (1) *pre-filtering*, which first materializes the valid subset $\mathcal{O}_{p_q}$ and then obtain the k-ANN restricted to $\mathcal{O}_{p_q}$ by brute-force scanning; and (2) *post-filtering*, which first retrieves k' candidates ($k' > k$) by ANN search on v_q and then filters them by c_q to obtain k valid results. Post-filtering is simple but may require a large k' when $Sel(p_q)$ is small or when predicate satisfaction is poorly aligned with the embedding neighbors in G.
(1-ii) Two-Phase Execution: VBASE [195] proposes a search algorithm based on relaxed monotonicity. It unfolds in two phases: it first navigates toward vectors close to v_q while disregarding c_q, and then iteratively expands the search frontier while taking c_q into account, so that the algorithm can continue exploring beyond the initial candidates when they contain insufficient valid results.
(2) Index Augmentation on PG: This category modifies/augments the PG index so that the search can traverse predicate-satisfying nodes more effectively and reduce early terminations caused by disconnected induced subgraphs.
(2-i) 2-Hop Augmentation. **(a)** ACORN [127] starts from a PG G built for $c_q = \emptyset$ and builds an augmented graph $\mathbf{G}$ by adding 2-hop connectivity: if (v_i, v_j) and (v_j, v_k) are edges in G, then v_k can be treated as an additional neighbor of v_i in $\mathbf{G}$. ACORN provides two variants, ACORN-γ and ACORN-1, which incorporate 2-hop information during graph construction and during query processing, respectively. Given $\mathbf{G}$, ACORN answers $q = (v_q, p_q)$ by traversing only nodes satisfying c_q, leveraging the augmented connectivity to reduce search failures. **(b)** NaviX [136] further proposes an adaptive 2-hop exploration strategy. It considers three heuristics: onehop-s, which explores only selected 1-hop neighbors of the current candidate; blind, which continues to explore 2-hop neighbors in the default scanning order of 1-hop neighbors until enough selected nodes are examined; and directed, which also explores 2-hop neighbors but first orders the 1-hop neighbors by their distances to the query vector v_q, biasing the search toward regions closer to v_q. Based on the local selectivity around the current candidate, NaviX adaptively chooses among these heuristics at each iteration, thereby improving robustness across different selectivities.
(2-ii) Composite Index and Joint Pruning. **(a)** NHQ [155] argues that treating filtering and vector search as two separate steps, i.e., pre-/post-filtering, is not a good approach. It therefore proposes to build a composite index over the proximity graph and performs joint pruning during traversal, where beam expansion considers both vector proximity to v_q and feasibility under c_q to prune unpromising branches early. To further improve navigability, NHQ introduces two navigable proximity graphs (NPGs) as indices, revising edge selection and routing to mitigate local traps caused by

nearby nodes violating c_q. With NPGs, the composite-index and joint-pruning traversal becomes more constraint-aware in search, reducing wasted expansions on infeasible neighbors. **(b)** JAG [172] improves robustness across different selectivities and filter types by building a graph that jointly captures vector proximity and attribute proximity. It introduces attribute and filter distances, turning a binary constraint into a continuous navigational signal. By optimizing the graph with respect to both vector similarity and attribute proximity, JAG reduces navigational dead-ends under filtering and aims to perform consistently across diverse predicates, such as label, range, subset, and Boolean filters.
(3) Index Introduction: UniFilter [171] differs from prior universal approaches by introducing an auxiliary predicate-aware AND/OR graph and integrating it with the PG built for $c_q = \emptyset$ into a unified index. It constructs an AND graph for categorical attributes and an OR graph for numerical attributes, where each predicate node is associated with the objects satisfying the corresponding predicate. Given a query predicate p_q, UniFilter finds a minimal coverage set of predicate nodes covering O_{p_q}, and links them to vector nodes through carefully selected cross-edges to form an enhanced PG. This enables traversal to move between predicate space and vector space, so that when invalid vector nodes are reached, the search can jump to regions containing valid objects, reducing wasted exploration. Since the original PG is preserved, the same index efficiently supports both unfiltered and filtered k-ANN queries.
(4) Workload-Aware Indices: SIEVE [94] builds a collection of indexes, each tailored to different predicate forms or selectivity ranges, motivated by the observation that a single graph may fail to preserve navigability for all filtered queries. To construct this collection under a memory budget, it proposes a three-dimensional analytical model capturing the relationships among index size, search time, and recall. It uses this model to guide index selection and parameterization jointly. At query time, the same model is used to dynamically choose both the index and its search parameters that provide the fastest search for the target recall.

4.2 Dedicated Index Approaches

Dedicated approaches build specialized indices for a given predicate family and can substantially improve search efficiency, but they often do so at the cost of higher index complexity and larger space or maintenance overhead.
Categorical Attributes: Dedicated index approaches for categorical attributes assume that $a_i \subseteq A_C$, $c_q \subseteq A_C$, and an object $o_i = (v_i, a_i)$ satisfies the predicate if $c_q \subseteq a_i$.
(1) Label-Aware PG Construction. Filtered-DiskANN [50] proposes two Vamana-based indices for categorical filtering. *FilteredVamana* incrementally inserts vectors and biases neighbor discovery and edge selection toward vectors with compatible labels, so that predicate-satisfying nodes remain well connected in the resulting graph. *StitchedVamana* first builds a separate Vamana graph for each label filter and then merges these graphs into a single index by stitching their edges together, followed by pruning to bound the out-degree. At query time, both methods perform beam search while restricting traversal to nodes satisfying the query predicate.
(2) Multiple PGs with a Navigating Graph. UNG [19] builds a PG G_f for each distinct label set f appearing in the dataset, where each object is maintained in the PG corresponding to its own label set. In addition, it builds a navigating graph G_N over these label sets, where an edge connects a label set to its minimum superset. Given a query predicate c_q, UNG first identifies the minimal supersets of c_q in G_N, and then searches the corresponding PGs. In this way, it bridges label containment and vector proximity through an explicit graph structure over categorical predicates.
(3) Elastic Index Selection.

ELI [180] selectively indexes only a subset of label sets to support all queries. Its key observation is that an index built for a label set L can also serve any query label set L_q with $L \subseteq L_q$, and the extra query cost is captured by an *elastic factor*: the ratio between the numbers of vectors matching L and L_q. Based on this, ELI formulates index selection as a

constrained optimization problem under space and query-efficiency requirements, and proposes greedy algorithms to decide which label sets should be indexed. Since the selected indexes are built on top of an underlying ANN structure, ELI is index-agnostic and offers a flexible space-performance trade-off.

(4) GPU-Oriented Hybrid Indexing. VecFlow [167] targets categorical filtered search on GPUs. Starting from an IVF-based index, it partitions posting lists according to their label distribution patterns. For groups with highly selective labels, VecFlow builds GPU-friendly graph indexes to accelerate candidate retrieval and avoid exhaustive scans over long posting lists; for the remaining groups, it falls back to brute-force scans over the posting lists. This design is particularly effective for long-tailed filters, where a few labels are frequent while many others are highly selective.

Numerical Attributes: Range Filters. Range filters assume that $a_i \in A_N$, $c_q = [l_q, r_q] \subseteq A_N$, and an object $o_i = (v_i, a_i)$ satisfies the predicate if $a_i \in c_q$.

(1) Range-Dedicated Index Decomposition. **(i)** SeRF [208] builds a segment graph by sorting objects according to their numerical attributes and constructing compressed graph indexes for a large collection of ranges. Given a query range, SeRF identifies the corresponding pre-built range structures and searches on them, so that the traversal is restricted to vectors whose attributes fall into the desired range. However, its compression is lossy, and the index is designed for static data. **(ii)** iRangeGraph [175] improves this by organizing the numerical domain into a segment tree. Each tree node corresponds to a range, and query processing decomposes c_q into $O(\log |A_N|)$ tree nodes and searches the corresponding indexes. Compared with SeRF, iRangeGraph offers a more systematic range decomposition and avoids building indexes for all possible ranges. **(iii)** SuperPostfiltering [35] follows a related idea by defining multiple overlapping ranges, building one index for each range, and answering a query by choosing a covering range and then applying post-filtering within that range-specific index. **(iv)** DIGRA [73] extends the range-decomposition idea toward dynamic settings. Built on a B-tree style organization of numerical attributes, it supports insertions and deletions while preserving efficient query processing on range-specific graph indexes. **(v)** WoW [164] targets dynamic settings by maintaining hierarchical range-to-range indexes of different sizes. A query is answered by combining multiple ranges that cover p_q, while updates only affect a bounded number of maintained range indexes.

(2) Unified Multi-Strategy Frameworks. UNIFY [97] supports pre-filtering, post-filtering, and hybrid filtering within a single PG. Its core structure, Segmented Inclusive Graph (SIG), partitions data by attribute values and guarantees that PG induced by any combination of segments is a subgraph of SIG, so that a query can reconstruct or directly search the relevant subgraph for its range. Its hierarchical variant HSIG incorporates an HNSW hierarchy to reduce filtering complexity while preserving support for different strategies within one unified index.

(3) Maintenance-Oriented Methods. Beyond DIGRA [73] and WoW [164] discussed before, there are also other methods explicitly targeting dynamic maintenance. RangePQ [192] represents a quantization-based alternative to graph-based RFANN indexes. Instead of building multiple graph structures for ranges, it uses quantization to support range-filtered search with improved space efficiency and maintenance support. For streaming scenarios, Dynamic Range-Filtering ANNS [130] further studies compact dynamic indexing by combining range-aware compression with labeled edges, so as to support many query ranges efficiently under continuous arrivals.

Numerical Attributes: Interval and Timestamp Filters. Numerical attributes in existing studies also include two variants.

(1) Interval Filtering. Interval filtering assumes that $a_i, c_q \subseteq A_N$, and an object $o_i = (v_i, a_i)$ satisfies the predicate p_q if $a_i \subseteq c_q$. Hi-PNG [178] addresses this setting by transforming each object interval $[l_i, r_i]$ into a point (l_i, r_i) in a two-dimensional space, and transforming the query interval $[l_q, r_q]$ into a query rectangle. It then builds a hierarchical interval-partition navigating graph over this transformed space, so that a query can first identify the relevant interval

Table 5. Summary of multi-vector similarity search approaches

Method	Graph	Quantization	Dense	Sparse	Multi-modality	Platform
Milvus [153]	✓	✓	✓	✓	✓	CPU/GPU
VBASE [195]	✓	✓	✓	✗	✓	CPU
DEG [184]	✓	✗	✓	✗	✓	CPU
FCVI [60]	✓	✓	✓	✗	✗	CPU
RadiusSearch [168]	✓	✓	✓	✗	✗	CPU
MUVERA [31]	✗	✗	✓	✗	✗	CPU
PLAID [134]	✗	✗	✓	✗	✗	CPU
ColBERTv2 [135]	✗	✗	✓	✗	✗	CPU/GPU
EMVB [120]	✗	✓	✓	✗	✗	CPU
IGP [15]	✓	✗	✓	✗	✗	CPU
OneSparse [26]	✗	✓	✓	✓	✓	CPU
AdaptiveSearch [194]	✓	✗	✓	✓	✓	CPU
Allan-Poe [96]	✓	✗	✓	✓	✓	GPU

partitions and then search the corresponding graph structures. In this way, Hi-PNG bridges interval containment and vector proximity through a hierarchical graph index.

(2) Timestamp Filtering. Timestamp filtering assumes that $a_i \subseteq A_N$, $c_q \in A_N$, and an object $o_i = (v_i, a_i)$ satisfies the predicate p_q if $c_q \in a_i$. TSANN [159] proposes a *timestamp graph* that maintains a unified graph index for all historical timestamps, instead of building one separate index for each timestamp. To preserve graph connectivity when vectors expire over time, it introduces backup neighbors during updates, and further compresses the historical neighbor information using a historic neighbor tree. As a result, TSANN supports timestamp-aware ANN search together with efficient updates and substantially reduced index space.

5 Multi-Vector Similarity Search

Multi-vector similarity search extends standard vector search from a single vector representation to multiple vectors. As defined earlier, existing studies mainly involve two basic matching patterns: *one-to-one vector matching*, where corresponding vectors are aggregated field by field, and *one-to-many vector matching*, where the final relevance is computed by fine-grained interactions between two vector sets.

Existing studies fall into four categories: (i) *multi-dense one-to-one vector search*, where each object is associated with multiple dense vectors and the final score is obtained by aggregating similarities of corresponding vector pairs; (ii) *multi-dense one-to-many vector search*, where each object and query are represented as sets of dense vectors, and the final relevance is computed by set-to-set interaction; (iii) *hybrid search*, where dense semantic embeddings are combined with sparse lexical vectors or attribute-aware vectors; and (iv) *one-modality multi-vector search*, where each object is associated with a single vector while the query contains multiple vectors, and the goal is to find objects that are jointly close to all or any of the query vectors. Table 5 summarizes these approaches from several perspectives, including whether they support proximity graph indexes or quantization-based indexes, whether they handle dense vectors or sparse vectors, whether they support vectors from different modalities, and the target platform they are designed for.

5.1 Multi-Dense Vector Search

(1) Multi-Dense One-to-One Vector Search: Multi-dense one-to-one vector search assumes that each object is associated with multiple dense vectors, and the final query distance (or similarity) is obtained by aggregating the distances between corresponding vector pairs. Existing approaches can be grouped into four categories as follows.

(1-i) Iterative Candidate Expansion: A straightforward way to support aggregated scoring is to search each vector space separately and then aggregate the candidate union. Milvus [153] follows this idea by iteratively expanding the per-vector candidate sets: it first retrieves the top-k' results for each vector, aggregates over their union, and then enlarges k' until

the final top-k no longer changes. This design is general and easy to deploy on top of existing ANN indexes, but it may incur repeated search overhead when small per-vector candidate sets do not adequately cover the aggregated top-k.
(1-ii) Multi-Index Traversal with Guidance: VBASE [195] implements the NRA algorithm natively on the ANN index scan operator, enabling multiple vector indexes to be traversed incrementally, so that it avoids repeatedly enlarging k' and restarting the search. To optimize traversal order, VBASE proceeds in rounds and maintains both local and global signals: a local priority queue stores results from the previous round to indicate which index is currently producing better candidates, while a global priority queue maintains overall candidates together with their aggregated scores. It tracks the average score of traversed entities for each index as a global quality signal, such that it can allocate more traversals in each round to high-quality indexes, while still visiting low-quality ones at least once to avoid local optima.
(1-iii) Two Dense Vector Case: For the special case of two dense vectors with a weighted aggregate score, DEG [184] proposes a dedicated graph index. Its key idea is to treat the two constituent distances as two objectives and use a Greedy Pareto Frontier Search algorithm to compute, for each node, a candidate neighbor set that covers approximate nearest neighbors under different weighted values. Based on this candidate set, DEG performs dynamic edge pruning: it assigns each retained edge an active range, indicating the interval of weighted values for which the edge should be used. At query time, given a query-specific weight, DEG extracts the searchable subgraph by keeping only edges whose active ranges contain the weight, and then performs graph search on this subgraph.
(1-iv) Attribute-as-Vector: FCVI [60] can be viewed as a special case of one-to-one multi-vector search. Each object is represented as (v_i, f_i) and each query as (q, F_q), where v_i, q are embeddings and f_i, F_q are filter vectors. Its objective jointly minimizes the distance between v_i and q while maximizing the similarity between f_i and F_q. FCVI applies a geometric transformation $\psi(v_i, f_i, \alpha)$ that injects the filter vector into the original space without changing dimensionality, moving vectors with similar filter values closer while largely preserving the original semantic structure. The transformed query is handled in the same way: reducing the two-vector search problem to ANN search on transformed vectors.
(2) Multi-Dense One-to-Many Vector Search: Multi-dense one-to-many vector search assumes that each object and query are associated with sets of dense vectors, and the final relevance is computed through interactions between two vector sets. One approach is to convert this problem into ANN-based searches. **(i)** MUVERA [31] transforms the problem into a single-vector retrieval problem by constructing fixed-dimensional encodings for queries and objects, so that off-the-shelf MIPS solvers can be used for candidate generation followed by exact re-ranking. Beyond single-vector proxies, some methods focus on optimizing late-interaction search more directly: **(ii)** PLAID [134] introduces centroid interaction and centroid pruning to filter low-quality candidates early, while **(iii)** ColBERTv2 [135] reduces the storage footprint of token-level representations through aggressive residual compression. **(iv)** EMVB [120] improves this by combining bit-vector pre-filtering with product quantization to accelerate late interaction and reduce memory usage,

and **(v)** IGP [15] builds a proximity graph for MaxSim-style retrieval and uses incremental greedy probing to obtain high-quality candidates with fewer probes. **(vi)** GEM [148] proposes a native graph-based index. It reduces redundancy by assigning each vector set only to the clusters of its most informative vectors, and builds a dual graph with intra-cluster graphs for local neighbor structure and a global graph for cross-cluster navigation. At query time, GEM performs a multi-entry beam search from multiple promising clusters while pruning paths that drift into irrelevant regions.
(3) One-Modality Multi-Vector Search: One-modality multi-vector search assumes that each object is represented by a single vector, while the query contains multiple vectors from the same modality. The goal is to retrieve objects close to all/any of the query vectors. RadiusSearch [168] addresses this setting by replacing the single-query distance in graph traversal with an all/any-radius, so that search is guided by multiple query vectors simultaneously.

RadiusSearch+ [168] further improves this: for any-k, it starts from 1-ANNs of all query vectors to cover multiple promising regions; for all-k, it first finds 1-ANN of an intersection vector,

thereby shortening the search path and improving efficiency.

5.2 Dense-Sparse Combined Similarity Search

Dense-sparse combined similarity search jointly uses dense vectors for semantic matching and sparse vectors for lexical matching. Existing approaches fall into two categories: unified graph indexes and multiple-index methods.

(1) Unified Graph Index: (i) [194] studies dense-sparse hybrid vectors using a single PG. To improve effectiveness, it introduces distribution alignment, which pre-samples dense and sparse vectors and analyzes their distance distributions so that the graph can better preserve the combined neighbor structure. To improve efficiency, it adopts an adaptive two-stage strategy: search first computes dense distances only, and computes the full hybrid distance involving the sparse part only for promising candidates; it further applies sparse-vector pruning to reduce the cost of sparse-distance evaluation. **(ii)** Allan-Poe [96] extends dense-sparse retrieval to a more general hybrid search setting on GPUs.

It builds a unified graph-based index with isolated heterogeneous edge storage, so that dense, sparse, and full-text retrieval paths are maintained in one graph while remaining separately addressable. To support efficient construction, it designs a GPU indexing pipeline with a warp-level hybrid distance kernel, RNG-IP joint pruning, and keyword-aware neighbor recycling, enabling the graph to capture hybrid neighbor structure efficiently. At query time, Allan-Poe employs a dynamic fusion framework that supports arbitrary combinations of retrieval paths and weights without reconstructing the index, and can further leverage logical edges derived from a knowledge graph for more complex queries.

(2) Multiple Indices: OneSparse [26] proposes a unified multi-index that combines an inverted index for sparse vectors with a quantization-based index for dense vectors. Instead of letting each index finish its own top-k' search and intersecting the results afterwards, OneSparse first narrows the search to matched posting lists and performs inter-index intersection before final scoring and ranking. To make this efficient, it reorders both sparse and dense posting lists by document IDs, so that candidates skipped in one index can be pruned early in the other during joint traversal.

6 Similarity Join

Similarity join is a fundamental operator for vector data management. Given two vector sets X and Y, an ϵ-similarity join returns all vector pairs $(x, y) \in X \times Y$ such that $\delta(x, y) \leq \epsilon$, while a k-NN join returns, for each vector in one set, the k nearest vectors in the other set.

Taxonomy. In the literature, similarity join methods can be broadly categorized into *exact* and *approximate* methods. Exact methods guarantee complete and correct join results, but their cost grows rapidly on high-dimensional dense vectors and large-scale datasets. Approximate methods trade recall for efficiency, and are thus more aligned with modern vector database workloads.

6.1 Exact Similarity Join

Many works study high-dimensional similarity joins, including ϵ-similarity join [16, 18, 25, 68, 79, 84, 113, 131, 139] and k-NN join [17, 150–152, 185]. However, the notion of "high dimensionality" has changed substantially over time. Many early studies [17, 84, 139, 185] treated tens of dimensions (e.g., around 40) as high-dimensional, whereas modern vector databases often manage hundreds or even thousands of dimensions. These exact methods rely on spatial or metric indexes, such as R-trees [18], metric-space indexes [68], and HDR tree and forest [150, 151], to prune non-promising

node pairs before verifying actual distances. Their high-level idea is to recursively join index nodes and use bounding regions, pivot-based lower bounds, or triangle inequality pruning to avoid comparing all vector pairs. [152] further proposes cluster-based batch update and pruning over HDR-Tree. Such methods can be effective in low or moderate dimensions, but their pruning power weakens rapidly as dimensionality grows, because bounding regions become less selective and index traversal incurs non-trivial overhead.

For exact high-dimensional dense vectors, the dominant paradigm is *filter-and-refine* (or sort-and-refine) [16, 25, 79, 131]. The filtering phase identifies candidate vector pairs that may satisfy the distance predicate, while the refinement phase computes exact distances only for the filtered candidates.

A representative line is the Epsilon Grid Order (EGO) family [16, 79]. Given a vector $p = (p_1, \ldots, p_d)$ and a threshold ϵ, EGO overlays a regular grid of side length ϵ and maps p to its cell coordinate $g(p) = (\lfloor p_1/\epsilon \rfloor, \ldots, \lfloor p_d/\epsilon \rfloor)$. Vectors are then ordered lexicographically by their cell coordinates: for two vectors p and q, $p <_{ego} q$ iff there exists a smallest dimension i such that $\lfloor p_i/\epsilon \rfloor < \lfloor q_i/\epsilon \rfloor$ and $\lfloor p_j/\epsilon \rfloor = \lfloor q_j/\epsilon \rfloor$ for all $j < i$. Under this order, every true join mate of p must lie within the ϵ-interval $[p - (\epsilon, \ldots, \epsilon)^T,\ p + (\epsilon, \ldots, \epsilon)^T]$ in the ordered file; equivalently, if $q <_{ego} p - (\epsilon, \ldots, \epsilon)^T$ or $p + (\epsilon, \ldots, \epsilon)^T <_{ego} q$, then $\|p - q\|_2 > \epsilon$, so (p, q) cannot be a join pair. Thus, EGO compares only pairs whose cell coordinates are sufficiently close in the induced order [16]. For two EGO-ordered sequences $P = \langle p_1, \ldots, p_k \rangle$ and $Q = \langle q_1, \ldots, q_m \rangle$, EGO defines the *active* dimension of a sequence as the first dimension i such that $\lfloor p_{1,i}/\epsilon \rfloor < \lfloor p_{k,i}/\epsilon \rfloor$ and $\lfloor p_{1,j}/\epsilon \rfloor = \lfloor p_{k,j}/\epsilon \rfloor$ for all $j < i$; dimensions $j < i$ are *inactive*. These enable recursive pruning: if P and Q share an inactive dimension j with $\left|\lfloor p_{1,j}/\epsilon \rfloor - \lfloor q_{1,j}/\epsilon \rfloor\right| \geq 2$, then the two sequences cannot contain any join pair and need not be recursively compared [16].

Super-EGO [79] improves this line as follows: it reorders dimensions so that more discriminative ones are examined earlier, uses more aggressive sequence-level pruning without materializing full boxes, and computes partial squared distances $S_t(a, b) = \sum_{\ell=1}^{t} (a_{\pi(\ell)} - b_{\pi(\ell)})^2$ under a dimension order π chosen to maximize early termination, aborting once $S_t(a, b) > \epsilon^2$. It also proposes to parallelize the processing of multiple sequence pairs.

FGF-Hilbert [131] improves EGO-based approaches from a cache perspective. It reorders candidate refinement according to a Fast General Form Hilbert space-filling curve. The key idea is that candidate pairs with nearby cell identifiers tend to access nearby memory regions. Processing them in a locality-preserving order substantially improves cache behavior across the memory hierarchy, thus reducing memory stalls during refinement.

DiskJoin [25] revisits exact similarity join approaches at billion scale. Unlike prior in-memory methods, DiskJoin targets SSD-resident vector datasets on a single machine. Its design reduces repetitive SSD accesses and read amplification through carefully scheduled data access, uses main memory as a dynamic cache with explicit cache-eviction management, and further applies probabilistic pruning to eliminate many vector pairs before exact distance evaluation.

6.2 Approximate Similarity Join

Approximate similarity join is more aligned with modern vector database workloads, since exact joins become prohibitively expensive as dimensionality and data scale grow. Existing approximate methods fall into two categories: (1) selection-based methods, which view similarity join as a collection of independent ϵ-range queries, i.e., for each vector in one dataset, search the other dataset for vectors within distance ϵ and union all returned pairs as the final join result; and (2) reuse-aware methods, which go beyond this view by reusing intermediate results across nearby queries.

(1) Selection-Based Method: This line can be further divided into four categories. (1-i) LSH-Based Methods. One major family is based on locality-sensitive hashing (LSH) [7, 90, 132, 186, 188]. At a high level, these methods first project each vector to one or multiple hash values so that nearby vectors collide with high probability, then treat the

join as an equi-join over hash buckets where colliding pairs are taken as candidates, and finally verify them by exact distance computation to remove false positives. In practice, they often employ multiple hash tables, concatenated hash functions, or multi-probe/collision-counting techniques to balance recall and candidate size.

(1-ii) Graph-Based Methods. HSJ [133] proposes an external strategy that repeatedly issues k-ANN queries with adaptively increased k until the threshold condition is covered. For each vector x, it initializes $k_{\text{init}} = ef + (1-\tau) \cdot ef$, runs k_{init}-ANN search on HNSW, and lets y_k be the least similar vector in the returned top-k set. If $\text{sim}(x, y_k) \geq \tau$, the current result may still be truncated before covering the full threshold answer, so it enlarges k as $k \leftarrow k + \frac{(1-\tau)\cdot k}{1-\text{sim}(x,y_k)}$ and reissues the ANN search. It stops only when $\text{sim}(x, y_k) < \tau$, and then verifies the returned set to keep vectors with similarity at least τ. (1-iii) Two-Stage Search. VBASE [195] supports approximate ϵ-range queries on ANN indexes through a two-stage framework based on relaxed monotonicity. It first performs a 1-ANN search to obtain a seed close to the query, and then incrementally expands from this seed while filtering candidates by the threshold predicate ϵ. Next, VBASE supports approximate similarity join through this ϵ-range search operator.

(1-iv) Learning-Based Pruning. Xling [160] strengthens independent-query approximate joins with a learned filter. It trains a regression-based metric-space filter to predict whether a query has enough ϵ-neighbors, and skips queries unlikely to contribute sufficient join results, thereby reducing unnecessary neighbor searches. Xling is engine-agnostic and can be plugged into loop-based similarity join methods.

(2) Reuse-Aware Join Processing: This line exploits the fact that nearby query vectors often have highly overlapping join results or similar traversal behaviors. **(2-i)** SimJoin [169] observes that similarity join is not merely a bag of independent range queries, because nearby query vectors often share highly overlapping join results. It defines the join window of a vector x_i as $J_i = \{y \in Y \mid \delta(x_i, y) \leq \epsilon\}$ and reuses results from already processed vectors. Its first key idea is join window sliding, which incrementally slides from the join window of one vector to that of an adjacent vector instead of restarting search from scratch; since exact adjacency is unavailable in high dimensions, SimJoin uses a proximity graph as an approximate adjacent graph. Its second key idea is join window order selection, which models transition costs between vectors as a weighted graph and chooses a low-cost processing order by solving an optimization problem. SimJoin also extends this framework to support k-similarity join. **(2-ii)** ES+MI+Adapt [83] extends SimJoin from output reuse to deeper traversal reuse. Its first idea is soft work sharing: instead of caching all in-range join results of a processed query, it caches only the closest visited data point, even if it is out of range, so that similar later queries can still reuse traversal effort under small thresholds while keeping the cache compact. Its third idea is an adaptive hybrid BFS-BestFS traversal for OOD queries: unlike SimJoin's BFS, which expands only in-range points, it keeps a bounded number of out-range points in the queue, allowing traversal to cross out-range walls and discover disconnected in-range regions. This hybrid mode is enabled only when a query is predicted to be OOD using a local distance-ratio heuristic, thereby improving recall without always incurring the extra traversal overhead.

7 Future Directions and Open Challenges

In this section, we aim to discuss prevalent research trends addressing various challenges in this domain and offer perspectives on potential future research opportunities.

(1) Toward Unified Query Processing Frameworks. A key future direction is to move toward unified vector query processing along two directions. (i) Unify processing across different query types in vector databases. As reviewed in this paper, similarity search, filtered similarity search, multi-vector similarity search, and similarity join are still largely supported by separate index designs, search strategies, and pruning mechanisms; even when some methods begin to blur these boundaries, they are usually limited to specific settings rather than a general framework. As a result,

practical workloads that combine vector similarity, structured predicates, multiple dense or sparse representations, and join matching still rely on multiple disconnected techniques. A natural direction is therefore to investigate whether one high-performance index, or one family of tightly integrated indexes, can support multiple query types efficiently.

This also motivates a declarative, non-procedural interface, similar to SQL, for expressing diverse vector queries uniformly, together with unified optimizer abstractions and cost models for comparing alternative execution strategies. (ii) Integrate vector query processing with traditional database systems. It can be achieved by treating vectors as native data types rather than external objects handled outside the database engine. Most existing approaches are still developed over vector datasets in isolation, so vector search remains largely separated from structured attributes, relations, and standard relational operators. A promising direction is thus to support vectors as first-class data types in general-purpose DBMSs, potentially through SQL-like interfaces with native vector predicates and operators, so that vector retrieval can be jointly optimized with filters, joins, grouping, and aggregation. This further raises several research directions, including how vector predicates should be represented in the query model, e.g., as approximate selections, top-k operators, or a new class of similarity-aware joins, how optimizers should reason jointly about vector retrieval and structured query processing, and how statistics and cost models should capture not only vector search cost but also its interaction with filters, joins, and aggregations.

(2) Bridging the Gap between Theoretical Guarantees and Practical Performance. Another important future direction is to bridge the gap between theoretical guarantees and practical performance along two aspects. (i) The gap between provable guarantees and state-of-the-art index design. As reviewed in this paper, graph indexes are among the most effective methods for similarity search in practice, yet most high-performance graph indexes are heuristic. Although several graph-based methods, such as MRNG, Vamana, FANNG, τ-MG, α-CG, and LMG, provide theoretical guarantees, these guarantees often rely on additional assumptions or expensive graph properties, and usually incur high construction cost due to costly edge selection and graph refinement. A natural direction is therefore to revisit graph index design by jointly balancing three objectives: theoretical guarantees, practical query performance, and index construction cost. Instead of directly adopting expensive exact theoretical constructions, future work may investigate approximate or relaxed constructions that preserve the most important provable properties while remaining practical to build, for example, through new pruning rules or multi-stage construction pipelines. More broadly, progress in this direction may also provide useful theoretical foundations for other query types reviewed in this paper, such as filtered similarity search, multi-vector similarity search, and similarity join. (ii) The gap between benchmark-based evaluation and theory-driven assessment. Existing methods are still mainly compared through empirical trade-offs on selected datasets and benchmark suites, using metrics such as recall, QPS, latency, and memory. While such evaluation is useful, it is highly data-dependent, and conclusions may vary across datasets, query distributions, and implementation details, making it difficult to understand the intrinsic strengths and weaknesses of an index without extensive experiments. Although some early attempts, such as query-hardness measures and analytical models, move toward more analytical characterization, they remain local to specific tasks. A promising direction is thus to develop theory-driven or structure-aware evaluation methods that complement benchmarking, for example, by studying whether graph properties, navigability measures, robustness indicators, or approximation bounds can serve as more direct predictors of query performance. Such metrics would not replace empirical evaluation, but could provide a more principled basis for comparing vector index designs.

(3) New Query Types and Workloads Triggered by LLMs, RAG, and Agentic Systems. LLMs, RAG, and agentic systems introduce new vector query types and workloads that go beyond the assumptions of current vector databases

in at least two aspects. (i) Multi-query workflows. Most existing methods assume that queries are issued and processed largely independently. In contrast, agentic systems often generate a sequence of correlated subqueries [85], whose later queries depend on earlier retrieval results, intermediate reasoning states, or tool outputs. Processing them independently may lead to substantial redundant work, while current vector query engines still lack a principled framework for multiple-query optimization (MQO), despite MQO having long been studied in traditional databases. A natural direction is therefore to move from query-at-a-time execution to workload-aware optimization over query sets or streams, for example, by generalizing reuse-aware ideas from similarity join so that traversal states and intermediate results can be shared across correlated agent-generated subqueries, and by revisiting MQO in the vector setting at the levels of candidate generation and graph traversal. This may further motivate new declarative abstractions for agentic retrieval workloads, where a vector database optimizes an entire retrieval pipeline rather than a single query.

(ii) Reliability, privacy, and security. When vector retrieval becomes part of a larger LLM or agentic workflow, retrieval must satisfy stronger reliability requirements, such as uncertainty detection, failure control, and abstention or clarification when retrieval is unreliable [21], while also handling privacy and security risks, including sensitive retrieved context, access-pattern leakage, and unsafe output exposure. Future work may therefore investigate secure and federated vector query processing frameworks that better control leakage, support filtered secure search, and scale to interactive workloads, as well as more robust query planning and execution strategies that can dynamically adjust search effort, trigger verification, or defer to safer fallback strategies when uncertainty is high.

8 Conclusion

Vector databases have become an important foundation for modern AI and data-intensive applications, such as RAG and agentic systems, where query processing over high-dimensional vectors is no longer limited to classical ANN search. To provide a structured understanding of this rapidly evolving area, this paper presents a comprehensive survey on query processing techniques in vector databases.

We first introduce the preliminaries and formalize four query types studied in vector databases, namely similarity search, filtered similarity search, multi-vector similarity search, and similarity join. We then review similarity search, which is the core primitive in vector databases. In particular, we summarize the two dominant lines of approaches, i.e., proximity graph and quantization, and further discuss several closely related directions, including distance computation acceleration, hard and out-of-distribution query processing, and secure similarity search. Next, we survey filtered similarity search, where vector retrieval is combined with structured predicates. We organize existing studies into universal and dedicated index approaches, and discuss how different methods support categorical, numerical, interval, and timestamp filters. We further review multi-vector similarity search, where either objects or queries are represented by multiple vectors. We summarize one-to-one and one-to-many matching for multi-vector objects, as well as one-modality multi-vector search and dense-sparse combined search, showing how richer matching semantics lead to new indexing and traversal designs beyond standard ANN search. After that, we survey similarity join, which extends vector retrieval from point queries to large-scale pairwise matching, by reviewing both exact and approximate methods. Finally, we discuss the research focus and the existing research gaps in query processing in vector databases, and identify several future directions and open challenges.

References

[1] 2011. NNDES: A Library for Efficient K-NN Graph Construction. https://code.google.com/archive/p/nndes/.

[2] Cecilia Aguerrebere, Mark Hildebrand, Ishwar Singh Bhati, Theodore Willke, and Mariano Tepper. 2024. Locally-adaptive quantization for

streaming vector search. *arXiv preprint arXiv:2402.02044* (2024).

[3] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2021. Quicker ADC : Unlocking the Hidden Potential of Product Quantization With SIMD. *IEEE Trans. Pattern Anal. Mach. Intell.* 43, 5 (2021), 1666–1677.

[4] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2016. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. In *42nd International Conference on Very Large Data Bases*, Vol. 9. 12.

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

[6] Martin Aumüller and Matteo Ceccarello. 2021. The role of local dimensionality measures in benchmarking nearest neighbor search. *Inf. Syst.* 101 (2021), 101807.

[7] Martin Aumüller and Matteo Ceccarello. 2022. Implementing Distributed Similarity Joins using Locality Sensitive Hashing. In *EDBT 2022*. OpenProceedings.org, 1:78–1:90.

[8] Artem Babenko and Victor Lempitsky. 2014. Additive quantization for extreme vector compression. In *CVPR*. 931–938.

[9] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence* 37, 6 (2014), 1247–1260.

[10] Artem Babenko and Victor Lempitsky. 2015. Tree quantization for large-scale similarity search and classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4240–4248.

[11] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 202–216.

[12] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree: An Index Structure for High-Dimensional Data. In *VLDB*. 28–39.

[13] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* (Mar 2012).

[14] Thierry Bertin-Mahieux, Daniel P. W. Ellis, Brian Whitman, and Paul Lamere. 2011. The Million Song Dataset. In *Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR 2011*. University of Miami, 591–596.

[15] Zheng Bian, Man Lung Yiu, and Bo Tang. 2025. IGP: Efficient Multi-Vector Retrieval via Proximity Graph Index. In *SIGIR*. ACM, 2524–2533.

[16] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. 2001. Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. ACM, 379–388.

[17] Christian Böhm and Florian Krebs. 2004. The k-Nearest Neighbour Join: Turbo Charging the KDD Process. *Knowl. Inf. Syst.* 6, 6 (2004), 728–749.

[18] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient Processing of Spatial Joins Using R-Trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*. ACM Press, 237–246.

[19] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. 2024. Navigating Labels and Vectors: A Unified Approach to Filtered Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 6 (2024), 246:1–246:27.

[20] Manos Chatzakis, Yannis Papakonstantinou, and Themis Palpanas. 2025. DARTH: Declarative Recall Through Early Termination for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 4 (2025), 242:1–242:26.

[21] Kaiwen Chen, Yueting Chen, Nick Koudas, and Xiaohui Yu. 2025. Reliable Text-to-SQL with Adaptive Abstention. *Proc. ACM Manag. Data* 3, 1 (2025), 69:1–69:30.

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

[23] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. SPTAG: A library for fast approximate nearest neighbor search. https://github.com/Microsoft/SPTAG.

[24] Xiaoyu Chen, Jinxiu Qu, Yitong Song, Shuhang Lu, Huiling Li, Minghui Jiang, Wei Zhou, Jianliang Xu, Xuanhe Zhou, and Fan Wu. 2026. Disk-Resident Graph ANN Search: An Experimental Evaluation.

[25] Yanqi Chen, Xiao Yan, Alexandra Meliou, and Eric Lo. 2025. DiskJoin: Large-scale Vector Similarity Join with SSD. *Proc. ACM Manag. Data* 3, 6 (2025), 1–27.

[26] Yaoqi Chen, Ruicheng Zheng, Qi Chen, Shuotao Xu, Qianxi Zhang, Xue Wu, Weihao Han, Hua Yuan, Mingqin Li, Yujing Wang, Jason Li, Fan Yang, Hao Sun, Weiwei Deng, Feng Sun, Qi Zhang, and Mao Yang. 2024. OneSparse: A Unified System for Multi-index Vector Search. In *WWW*. ACM, 393–402.

[27] Rongxin Cheng, Yifan Peng, Xingda Wei, Hongrui Xie, Rong Chen, Sijie Shen, and Haibo Chen. 2024. Characterizing the Dilemma of Performance and Index Size in Billion-Scale Vector Search and Breaking It with Second-Tier Memory. *CoRR* abs/2405.03267 (2024).

[28] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private Information Retrieval. In *FOCS*. IEEE Computer Society, 41–50.

[29] Liwei Deng, Penghao Chen, Ximu Zeng, Tianfu Wang, Yan Zhao, and Kai Zheng. 2024. Efficient Data-aware Distance Comparison Operations for High-Dimensional Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 18, 3 (2024), 812–821.

[30] Yangshen Deng, Zhengxin You, Long Xiang, Qilong Li, Peiqi Yuan, Zhaoyang Hong, Yitao Zheng, Wanting Li, Runzhong Li, Haotian Liu, Kyriakos Mouratidis, Man Lung Yiu, Huan Li, Qiaomu Shen, Rui Mao, and Bo Tang. 2025. AlayaDB: The Data Foundation for Efficient and Effective Long-context LLM Inference. In *Companion of the 2025 International Conference on Management of Data*. ACM, 364–377.

[31] Laxman Dhulipala, Majid Hadian, Rajesh Jayaram, Jason Lee, and Vahab Mirrokni. 2024. MUVERA: Multi-Vector Retrieval via Fixed Dimensional Encodings. *CoRR* abs/2405.19504 (2024).

[32] Wei Dong, Moses Charikar, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the*

20th International Conference on World Wide Web, WWW 2011. ACM, 577–586.

[33] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2025. The faiss library. *IEEE Transactions on Big Data* (2025).

[34] Hao Duan, Yitong Song, Bin Yao, and Anqi Liang. 2025. PGTuner: An Efficient Framework for Automatic and Transferable Configuration Tuning of Proximity Graphs. *Proc. ACM Manag. Data* 3, 4 (2025), 261:1–261:27.

[35] Joshua Engels, Benjamin Landrum, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2024. Approximate Nearest Neighbor Search with Window Filters. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

[36] Zeheng Fan, Yuxiang Zeng, Zhuanglin Zheng, and Yongxin Tong. 2025. FedVSE: A Privacy-Preserving and Efficient Vector Search Engine for Federated Databases. *Proc. VLDB Endow.* 18, 12 (2025), 5371–5374.

[37] Zeheng Fan, Yuxiang Zeng, Zhuanglin Zheng, Binhan Yang, and Yongxin Tong. 2025. FedVS: Towards Federated Vector Similarity Search with Filters. In *SIGKDD*. ACM, 579–590.

[38] Robert W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* 5, 6 (1962), 345.

[39] Cong Fu and Deng Cai. 2016. EFANNA : An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph. *CoRR* abs/1609.07228 (2016).

[40] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.* 44, 8 (2022), 4139–4150.

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

[42] Georg Fuchsbauer, Riddhi Ghosal, Nathan Hauke, and Adam O'Neill. 2022. Approximate Distance-Comparison-Preserving Symmetric Encryption. In *SCN 2022 (Lecture Notes in Computer Science, Vol. 13409)*. Springer, 117–144.

[43] Teddy Furon, Hervé Jégou, Laurent Amsaleg, and Benjamin Mathon. 2013. Fast and secure similarity search in high dimensional space. In *WIFS 2013*. IEEE, 73–78.

[44] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality Sensitive Hashing Scheme Based on Dyanmic Collision Counting. In *SIGMOD*. 541–552.

[45] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2025. Practical and asymptotically optimal quantization of high-dimensional vectors in Euclidean space for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

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

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

[48] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization. *IEEE TPAMI* 36, 4 (2013), 744–755.

[49] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.

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

[51] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. 2025. SymphonyQG: towards symphonious integration of quantization and graph for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[52] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik PA Lensch. 2022. GGNN: Graph-based gpu nearest neighbor search. *IEEE Transactions on Big Data* 9, 1 (2022), 267–279.

[53] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 855–864.

[54] Yunguo Guan, Rongxing Lu, Yandong Zheng, Jun Shao, and Guiyi Wei. 2021. Toward Oblivious Location-Based k-Nearest Neighbor Query in Smart Cities. *IEEE Internet Things J.* 8, 18 (2021), 14219–14231.

[55] Hao Guo and Youyou Lu. 2025. Achieving Low-Latency Graph-Based Vector Search via Aligning Best-First Search Algorithm with SSD. In *OSDI*. USENIX Association, 171–186.

[56] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020 (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 3887–3896.

[57] Bernal Jimenez Gutierrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. 2024. HippoRAG: Neurobiologically Inspired Long-Term Memory for Large Language Models. In *NeurIPS*.

[58] Yikun Han, Chunjiang Liu, and Pengfei Wang. 2023. A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge. *CoRR* abs/2310.11703 (2023).

[59] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*. IEEE Computer Society, 5713–5722.

[60] Alireza Heidari and Wei Zhang. 2025. Filter-Centric Vector Indexing: Geometric Transformation for Efficient Filtered Vector Search. *CoRR*

abs/2506.15987 (2025).

[61] Nan Hou, Kangfei Zhao, Jiadong Xie, and Jeffrey Xu Yu. 2026. Beyond Linear LLM Invocation: An Efficient and Effective Semantic Filter Paradigm.

[62] Xiao Hu, Ke Yi, and Yufei Tao. 2019. Output-Optimal Massively Parallel Algorithms for Similarity Joins. *ACM Trans. Database Syst.* 44, 2 (2019), 6:1–6:36.

[63] Zhiyuan Hua, Qiji Mo, Zebin Yao, Lixiao Cui, Xiaoguang Liu, Gang Wang, Zijing Wei, Xinyu Liu, Tianxiao Tang, Shaozhi Liu, and Lin Qu. 2025. Dynamically Detect and Fix Hardness for Efficient Approximate Nearest Neighbor Search. *CoRR* abs/2510.22316 (2025).

[64] Zhiyuan Hua, Qiji Mo, Zebin Yao, Lixiao Cui, Xiaoguang Liu, Gang Wang, Zijing Wei, Xinyu Liu, Tianxiao Tang, Shaozhi Liu, and Lin Qu. 2025. Dynamically Detect and Fix Hardness for Efficient Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 6 (2025), 1–28.

[65] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2016. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *PVLDB* 9, 1 (2016), 1–12.

[66] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.

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

[68] Edwin H. Jacox and Hanan Samet. 2008. Metric space similarity joins. *ACM Trans. Database Syst.* 33, 2 (2008), 7:1–7:38.

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

[70] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2024. Bridging Software-Hardware for CXL Memory Disaggregation in Billion-Scale Nearest Neighbor Search. *ACM Trans. Storage* 20, 2 (2024), 10:1–10:30.

[71] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.

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

[73] Mengxu Jiang, Zhi Yang, Fangyuan Zhang, Guanhao Hou, Jieming Shi, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter. *Proc. ACM Manag. Data* 3, 3 (2025), 148:1–148:26.

[74] Wenqi Jiang, Hang Hu, Torsten Hoefler, and Gustavo Alonso. 2025. Fast Graph Vector Search via Hardware Acceleration and Delayed-Synchronization Traversal. *Proc. VLDB Endow.* 18, 11 (2025), 3797–3811.

[75] Huijun Jin, Jieun Lee, Shengmin Piao, Sangmin Seo, and Sanghyun Park. 2026. PRO-HNSW: Proactive Repair and Optimization for High-Performance Dynamic HNSW Indexes. (2026).

[76] Zhongming Jin, Debing Zhang, Yao Hu, Shiding Lin, Deng Cai, and Xiaofei He. 2014. Fast and Accurate Hashing Via Iterative Nearest Neighbors Expansion. *IEEE Trans. Cybern.* 44, 11 (2014), 2167–2177.

[77] Zhi Jing, Yongye Su, Yikun Han, Bo Yuan, Haiyun Xu, Chunjiang Liu, Kehai Chen, and Min Zhang. 2024. When Large Language Models Meet Vector Databases: A Survey. *CoRR* abs/2402.01763 (2024).

[78] Yannis Kalantidis and Yannis Avrithis. 2014. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2321–2328.

[79] Dmitri V. Kalashnikov. 2013. Super-EGO: fast multi-dimensional similarity join. *VLDB J.* 22, 4 (2013), 561–585.

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

[81] Leonid V. Kantorovich. 1960. Mathematical Methods of Organizing and Planning Production. *Management Science* 6 (1960), 366–422.

[82] Norio Katayama and Shin'ichi Satoh. 1997. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *SIGMOD*. 369 – 380.

[83] Kyoungmin Kim, Lennart Roth, Liang Liang, and Anastasia Ailamaki. 2026. Work Sharing and Offloading for Efficient Approximate Threshold-based Vector Join.

[84] Nick Koudas and Kenneth C. Sevcik. 2000. High Dimensional Similarity Joins: Algorithms and Performance Evaluation. *IEEE Trans. Knowl. Data Eng.* 12, 1 (2000), 3–18.

[85] Udesh Kumarasinghe, Tyler Liu, Chunwei Liu, and Walid G. Aref. 2026. iPDB - Optimizing SQL Queries with ML and LLM Predicates. *CoRR* abs/2601.16432 (2026).

[86] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*.

[87] Binhong Li, Xiao Yan, and Shangqi Lu. 2025. Fast-Convergent Proximity Graphs for Approximate Nearest Neighbor Search. *CoRR* abs/2510.05975 (2025).

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

[89] Hui Li, Shiyuan Deng, Xiao Yan, Xiangyu Zhi, and James Cheng. 2025. SAQ: Pushing the Limits of Vector Quantization through Code Adjustment and Dimension Segmentation. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–25.

[90] Hangyu Li, Sarana Nutanong, Hong Xu, Chenyun Yu, and Foryu Ha. 2019. C2Net: A Network-Efficient Approach to Collision Counting LSH Similarity Join. *IEEE Trans. Knowl. Data Eng.* 31, 3 (2019), 423–436.

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

[92] Mocheng Li, Xiao Yan, Baotong Lu, Yue Zhang, James Cheng, and Chenhao Ma. 2025. Attribute Filtering in Approximate Nearest Neighbor Search: An In-depth Experimental Study. *Proc. ACM Manag. Data* 3, 6 (2025), 1–26.

[93] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *IEEE Trans. Knowl. Data Eng.* 32, 8 (2020), 1475–1488.

[94] Zhaoheng Li, Silu Huang, Wei Ding, Yongjoo Park, and Jianjun Chen. 2025. SIEVE: Effective Filtered Vector Search with Collection of Indexes. *Proc. VLDB Endow.* 18, 11 (2025), 4723–4736.

[95] Zhonggen Li, Xiangyu Ke, Yifan Zhu, Bocheng Yu, Baihua Zheng, and Yunjun Gao. 2025. Scalable Graph Indexing using GPUs for Approximate Nearest Neighbor Search. *CoRR* abs/2508.08744 (2025).

[96] Zhonggen Li, Yougen Li, Yifan Zhu, Zhaoqiang Chen, and Yunjun Gao. 2025. All-in-one Graph-based Indexing for Hybrid Search on GPUs. *CoRR* abs/2511.00855 (2025).

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

[98] Petro Liashchynskyi and Pavlo Liashchynskyi. 2019. Grid search, random search, genetic algorithm: a big comparison for NAS. *arXiv preprint arXiv:1912.06059* (2019).

[99] Yanjun Lin, Kai Zhang, Zhenying He, Yinan Jing, and X. Sean Wang. 2025. Survey of Filtered Approximate Nearest Neighbor Search over the Vector-Scalar Hybrid Data. *CoRR* abs/2505.06501 (2025).

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

[101] Yingfan Liu, Jiangtao Cui, Zi Huang, Hui Li, and Hengtao shen. 2014. SK-LSH : an efficient index structure for approximate nearest neighbor search. *PVLDB* 7, 9 (2014), 745–756.

[102] Yingfan Liu, Hao Wei, and Hong Cheng. 2018. Exploiting lower bounds to accelerate approximate nearest neighbor search on high-dimensional data. *Inf. Sci.* 465 (2018), 484–504.

[103] Ying Liu, Dengsheng Zhang, Guojun Lu, and Wei-Ying Ma. 2007. A survey of content-based image retrieval with high-level semantics. *Pattern Recognit.* 40, 1 (2007), 262–282.

[104] Yingfan Liu, Yandi Zhang, Jiadong Xie, Hui Li, Jeffrey Xu Yu, and Jiangtao Cui. 2025. Privacy-preserving approximate nearest neighbor search on high-dimensional data. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 3017–3029.

[105] Jiahao Lou, Quan Yu, Shufeng Gong, Song Yu, Yanfeng Zhang, and Ge Yu. 2025. DGAI: Decoupled On-Disk Graph-Based ANN Index for Efficient Updates and Queries. *CoRR* abs/2510.25401 (2025).

[106] Kejing Lu, Chuan Xiao, and Yoshiharu Ishikawa. 2024. Probabilistic Routing for Graph-Based Approximate Nearest Neighbor Search. In *ICML (Proceedings of Machine Learning Research)*. PMLR / OpenReview.net, 33177–33195.

[107] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*. 950–961.

[108] P. C. Mahalanobis. 2018. On the Generalised Distance in Statistics. *Sankhya A* 80, 1 (12 2018), 1–7. Reprint of the 1936 original.

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

[110] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.

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

[112] Julieta Martinez, Shobhit Zakhmi, Holger H Hoos, and James J Little. 2018. Lsq++: Lower running time and higher recall in multi-codebook quantization. In *Proceedings of the European conference on computer vision (ECCV)*. 491–506.

[113] Ahmed Metwally and Christos Faloutsos. 2012. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *Proc. VLDB Endow.* 5, 8 (2012), 704–715.

[114] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *27th Annual Conference on Neural Information Processing Systems 2013*. 3111–3119.

[115] Stanley Milgram. 1967. The small world problem. *Psychology today* 2, 1 (1967), 60–67.

[116] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F. Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-Throughput Vector Similarity Search in Knowledge Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 197:1–197:25.

[117] Stanislav Morozov and Artem Babenko. 2019. Unsupervised neural quantization for compressed-domain similarity search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 3036–3045.

[118] Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE TPAMI* 36, 11 (2014), 2227–2240.

[119] Javier Alvaro Vargas Muñoz, Marcos André Gonçalves, Zanoni Dias, and Ricardo da Silva Torres. 2019. Hierarchical Clustering-Based Graphs for Large Scale Approximate Nearest Neighbor Search. *Pattern Recognit.* 96 (2019).

[120] Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. 2024. Efficient Multi-vector Dense Retrieval with Bit Vectors. In *ECIR (Lecture Notes in Computer Science, Vol. 14609)*. Springer, 3–17.

[121] Nasser M Nasrabadi and Robert A King. 1988. Image coding using vector quantization: A review. *IEEE Transactions on communications* 36, 8 (1988), 957–971.

[122] Naoki Ono and Yusuke Matsui. 2023. Relative NN-Descent: A Fast Index Construction for Graph-Based Approximate Nearest Neighbor Search. In *Proceedings of the 31st ACM International Conference on Multimedia, MM 2023*. ACM, 1659–1667.

[123] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2024. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. In *ICDE*. IEEE, 4236–4247.

[124] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *VLDB J.* 33, 5 (2024), 1591–1615.

[125] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Companion of the 2024 International Conference on Management of Data*. ACM, 597–604.

[126] Liana Patel, Siddharth Jha, Melissa Z. Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Towards AI-Based Data Analytics with Accuracy Guarantees. *Proc. VLDB Endow.* 18, 11 (2025), 4171–4184.

[127] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proc. ACM Manag. Data* 2, 3 (2024), 120.

[128] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proc. ACM Manag. Data* 1, 1 (2023), 54:1–54:27.

[129] Yanguo Peng, Jiangtao Cui, Hui Li, and Jianfeng Ma. 2017. A reusable and single-interactive model for secure approximate k-nearest neighbor query in cloud. *Information Sciences* 387 (2017), 146–164.

[130] Zhencan Peng, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2025. Dynamic Range-Filtering Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 18, 10 (2025), 3256–3268.

[131] Martin Perdacher, Claudia Plant, and Christian Böhm. 2019. Cache-oblivious High-performance Similarity Join. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019*. ACM, 87–104.

[132] Sébastien Rivault, Mostafa Bamha, Sébastien Limet, and Sophie Robert. 2022. A Scalable Similarity Join Algorithm Based on MapReduce and LSH. *Int. J. Parallel Program.* 50, 3-4 (2022), 360–380.

[133] Douglas Rolins Santana and Leonardo Andrade Ribeiro. 2023. Approximate Similarity Joins over Dense Vector Embeddings. In *Proceedings of the 38th Brazilian Symposium on Databases, SBBD 2023, Belo Horizonte, MG, Brazil, September 25-29, 2023*. SBC, 51–62.

[134] Keshav Santhanam, Omar Khattab, Christopher Potts, and Matei Zaharia. 2022. PLAID: An Efficient Engine for Late Interaction Retrieval. In *CIKM*. ACM, 1747–1756.

[135] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. 2022. ColBERTv2: Effective and Efficient Retrieval via Lightweight Late Interaction. In *NAACL*. Association for Computational Linguistics, 3715–3734.

[136] Gaurav Sehgal and Semih Salihoglu. 2025. NaviX: A Native Vector Index Design for Graph DBMSs With Robust Predicate-Agnostic Search Performance. *Proc. VLDB Endow.* 18, 11 (2025), 4438–4450.

[137] Sacha Servan-Schreiber, Simon Langowski, and Srinivas Devadas. 2022. Private approximate nearest neighbor search with sublinear communication. In *S&P 2022*. IEEE, 911–929.

[138] Jifan Shi, Jianyang Gao, James Xia, Tamás Béla Fehér, and Cheng Long. 2026. GPU-Native Approximate Nearest Neighbor Search with IVF-RaBitQ: Fast Index Build and Search. *arXiv preprint arXiv:2602.23999* (2026).

[139] Kyuseok Shim, Ramakrishnan Srikant, and Rakesh Agrawal. 1997. High-Dimensional Similarity Joins. In *ICDE*. IEEE Computer Society, 301–311.

[140] Chanop Silpa-Anan and Richard I. Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2008)*. IEEE Computer Society.

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

[142] Gurpreet Singh. 2013. A study of encryption algorithms (RSA, DES, 3DES and AES) for information security. *International Journal of Computer Applications* 67, 19 (2013).

[143] Yitong Song, Pengcheng Zhang, Chao Gao, Bin Yao, Kai Wang, Zongyuan Wu, and Lin Qu. 2025. TRIM: Accelerating High-Dimensional Vector Similarity Search with Enhanced Triangle-Inequality-Based Pruning. *CoRR* abs/2508.17828 (2025).

[144] Yitong Song, Xuanhe Zhou, Christian S. Jensen, and Jianliang Xu. 2026. Vector Search for the Future: From Memory-Resident, Static Heterogeneous Storage, to Cloud-Native Architectures.

[145] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2015. SRS: Solving c-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *PVLDB* 8, 1 (2015), 1–12.

[146] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*. 563–576.

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

[148] Yao Tian, Zhoujin Tian, Xi Zhao, Ruiyuan Zhang, and Xiaofang Zhou. 2026. GEM: A Native Graph-based Index for Multi-Vector Retrieval.

[149] Yao Tian, Ziyang Yue, Ruiyuan Zhang, Xi Zhao, Bolong Zheng, and Xiaofang Zhou. 2023. Approximate Nearest Neighbor Search in High Dimensional Vector Databases: Current Research and Future Directions. *IEEE Data Eng. Bull.* 47, 3 (2023), 39–54.

[150] Nimish Ukey, Zhengyi Yang, Wenke Yang, Binghao Li, and Runze Li. [n. d.]. kNN Join for Dynamic High-Dimensional Data: A Parallel Approach. In *Australasian Database Conference, series = Lecture Notes in Computer Science, pages = 3–16, publisher = Springer, year = 2023.*

[151] Nimish Ukey, Guangjian Zhang, Zhengyi Yang, Binghao Li, Wei Li, and Wenjie Zhang. 2023. Efficient continuous kNN join over dynamic high-dimensional data. *World Wide Web (WWW)* 26, 6 (2023), 3759–3794.

[152] Nimish Ukey, Guangjian Zhang, Zhengyi Yang, Xiaoyang Wang, Binghao Li, Serkan Saydam, and Wenjie Zhang. 2024. A Cluster-Based Approach to kNN Join Over Batch-Dynamic High-Dimensional Data. In *Advanced Data Mining and Applications - 20th International Conference (Lecture Notes in Computer Science)*. Springer, 81–96.

[153] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD '21*. ACM, 2614–2627.

[154] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. 2018. A Survey on Learning to Hash. *IEEE Trans. Pattern Anal. Mach. Intell.* 40, 4 (2018), 769–790.

[155] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2022. Navigable Proximity Graph-Driven Native Hybrid Queries with Structured and Unstructured Constraints. *CoRR* abs/2203.13601 (2022).

[156] Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Yifan Zhu, and Wenchao Zhou. 2025. Accelerating Graph Indexing for ANNS on Modern CPUs. *Proc. ACM Manag. Data* 3, 3 (2025), 123:1–123:29.

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

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

[159] Yuxiang Wang, Ziyuan He, Yongxin Tong, Zimu Zhou, and Yiman Zhong. 2025. Timestamp Approximate Nearest Neighbor Search over High-Dimensional Vector Data. In *ICDE*. IEEE Computer Society, 3043–3055.

[160] Yifan Wang, Vyom Pathak, and Daisy Zhe Wang. 2024. Xling: A Learned Filter Framework for Accelerating High-Dimensional Approximate Similarity Join. *CoRR* abs/2402.13397 (2024).

[161] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Graph-and Tree-based Indexes for High-dimensional Vector Similarity Search: Analyses, Comparisons, and Future Directions. *Data Engineering* (2023), 3–21.

[162] Zeyu Wang, Qitong Wang, Xiaoxing Cheng, Peng Wang, Themis Palpanas, and Wei Wang. 2024. Steiner-Hardness: A Query Hardness Measure for Graph-Based ANN Indexes. *Proc. VLDB Endow.* 17, 13 (2024), 4668–4682.

[163] Zeyu Wang, Haoran Xiong, Zhenying He, Peng Wang, and Wei Wang. 2024. Distance Comparison Operators for Approximate Nearest Neighbor Search: Exploration and Benchmark. *CoRR* abs/2403.13491 (2024).

[164] Ziqi Wang, Jingzhe Zhang, and Wei Hu. 2025. WoW: A Window-to-Window Incremental Index for Range-Filtering Approximate Nearest Neighbor Search. *CoRR* abs/2508.18617 (2025).

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

[166] Wai Kit Wong, David Wai-lok Cheung, Ben Kao, and Nikos Mamoulis. 2009. Secure kNN computation on encrypted databases. In *SIGMOD*. 139–152.

[167] Jingyi Xi, Chenghao Mo, Ben Karsin, Artem M. Chirkin, Mingqin Li, and Minjia Zhang. 2025. VecFlow: A High-Performance Vector Data Management System for Filtered-Search on GPUs. *Proc. ACM Manag. Data* 3, 4 (2025), 271:1–271:27.

[168] Jiadong Xie, Jeffrey Liang, Siyi Teng, Jeffrey Xu Yu, and Yingfan Liu. 2026. Breaking the Single-Reference-Vector Barrier in Approximate Nearest Neighbor Search. In *Proceedings of the ACM on Web Conference 2026, WWW*. ACM.

[169] Jiadong Xie, Jeffrey Xu Yu, and Yingfan Liu. 2025. Fast Approximate Similarity Join in Vector Databases. *Proc. ACM Manag. Data* 3, 3 (2025), 158:1–158:26.

[170] Jiadong Xie, Jeffrey Xu Yu, and Yingfan Liu. 2025. Graph Based K-Nearest Neighbor Search Revisited. *ACM TODS* (May 2025).

[171] Jiadong Xie, Jeffrey Xu Yu, Siyi Teng, and Yingfan Liu. 2025. Beyond Vector Search: Querying With and Without Predicates. *Proc. ACM Manag. Data* 3, 6 (2025), 300:1–300:26.

[172] Haike Xu, Guy Blelloch, Laxman Dhulipala, Lars Gottesbüren, Rajesh Jayaram, and Jakub Łącki. 2026. JAG: Joint Attribute Graphs for Filtered Nearest Neighbor Search.

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

[174] Qian Xu, Juan Yang, Feng Zhang, Junda Pan, Kang Chen, Youren Shen, Amelie Chi Zhou, and Xiaoyong Du. 2025. Tribase: A Vector Data Query Engine for Reliable and Lossless Pruning Compression using Triangle Inequalities. *Proc. ACM Manag. Data* 3, 1 (2025), 82:1–82:28.

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

[176] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *SOSP*. ACM, 545–561.

[177] Ming Yang, Yuzheng Cai, and Weiguo Zheng. 2024. CSPG: Crossing Sparse Proximity Graphs for Approximate Nearest Neighbor Search. In *NeurIPS*.

[178] Ming Yang, Yuzheng Cai, and Weiguo Zheng. 2025. Hi-PNG: Efficient Interval-Filtering ANNS via Hierarchical Interval Partition Navigating Graph. In *SIGKDD*. 3518–3529.

[179] Mingyu Yang, Wentao Li, Jiabao Jin, Xiaoyao Zhong, Xiangyu Wang, Zhitao Shen, Wei Jia, and Wei Wang. 2025. Effective and General Distance Computation for Approximate Nearest Neighbor Search. In *ICDE*. IEEE, 1098–1110.

[180] Mingyu Yang, Wenxuan Xia, Wentao Li, Raymond Chi-Wing Wong, and Wei Wang. 2025. Elastic Index Selection for Label-Hybrid AKNN Search.

[181] Shuo Yang, Jiadong Xie, Yingfan Liu, Jeffrey Xu Yu, Xiyue Gao, Qianru Wang, Yanguo Peng, and Jiangtao Cui. 2025. Revisiting the Index Construction of Proximity Graph-Based Approximate Nearest Neighbor Search. *PVLDB* 18, 6 (2025), 1825–1838.

[182] Tiannuo Yang, Wen Hu, Wangqi Peng, Yusen Li, Jianguo Li, Gang Wang, and Xiaoguang Liu. 2024. Vdtuner: Automated performance tuning for vector data management systems. In *ICDE*. IEEE, 4357–4369.

[183] Peiqi Yin, Xiao Yan, Qihui Zhou, Hui Li, Xiaolu Li, Lin Zhang, Meiling Wang, Xin Yao, and James Cheng. 2025. Gorgeous: Revisiting the Data Layout for Disk-Resident High-Dimensional Vector Search. *CoRR* abs/2508.15290 (2025).

[184] Ziqi Yin, Jianyang Gao, Pasquale Balsebre, Gao Cong, and Cheng Long. 2025. DEG: Efficient Hybrid Vector Search Using the Dynamic Edge Navigation Graph. *Proc. ACM Manag. Data* 3, 1 (2025), 29:1–29:28.

[185] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. 2007. Efficient index-based KNN join processing for high-dimensional data. *Inf. Softw. Technol.* 49, 4 (2007), 332–344.

[186] Chenyun Yu, Sarana Nutanong, Hangyu Li, Cong Wang, and Xingliang Yuan. 2017. A Generic Method for Accelerating LSH-Based Similarity Join Processing. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 712–726.

[187] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2022. GPU-accelerated proximity graph approximate nearest Neighbor search and construction. In *ICDE*. IEEE, 552–564.

[188] Xingliang Yuan, Xinyu Wang, Cong Wang, Chenyun Yu, and Sarana Nutanong. 2017. Privacy-Preserving Similarity Joins Over Encrypted Data. *IEEE Trans. Inf. Forensics Secur.* 12, 11 (2017), 2763–2775.

[189] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-time OLAP Database System at Alibaba Cloud. *Proc. VLDB Endow.* 12, 12 (2019), 2059–2070.

[190] Jingtao Zhan, Jiaxin Mao, Yiqun Liu, Jiafeng Guo, Min Zhang, and Shaoping Ma. 2021. Jointly optimizing query encoder and product quantization to improve retrieval performance. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 2487–2496.

[191] Cheng Zhang, Jianzhi Wang, Wan-Lei Zhao, and Shihai Xiao. 2025. Highly Efficient Disk-based Nearest Neighbor Search on Extended Neighborhood Graph. In *SIGIR5*. ACM, 2513–2523.

[192] Fangyuan Zhang, Mengxu Jiang, Guanhao Hou, Jieming Shi, Hua Fan, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 3 (2025), 152:1–152:26.

[193] Guibin Zhang, Muxin Fu, Guancheng Wan, Miao Yu, Kun Wang, and Shuicheng Yan. 2025. G-Memory: Tracing Hierarchical Memory for Multi-Agent Systems. *CoRR* abs/2506.07398 (2025).

[194] Haoyu Zhang, Jun Liu, Zhenhua Zhu, Shulin Zeng, Maojia Sheng, Tao Yang, Guohao Dai, and Yu Wang. 2024. Efficient and Effective Retrieval of Dense-Sparse Hybrid Vectors using Graph-based Approximate Nearest Neighbor Search. *CoRR* abs/2410.20381 (2024).

[195] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023*. USENIX Association, 377–395.

[196] Xinyi Zhang, Qichen Wang, Cheng Xu, Yun Peng, and Jianliang Xu. 2024. FedKNN: Secure Federated k-Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 1 (2024), V2mod011:1–V2mod011:26.

[197] Zhilin Zhang, Ke Wang, Chen Lin, and Weipeng Lin. 2018. Secure Top-k Inner Product Retrieval. In *CIKM 2018*. ACM, 77–86.

[198] Weichen Zhao, Yuncheng Lu, Yao Tian, Hao Zhang, Jiehui Li, Minghao Zhao, Yakun Li, and Weining Qian. 2026. Optimizing SSD-Resident Graph Indexing for High-Throughput Vector Search. *CoRR* abs/2602.22805 (2026).

[199] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate nearest neighbor search on gpu. In *ICDE*. IEEE, 1033–1044.

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

[201] Yandong Zheng, Rongxing Lu, and Jun Shao. 2019. Achieving efficient and privacy-preserving k-NN query for outsourced ehealthcare data. *Journal of Medical Systems* 43 (2019), 1–13.

[202] Yandong Zheng, Rongxing Lu, Songnian Zhang, Jun Shao, and Hui Zhu. 2024. Achieving Practical and Privacy-Preserving kNN Query over Encrypted Data. *IEEE TDSC* (2024).

[203] Mingxun Zhou, Elaine Shi, and Giulia Fanti. 2024. Pacmann: Efficient Private Approximate Nearest Neighbor Search. *IACR Cryptol. ePrint Arch.* (2024), 1600.

[204] Wenyang Zhou, Jiadong Xie, Yingfan Liu, Zhihao Yin, Jeffrey Xu Yu, Hui Li, Zhangqian Mu, Xiaotian Qiao, and Jiangtao Cui. 2026. Fast Tuning the Index Construction Parameters of Proximity Graphs in Vector Databases.

[205] Jiaxu Zhu, Jiayu Yuan, Kaiwen Yang, Xiaobao Chen, Shihuan Yu, Hongchang Lv, Yan Li, and Bolong Zheng. 2025. An Experimental Evaluation of Hybrid Querying on Vectors. *Proc. VLDB Endow.* 19, 2 (2025), 183–195.

[206] Youwen Zhu, Rui Xu, and Tsuyoshi Takagi. 2013. Secure k-NN computation on encrypted cloud data without sharing key with query users. In *SCC@ASIACCS*. ACM, 55–60.

[207] Zeqi Zhu, Zeheng Fan, Yuxiang Zeng, Yexuan Shi, Yi Xu, Mengmeng Zhou, and Jin Dong. 2024. FedSQ: A Secure System for Federated Vector Similarity Queries. *Proc. VLDB Endow.* 17, 12 (2024), 4441–4444.

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