



WHITE PAPER · DATA STORAGE ENGINES

Vector Storage Engines

Indexes, storage and the trade-offs behind similarity search

White Paper 4 · Data Storage Engines series · Version 1.0 · 26 October 2026

Bassirah Technology — Innovation & R&D

Vector Storage Engines: Indexes, Storage and the Trade-offs Behind Similarity Search

White Paper 4 · Data Storage Engines series · Bassirah Technology — Innovation & R&D · 26 October 2026

Document control

Field	Value
Series	Data Storage Engines, paper 4 of 6
Author	Bassirah Technology — Innovation & R&D
Version	1.0 · 26 October 2026
Scope	A technology survey. It describes published research and publicly documented systems; it does not report Bassirah benchmarks or customer results.

Contents

1. Abstract.....	4
2. Introduction	4
2.1 Embeddings and similarity search in AI applications	4
2.2 Exact versus approximate search.....	4
2.3 Scope.....	4
2.4 How this paper builds on papers 1 to 3.....	5
3. Vector search fundamentals	5
3.1 Similarity metrics.....	5
3.2 Recall, latency, memory and build time.....	5
3.3 The curse of dimensionality	5
3.4 Evaluation methodology and benchmarks	5
4. Index families	6
4.1 Proximity graphs and HNSW.....	6
4.2 NSG, Vamana and design lessons	6
4.3 Partition-based indexes: IVF, SPANN and SOAR.....	6
4.4 Hashing, trees and adaptive search	6
5. Compression and quantization.....	7
5.1 Product quantization	7
5.2 Scalar and binary quantization.....	7
5.3 Quantization with error bounds and anisotropic loss	7
5.4 Reranking with full vectors.....	7
6. Memory, disk and hardware	8
6.1 SSD-resident indexes	8
6.2 GPUs and CPUs.....	8
7. Filtering, hybrid search and updates.....	8

7.1 Attribute filtering	8
7.2 Filter-aware indexes	9
7.3 Hybrid lexical and vector ranking	9
7.4 Inserts, deletes and freshness.....	9
8. System architectures	10
8.1 Purpose-built vector databases	10
8.2 Vector search inside relational databases	10
8.3 Search engines and document databases	10
8.4 In-memory and analytical engines.....	11
8.5 Embedded engines and columnar formats	11
9. Comparative analysis.....	11
10. Open challenges and future opportunities.....	12
10.1 Filtered and hybrid search quality	12
10.2 Freshness and updates at scale	12
10.3 The cost of memory versus disk	13
10.4 Benchmark realism and out-of-distribution queries	13
10.5 Embedding-model drift and re-indexing	13
10.6 Multi-vector and multimodal retrieval	13
10.7 Security, privacy and access control	13
10.8 Convergence into general-purpose databases	14
11. What this means for platform decisions	14
References	14

1. Abstract

Machine-learning models turn text, images and other unstructured data into embedding vectors, and many AI applications, including retrieval-augmented generation, depend on finding the stored vectors closest to a query vector. Exact search compares the query with every vector, which becomes slow and costly as collections grow, so most systems use approximate nearest neighbor (ANN) indexes that trade a controlled loss of accuracy for speed. This paper examines how vector storage engines make that trade. It reviews the main index families (proximity graphs such as HNSW and DiskANN, partition-based indexes such as IVF and SPANN, and older hashing and tree methods), compression by quantization, and the placement of indexes in memory, on SSDs and on GPUs. It then covers attribute filtering, hybrid lexical and vector search, and updates, and describes how purpose-built vector databases, general-purpose databases, search engines and embedded formats package these techniques. The paper closes with open challenges, framed as research directions and opportunities, and with notes for platform decisions.

2. Introduction

2.1 Embeddings and similarity search in AI applications

An embedding model maps an item, such as a passage of text or an image, to a vector of a few hundred to a few thousand dimensions, so that similar items lie close together. Search becomes a geometric problem: given a query vector, return the k stored vectors nearest to it. Recommendation, image search and question answering have long used this pattern [1]. Retrieval-augmented generation (RAG) made it central to large language model applications by pairing a generator with a dense vector index of passages consulted at inference time [2].

Pan, Wang and Li count more than twenty commercial vector database management systems introduced within five years, and note that vectors are large, comparisons costly, natural partitioning absent and hybrid attribute-vector queries hard to answer well [3].

2.2 Exact versus approximate search

Exact k -nearest-neighbor search scans every vector and is always correct, but its cost grows linearly with the number of vectors and their dimension. ANN indexes avoid the full scan by visiting only a promising subset of vectors, by comparing compressed codes instead of full vectors, or both. The result may miss some true neighbors, so ANN systems are judged by recall, the fraction of true neighbors returned, alongside latency, throughput, memory and build time [4]. Most engines still offer exact search for small collections or highly selective filters [5], [6], [7].

2.3 Scope

This paper covers the index structures, compression schemes and storage layouts used for dense vector similarity search, and the systems that package them. It concentrates on peer-reviewed work from 2021 to 2026 and on official documentation, citing earlier papers where they introduced an idea still in use. It does not report benchmark figures. Where a paper reports results, we describe them qualitatively and attribute them to the authors.

2.4 How this paper builds on papers 1 to 3

Paper 1 placed vector engines in Class D of a five-class taxonomy and named ANN index memory size and search latency as their bottlenecks. This paper opens Class D, and the earlier papers reappear inside it. Milvus manages dynamic vector data with an LSM-based structure, buffering inserts in memory, flushing immutable segments and merging them later [1], the pattern paper 2 examined. ClickHouse builds vector indexes per data part and rebuilds them as parts merge [7], and the Lance format targets the random access that AI workloads need from columnar storage [8], both topics of paper 3.

3. Vector search fundamentals

3.1 Similarity metrics

Three metrics dominate: Euclidean (L2) distance, inner product and cosine similarity, with Hamming distance for binary vectors [5], [6]. Cosine similarity equals inner product on unit-length vectors, which is why some engines normalize vectors at index time [9]. Maximum inner product search, in which vector length carries meaning, is common in recommendation and benefits from quantizers designed for it [10]. An index built for one metric generally serves only queries that use it [7].

3.2 Recall, latency, memory and build time

Every ANN index exposes parameters that move it along a trade-off curve. In a graph index, a larger search beam (ef or ef_search) visits more nodes and raises recall at the cost of latency; in a partition index, probing more partitions has the same effect [5]. Build parameters, such as the maximum number of neighbors per node (M) and the build beam (efConstruction), raise index quality at the cost of build time and memory [11], [6]. Compression reduces memory but adds approximation error, which reranking with full vectors can recover at extra cost (Section 5.4). The Faiss authors frame vector search as trade-offs among accuracy, speed, memory and build time, and choose an index starting from the memory constraint [12].

3.3 The curse of dimensionality

In high dimensions, distances concentrate. Beyer et al. showed that, under broad conditions, the distance to the nearest neighbor approaches the distance to the farthest as dimensionality grows, and that many indexing studies used workloads where nearest neighbors were not meaningful [13]. They also identified workloads in which nearest neighbors remain meaningful, and the practical success of ANN indexes on learned embeddings suggests such structure is common. The effect still explains why tree-based partitioning, effective in low dimensions, loses its advantage on embeddings [12], [3].

3.4 Evaluation methodology and benchmarks

ANN-Benchmarks standardized in-memory evaluation by running many algorithms over parameter sweeps and plotting recall against throughput on common datasets [4]. The NeurIPS'21 Big-ANN challenge moved to billion-scale datasets, with tracks for limited memory, memory plus SSD, and custom hardware [14]. The NeurIPS'23 competition shifted to smaller datasets and more realistic tasks: filtered search, out-of-distribution queries, sparse vectors and streaming updates [15]. Experimental studies of graph indexes followed [16], [17]. The common lesson is that results depend on the dataset, the query distribution and the operating point, so an index should be judged on data that resemble the target workload.

4. Index families

4.1 Proximity graphs and HNSW

A proximity graph connects each vector to some of its neighbors. A query starts at an entry point and moves greedily toward closer neighbors, keeping a beam of candidates until no closer vector is found. Hierarchical Navigable Small World (HNSW) graphs add a hierarchy: each element receives a maximum layer drawn from an exponentially decaying distribution, upper layers hold fewer elements with long-range links, and search descends to the bottom layer, which contains every element [11]. The authors report logarithmic complexity scaling and attribute much of the method's robustness, especially on clustered data, to a heuristic that selects diverse neighbors rather than simply the closest [11].

HNSW builds incrementally, so vectors can be inserted without a rebuild, one reason it is the default graph index in most systems described in Section 8. Its costs are memory, since the graph and usually the full vectors stay in RAM, and random memory access during traversal.

4.2 NSG, Vamana and design lessons

Other graph indexes invest more in graph shape. NSG approximates a monotonic relative neighborhood graph, aiming for connectivity, low out-degree, short search paths and a small index [18]. DiskANN introduced Vamana, built in two passes with a pruning parameter that keeps long-range edges, so that searches converge in few hops, which matters when each hop is an SSD read [19].

Wang et al. compared thirteen graph algorithms in a uniform environment and assembled an optimized algorithm from the principles they derived [16]. Azizi, Echiabi and Palpanas classify graph methods by five design paradigms (seed selection, incremental insertion, neighborhood propagation, neighborhood diversification and divide-and-conquer) and report that the stronger methods typically combine incremental insertion with neighborhood diversification, while the choice of base graph can limit scalability [17].

4.3 Partition-based indexes: IVF, SPANN and SOAR

An inverted file (IVF) index clusters vectors with k-means and stores each cluster as a posting list keyed by its centroid. A query is compared with the centroids, and only the nearest few lists are scanned [20], [12]. IVF indexes are compact, cheap to build, combine naturally with compression, and suit disk and GPU layouts because posting lists are read sequentially. Their weakness is the boundary: a true neighbor in an unprobed cluster is missed.

Recent work assigns some vectors to several lists. SPANN keeps centroids in memory and posting lists on SSD, balances list lengths with hierarchical clustering, adds boundary vectors to neighboring lists and prunes lists per query [21]. SOAR assigns vectors to additional partitions chosen with an orthogonality-amplified residual loss, so each assignment covers cases where the others perform poorly; its authors report better accuracy at a given search cost while keeping indexing fast and memory low [22].

4.4 Hashing, trees and adaptive search

Locality-sensitive hashing (LSH) uses random projections so that nearby vectors tend to share buckets, with statistical guarantees under assumptions about the data. The Faiss authors note that random partitioning is generally outperformed by data-aware partitioning such as k-means, and that LSH survives mainly as a way to produce compact binary codes [12]. Tree-

based indexes such as k-d trees and hierarchical k-means trees remain useful at moderate dimensionality but lose pruning power on high-dimensional embeddings [12], [3].

Most indexes apply the same search budget to every query, although the effort needed varies widely. Li et al. train lightweight models that use intermediate search results to predict when each query can stop, improving the latency-accuracy trade-off without changing the index [23]. Weaviate applies a simpler version, sizing the HNSW beam from each query's result limit [24].

5. Compression and quantization

5.1 Product quantization

Product quantization (PQ) splits each vector into subvectors and quantizes each with its own small codebook, so a vector becomes a short code of centroid indices. Distances are estimated from precomputed lookup tables, and PQ combined with an inverted file made billion-scale search feasible in memory [20]. Optimized product quantization (OPQ) learns a rotation before splitting, reducing distortion [25]. The Faiss library implements PQ alongside additive and residual quantizers and documents the trade-off between code size, accuracy and encoding time among them [12].

5.2 Scalar and binary quantization

Scalar quantization maps each dimension to a small integer, typically 8 or 4 bits; binary quantization keeps one bit per dimension. Both need little training and run fast with SIMD instructions, and both are now common in production. Elasticsearch offers int8, int4 and binary (BBQ) variants of its HNSW and flat indexes [9], MongoDB Vector Search offers scalar and binary quantization [26], and ClickHouse can quantize its HNSW graph to bfloat16, int8 or one bit [7].

5.3 Quantization with error bounds and anisotropic loss

PQ works well empirically but offers no theoretical error bound, and Gao and Long observe that it can fail badly on some real datasets [27]. RaBitQ applies a random rotation and encodes a D-dimensional vector as a D-bit string, with a distance estimator that carries a sharp error bound; the authors report a better accuracy-efficiency trade-off than PQ on their datasets [27]. An extension uses B bits per dimension for flexible compression and proves asymptotic optimality in the trade-off between space and error bound; its authors report higher accuracy than baselines at the same memory [28]. RaBitQ-based indexes appear in Milvus and LanceDB documentation [29], [30].

A different idea targets inner-product search. Quantizers usually minimize reconstruction error, but what matters is error that changes the ranking of the top results. Guo et al. proposed an anisotropic loss that penalizes the residual component parallel to the data point more heavily than the orthogonal component, released in the ScaNN library [10], on which SOAR also builds [22].

5.4 Reranking with full vectors

Compression and reranking work as a pair: the index retrieves extra candidates using compressed codes, then rescores them with full-precision vectors. Faiss provides refinement indexes for this [12], Elasticsearch an oversampling factor [9], and LanceDB a refine factor [30];

MongoDB keeps full-fidelity vectors on disk for rescoring [26], and ClickHouse rescores with full-precision vectors by default [7]. Reranking turns memory into a placement question: compact codes in RAM, full vectors on cheaper storage.

6. Memory, disk and hardware

6.1 SSD-resident indexes

Keeping a graph and full vectors in RAM becomes expensive at scale. DiskANN stores the Vamana graph and full vectors on SSD and keeps PQ codes in memory to steer the search [19]. Each node's full vector sits in the same disk sector as its neighbor list, so reranking needs no extra reads. The authors report high recall at low latency on a billion-point dataset with a single workstation and an inexpensive SSD [19]. SPANN takes the partition route, keeping only centroids in memory and reading a few posting lists per query, and its authors report better latency than DiskANN at equal memory on their datasets [21].

Starling adapts disk-resident graphs to vector databases that split data into many segments, each with limited memory and disk. It reorders the on-disk graph for locality, adds a small in-memory navigation graph and uses a block-oriented search to reduce I/O; its authors report higher throughput and lower latency than earlier disk-based methods at the same accuracy [31]. Products follow these designs: Milvus documents a DiskANN index [29], OpenSearch a DiskANN-style plugin method [32], Elasticsearch a clustering-based, binary-quantized disk index [9], and Weaviate an SPFresh-based index that keeps only compressed centroids in memory [24].

6.2 GPUs and CPUs

GPUs suit the dense arithmetic of brute-force and IVF-PQ search. Johnson, Douze and Jégou identified k-selection and memory-hierarchy use as the bottlenecks of earlier GPU approaches and designed an open-source GPU k-selection algorithm for brute-force, IVF and PQ-based search [33]. Graph traversal is harder to parallelize because each hop depends on the last, which leaves GPUs at a disadvantage without new techniques [12]. CAGRA is a proximity graph and search algorithm designed for GPU parallelism; its authors report faster graph construction than CPU HNSW and higher throughput for large and small query batches at comparable accuracy [34].

On CPUs, distance computation dominates, and modern instruction sets accelerate it. Milvus applies cache-aware and SIMD-aware optimizations [1]. Faiss interleaves 4-bit PQ codes so that lookup-table operations run inside SIMD registers, a layout inspired by ScaNN [12], and RaBitQ estimates distances with bitwise or SIMD operations [27]. Redis offers SVS-VAMANA, a graph index from an Intel project, whose LVQ and LeanVec compression relies on Intel optimizations that are unavailable in Redis Open Source and on other hardware, where it falls back to 8-bit scalar quantization [6].

7. Filtering, hybrid search and updates

7.1 Attribute filtering

Real queries ask for the nearest vectors that also match a tenant, a date range or a permission. Three strategies exist [3], [35]. Post-filtering runs the ANN search and discards non-

matching results, which can return fewer than k results when the filter is selective; pgvector documents this behavior and adds iterative index scans that keep searching until enough rows pass [5]. Pre-filtering evaluates the filter first, usually with a conventional index, and searches only matching vectors, exactly if few remain. In-filtering applies the predicate during graph or partition traversal.

Engines increasingly choose at run time. Redis switches between a batched mode, which pages through neighbors until enough pass the filter, and brute force over small matching sets [6]. Qdrant uses payload-index cardinality estimates to choose between full scan and HNSW traversal [36]. Lucene, as documented by Elasticsearch, applies filters inside the kNN search and falls back to exact scoring in a segment when the filter matches few documents [37]. Chronis et al. stress that the goal is stable recall across queries [35].

7.2 Filter-aware indexes

When a selective filter is unrelated to vector proximity, a graph walk restricted to matching nodes can lose connectivity. Filtered-DiskANN builds graphs whose edges reflect both geometry and label sets, in streaming and batch variants, queryable from SSD [38]. ACORN adds edges at construction so that the subgraph matching any predicate stays navigable, without knowing predicates in advance; its authors report higher throughput at fixed recall than prior methods [39]. These ideas have reached products: Qdrant adds filter-aware edges for indexed payload fields and offers ACORN for combinations of strict filters [36], Weaviate uses an ACORN-inspired strategy by default from version 1.34 [40], and Elasticsearch applies an ACORN heuristic, a preview feature, by default to new indices [37]. Li et al. compared filtering algorithms across attribute types and selectivity levels and found that pruning, entry-point selection and edge-filtering costs drive performance [41].

7.3 Hybrid lexical and vector ranking

Dense vectors capture meaning but can miss exact terms, identifiers and rare names that lexical retrieval such as BM25 handles well. Hybrid search runs both and fuses the ranked lists. Reciprocal rank fusion (RRF) sums, for each document, a score that decays with its rank in each list, without score normalization [42]. Bruch, Gai and Ingber found RRF sensitive to its parameters, and found that a convex combination of normalized scores outperformed it in and out of domain with only a small training set [43]. Elasticsearch, OpenSearch, MongoDB and Redis all document hybrid search [42], [32], [44], [6].

7.4 Inserts, deletes and freshness

Embeddings arrive continuously, and documents are corrected or withdrawn. Graph deletion is hard because removing a node can break paths through it. FreshDiskANN, a 2021 preprint, marks deleted nodes and consolidates the graph in batches, and its authors report sustaining concurrent inserts, deletes and searches without loss of search quality [45]. IP-DiskANN, a 2025 preprint, processes each insertion and deletion in place and reports stable recall over long update sequences [46]. ClickHouse avoids the problem by falling back to exact search for parts with pending deletes [7].

Partition indexes face a different problem: as data shifts, posting lists become unbalanced. SPFresh's LIRE protocol splits oversized partitions and reassigns only vectors near partition boundaries, avoiding periodic global rebuilds; its authors report steadier latency and accuracy than rebuild-based approaches with fewer resources [47]. Freshness also depends on the

pipeline: Weaviate's asynchronous indexing delays visibility briefly so that writes return quickly [24].

8. System architectures

8.1 Purpose-built vector databases

Milvus was designed as a vector data management system rather than a library. It builds on Faiss and adds an LSM-based storage engine with snapshot isolation, CPU and GPU optimizations, attribute filtering, multi-vector queries and distribution across nodes [1]. Manu, its cloud-native successor, relaxes data-model and consistency constraints in exchange for elasticity and evolvability: it exposes the write-ahead log and binlog as backbone services, treats writers as log publishers and search components as independent subscribers, and uses multi-version concurrency control with a delta consistency model [48].

Qdrant pairs HNSW with payload indexes and filter-aware edges and lets users tier payload indexes between memory and disk [36]. Weaviate offers flat, HNSW, dynamic and disk-based HFresh indexes, keeps an inverted index beside each shard's vector index for filtering and keyword search, and recommends lightweight flat indexes for per-tenant data [24], [40].

8.2 Vector search inside relational databases

The alternative adds vectors to a database that already holds the rest of the data. AnalyticDB-V exposed ANN algorithms as physical operators in an analytical engine, with accuracy-aware, cost-based optimization of hybrid queries [49]. VBase observed that vector indexes lack the monotonicity of conventional indexes, which forces systems to guess how many neighbors to retrieve before applying relational predicates, and identified a weaker property, relaxed monotonicity, that integrates vector search with relational operators while keeping top-k semantics [50].

pgvector is the common route for PostgreSQL. It performs exact search by default and adds HNSW and IVFFlat indexes, half-precision, binary and sparse vector types, and iterative index scans for filtered queries; its HNSW index takes M and $ef_construction$ at build time, ef_search at query time, and indexes single-precision vectors of up to 2,000 dimensions [5]. YugabyteDB, a PostgreSQL-compatible distributed database, accepts pgvector syntax and, according to its engineering blog, runs a per-tablet vector LSM that indexes new vectors in an in-memory HNSW buffer, flushes immutable chunks to disk and merges per-tablet top-k results [51]. Zhang, Liu and Wang asked whether the gap between relational vector extensions and specialized libraries is fundamental; comparing a PostgreSQL extension with Faiss, they traced it to implementation issues and proposed ways to close it [52].

8.3 Search engines and document databases

Search engines built on Lucene added vectors to an inverted-index architecture. Lucene builds HNSW graphs per index segment, so vector indexes follow the segment lifecycle of writes and merges, much as SSTables do in paper 2 [37]. Elasticsearch defaults to quantized index types for float vectors [9], and OpenSearch offers HNSW and IVF methods through Faiss and Lucene engines [32].

MongoDB separates search from the database. Its search process, mongot, is built on Lucene, runs beside the database process, synchronizes indexes through change streams and serves

queries that the database proxies to it [53]. MongoDB Vector Search uses HNSW for approximate search and supports exact search, pre-filtering on indexed fields and hybrid search [44]. MongoDB Atlas runs mongot as a managed service; for MongoDB Community Edition, Search and Vector Search became generally available in June 2026, with mongot distributed under the SSPL [53], [54].

8.4 In-memory and analytical engines

Redis maintains secondary indexes with flat, HNSW and SVS-VAMANA vector fields over hashes and JSON documents, combines vector queries with tag, numeric, text and geospatial filters, and in clusters lets each shard return a fraction of the requested neighbors, trading accuracy for speed [6].

ClickHouse brings vector search to a columnar engine. Exact search is a brute-force scan; the vector similarity index, available from version 25.8, builds an HNSW graph per data part, which can be quantized while the stored column stays unchanged [7]. Filters apply after vector search by default, with an option to pre-filter and a multiplier that fetches extra candidates to offset post-filter losses [7].

8.5 Embedded engines and columnar formats

LanceDB applies to vectors the embedded model that paper 3 described for DuckDB. It stores data in the Lance columnar format, designed for both scans and random access [8], and builds IVF-based indexes with PQ, RaBitQ or scalar-quantized HNSW within partitions, with index files on disk [30]. It suits applications that keep vectors beside other columnar data without a separate server.

9. Comparative analysis

Table 1 compares index families and system architectures discussed in this paper. The ratings are qualitative and relative to each other. Actual behavior depends on data, embedding model, query distribution, filters and configuration.

Index family or architecture	Memory footprint	Recall control	Update support	Filtering support	Best-fit workloads
Exact (flat) search	Low (no index structure)	Exact results	High	High (filter first, then scan)	Small collections, per-tenant data, highly selective filters, ground truth for evaluation
In-memory graph (HNSW and variants)	High (graph plus vectors; reducible by quantization)	High; tuned at query time with the search beam	Medium (incremental inserts; deletes need cleanup)	Medium; High with filter-aware edges or ACORN-style traversal	Low-latency search on collections that fit in RAM

Index family or architecture	Memory footprint	Recall control	Update support	Filtering support	Best-fit workloads
Partition index (IVF with PQ or RaBitQ)	Low to Medium	Medium; tuned with probes and reranking	Medium (cheap assignment; drift unbalances lists)	Medium	Large collections under memory budgets; GPU and batch search
SSD-resident graph (DiskANN, Starling)	Low in RAM; data on SSD	High, at higher latency than in-memory graphs	Medium (batch consolidation or in-place updates)	Medium to High (Filtered-DiskANN)	Collections that exceed RAM where SSD latency is acceptable
SSD-resident partitions (SPANN, SPFresh)	Low in RAM; data on SSD	Medium to High	Medium to High (incremental rebalancing)	Medium	Very large, continuously updated collections
GPU graph or IVF (CAGRA, Faiss GPU)	High in accelerator memory	High	Low to Medium (batch rebuilds)	Low to Medium	High-throughput batch search; fast index builds
Vector index in a general-purpose database (pgvector, MongoDB, Redis, ClickHouse)	Medium to High (engine-dependent)	Medium to High	High (transactional or near-real-time writes)	High expressiveness (predicates on co-located data); recall depends on strategy	Applications whose data and access control already live in that database
Purpose-built distributed vector database (Milvus/Manu, Qdrant, Weaviate)	Configurable (memory, disk and quantization tiers)	High	Medium to High	High	Large or multi-tenant vector workloads that need dedicated scaling and index choice

Table 1: Qualitative comparison of vector index families and system architectures.

10. Open challenges and future opportunities

10.1 Filtered and hybrid search quality

Challenge. Filters and lexical signals are the norm in production, yet recall under selective or uncorrelated filters remains unstable, and fused rankings are sensitive to fusion choices [35], [41], [43]. **Research direction.** Filter-aware graphs, predicate-agnostic construction and cost-based choice among filtering strategies are advancing [38], [39], [50], and the NeurIPS'23 competition made filtered search a benchmark track [15]. **Opportunity.** Query optimizers that accept a recall target and choose the filtering strategy and search budget to meet it would make filtered retrieval predictable without per-query tuning.

10.2 Freshness and updates at scale

Challenge. Graph indexes resist deletion, partition indexes drift out of balance, and asynchronous pipelines delay visibility [45], [47], [24]. **Research direction.** In-place graph updates, incremental partition rebalancing and log-structured system designs address each

part [46], [47], [48]. **Opportunity.** As with LSM-trees in paper 2, engines that expose freshness lag and physical-deletion delay as explicit, bounded properties would help teams meet both user expectations and deletion obligations in RAG systems.

10.3 The cost of memory versus disk

Challenge. In-memory graphs give low latency but make RAM the dominant cost, and segment-based databases constrain memory per segment [31]. **Research direction.** SSD-resident graphs and partitions, quantization with provable error bounds, and reranking from cheaper storage reduce the RAM footprint [19], [21], [28]. **Opportunity.** Placing graph, codes and full vectors across RAM, SSD and object storage by access frequency, with recall and latency targets as constraints, parallels the tiering questions paper 3 raised for analytical data.

10.4 Benchmark realism and out-of-distribution queries

Challenge. Many evaluations draw queries from the same distribution as the data. Cross-modal queries, such as text against image embeddings, are out of distribution, and Chen et al. show that this breaks assumptions behind existing graph indexes [55]. **Research direction.** Query-guided graphs such as RoarGraph [55], out-of-distribution and streaming tracks in Big-ANN [15], and broader experimental studies [17] move evaluation closer to practice. **Opportunity.** Benchmarks built from real query logs, filters and update streams, reported across the whole trade-off curve, would match how systems are actually used.

10.5 Embedding-model drift and re-indexing

Challenge. Replacing an embedding model normally means re-encoding the corpus and rebuilding every index, which is costly and disruptive, so systems tend to stay on older models. **Research direction.** Drift-Adapter learns a lightweight mapping from new query embeddings into the legacy space, so the existing index keeps serving; its author reports recall close to full re-embedding on the tested corpora [56]. **Opportunity.** Treating the embedding model version as index metadata, with tooling for dual serving, adapters and staged backfills, would turn model upgrades into routine migrations.

10.6 Multi-vector and multimodal retrieval

Challenge. Late-interaction models such as ColBERTv2 represent each document as many token vectors, multiplying storage and search cost [57], and multimodal models add cross-modal queries [55]. **Research direction.** PLAID prunes candidates with centroid interactions before full scoring, and its authors report lower latency than the unoptimized ColBERTv2 pipeline without loss of retrieval quality [57]. Weaviate encodes multi-vector embeddings as single vectors for search and rescoring exactly [24]. **Opportunity.** Storage layouts and indexes designed for sets of vectors per item, rather than adapted from single-vector indexes, remain an open area.

10.7 Security, privacy and access control

Challenge. Embeddings are not anonymous: Morris et al. reconstructed much of the original text from text embeddings, including personal names from clinical notes [58]. Enforcing permissions also forces a choice between per-user indexes, which duplicate data, and shared indexes with post-filtering, which add latency [59]. **Research direction.** HoneyBee uses the role structure of role-based access control to build overlapping partitions that trade memory for latency in a controlled way [59]. **Opportunity.** Vector stores should be governed like the source

records, with equal encryption, retention and deletion policies, and with access control evaluated inside retrieval rather than after it.

10.8 Convergence into general-purpose databases

Challenge. A separate vector database adds synchronization, consistency and operational overhead, while extensions inside general-purpose databases have historically trailed specialized systems [52]. **Research direction.** VBase and AnalyticDB-V show how vector operators can join relational query processing [50], [49], and Zhang et al. trace the remaining gap in PostgreSQL to implementation issues rather than fundamental limits [52]. **Opportunity.** As vector search becomes a database feature, the question shifts from which vector database to adopt to where vectors should live relative to the data, permissions and transactions they depend on.

11. What this means for platform decisions

Retrieval quality depends on the fit between index, filters, freshness requirements and data, more than on the vector engine alone. In Bassirah's platform work, vector search starts inside an existing database where it fits: PostgreSQL with pgvector; MongoDB, a Strategic Technology Partner, either as partner-led MongoDB Atlas Vector Search or as self-managed MongoDB Community Server with mongot, which is source-available under the SSPL; or Redis, a Strategic Technology Partner, where low latency matters. A dedicated vector engine is considered when scale, filtering or update patterns call for one. In each case, recall, latency and cost are measured on the customer's own data and queries before adoption, retrieval quality in production is evaluated and observed with Langfuse, a Strategic Technology Partner, and no outcome is assumed from published results.

References

1. Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., Wang, X., Guo, X., Li, C., Xu, X., et al. (2021). Milvus: A Purpose-Built Vector Data Management System. SIGMOD, 2614–2627. (<https://doi.org/10.1145/3448016.3457550>)
2. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S. and Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems 33 (NeurIPS). (<https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html>)
3. Pan, J. J., Wang, J. and Li, G. (2024). Survey of vector database management systems. The VLDB Journal, 33(5), 1591–1615. (<https://doi.org/10.1007/s00778-024-00864-x>)
4. Aumüller, M., Bernhardsson, E. and Faithfull, A. (2020). ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. Information Systems, 87, 101374. (<https://doi.org/10.1016/j.is.2019.02.006>)
5. pgvector project (2026). pgvector: Open-source vector similarity search for Postgres. Official repository and documentation, accessed 26 October 2026. (<https://github.com/pgvector/pgvector>)
6. Redis (2026). Vector search concepts. Official documentation, accessed 26 October 2026. (<https://redis.io/docs/latest/develop/ai/search-and-query/vectors/>)

7. ClickHouse (2026). Exact and Approximate Vector Search. Official documentation, accessed 26 October 2026. (<https://clickhouse.com/docs/engines/table-engines/mergetree-family/annindexes>)
8. Pace, W., She, C., Xu, L., Jones, W., Lockett, A., Wang, J. and Shah, R. (2025). Lance: Efficient Random Access in Columnar Storage through Adaptive Structural Encodings. arXiv preprint 2504.15247. (<https://arxiv.org/abs/2504.15247>)
9. Elastic (2026). Dense vector field type. Official Elasticsearch documentation, accessed 26 October 2026. (<https://www.elastic.co/docs/reference/elasticsearch/mapping-reference/dense-vector>)
10. Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F. and Kumar, S. (2020). Accelerating Large-Scale Inference with Anisotropic Vector Quantization. ICML, PMLR 119, 3887–3896. (<https://proceedings.mlr.press/v119/guo20h.html>)
11. Malkov, Y. A. and Yashunin, D. A. (2020). Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence, 42(4), 824–836. (<https://doi.org/10.1109/TPAMI.2018.2889473>)
12. Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L. and Jégou, H. (2026). The Faiss Library. IEEE Transactions on Big Data, 12(2), 346–361. (<https://doi.org/10.1109/TBDATA.2025.3618474>)
13. Beyer, K., Goldstein, J., Ramakrishnan, R. and Shaft, U. (1999). When Is "Nearest Neighbor" Meaningful? Database Theory — ICDT'99, Lecture Notes in Computer Science, 217–235. (https://doi.org/10.1007/3-540-49257-7_15)
14. Simhadri, H. V., Williams, G., Aumüller, M., Douze, M., Babenko, A., Baranchuk, D., Chen, Q., Hosseini, L., Krishnaswamy, R., Srinivasa, G., et al. (2022). Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. Proceedings of the NeurIPS 2021 Competitions and Demonstrations Track, PMLR 176, 177–189. (<https://proceedings.mlr.press/v176/simhadri22a.html>)
15. Simhadri, H. V., Aumüller, M., Ingber, A., Douze, M., Williams, G., Manohar, M. D., Baranchuk, D., Liberty, E., Liu, F., Landrum, B., et al. (2024). Results of the Big ANN: NeurIPS'23 competition. arXiv preprint 2409.17424. (<https://arxiv.org/abs/2409.17424>)
16. Wang, M., Xu, X., Yue, Q. and Wang, Y. (2021). A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. PVLDB, 14(11), 1964–1978. (<https://doi.org/10.14778/3476249.3476255>)
17. Azizi, I., Echihabi, K. and Palpanas, T. (2025). Graph-Based Vector Search: An Experimental Evaluation of the State-of-the-Art. Proceedings of the ACM on Management of Data (SIGMOD), 3(1). (<https://doi.org/10.1145/3709693>)
18. Fu, C., Xiang, C., Wang, C. and Cai, D. (2019). Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. PVLDB, 12(5), 461–474. (<https://doi.org/10.14778/3303753.3303754>)
19. Subramanya, S. J., Devvrit, F., Simhadri, H. V., Krishnaswamy, R. and Kadekodi, R. (2019). DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. Advances in Neural Information Processing Systems 32 (NeurIPS). (<https://papers.nips.cc/paper/2019/hash/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Abstract.html>)

20. Jégou, H., Douze, M. and Schmid, C. (2011). Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1), 117–128. (<https://doi.org/10.1109/TPAMI.2010.57>)
21. Chen, Q., Zhao, B., Wang, H., Li, M., Liu, C., Li, Z., Yang, M. and Wang, J. (2021). SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. *Advances in Neural Information Processing Systems 34 (NeurIPS)*. (<https://proceedings.neurips.cc/paper/2021/hash/299dc35e747eb77177d9cea10a802da2-Abstract.html>)
22. Sun, P., Simcha, D., Dopson, D., Guo, R. and Kumar, S. (2023). SOAR: Improved Indexing for Approximate Nearest Neighbor Search. *Advances in Neural Information Processing Systems 36 (NeurIPS)*. (https://proceedings.neurips.cc/paper_files/paper/2023/hash/0973524e02a712af33325d0688ae6f49-Abstract.html)
23. Li, C., Zhang, M., Andersen, D. G. and He, Y. (2020). Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. *SIGMOD*, 2539–2554. (<https://doi.org/10.1145/3318464.3380600>)
24. Weaviate (2026). Vector indexing. Official documentation, accessed 26 October 2026. (<https://docs.weaviate.io/weaviate/concepts/vector-index>)
25. Ge, T., He, K., Ke, Q. and Sun, J. (2014). Optimized Product Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(4), 744–755. (<https://doi.org/10.1109/TPAMI.2013.240>)
26. MongoDB (2026). Vector Quantization. Official MongoDB Vector Search documentation, accessed 26 October 2026. (<https://www.mongodb.com/docs/atlas/atlas-vector-search/vector-quantization/>)
27. Gao, J. and Long, C. (2024). RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data (SIGMOD)*, 2(3). (<https://doi.org/10.1145/3654970>)
28. Gao, J., Gou, Y., Xu, Y., Yang, Y., Long, C. and Wong, R. C.-W. (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 (SIGMOD)*, 3(3). (<https://doi.org/10.1145/3725413>)
29. Milvus project (2026). Index Explained. Official documentation, accessed 26 October 2026. (<https://milvus.io/docs/index-explained.md>)
30. LanceDB (2026). Vector Indexes. Official documentation, accessed 26 October 2026. (<https://docs.lancedb.com/indexing/vector-index>)
31. Wang, M., Xu, W., Yi, X., Wu, S., Peng, Z., Ke, X., Gao, Y., Xu, X., Guo, R. and Xie, C. (2024). Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proceedings of the ACM on Management of Data (SIGMOD)*, 2(1). (<https://doi.org/10.1145/3639269>)
32. OpenSearch project (2026). Methods and engines; Hybrid search. Official documentation, accessed 26 October 2026. (<https://docs.opensearch.org/latest/mappings/supported-field-types/knn-methods-engines/>; <https://docs.opensearch.org/latest/vector-search/ai-search/hybrid-search/index/>)
33. Johnson, J., Douze, M. and Jégou, H. (2021). Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7(3), 535–547. (<https://doi.org/10.1109/TBDATA.2019.2921572>)

34. Ootomo, H., Naruse, A., Nolet, C., Wang, R., Feher, T. and Wang, Y. (2024). CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. IEEE ICDE, 4236–4247. (<https://doi.org/10.1109/ICDE60146.2024.00323>)
35. Chronis, Y., Caminal, H., Papakonstantinou, Y., Özcan, F. and Ailamaki, A. (2025). Filtered Vector Search: State-of-the-Art and Research Opportunities. PVLDB, 18(12), 5488–5492. (<https://doi.org/10.14778/3750601.3750700>)
36. Qdrant (2026). Indexing. Official documentation, accessed 26 October 2026. (<https://qdrant.tech/documentation/concepts/indexing/>)
37. Elastic (2026). Filter approximate kNN results. Official Elasticsearch documentation, accessed 26 October 2026. (<https://www.elastic.co/docs/solutions/search/vector/knn/filtered-knn-search>)
38. Gollapudi, S., Karia, N., Sivashankar, V., Krishnaswamy, R., Begwani, N., Raz, S., Lin, Y., Zhang, Y., Mahapatro, N., Srinivasan, P., et al. (2023). Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. Proceedings of the ACM Web Conference (WWW), 3406–3416. (<https://doi.org/10.1145/3543507.3583552>)
39. Patel, L., Krafft, P., Guestrin, C. and Zaharia, M. (2024). ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. Proceedings of the ACM on Management of Data (SIGMOD), 2(3). (<https://doi.org/10.1145/3654923>)
40. Weaviate (2026). Filtering. Official documentation, accessed 26 October 2026. (<https://docs.weaviate.io/weaviate/concepts/filtering>)
41. Li, M., Yan, X., Lu, B., Zhang, Y., Cheng, J. and Ma, C. (2025). Attribute Filtering in Approximate Nearest Neighbor Search: An In-depth Experimental Study. Proceedings of the ACM on Management of Data (SIGMOD), 3(6). (<https://doi.org/10.1145/3769763>)
42. Elastic (2026). Reciprocal rank fusion. Official Elasticsearch documentation, accessed 26 October 2026. (<https://www.elastic.co/docs/reference/elasticsearch/rest-apis/reciprocal-rank-fusion>)
43. Bruch, S., Gai, S. and Ingber, A. (2023). An Analysis of Fusion Functions for Hybrid Retrieval. ACM Transactions on Information Systems, 42(1). (<https://doi.org/10.1145/3596512>)
44. MongoDB (2026). MongoDB Vector Search Overview. Official documentation, accessed 26 October 2026. (<https://www.mongodb.com/docs/atlas/atlas-vector-search/vector-search-overview/>)
45. Singh, A., Subramanya, S. J., Krishnaswamy, R. and Simhadri, H. V. (2021). FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. arXiv preprint 2105.09613. (<https://arxiv.org/abs/2105.09613>)
46. Xu, H., Manohar, M. D., Bernstein, P. A., Chandramouli, B., Wen, R. and Simhadri, H. V. (2025). In-Place Updates of a Graph Index for Streaming Approximate Nearest Neighbor Search. arXiv preprint 2502.13826. (<https://arxiv.org/abs/2502.13826>)
47. Xu, Y., Liang, H., Li, J., Xu, S., Chen, Q., Zhang, Q., Li, C., Yang, Z., Yang, F., Yang, Y., et al. (2023). SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. ACM SOSP, 545–561. (<https://doi.org/10.1145/3600006.3613166>)
48. Guo, R., Luan, X., Xiang, L., Yan, X., Yi, X., Luo, J., Cheng, Q., Xu, W., Luo, J., Liu, F., et al. (2022). Manu: A Cloud Native Vector Database Management System. PVLDB, 15(12), 3548–3561. (<https://doi.org/10.14778/3554821.3554843>)
49. Wei, C., Wu, B., Wang, S., Lou, R., Zhan, C., Li, F. and Cai, Y. (2020). AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. PVLDB, 13(12), 3152–3165. (<https://doi.org/10.14778/3415478.3415541>)

50. Zhang, Q., Xu, S., Chen, Q., Sui, G., Xie, J., Cai, Z., Chen, Y., He, Y., Yang, Y., Yang, F., et al. (2023). VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. USENIX OSDI, 377–395.
(<https://www.usenix.org/conference/osdi23/presentation/zhang-qianxi>)
51. YugabyteDB, Inc. (2026). Explore YugabyteDB's Vector Indexing Architecture. Official engineering blog, accessed 26 October 2026.
(<https://www.yugabyte.com/blog/yugabytedb-vector-indexing-architecture/>)
52. Zhang, Y., Liu, S. and Wang, J. (2024). Are There Fundamental Limitations in Supporting Vector Data Management in Relational Databases? A Case Study of PostgreSQL. IEEE ICDE, 3640–3653. (<https://doi.org/10.1109/ICDE60146.2024.00280>)
53. MongoDB (2026). Self-Managed MongoDB Search and Vector Search (About mongot). Official documentation, accessed 26 October 2026.
(<https://www.mongodb.com/docs/search/self-managed/current/>)
54. MongoDB (2026). Now GA: MongoDB Search and Vector Search for Community Edition. Official product update, 30 June 2026, accessed 26 October 2026.
(<https://www.mongodb.com/products/updates/now-ga-mongodb-search-and-vector-search-for-community-edition/>)
55. Chen, M., Zhang, K., He, Z., Jing, Y. and Wang, X. S. (2024). RoarGraph: A Projected Bipartite Graph for Efficient Cross-Modal Approximate Nearest Neighbor Search. PVLDB, 17(11), 2735–2749. (<https://doi.org/10.14778/3681954.3681959>)
56. Vejendla, H. (2025). Drift-Adapter: A Practical Approach to Near Zero-Downtime Embedding Model Upgrades in Vector Databases. Proceedings of EMNLP, 15949–15960.
(<https://doi.org/10.18653/v1/2025.emnlp-main.805>)
57. Santhanam, K., Khattab, O., Potts, C. and Zaharia, M. (2022). PLAID: An Efficient Engine for Late Interaction Retrieval. ACM CIKM, 1747–1756.
(<https://doi.org/10.1145/3511808.3557325>)
58. Morris, J. X., Kuleshov, V., Shmatikov, V. and Rush, A. M. (2023). Text Embeddings Reveal (Almost) As Much As Text. Proceedings of EMNLP, 12448–12460.
(<https://doi.org/10.18653/v1/2023.emnlp-main.765>)
59. Zhong, H., Lentz, M., Narodytska, N., Szekeres, A. and Rong, K. (2026). HONEYBEE: Efficient Role-based Access Control for Vector Databases via Dynamic Partitioning. Proceedings of the ACM on Management of Data (SIGMOD), 4(1). (<https://doi.org/10.1145/3786625>)

Product and project names are trademarks of their respective owners. Their mention describes publicly documented technology and does not imply a business relationship, except where Bassirah names a Strategic Technology Partner.