TEngineDB-V: An OLAP-Native Vector Search System for Large-k Workloads at Tencent

Xufei Wu*
Shanghai Jiao Tong Univ.
wuxufei0807@gmail.com

Pengcheng Zhang*
Tencent Inc.
petrizhang@tencent.com

Yitong Song
Hong Kong Baptist Univ.
ytsong@comp.hkbu.edu.hk

Xiaobo Zhang
Shanghai Jiao Tong Univ.
strange_uncle@sjtu.edu.cn

Anqi Liang
HKUST(GZ)
anqiliang9657@gmail.com

Kai Wang
Shanghai Jiao Tong Univ.
w.kai@sjtu.edu.cn

Jijun Du
Tencent Inc.
jijundu@tencent.com

Yidi Xiong
Tencent Inc.
eddyxiong@tencent.com

Guangxu Cheng
Tencent Inc.
andrewcheng@tencent.com

Zhe Chen
Tencent Inc.
octopuschen@tencent.com

Peng Chen
Tencent Inc.
pengchen@tencent.com

Guoliang Li
Tsinghua Univ.
liguoliang@tsinghua.edu.cn

Xuanhe Zhou
Shanghai Jiao Tong Univ.
zhouxuanhe@sjtu.edu.cn

Fan Wu
Shanghai Jiao Tong Univ.
fwu@cs.sjtu.edu.cn

ABSTRACT

Vector search systems are essential infrastructure for modern data-driven applications. Large-k analytical vector search, which retrieves $k = 10^3$–10^5 results for analytics (e.g., aggregation, filtering, joins), is increasingly important for emerging workloads, including LLM data management and advertising analysis at Tencent. Existing systems remain inadequate: specialized vector databases often cap k (e.g., $k \leq 10^4$) to satisfy tail-latency constraints and offer limited analytical support, while OLAP systems typically embed per-segment vector indexes as black boxes, causing severe read/-compute amplification and preventing native query optimization.

This paper presents TEngineDB-V, an OLAP-native vector search system for large-k workloads. TEngineDB-V makes vector search a first-class analytical primitive in Tencent's OLAP engine through a global segment-decoupled index materialized as relational tables, eliminating scatter–gather execution, reducing amplification, and enabling native storage optimizations. It decomposes IVFPQ-based search into relational operators, integrates OLAP optimizations, and introduces DPPQ, which combines direction-aware quantization with hierarchical residual refinement to improve recall while preserving relational efficiency. TEngineDB-V further incorporates index-aware query rewriting and a distributed-aware cost model for efficient distributed execution. Experiments show that TEngineDB-V achieves up to a 145× speedup over competitive systems such as StarRocks, and up to a 52× improvement in 10-billion-scale production deployments.

PVLDB Reference Format:
Xufei Wu, Pengcheng Zhang, Yitong Song, Xiaobo Zhang, Anqi Liang, Kai Wang, Jijun Du, Yidi Xiong, Guangxu Cheng, Zhe Chen, Peng Chen, Guoliang Li, Xuanhe Zhou, and Fan Wu. TEngineDB-V: An OLAP-Native Vector Search System for Large-k Workloads at Tencent. PVLDB, 19(12): 3955 - 3968, 2026.
doi:10.14778/3827998.3828008

*Equal contribution.

Table 1: Comparison of small-k and large-k workloads.

Workload	k	Latency	Serving	Applications
Small-k	1 ~ 1K	Low	Answers	Image Retrieval, RAG
Large-k	1K ~ 100K	Moderate	Exploratory	Training Set Retrieval, Advertising Data Analysis

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/petrizhang/relify.

1 INTRODUCTION

Vector search has become a fundamental building block of modern database and multimodal search systems [4, 8, 13, 15, 32, 34, 46–48]. It typically involves two common query types: Approximate k-Nearest Neighbor Search (ANNS), which retrieves the top-k most similar vectors from a large-scale dataset, and Filtered ANNS (FANNS), which incorporates attribute-based predicates to restrict the search scope and returns the top-k most similar vectors within the filtered subset. With the emergence of new applications, vector search workloads are no longer confined to small-k scenarios. Instead, many applications require a large number of results, typically ranging from 10^3 to 10^5. For example, in our production environment at Tencent, we maintain a corpus of over 10 billion images within TEngineDB-V to serve the training set retrieval and analysis requests for Text-to-Image/Visual model developers [37–39, 41]. They typically retrieve 10^4–10^5 images to build domain-specific fine-tuning datasets [17, 40]. They also use our system to compute aggregate statistics that characterize the quality and distribution of training sets that either semantically correspond to particular text prompts or are visually similar to given reference images.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828008

Table 2: Comparison of representative specialized and database-integrated vector search systems. "Segment-coupled" indicates per-segment local indexing, whereas "Segment-decoupled" denotes a globally organized index across segments.

System	Index Persistence	Index Organization	Index Type	Max k	Execution Operator	Native Integration
Milvus [50]	Binary files	Segment-coupled	HNSW, Vamana, IVFPQ, IVFRaBitQ	16K (default)	Dedicated	Specialized
PGVector [34]	Binary files	Segment-decoupled	HNSW, IVFFlat	10K	Dedicated	✗
GaussDB-Vector [48]	Binary files	Segment-coupled	Vamana, IVF-Based	Not disclosed	Dedicated	✗
ElasticSearch [15]	Binary files	Segment-coupled	HNSW, BBQ	10K	Dedicated	✗
Doris [4]	Binary files	Segment-coupled	HNSW, IVF-Based	Unlimited	Dedicated	✗
StarRocks [47]	Binary files	Segment-coupled	HNSW, IVFPQ	Unlimited	Dedicated	✗
SingleStore-V [8]	Binary files	Segment-coupled	HNSW, IVFFlat, IVFPQ	Not disclosed	Dedicated	✗
BlendHouse [32]	Binary files	Segment-coupled	HNSW, Vamana, IVF-Based	Unlimited	Dedicated	✗
TEngineDB-V (ours)	**Tables**	**Segment-decoupled**	**IVFPQ, IVFDPPQ (ours)**	**Unlimited**	**Relational**	✓

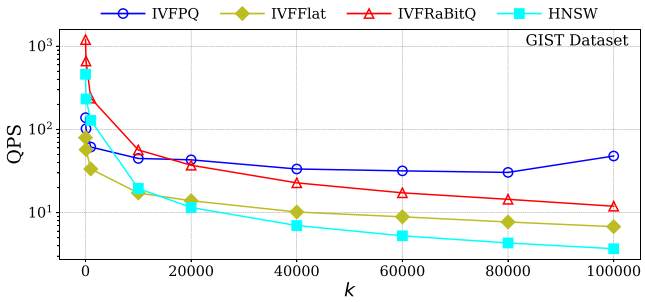


Figure 1: QPS comparison of vector indexes under varying k.

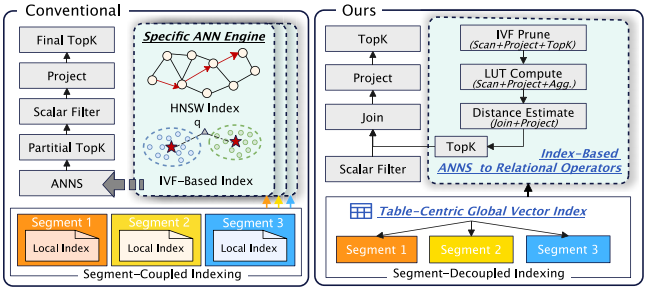


Figure 2: Relational vector search paradigm.

As shown in Table 1, unlike traditional small-k workloads, which return only a few results under strict latency constraints [5], large-k workloads target exploratory analytics over substantial similarity sets. Their latency requirements are inherently more relaxed, as the computational cost grows significantly with k. This shift introduces new requirements to the underlying systems: (1) *Scalability under large result cardinality.* The system must efficiently process numerous results within acceptable latency, while avoiding excessive read/compute amplification. (2) *Native integration with analytical execution.* Large-k retrieval is rarely the final step. Instead, the retrieved similarity sets are further processed by downstream analytical operators (e.g., filtering, aggregation, join). Therefore, *vector search should be tightly integrated into the analytical engine as a native operator* [22] to enable joint optimization with other analytical operators. However, existing vector search systems, including both specialized systems [1, 9, 50] and general-purpose database extensions [4, 8, 15, 32, 34, 47, 48], are primarily optimized for small-k workloads and lack deep co-optimization between vector search operators and the native analytical pipelines. Therefore, they exhibit the following two main limitations.

Limitation 1: Limited Scalability to Large-k Serving. As shown in Table 2, many representative vector search systems [15, 34, 50] impose explicit upper bounds on k (typically 100–16K) to control tail latency. Although some systems (e.g., StarRocks [47], Doris [4]) do not enforce a hard limit, they exhibit poor scalability when extended to large-k retrieval, primarily due to two factors. (1) *Segment-coupled indexing and scatter–gather execution.* Many systems [4, 22, 32, 47, 50] maintain independent vector indexes per data segment and employ scatter–gather execution for top-k vector similarity search. When a table is split into N segments, this forces the system to fetch at least k records from each segment, inflating the result volume to $N \times k$ (e.g., 200M records for k=100,000 and N=2,000). This amplification cascades into excessive disk I/O, network traffic, and global merge costs. (2) *Degradation of graph-based search under large-k.* Most systems rely on graph-based indexes (e.g., HNSW [31]) for efficient small-k retrieval. However, as k increases, the candidate set expands rapidly, causing substantial computation for both vector evaluation and intermediate result maintenance (see details in Figure 1).

Limitation 2: Loose Integration with Underlying Systems. To support vector search and analytics, popular OLAP systems typically integrate vector indexes as a third-party component (e.g., using the Faiss [1] library) that is loosely coupled with the system [4, 22, 47]. These indexes are stored as isolated files, and their execution logic remains black-box to the optimizer and compute engine. Therefore, general-purpose database optimizations cannot be directly applied to the index search. For example, systems typically invoke a black-box `IVFScan` operator to probe top-n clusters, making the following general-purpose database optimizations, including cardinality estimation (lack visibility into cluster distributions), I/O pruning (fail to access specific clusters on disk selectively), and fine-grained caching (cache granularity in index file level rather than individual clusters), infeasible.

To this end, we propose `TEngineDB-V`, an OLAP-native large-k vector search engine built on the production-grade OLAP system `TEngineDB` deployed at Tencent. As illustrated in Figure 2, `TEngineDB-V` introduces a new paradigm for large-k vector search: it materializes index structures as relational tables and executes vector search through composable relational operators.

Specifically, (1) at storage layer, `TEngineDB-V` builds a segment-decoupled global index spanning all data segments, which is fully decoupled from raw data and materialized as standard relational tables, enabling native optimizations such as columnar compression and fine-grained caching. (2) At the computation layer, `TEngineDB-V`

adopts a relational execution paradigm, transforming IVFPQ-based search into a composable sequence of relational operators and enabling native optimizations such as late materialization and runtime filters. This design is particularly effective for large-k workloads, as it leverages the multi-threaded OLAP engine to parallelize key operations (e.g., inverted-list scanning, LUT construction, and distance computation), avoiding the scalability limitations of traditional single-threaded execution. However, traditional IVFPQ-based search often relies on low-precision distance estimation and therefore requires refining $2k$–$3k$ candidate vectors to ensure accuracy, making refinement increasingly expensive as k grows. To address this issue, we develop DPPQ, which improves distance estimation accuracy by redesigning the quantization mechanism and integrating refinement directly into fused relational operators, avoiding the costly large-scale raw-vector refinement stage. (3) At the control layer, we design an index-aware query rewriting module that injects vector index semantics into the relational optimizer and enumerates logically equivalent plans. A distributed-aware cost model further evaluates these plans by jointly modeling CPU, memory, and network costs. For large-k hybrid queries, this optimization is essential since substantial intermediate results amplify join, materialization, and network overheads.

Product Impact. TEngineDB-V has been widely deployed in production environments at Tencent, powering advertising analysis, LLM services, and multimodal data analytics. Our largest dedicated cluster for large-k workloads comprises 30+ compute nodes managing 100TB of data with about 10 billion image embeddings, consistently delivering <5s latency for top-100,000 search.

Contributions. Our contributions are summarized as follows.

(1) **We propose TEngineDB-V, the first OLAP-native vector search system designed for large-k workloads.** It introduces a new paradigm that tightly integrates IVFPQ-style indexing with relational storage and execution, enabling scalable large-result retrieval within production-grade OLAP infrastructures.
(2) **We design a segment-decoupled global indexing framework with table-centric materialization.** These designs eliminate exhaustive scatter-gather execution across all segments and substantially reduce read/compute amplification under large-k workloads, while enabling native storage-level optimizations.
(3) **We present a relational execution paradigm for vector search with an enhanced quantization method.** We decompose IVFPQ-based search into composable relational operators, allowing seamless integration and OLAP-native optimizations. To reduce quantization error, we develop DPPQ, which leverages direction-aware quantization and hierarchical residual refinement to improve accuracy while keeping relational efficiency.
(4) **We develop an index-aware distributed optimization framework.** The framework injects vector index semantics into logical plans and jointly models CPU, memory, and network costs to select efficient execution plans in distributed environments.
(5) **We conduct extensive experimental and production evaluations at billion scale.** Experimental results demonstrate up to a 145× speedup over competitive systems under large-k workloads, while 10-billion-scale production deployment achieves up to a 52× performance gain.

2 PRELIMINARIES

This section first presents two typical vector search query types, followed by an overview of representative vector search systems and the IVFPQ index.

2.1 Two Query Types for Vector Search

ANNS. Given a dataset $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$ and a query vector $\mathbf{q} \in \mathbb{R}^d$, k-nearest neighbor search (kNNS) returns a subset $S_k \subset X$ with $|S_k| = k$ such that for any $\mathbf{x} \in S_k$ and $\mathbf{y} \in X \setminus S_k$, $\text{dist}(\mathbf{q}, \mathbf{x}) \leq \text{dist}(\mathbf{q}, \mathbf{y})$. In high-dimensional spaces, exact kNNS is computationally prohibitive due to the curse of dimensionality [31]. Approximate nearest neighbor search (ANNS) relaxes the accuracy requirement by returning vectors whose distances are close to those of the true top-k neighbors with high probability, thereby substantially reducing computational overhead while maintaining strong empirical accuracy [27].

FANNS. Filtered Approximate Nearest Neighbor Search (FANNS) augments ANNS with attribute predicates evaluated alongside vector similarity [28, 44, 54, 58]. Formally, given a filtering predicate $P : X \to \{\text{true}, \text{false}\}$, let $X_P = \{\mathbf{x} \in X \mid P(\mathbf{x}) = \text{true}\}$ be the subset of vectors that satisfy the predicate. FANNS identifies a subset $S_k \subset X_P$ of size $\min(k, |X_P|)$ such that $\text{dist}(\mathbf{q}, \mathbf{x}) \leq \text{dist}(\mathbf{q}, \mathbf{y})$ for any $\mathbf{x} \in S_k$ and $\mathbf{y} \in X_P \setminus S_k$.

2.2 Existing Vector Search Systems

Specialized Vector Search Systems and Libraries. Milvus [50], Pinecone [35], Weaviate [53], Qdrant [36], FAISS [1], and SPTAG [9] are specialized ANNS systems or libraries built on PQ- or graph-based indexes (e.g., HNSW [31]), primarily optimized for latency-sensitive small-k workloads. As k grows, the number of maintained candidates and intermediate results increases substantially, enlarging the effective search space and diminishing pruning effectiveness. Therefore, under large-k analytical workloads, they often suffer from significantly reduced query efficiency.

Database-Integrated Vector Search Systems. To bridge vector search and relational analytics, recent database and analytical systems have introduced vector search extensions within their architectures, including Elastic Search [15], PGVector [34], GaussDB-vector [48], Doris [4], StarRocks [47], SingleStore-V [8], and BlendHouse [32]. In these designs, vector search is implemented as a specialized module whose execution logic remains opaque to the optimizer and compute engine. As a result, despite the availability of mature OLAP optimizations, e.g., columnar compression and vectorized execution, these modules cannot fully exploit them.

Under large-k workloads, these systems often exhibit severe performance degradation. This is largely due to segment-coupled indexing and scatter–gather execution: each data segment maintains an independent local index, and execution requires probing all segments and merging their partial results. As k increases, this fragmented execution amplifies I/O, intermediate materialization, and network overhead, leading to serious read/compute amplification.

2.3 IVFPQ Algorithms

IVFPQ [24] is one of the most widely adopted indexing structures for large-scale ANNS. As our system is built upon an IVFPQ-based search, we briefly introduce its fundamental principles below.

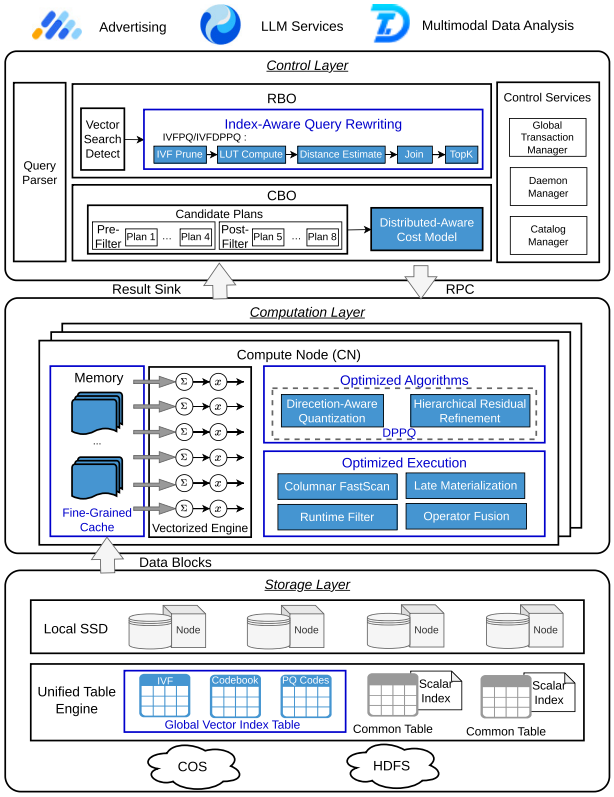


Figure 3: The Architecture of TEngineDB-V.

IVFPQ uses k-means clustering [25] to partition vectors into k clusters and searches the top-N_{probe} clusters closest to the query. To further accelerate distance computation within each cluster, Product Quantization (PQ) is applied to encode the vectors. The d-dimensional vector is partitioned into m disjoint subspaces, each of dimension d/m. For each subspace, a separate codebook with C centroids is learned using k-means clustering. Each vector is therefore represented by an m-dimensional code, where each dimension is an integer in $\{1, \ldots, C\}$ indicating the assigned subspace centroid.

During query execution, the query vector q is decomposed into m sub-vectors q_i. For each subspace, a lookup table (LUT) is constructed by computing the squared distances between q_i and all C centroids. The approximate squared distance between q and a vector x is then computed as $\text{dist}(q, x)^2 = \sum_{i=1}^{m} d(j, i)$, where j is the PQ code index of x in the i-th subspace, and $d(j, i)$ is the corresponding entry in the precomputed distance table. This formulation allows efficient evaluation using only mC (d/m)-dimensional distance computations to construct the LUT and mn table lookups to score n candidate vectors. For ANNS, the vectors with the k smallest distances are returned.

3 SYSTEM OVERVIEW

3.1 System Architecture

Figure 3 presents the system architecture of TEngineDB-V, a cloud-native multi-modal data warehouse adopting a disaggregated storage paradigm where control plane, compute nodes, and storage layer operate as independent services.

Control Layer. The control layer integrates cluster-level management services (transaction, daemon, and catalog managers). Furthermore, it parses incoming SQL statements and performs both rule-based and cost-based optimizations (RBO and CBO) to generate efficient execution plans. Specifically, the RBO module recognizes vector/hybrid queries and rewrites them into index-aware operators. Instead of treating vector index search as a black-box operation, TEngineDB-V decomposes IVFPQ search into explicit relational operators. These operators can be natively optimized by the underlying OLAP compute engine. Second, for FANNS queries, the CBO module enumerates alternative physical plans in consideration of various execution strategies (i.e., pre-filtering vs. post-filtering, left join vs. right join, and broadcast join vs. shuffle join). A distributed-aware cost model is developed to jointly estimate CPU, memory, and network I/O costs based on data distribution and statistics, selecting the lowest-cost plan. The finalized distributed physical plan is then dispatched to the computation layer via RPC.

Computation Layer. The computation layer executes distributed plans across a cluster of compute nodes. We materialize vector indexes as standard relational tables (see storage layer) to leverage native OLAP optimizations: (1) *Late Materialization*, which separates similarity computation from payload retrieval via a join operation; (2) *Runtime Filters*, which prune irrelevant data at the source to reduce I/O and network traffic; (3) *Columnar FastScan*, which aligns FastScan with the engine's columnar layout to maximize SIMD efficiency; (4) *Operator Fusion*, which merges several operators into a single SIMD-optimized physical operator. Building on this OLAP-native design, we further introduce DPPQ, which integrates direction-aware quantization (preserves directional fidelity to stabilize similarity ranking) with hierarchical residual refinement (progressively captures finer-grained directional components to mitigate quantization error), to improve retrieval accuracy.

Storage Layer. The storage layer adopts a three-tier architecture (i.e., remote storage, on-SSD cache, and in-memory cache) to provide scalable and low-latency access of data and vector indexes. Built upon a unified table engine, TEngineDB-V materializes both base data and vector indexes as relational tables. Instead of maintaining per-segment index files, it employs a *segment-decoupled global indexing*: a single global index is built over the entire dataset, and its structures (IVF centroids, PQ codebooks, and quantized vector codes) are stored as sharded relational tables. This table-centric design eliminates the exhaustive scatter-gather execution inherent in per-segment indexing. Moreover, we adopt a two-tier cache to improve remote storage access latency and throughput, where remote data files are logically split into fine-grained, fixed-size blocks and cached in both local SSD disk and memory.

Workflow. When a vector/hybrid SQL query arrives, the control layer parses the query and performs index-aware query rewriting and cost-based optimization to generate a distributed physical plan. The optimized plan is then dispatched to the computation layer, where compute nodes execute it using relational operators with various OLAP-native optimizations. During execution, required data and index segment pages are fetched from the storage layer through globally indexed, table-centric access. Finally, the vector search results are produced and returned through the result sink.

3.2 SQL Dialects

TEngineDB-V offers a flexible, easy-to-use SQL interface for manipulating tables/indexes and performing vector search or complex analytics. For example, the following SQL creates a table that is hash-sharded into 512 buckets by the primary key id, with each row containing an embedding vector and several structured attributes.

```
CREATE TABLE images (
    id bigint,
    embedding array<float>,
    category varchar,
    author_id bigint
)
PRIMARY KEY(id)
DISTRIBUTED BY HASH(id) BUCKETS 512;
```

The following statement builds a global IVFPQ index on `embedding` paired with primary key `id`.

```
CREATE GLOBAL INDEX vector_index
ON images(embedding,id) USING VECTOR
REFRESH IMMEDIATE ASYNC EVERY (INTERVAL 1 day)
PROPERTIES ("index_type"="ivfpq", "dim"="768",
    "nbits"="4", "nlist"="1024");
```

We adopt an asynchronous strategy for index refreshing: the system checks the base table at most once per day and triggers a full background rebuild upon detecting any changes (Section 4.2).

Finally, the following SQL demonstrates a complex query pattern frequently observed in our production workloads. It presents a hybrid query that combines structured filtering (e.g., high-resolution images with width > 1024) with vector search, and joins the retrieved visually similar images with campaign metadata to retain ad creatives from a specific industry for advertising analysis.

```
SELECT t.image_id, t.category, t.dist
FROM (
   SELECT image_id, category, campaign_id,
          approx_l2_distance(embedding,
              [0.1,0.2,...]) AS dist
   FROM images WHERE width > 1024
   ORDER BY dist LIMIT 1000000
) t
JOIN campaigns c ON t.campaign_id = c.id
WHERE c.industry = 'automotive';
```

4 UNIFIED TABLE STORAGE ENGINE

In modern analytical databases and vector management systems [4, 8, 15, 32, 47, 48, 50], data is typically horizontally split into multiple segments based on scalar attributes (e.g., primary keys or timestamps), and vector indexes are constructed independently for each segment. These indexes are commonly stored as proprietary binary files (or sidecar files) tightly bound to their corresponding data segments. We refer to this design as *segment-coupled indexing*. However, this design introduces two fundamental limitations.

(1) Segment-coupled indexing enforces a scatter-gather execution, leading to severe amplification. When the dataset is sharded into thousands of segment-local indexes, query processing must broadcast the query to all segments, compute local top-k candidates, and perform a centralized global merge. This architecture scales poorly as either k or data volume increases. As they grow, both local search costs and global merging overhead expand significantly, amplifying overall query latency.

(2) Storing the vector index as an isolated file prevents the system from leveraging native optimizations. The index is maintained in a specialized format, which is incompatible with the standard data storage engine. As a result, general-purpose optimizations such as columnar compression and fine-grained caching cannot be directly applied to index data. Instead, the compute engine accesses the index through rigid, black-box APIs, which limits fine-grained resource scheduling and operator fusion, and ultimately constrains execution efficiency.

Motivated by these, TEngineDB-V adopts a unified storage architecture centered on two core designs: *Segment-Decoupled Global Indexing*, which builds a global index to eliminate scatter–gather execution, and *Table-Centric Index Storage*, which materializes indexes as relational tables to enable native optimizations. We first introduce the overall data organization and multi-tier caching mechanism in Section 4.1, followed by detailed descriptions of these two designs in Sections 4.2 and 4.3.

4.1 Data Organization and Multi-Tier Cache

Table Data Organization. Following a similar storage design to popular OLAP systems Doris [4] and StarRocks [47], TEngineDB-V tables are horizontally partitioned (typically by time or range) to enable pruning. Each partition is then hash-bucketed into shards. One shard serves as the fundamental unit of physical data distribution and cluster scaling, mapping directly to an independent path on remote storage (COS/HDFS). Within each shard, rowsets represent immutable, atomic ingestion batches tagged with monotonic version numbers for MVCC; each rowset comprises one or more segments. A segment is a self-contained columnar file applying three optimization layers: flexible encoding (dictionary, bit-shuffle, RLE, etc.) at the column level, compression (LZ4, ZSTD, etc.) at the page level, and auxiliary indexes (ZoneMap, Bloom filters, etc.) at the file level to minimize I/O.

Multi-Tier Fine-Grained Cache. To mitigate remote storage access latency, each compute node maintains a two-tier, fine-grained caching system consisting of both in-memory and on-SSD caches. Raw data is logically split into fixed-size blocks (typically 1 MB) for fine-grained caching. A hierarchical lookup strategy is used for data access: queries first probe the memory tier, then fall back to the local SSD cache, and finally retrieve data from remote storage if necessary. When a block is evicted from memory, it is asynchronously demoted to the SSD tier; data evicted from the SSD tier is permanently discarded. Furthermore, TEngineDB-V enables cross-node cache access during cluster scaling, allowing new nodes to fetch data from existing peers before resorting to remote storage, thereby eliminating cold-start penalties during elastic expansion.

4.2 Segment-Decoupled Global Indexing

As illustrated in Figure 4, rather than constructing independent local indexes for each data segment, we build a single global vector index that spans the entire dataset. This index is physically decoupled from the underlying data segments and stored as a separate

table. This architecture is termed as *segment-decoupled indexing*, which fundamentally eliminates the exhaustive scatter–gather execution pattern inherent in segment-coupled paradigm. Rather than broadcasting queries to thousands of segment-local indexes and merging their intermediate top-k results, the query directly navigates the global index to identify relevant clusters and the search cost only scales with probed cluster count, significantly reducing read/compute amplification.

However, constructing and refreshing large-scale global index is non-trivial, as it requires aggregating and clustering massive volumes of vectors across all data partitions and refresh the globally consolidated structure, imposing immense computational, network, and disk I/O overhead. We address these challenges as follows.

Global Index Construction. We leverage Apache Spark [56] to build the global index. To construct the index, we submit Spark jobs to read our data files and perform distributed clustering to generate the IVF and PQ metadata (e.g., centroids and codebooks). The system then associates each vector with an IVF cluster ID and encodes it into PQ codes, bulk-loading the resulting records into `TEngineDB-V`'s index tables. Using Spark offers distributed scalability for billion-scale vector clustering/encoding and fault-tolerant execution that ensures robust index construction.

Global Index Refreshing. In the segment-decoupled architecture, handling updates requires periodically rebuilding the global index, which is expensive. To avoid blocking online queries, we perform index refresh asynchronously via background Spark jobs and apply the new index through an atomic swap. As a result, `TEngineDB-V` adopts an eventual consistency model: during the refresh interval, the index may lag behind the base table (e.g., deleted IDs may still appear, and newly inserted vectors are not visible until the next rebuild). This design is well-suited for analytical workloads with infrequent updates (days/weeks). A similar approach can also be found in the materialized view refresh mechanisms of open-source OLAP systems like Doris [4] and StarRocks [47], as well as in commercial products such as Snowflake [14].

Rethinking Global Index Storage. While a unified global index eliminates scatter–gather overhead, it introduces a new systems challenge: how to efficiently materialize and access this large, consolidated index structure within storage engine. If the global index is stored as a single big opaque binary file, it would not only create I/O bottlenecks but also remain outside the scope of the engine's native storage optimizations. This architectural tension directly motivates our second core design: *Table-Centric Index Storage*.

4.3 Table-Centric Index Storage

`TEngineDB-V` adopts a table-centric index storage design, in which the global index is materialized as relational tables and sharded across segments instead of segment-coupled, opaque binary file.

Index Tables. As illustrated in Figure 4, we decompose the IVFPQ index into three relational tables: (1) an *IVF centroid table*, storing cluster identifiers (CIDs) and their centroids for cluster probing, sharded by CID; (2) a *PQ codebook table*, maintaining subspace codebooks and not sharded; and (3) a *quantized vector table*, mapping CIDs to vector identifiers (PIDs) and storing PQ codes, also sharded by CID. PQ codes from different subspaces are materialized as separate columns to align with the columnar execution model

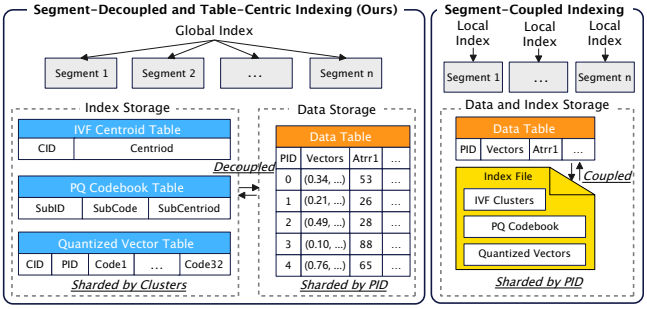


Figure 4: Storage architectural comparison.

and enable SIMD-friendly computation. Sharding by CID further ensures that only relevant shards are accessed during IVF probing.

Native Optimizations. Our table-centric design enables the global index to fully leverage native engine optimizations along the storage and execution pipeline. (1) Columnar Compression. Since index tables are stored in columnar format, they automatically benefit from built-in compression schemes (e.g., dictionary encoding and run-length encoding). This reduces storage footprint and disk I/O without introducing custom, index-specific compression logic. (2) Fine-Grained Cache. Rather than loading an entire index file into memory, `TEngineDB-V` reads and caches only the required shards and blocks during query processing. This fine-grained caching mechanism reduces unnecessary I/O and improves memory efficiency. (3) Vectorized Execution. Since index data is managed as relational tables, IVF filtering, distance computation, and top-k selection can be executed using the engine's vectorized operators, improving compute efficiency and CPU cache locality.

Rethinking Query Execution. Materializing the vector index as relational tables fundamentally reshapes the execution paradigm. Rather than invoking a specialized index interface, vector search is expressed through standard relational operators, enabling cluster routing, distance computation, and relational filtering to be jointly optimized by the SQL engine. This redesign raises three key challenges: (1) *how to faithfully express and efficiently execute IVFPQ search within relational operators*; (2) *how to mitigate the approximation errors inherent in PQ while preserving efficient relational execution*; and (3) *how to fully exploit OLAP-native optimizations to accelerate search*. We address these challenges in the next section.

5 OLAP-NATIVE COMPUTATION LAYER

To address the three challenges outlined above, this section presents the OLAP-native computation layer of `TEngineDB-V`. Section 5.1 describes how classical IVFPQ search is adapted to the relational execution paradigm through operator-level decomposition. Section 5.2 then improves retrieval accuracy within this framework by introducing `DPPQ`. Finally, Section 5.3 presents OLAP-native optimizations for relational execution.

5.1 New Paradigm of Relational Execution

As illustrated in Figure 2, `TEngineDB-V` executes IVFPQ-based vector search as a sequence of relational operators over the table-centric global index. Note (1) Graph-based methods [10, 16, 23, 31, 33, 43, 45, 51, 52, 58] are not considered as their poor performance under large-k workloads (Figure 1); (2) RaBitQ [18, 19] is another

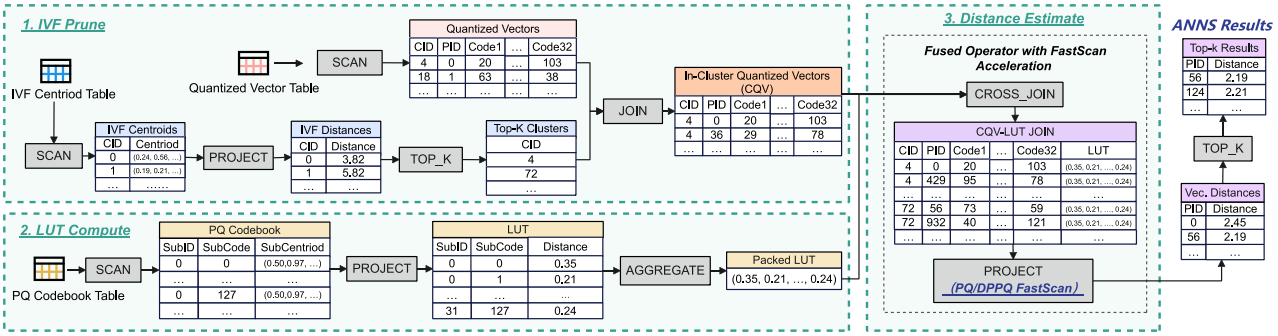


Figure 5: Relational execution in TEngineDB-V.

quantization approach with strong empirical performance, but its processing relies on complex operations, including per-cluster normalization, matrix multiplication, and bitwise operations, which are not naturally adaptable to standard relational operators and may be less efficient when adapted to relational execution.

Decomposing IVFPQ Search to Relational Operators. As illustrated in Figure 5, IVFPQ search is mapped into three composable relational stages: (1) IVF Prune. A SCAN over the IVF centroid table is first performed, followed by PROJECT to compute cluster–query distances and Top_K to identify the N_{probe} nearest clusters. A subsequent JOIN between the selected clusters and the quantized vectors is performed to retrieve the quantized vectors within these clusters. (2) LUT Compute. A SCAN over the PQ codebook table is executed, and PROJECT constructs the query-specific LUT. AGGREGATE then converts these multi-row results into a single-row array to avoid memory replication during distance evaluation. (3) Distance Estimate. A CROSS_JOIN broadcasts the LUT to candidate vectors, and PROJECT looks up and accumulates subspace distances to compute approximate distances. To further improve efficiency, we design a fused operator that consolidates these logical steps and accelerates execution using FastScan (Section 5.3). A final Top_K selects the vectors with the smallest distances.

5.2 DPPQ: Bridging Accuracy and Relational Execution

Although IVFPQ can be efficiently executed within a relational framework, its achievable recall is fundamentally limited under large-k workloads due to inherent quantization errors. A common remedy is to introduce a refinement phase that re-computes exact distances for an expanded candidate set (e.g., $2k$–$3k$ vectors with the smallest PQ distances). However, this approach requires materializing and re-evaluating high-dimensional raw vectors for a large candidate set, which quickly dominates query latency under large-k workloads and negates the efficiency benefits of relational IVFPQ execution. Therefore, instead of relying on costly post-refinement, we enhance quantization accuracy at its source by redesigning the quantization mechanism and integrating refinement directly into the fused operator, avoiding large-scale raw-vector materialization.

To address this, we propose DPPQ, a **D**irectional **P**rogressive **P**roduct **Q**uantization scheme that improves retrieval accuracy while remaining fully compatible with relational execution, eliminating the need for raw-vector materialization.

Direction-Aware Quantization. Unlike conventional PQ, which minimizes the mean squared reconstruction error $|\mathbf{x}-\hat{\mathbf{x}}|^2$ (where $\hat{\mathbf{x}}$ denotes the quantized vector), DPPQ explicitly prioritizes directional alignment. Specifically, for a given vector $\mathbf{x}$, instead of directly minimizing Euclidean reconstruction error, DPPQ minimizes the angular deviation between the normalized vector $\frac{\mathbf{x}}{|\mathbf{x}|}$ and its quantized representation. The intuition is that nearest neighbor ranking is often more sensitive to directional similarity than to absolute magnitude differences. Concretely, a high-dimensional vector $\mathbf{x} \in \mathbb{R}^D$ is partitioned into m sub-vectors, $\mathbf{x} = [\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}]$. Each sub-vector is decomposed into a scalar norm and a unit direction:

$$\mathbf{x}^{(j)} = \|\mathbf{x}^{(j)}\| \cdot \frac{\mathbf{x}^{(j)}}{\|\mathbf{x}^{(j)}\|} = \rho^{(j)} \cdot \mathbf{u}^{(j)} \tag{1}$$

where $\rho^{(j)}$ denotes the Euclidean norm and $\mathbf{u}^{(j)}$ is the unit direction. The directional components $\mathbf{u}^{(j)}$ are encoded using a PQ-style subspace clustering scheme. The corresponding magnitudes $\rho^{(j)}$ are retained to preserve scale information and enable accurate distance reconstruction. The Euclidean distance is then reconstructed via the law of cosines by combining the approximated dot product with the precomputed norms of the query and base vectors:

$$\|\mathbf{q}-\mathbf{x}\|^2 \approx \|\mathbf{q}\|^2 + \|\mathbf{x}\|^2 - 2\sum_{j=1}^{m} \rho^{(j)} \langle \mathbf{q}^{(j)}, \mathbf{u}^{(j)} \rangle. \tag{2}$$

Hierarchical Residual Refinement. To further reduce quantization error, DPPQ employs a hierarchical residual quantization scheme that progressively refines the directional component. Let $\mathbf{r}_0 = \mathbf{x}$ denote the original vector. At layer l ($1 \le l \le L$), we quantize the residual $\mathbf{r}_{l-1}$ using a directional-aware quantizer $Q_l(\cdot)$ and compute the new residual as $\mathbf{r}_l = \mathbf{r}_{l-1} - Q_l(\mathbf{r}_{l-1})$. This procedure is applied recursively across L layers. The final approximation is therefore given by $\hat{\mathbf{x}} = \sum_{l=1}^{L} Q_l(\mathbf{r}_{l-1})$. By successively encoding finer residual components, this hierarchical refinement progressively reduces quantization error.

IVFDPPQ Index Construction. Algorithm 1 describes the construction of IVFDPPQ, consisting of a coarse clustering phase followed by multi-epoch residual encoding. We first partition the base vectors using K-Means [25] to obtain coarse centroids and build the IVF structure. Then, over E epochs, residual vectors are hierarchically quantized: each residual is split into M sub-vectors, decomposed into unit directions and scalar magnitudes, and the directions are encoded using PQ codebooks trained per epoch. After each epoch,

residuals are reconstructed and updated by subtracting the decoded approximations.

Decomposing IVFDPPQ Search to Relational Operators. DPPQ is integrated with the IVF structure for query execution. As illustrated in Figure 5, IVFDPPQ follows the same relational pipeline as conventional IVFPQ. The key distinction lies in the fused operator, which incorporates additional execution logic to retrieve the associated $\rho^{(j)}$ values for each candidate vector to reconstruct the DPPQ distance. When residual refinement is enabled, the operator iteratively executes these lookup and calculation steps for *epoch* rounds to progressively refine the estimated distance.

As demonstrated in Section 7.4, under the same bit budget, DPPQ achieves a higher recall upper bound than the state-of-the-art RaBitQ, highlighting its superior quantization effectiveness.

5.3 OLAP-Native Optimizations

Building upon the table-centric storage and relational execution, we introduce additional optimizations that deeply integrate with the underlying OLAP engine to accelerate large-k vector search.

(1) Late Materialization. We decouple vector retrieval from attribute fetching via a JOIN operator that forms a two-phase execution pipeline. In the first phase, the system searches the vector index to obtain the global top-k vector IDs. In the second phase, these IDs are joined with the base table to fetch the corresponding payload columns. Since the base table is hash-bucketed by vector ID, this join is executed efficiently as a hash join without shuffling data of the base table across compute nodes. This design inherently establishes a late materialization paradigm, preventing non-vector payload columns from participating in the I/O and network transmission during the similarity search phase, which is particularly beneficial for wide OLAP tables with numerous attributes.

(2) Join Runtime Filter. Join runtime filters [7, 29] are used to further optimize the execution of hash join operators in TEngineDB-V. During execution, TEngineDB-V generates runtime filters (including SIMD-accelerated Bloom filters and Min/Max ranges) on the build side and pushes them down to probe-side operators, filtering data before I/O and network transmission. This proves particularly effective for late materialization in vector search, where selective top-k IDs aggressively prune the base table, minimizing redundant row access overhead for vector search.

(3) Top-K Runtime Filter. During distributed execution, the merge coordinator maintains a dynamic threshold, determined by the current k-th highest similarity score among received partial results. As stronger candidates arrive, the threshold is updated and pushed to worker nodes, which use it to prune non-competitive vectors during local top-k search. By filtering candidates at the source, this mechanism substantially reduces cross-node data transfer under large-k workloads.

(4) Columnar PQ FastScan. PQ distance computation is commonly accelerated by FastScan [2], which uses SIMD instructions to perform efficient distance lookups and accumulation. Conventional FastScan [2, 3] assumes a row-oriented layout where PQ codes of 32 consecutive vectors are interleaved (packed) to enable 256-bit SIMD (e.g., AVX2) operations. However, OLAP compute engines [4, 22, 47], typically adopt an Arrow [49]-like columnar storage layout, creating a fundamental mismatch with the horizontal, interleaved data organization required by traditional FastScan.

Algorithm 1: Hierarchical Quantization of IVFDPPQ

Input: Base vector set V, Cluster count $Nlist$, Epochs E, Sub-spaces M, Bits per subspace b
Output: IVFDPPQ Index Data

$K_{sub} \leftarrow 2^b$;
$d_{sub} \leftarrow D/M$;
// Step 1: Coarse KMeans Clustering
$C \leftarrow \text{KMeansTrain}(V, K_c)$
$A \leftarrow \text{KMeansAssign}(V, C)$
// Step 2: Multi-Epoch Training and Encoding
for $e \leftarrow 1$ **to** E **do**
  // Decompose sub-vectors
  $u_e, \rho_e \leftarrow \text{NormalizeSubvectors}(V, M)$;
  // Train PQ codebook for unit direction and encode
  $CB_e \leftarrow \text{PQTrain}(u_e, M, b)$;
  $P_e \leftarrow \text{PQEncode}(u_e, CB_e)$;
  // Decode, denormalize and update residuals
  $\hat{u_e} \leftarrow \text{PQDecode}(P_e, CB_e)$;
  $\hat{u_e} \leftarrow \text{DenormalizeSubvectors}(\hat{u_e}, \rho_e, M)$;
  $V \leftarrow V - \hat{R}^{(e)}$;
  $CB_{all} \leftarrow CB_{all} \cup \{CB^{(e)}\}$;
  $P_{all} \leftarrow P_{all} \cup \{P^{(e)}\}$;
  $\rho_{all} \leftarrow \rho_{all} \cup \{\rho_e\}$;
return $C, CB_{all}, (A, P_{all}, \rho_{all})$;

To bridge this gap, we develop a Columnar FastScan mechanism that adapts the FastScan algorithm to a columnar data layout without sacrificing SIMD efficiency. Recall that by our table-centric design, instead of storing PQ codes in an interleaved row-wise format, we organize the quantized codes as independent columns for different sub-quantizers (subspaces). Furthermore, the compute engine processes data in Arrow-like columnar chunks of 4,096 rows as its basic processing unit. Therefore, within each chunk, PQ codes with m sub-quantizers are organized as separate arrays (columns), each containing the quantized indices for 4,096 vectors.

Our SIMD-optimized implementation operates directly on this vertical layout: for each sub-quantizer column, the kernel loads 32 consecutive codes and performs batched lookups and accumulations using a precomputed 32-byte LUT. Empirically, this columnar design incurs only about 5% overhead versus row-oriented FastScan, while achieving a 4×–5× speedup over non-FastScan execution, balancing OLAP compatibility with computational efficiency.

(5) Operator Fusion. To reduce the overhead of decomposed relational execution, we design a fused physical operator for distance estimation. The operator combines four logical stages: (1) LUT row replication, (2) LUT-based distance projection, (3) norm-based scaling for DPPQ, and (4) final summation. It first consumes the single-row LUT and caches it in memory as a local pipeline breaker, then streams the quantized vector table directly. During streaming, it processes each input chunk in one pass, using SIMD vectorization to perform LUT lookups, norm scaling, and distance accumulation against the cached LUT state. This single-pass execution avoids materializing massive intermediate results. Together with vectorized execution [6] over columnar chunks, the fused operator improves CPU cache locality and reduces per-tuple computation overhead compared with traditional volcano-style execution [21].

6 INDEX-AWARE DISTRIBUTED OPTIMIZATION

At control layer, the query optimizer of TEngineDB-V builds upon the Cascades framework [20, 21, 42] and extends it to jointly reason about vector-index execution semantics and distributed plan generation. This enables execution plans that are both *index-aware* and *communication-efficient*, essential for large-k vector search.

When a SQL query containing a vector predicate is submitted, it is first transformed into a logical operator tree and processed by the rule-based optimizer (RBO). The RBO performs deterministic, *index-aware query rewriting* on the initial logical operator tree, injecting specialized vector-index execution logic to establish a baseline execution strategy. The rewritten plan is then inserted into the Memo of the cost-based optimizer (CBO) under the Cascades framework. During optimization, the CBO expands the Memo to explore equivalent logical formulations and candidate physical implementations. To evaluate these alternatives, we integrate a *distributed-aware cost model* that jointly estimates CPU cost, memory pressure, network overhead, and recall impact. This enables the optimizer to select the globally optimal physical plan from the expanded search space.

6.1 Index-Aware Query Rewriting

Upon receiving a high-level vector search request, TEngineDB-V rewrites it into an index-aware logical plan and enumerates equivalent alternatives within the Cascades framework.

Logical Plans for ANNS. For pure ANNS, rewriting is triggered when the RBO detects a canonical pattern in the logical plan, i.e., a vector distance computation followed by a Top_K operator. Upon matching this pattern, the RBO applies certain rules to rewrite the generic operators with specialized vector search operators that embed index semantics (Figure 5). The rewritten plan is inserted into the corresponding equivalence classes in the Memo, allowing the CBO to evaluate it alongside standard relational alternatives.

Logical Plans for FANNS. For FANNS queries that combine vector search with scalar predicates, a single static rewrite is insufficient, as the optimal strategy depends on both filter selectivity and data distribution. Our RBO first rewrites the query into a canonical *pre-filter* plan by pushing scalar predicates below distance estimation. Once inserted into the Cascades Memo, transformation rules are triggered to explore the *post-filter* alternative and other equivalent forms. As shown in Figure 6, we expand the search space along three orthogonal dimensions: (1) Logical order. *Pre-filter* (scalar before distance) vs. *post-filter* (scalar after distance). (2) Join order. Two hash join configurations by swapping the build/probe side between vector and scalar intermediates. (3) Data exchange. Two distributed strategies: *broadcast* and *shuffle* joins.

These three binary dimensions yield eight candidate physical plans ($2 \times 2 \times 2$), all explicitly maintained in the Memo. This enumeration defines a complete execution search space, over which a *distributed-aware cost model* is used to select the optimal plan.

6.2 Distributed-Aware Cost Model

Unlike existing systems that primarily model CPU cost [32, 34, 48, 50, 54], we adopt a weighted cost model that jointly evaluates CPU, memory, network cost when evaluating alternative logical plans:

$$Cost = w_{\text{cpu}} \cdot Cost_{\text{cpu}} + w_{\text{mem}} \cdot Cost_{\text{mem}} + w_{\text{net}} \cdot Cost_{\text{net}}. \quad (3)$$

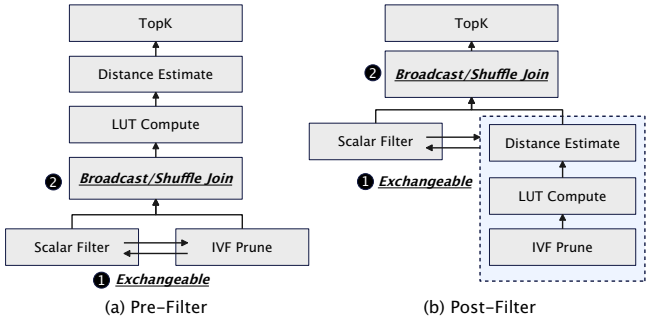


Figure 6: Execution Plans for FANNS.

The weights are calibrated to reflect the architectural characteristics of TEngineDB-V. By default, we set $w_{\text{cpu}} = 0.5$, $w_{\text{mem}} = 2.0$, and $w_{\text{net}} = 1.5$, assigning higher penalties to memory pressure and network communication in distributed environments.

Recall that we classify the eight candidate FANNS plans along three orthogonal dimensions. For each dimension, we then analyze the corresponding CPU, memory, and network bottlenecks, with the overall cost breakdown summarized in Table 3.

Pre-Filter and Post-Filter. Pre-filtering reduces the vector search space to N_F rows before distance evaluation, thus bounding the number of in-cluster candidates M_p and ensuring recall correctness. However, when the scalar filter is unselective (i.e., $N_F \approx N$), both the join cost and memory usage become dominant, leading to CPU-bound execution. In contrast, post-filtering performs vector pruning first and joins only the top-K_{ann} candidates, with CPU cost dominated by $M_p \cdot C_{dist}$ (i.e., distance computations on M_p rows). Under highly selective scalar predicates, K_{ann} must be set to a larger value to preserve recall, which increases join and memory costs and becomes inefficiency. Otherwise, insufficient setting of K_{ann} may lead to recall degradation.

TEngineDB-V adaptively sets K_{ann} based on the predicate selectivity estimated by the optimizer (e.g., retrieving $2k$ candidates for an estimated 50% selectivity), while also allowing manual overrides. Under the post-filtering strategy, instead of expanding the candidate pool using iterative search, the system does not strictly enforce returning exactly k results. If the final filtered yield falls below k but remains above a configurable threshold (defaulting to 20% of k), it simply returns the available subset, as minor result reductions rarely impact the effectiveness of large-k query workloads. However, if the yield drops below this threshold, the execution dynamically falls back to a pre-filtering plan.

Build-Side Selection. In hash join execution, the build side can be either the vector or scalar intermediate results. As shown in Table 3, memory usage scales with the build-side cardinality and row width (e.g., $M_p \cdot R_V$, $K_{ann} \cdot R_V$, or $N_F \cdot R_F$), while CPU cost scales with the build size for hash construction and with the probe size for probing. Hence, assigning the smaller intermediate result to the build side is crucial for minimizing memory and construction overhead. When the vector candidate count is much smaller ($M_p \ll N_F$), building the hash table on the vector side is optimal. Conversely, when a selective scalar predicate results in a small filtered set ($N_F \ll M_p$), building on the scalar side is preferable.

Broadcast Join and Shuffle Join. The network cost differentiates broadcast and shuffle joins. For broadcast joins (Plans 1, 3, 5, 7), the

Table 3: Notation used in the cost model (left) and analytical cost breakdown of the eight FANNS execution plans (right).

Symbol	Meaning
N	Total rows
N_f	Rows after filter
M_p	Rows after IVF Pruning
K_{ann}	Post-filter candidates
R_F	Scalar row width
R_V	Vector row width
C_{build}	Build cost
C_{probe}	Probe cost
C_{hash}	Hash cost
C_{dist}	Distance cost
P	# Nodes

Plan	CPU Cost		Memory	Network
	ANNS	Join		
Plan 1 (Pre-Filter-Broadcast-Probe)	M_pC_{dist}	$M_pC_{build} + N_fC_{probe}$	M_pR_V	PM_pR_V
Plan 2 (Pre-Filter-Shuffle-Probe)	M_pC_{dist}	$(N_f + M_p)C_{hash} + M_pC_{build} + N_fC_{probe}$	M_pR_V	$N_fR_F + M_pR_V$
Plan 3 (Pre-Filter-Broadcast-Build)	N_fC_{dist}	$N_fC_{build} + M_pC_{probe}$	N_fR_F	PN_fR_F
Plan 4 (Pre-Filter-Shuffle-Build)	N_fC_{dist}	$(N_f + M_p)C_{hash} + N_fC_{build} + M_pC_{probe}$	N_fR_F	$M_pR_V + N_fR_F$
Plan 5 (Post-Filter-Broadcast-Probe)	M_pC_{dist}	$K_{ann}C_{build} + N_fC_{probe}$	$K_{ann}R_V$	$PK_{ann}R_V$
Plan 6 (Post-Filter-Shuffle-Probe)	M_pC_{dist}	$(N_f + K_{ann})C_{hash} + K_{ann}C_{build} + N_fC_{probe}$	$K_{ann}R_V$	$N_fR_F + K_{ann}R_V$
Plan 7 (Post-Filter-Broadcast-Build)	M_pC_{dist}	$N_fC_{build} + K_{ann}C_{probe}$	N_fR_F	PN_fR_F
Plan 8 (Post-Filter-Shuffle-Build)	M_pC_{dist}	$(K_{ann} + N_f)C_{hash} + N_fC_{build} + K_{ann}C_{probe}$	N_fR_F	$K_{ann}R_V + N_fR_F$

Table 4: Statistics of dataset.

Dataset	Dimension	#Vectors	#Queries	Source
GloVe	100	1,183,514	10,000	Texts
SIFT1M	128	1,000,000	10,000	Image
SIFT1B	128	1,000,000,000	10,000	Image
Wikipedia	768	35,312,651	10,000	Texts
Tencent-Image	768	10,000,000,000	1,000	Images
GIST	960	1,000,000	1,000	Images

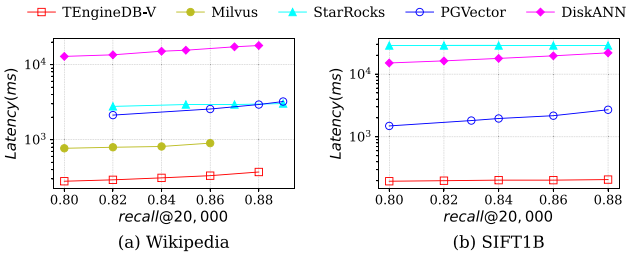
network cost scales linearly with the node size times the build-side size (e.g., $P \cdot M_p \cdot R_V$ or $P \cdot N_F \cdot R_F$). Therefore, broadcast is efficient only when the build side is sufficiently small. Otherwise, its cost grows proportionally with node scale and quickly dominates total latency. In contrast, shuffle joins (Plans 2, 4, 6, 8) repartition both inputs across nodes, incurring a network cost proportional to the combined size of the intermediate results (e.g., $N_F \cdot R_F + M_p \cdot R_V$). Although shuffle avoids data replication and distributes memory pressure more evenly, it introduces additional hash partitioning overhead, such as $(N_F + M_p) \cdot C_{hash}$, cost to calculate hash codes for both vector search and scalar intermediate results. Consequently, shuffle joins become necessary when intermediate results are large or node scale amplifies broadcast cost.

7 EXPERIMENTS

7.1 Experiments Setting

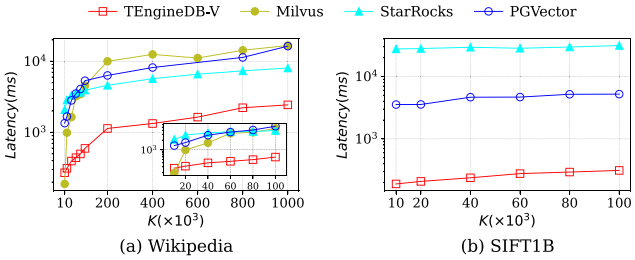
Datasets. We include low- to medium-dimensional image and text embeddings such as GloVe (100D), SIFT1M/SIFT1B (128D), and GIST (960D), as well as higher-dimensional textual embeddings from Wikipedia (768D). To assess scalability at extreme scale, we further incorporate Tencent-Image (768D), a real-world production dataset containing 10 billion vectors.

Compared Systems. Both specialized and database-integrated vector search systems are compared. For specialized vector search systems, we use Milvus [50], a widely adopted and representative vector database. For database-integrated vector search systems, we evaluate StarRocks [47] and PGVector [34]. StarRocks is a distributed analytical database for real-time OLAP workloads. PGVector is a widely adopted PostgreSQL extension [57] that enables vector similarity search within relational databases. We incorporate the DiskANN library [23] to assess performance within the broader context of disk-resident indexing techniques [11, 46]. All evaluated systems are open-source and compared at medium recall levels (0.8–0.9), as in our target workloads vector search serves only as a coarse candidate generation stage rather than the final accuracy-critical results.



(a) Wikipedia (b) SIFT1B

Figure 7: Latency–Recall Trade-off Across Systems.



(a) Wikipedia (b) SIFT1B

Figure 8: Latency Scalability with Increasing K.

Parameter Settings. All systems are deployed using the same 56-shard configuration to ensure fair comparisons. All systems except DiskANN utilize the IVFPQ index. For the SIFT1B dataset, the data space is partitioned into 3000 clusters, and vectors are compressed using 32 subspaces with 8-bit quantization per subspace. For the Wikipedia dataset, the data is divided into 6000 clusters, utilizing 192 subspaces with 8-bit quantization. For DiskANN, the graph degree is set to 48, the build list size is 128, and the beam list parameter is 4 for both datasets. The search list size is dynamically adjusted, ranging from 36,000 to 58,500 for the SIFT1B dataset, and from 30,000 to 45,000 for the Wikipedia dataset.

Implementation. Experiments are conducted on the machine equipped with an Intel Xeon Platinum 8576C processor (56 cores, 2.0 GHz), 487 GB of memory, and a 7 TB NVMe SSD. The case study is performed on a distributed cluster consisting of 10 machines, each with the exact same hardware configuration.

7.2 ANNS Performance

We compare the ANNS performance with k = 20,000. The results in Figure 7 (Milvus does not report results on SIFT1B due to out-of-memory errors) show `TEngineDB-V` consistently achieves superior latency–recall trade-off compared to all baselines. As shown in Figure 7(a), at a recall of 0.86, `TEngineDB-V` achieves a latency of 332ms, representing speedups of 2.7×, 8.9×, 7.7×, and 50.0× over Milvus, StarRocks, PGVector, and DiskANN library on the Wikipedia

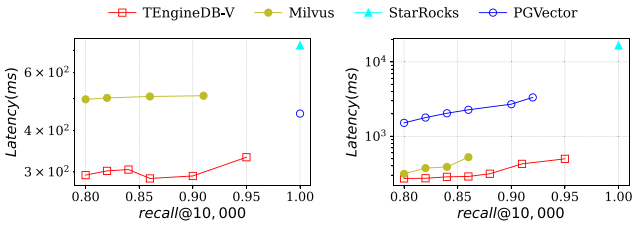


Figure 9: FANNS Latency–Recall Trade-off on Wikipedia.

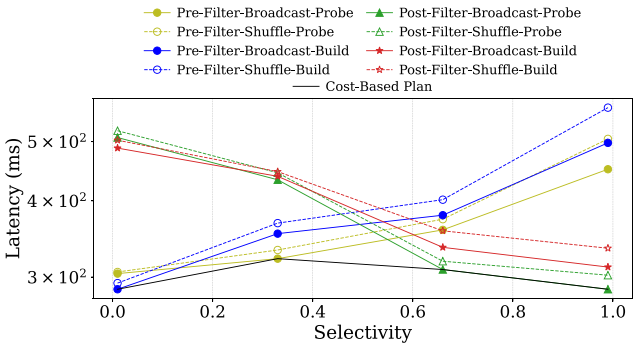


Figure 10: Performance of the eight FANNS execution plans and the cost-based plan under varying selectivity.

dataset. Similarly, in Figure 7(b), at a recall of 0.88, TEngineDB-V achieves a latency of 208ms, representing speedups of 137.1×, 13×, and 104.7× over StarRocks, PGVector, and DiskANN on SIFT1B. The performance gains mainly attribute to TEngineDB-V 's segment-decoupled global indexing and relational execution framework: (1) the relational execution framework improves per-segment execution efficiency through multi-threaded parallel processing; (2) the segment-decoupled global index selectively probes only relevant segments, significantly reducing cross-segment intermediate-result merging and global top-k sorting overhead.

We also evaluate the performance under increasing k values. On Wikipedia, k ranges from 10^4 to 10^6, while on SIFT1B, k ranges from 10^4 to 10^5, (Figure 8). Milvus does not report results on SIFT1B due to out-of-memory errors, as discussed earlier. Across both datasets, TEngineDB-V consistently achieves lower latency than all baselines as k increases. For example, when $k = 8 \times 10^5$, TEngineDB-V achieves a latency of 2222ms, representing speedups of 6.4×, 3.3×, and 5.1× over Milvus, StarRocks, and PGVector on Wikipedia, respectively. Similarly, when $k = 10^4$, it achieves a latency of 189ms, representing speedups of 145.5× and 18.7× over StarRocks and PGVector on SIFT1B. These results demonstrate the strong scalability of TEngineDB-V with respect to increasing k, validating the effectiveness of OLAP-native ANNS in large-k workloads.

7.3 FANNS Performance

We evaluate FANNS performance on the Wikipedia dataset under predicate selectivities of 1% and 99%. As shown in Figure 9, TEngineDB-V consistently delivers the best latency-recall trade-off across selectivity settings. Under 1% selectivity, TEngineDB-V outperforms Milvus by 1.7× at about 0.9 recall; under 99% selectivity, it outperforms Milvus and PGVector by 1.6× and 7.5×, respectively,

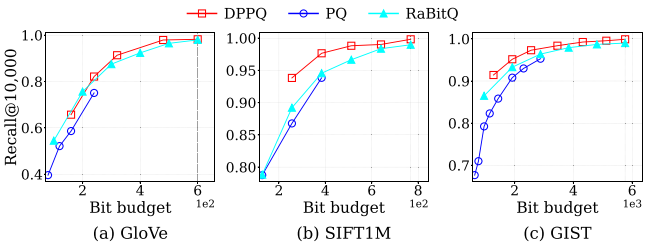


Figure 11: Comparison of different quantization algorithms.

at about 0.85 recall. These gains come from TEngineDB-V's cost-based, distributed-aware FANNS optimizer, which selects efficient execution plans under different filtering conditions. We further validate effectiveness of the cost-based strategy with eight alternative plans across selectivities from 1% to 99% (Figure 10). Since no single plan dominates across all selectivities, the cost model achieves consistently strong performance by dynamically selecting the most suitable plan for each workload.

7.4 In-Depth Analysis of DPPQ

7.4.1 Quantization performance. We compare the quantization performance of DPPQ against PQ and RaBitQ under equal per-vector bit budgets. To ensure fairness, we define the bit budget used to represent each vector, which determines its theoretical compression capacity. For PQ, the bit budget equals the number of sub-vectors multiplied by the bits allocated per sub-vector. For RaBitQ, it equals the vector dimensionality multiplied by the bits assigned per dimension (typically one bit for binary quantization). For DPPQ, the bit budget extends that of PQ by introducing an expansion factor corresponding to the number of refinement epochs, reflecting its progressive quantization mechanism.

Figure 11 illustrates the trade-off between Recall@10,000 and bit budget across three datasets. Under the same bit budget, DPPQ consistently achieves higher recall, while requiring one additional FP32 scaling factor per subspace at each refinement epoch. For example, on SIFT1M, at a bit budget of 400 bits per vector, DPPQ improves recall by 4.1% and 3.2% over PQ and RaBitQ, respectively. Similar improvements are observed at lower bit budgets. Even on lower-dimensional datasets such as GloVe, DPPQ achieves comparable performance to RaBitQ and consistently outperforms PQ. These results demonstrate DPPQ's effectiveness across datasets with diverse dimensionalities and data distributions, highlighting the superior quantization performance of DPPQ due to its direction-aware quantization and progressive refinement.

7.4.2 Convergence Analysis. We further analyze the effect of hierarchical residual refinement in DPPQ. Figure 12 reports quantization error (MSE) over refinement epochs under different bit budgets, where m is the number of sub-vectors and b is the number of bits per sub-vector. Across configurations and datasets, the error drops sharply within the first three epochs and stabilizes after only a few iterations, showing that progressive refinement rapidly captures finer-grained directional structure without extensive training. Higher bit budgets further accelerate convergence, and the consistent trends across datasets demonstrate the robustness of the direction-aware progressive optimization strategy.

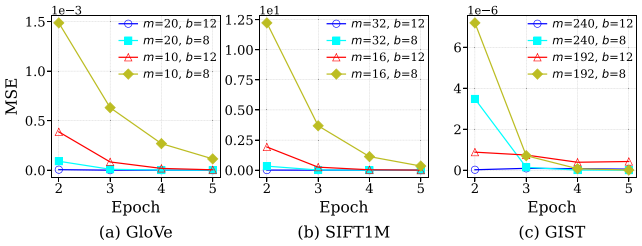


Figure 12: Convergence analysis of DPPQ.

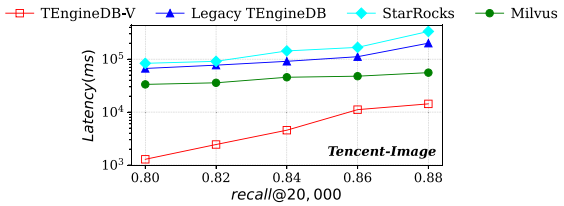


Figure 13: Production performance at 10-billion scale.

7.5 Case Study

To evaluate the real-world impact of TEngineDB-V, we compare with Legacy TEngineDB, StarRocks, and Milvus on *Tencent-Image*, a production-scale dataset deployed in Tencent's internal image retrieval system. The legacy TEngineDB adopts a traditional segment-coupled indexing framework, where each segment maintains an independent local vector index. As shown in Figure 13, TEngineDB-V consistently outperforms Legacy TEngineDB, StarRocks, and Milvus for $k = 20{,}000$, a typical setting for large-scale exploratory retrieval. At a recall of 0.88, TEngineDB-V achieves a 14×, 23×, and 4× latency reduction compared to the legacy system, StarRocks, and Milvus, with absolute latencies of 14.2s, 200s, 333s, and 56s, respectively. This performance gap further widens at a 0.8 recall, where TEngineDB-V outperforms the legacy system by 52×, StarRocks by 65×, and Milvus by 26×. These results validate the gains in production efficiency of TEngineDB-V.

8 LESSONS LEARNED

Limitations of Segment-Coupled Indexing. Our experience indicates that segment-coupled indexing is unsuitable for large-k workloads. Legacy TEngineDB, built upon this architecture, performed well for millions of vectors with hundreds of segments, yet encountered severe scalability bottlenecks beyond 1B rows and 2,000 segments. Two fundamental issues drove this limitation: *(1) k-amplification overhead* including excessive disk I/O, network traffic, and global merge costs. *(2) Index cache thrashing*. File-granularity index caching was used to adapt our three-tier storage hierarchy (Remote → Local SSD → Memory). However, this design becomes a liability when SSD and memory capacity are constrained: a single query must access all index files for local searching, triggering frequent index file evictions and reloads that degrade query latency.
Incremental Optimizations Are Not Enough. Three incremental optimizations were investigated to address these issues. First, we increased segment capacity from 2GB to 20GB to reduce N and lower the $N \times k$ amplification. However, this increased write amplification, compaction costs, and data re-distribution overhead during node failures or cluster scaling, rendering it impractical for production. Second, we implemented two-round search: first searching only vector IDs to find the global top-k, then fetching payloads. Yet this still requires reading the full vector index from every segment and performing heavy global merges. Third, we tested IVF-cluster-level caching instead of file-level caching, but this only improved intra-segment cache granularity without alleviating thrashing across 2,000+ independent segments. These practical constraints forced us to abandon the segment-coupled paradigm entirely, leading to our segment-decoupled design with table-centric index storage. By maintaining a global index and leveraging the native multi-tier fine-grained cache, we eliminate the $N \times k$ I/O explosion and critical cache thrashing inherent in the coupled design, achieving stable performance regardless of k magnitude.
Complex Similarity Search and Hybrid Workloads. Our experience deploying TEngineDB-V reveals two trends. First, similarity queries have grown increasingly complex, moving beyond simple top-k retrieval toward complex patterns including similarity-bounded range aggregation [26, 28] and similarity joins [12, 30, 55]. Second, it is increasingly common to run similarity search and structured analytics simultaneously on the same multi-modal dataset. For instance, our users are performing large-k image retrieval while concurrently generating BI reports on structured image metadata (file size, source, creation time, etc.). By decomposing vector search into relational operators, TEngineDB-V enables unified execution plans in which vector search is jointly optimized with other relational operators such as joins and filters. The optimizer can dynamically reorder, interleave, or push these operators into vector search as runtime filters based on predicate selectivity and execution cost, enabling early pruning and reducing unnecessary candidate materialization. These optimizations extend beyond the FANNS cases in Table 3 and generalize to more complex production workloads.

9 CONCLUSION

This paper presents TEngineDB-V, an OLAP-native vector search system tailored for large-k workloads. TEngineDB-V establishes a new execution paradigm that materializes segment-decoupled global indexes as relational tables and executes vector search via relational operators. Building upon this, we develop DPPQ to improve quantization accuracy while preserving relational efficiency, design an index-aware distributed optimization framework to ensure efficient execution across diverse workloads, and introduce several OLAP-native optimizations to further accelerate query processing. Extensive experiments and real-world production deployments validate the substantial performance gains of TEngineDB-V.

ACKNOWLEDGMENTS

Xuanhe Zhou is the corresponding author. Xufei Wu conducted this work during his Tencent internship, where Pengcheng Zhang was his mentor. This work was supported in part by National Key R&D Program of China (No. 2023YFB4502400), China NSF grant (No. 62441236, 62372296, 62432007, U25A6024, U25A20437, 62525202, 62232009, 62502304), Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM103), Tencent Rhino Bird Key Research Project, Shenzhen Project (CJGJZD20230724093403007), Zhongguancun Lab, and Beijing National Research Center for Information Science and Technology (BNRist), Tencent, Shanghai Jiao Tong University AI for Engineering Initiative.

REFERENCES

[1] Meta AI. 2017. FAISS. https://ai.facebook.com/tools/faiss.

[2] 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 *PVLDB*, Vol. 9. 12.

[3] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2019. Quicker adc: Unlocking the hidden potential of product quantization with simd. *IEEE TPAMI* 43, 5 (2019), 1666–1677.

[4] Apache Doris Community. 2025. Vector Search in Apache Doris. https://doris.apache.org/zh-CN/docs/4.x/ai/vector-search/overview/. Accessed: 2025-12.

[5] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.

[6] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[7] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.

[8] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. Singlestore-v: An integrated vector database system in singlestore. *PVLDB* 17, 12 (2024).

[9] 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.* Microsoft Research, GitHub repository: https://github.com/Microsoft/SPTAG.

[10] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2111.08566* (2021).

[11] 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. *arXiv preprint arXiv:2603.01779* (2026).

[12] Yanqi Chen, Xiao Yan, Alexandra Meliou, and Eric Lo. 2025. DiskJoin: Large-scale Vector Similarity Join with SSD. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–27.

[13] ClickHouse, Inc. 2024. Approximate Nearest Neighbor Search with Vector Similarity Indexes. https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/annindexes. Accessed: 2025-12.

[14] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[15] Elastic. 2023. kNN Search and Vector Fields in Elasticsearch. https://www.elastic.co/guide/en/elasticsearch/reference/current/knn-search.html. Accessed: 2025-12.

[16] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. In *PVLDB*, Vol. 12. VLDB Endowment, 416–474.

[17] Samir Yitzhak Gadre, Gabriel Ilharco, Alex Fang, Jonathan Hayase, Georgios Smyrnis, Thao Nguyen, Ryan Marten, Mitchell Wortsman, Dhruba Ghosh, Jieyu Zhang, et al. 2023. Datacomp: In search of the next generation of multimodal datasets. *Advances in Neural Information Processing Systems* 36 (2023), 27092–27112.

[18] 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. *ACM SIGMOD* 3, 3 (2025), 1–26.

[19] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *ACM SIGMOD* 2, 3 (2024), 1–27.

[20] Goetz Graefe et al. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[21] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*. IEEE, 209–218.

[22] Yuxing Han, Yu Lin, Yifeng Dong, Xuanhe Zhou, Xindong Peng, Xinhui Tian, Zhiyuan You, Yingzhong Guo, Xi Chen, Weiping Qu, et al. 2026. ByteHouse: A Cloud-Native OLAP Engine with Incremental Computation and Multi-Modal Retrieval. *ACM SIGMOD* (2026).

[23] 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. *Advances in Neural Information Processing Systems* 32 (2019).

[24] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE TPAMI* 33, 1 (2010), 117–128.

[25] K Krishna and M Narasimha Murty. 1999. Genetic K-means algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 29, 3 (1999), 433–439.

[26] Hai Lan, Shixun Huang, Zhifeng Bao, and Renata Borovica-Gajic. 2024. Cardinality estimation for similarity search on high-dimensional data objects: The impact of reference objects. *Proceedings of the VLDB Endowment* 18, 3 (2024), 544–556.

[27] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *TKDE* 32, 8 (2019), 1475–1488.

[28] Anqi Liang, Pengcheng Zhang, Bin Yao, Zhongpu Chen, Yitong Song, and Guangxu Cheng. 2024. UNIFY: Unified Index for Range Filtered Approximate Nearest Neighbors Search. *Proceedings of the VLDB Endowment* 18, 4 (2024), 1118–1130.

[29] Guy M Lohman. 1988. R* optimizer validation and performance evaluation for distributed queries. *Readings in Database Systems* (1988), 219.

[30] Youzhong Ma, Shijie Jia, and Yongxin Zhang. 2017. A novel approach for high-dimensional vector similarity join query. *Concurrency and Computation: Practice and Experience* 29, 5 (2017), e3952.

[31] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE TPAMI* 42, 4 (2018), 824–836.

[32] Zhaojie Niu, Xinhui Tian, Xindong Peng, and Xing Chen. 2025. BlendHouse: A cloud-native vector database system in ByteHouse. In *IEEE ICDE*. IEEE, 4332–4345.

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

[34] pgvector authors. 2023. pgvector: Open-source vector similarity search for Postgres. https://github.com/pgvector/pgvector. Accessed: 2025-12.

[35] Pinecone. 2021. *Pinecone.io.* https://www.pinecone.io/.

[36] Qdrant. 2021. *qdrant.tech.* https://qdrant.tech/.

[37] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PmLR, 8748–8763.

[38] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125* 1, 2 (2022), 3.

[39] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 10684–10695.

[40] Nataniel Ruiz, Yuanzhen Li, Varun Jampani, Yael Pritch, Michael Rubinstein, and Kfir Aberman. 2023. Dreambooth: Fine tuning text-to-image diffusion models for subject-driven generation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 22500–22510.

[41] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily L Denton, Kamyar Ghasemipour, Raphael Gontijo Lopes, Burcu Karagol Ayan, Tim Salimans, et al. 2022. Photorealistic text-to-image diffusion models with deep language understanding. *Advances in neural information processing systems* 35 (2022), 36479–36494.

[42] Mohamed A Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, et al. 2014. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 337–348.

[43] Yitong Song, Kai Wang, Bin Yao, Zhida Chen, Jiong Xie, and Feifei Li. 2024. Efficient Reverse k Approximate Nearest Neighbor Search Over High-Dimensional Vectors. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4262–4274.

[44] Yitong Song, Bin Yao, Zhida Chen, Xin Yang, Jiong Xie, Feifei Li, and Mengshi Chen. 2025. Efficient top-k spatial-range-constrained approximate nearest neighbor search on geo-tagged high-dimensional vectors. *The VLDB Journal* 34, 1 (2025), 14.

[45] 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. *ACM SIGMOD* 3, 6 (2025), 1–26.

[46] 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. In *Companion of the International Conference on Management of Data*. 580–587.

[47] StarRocks Community. 2024. Vector Index in StarRocks. https://docs.starrocks.io/docs/using_starrocks/vector-index/. Accessed: 2025-12.

[48] Ji Sun, Guoliang Li, James Pan, Jiang Wang, Yongqing Xie, Ruicheng Liu, and Wen Nie. 2025. GaussDB-Vector: A Large-Scale Persistent Real-Time Vector Database for LLM Applications. *PVLDB* 18, 12 (2025), 4951–4963.

[49] The Apache Software Foundation. 2025. Apache Arrow: A cross-language development platform for in-memory data. https://arrow.apache.org/. Accessed: 2025-10-24.

[50] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *ACM SIGMOD*. 2614–2627.

[51] 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. *arXiv preprint arXiv:2203.13601* (2022).

[52] 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. *ACM SIGMOD* 2, 1 (2024), 1–27.

[53] Weaviate. 2019. *Weaviate.io*. https://weaviate.io/developers/weaviate/concepts/vector-index.

[54] 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. *PVLDB* 13, 12 (2020), 3152–3165.

[55] Jiadong Xie, Jeffrey Xu Yu, and Yingfan Liu. 2025. Fast approximate similarity join in vector databases. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[56] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*.

[57] Yunan Zhang, Shige Liu, and Jianguo Wang. 2024. Are there fundamental limitations in supporting vector data management in relational databases? A case study of PostgreSQL. In *IEEE ICDE*. IEEE.

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