SQL-Native Vector Search at Billion Scale in Presto

Zhichen Xu
zhichenxu@meta.com
Meta Platforms, Inc., USA

Ge Gao
gaoge@meta.com
Meta Platforms, Inc., USA

Ke Wang
wangke@meta.com
Meta Platforms, Inc., USA

Aakash Deep
aakashda@meta.com
Meta Platforms, Inc., USA

Junjie Qi
jqi@meta.com
Meta Platforms, Inc., USA

Jort Gemmeke
jgemmeke@meta.com
Meta Platforms, Inc., USA

Amit Dutta
adutta@meta.com
Meta Platforms, Inc., USA

Xiaoxuan Meng
xiaoxmeng@meta.com
Meta Platforms, Inc., USA

Trang Nguyen
trangnmnguyen@meta.com
Meta Platforms, Inc., USA

Jiayin Yan
yjyolande@meta.com
Meta Platforms, Inc., USA

Xiao Du
duxiao@meta.com
Meta Platforms, Inc., USA

Vivek Gaur
vsgaur@meta.com
Meta Platforms, Inc., USA

Kaushik Ravichandran
kaushikr@meta.com
Meta Platforms, Inc., USA

Vishal Gandhi
vishalg@meta.com
Meta Platforms, Inc., USA

ABSTRACT

We present Presto Vector Search, a SQL-native distributed vector search system built on the observation that partition-based vector search decomposes naturally into relational algebra—partitioning as scalar functions, index construction as `GROUP BY` aggregation, search as equi-joins—with the local index type (FLAT, IVF, RaBitQ, HNSW) as a pluggable parameter within the aggregate. This decomposition enables pre-filter indexing, arbitrary joins and aggregations on results, and optimizer-driven zero-shuffle distributed execution as direct consequences of the relational mapping. A two-level API—a declarative table function rewritten by the optimizer into distributed SQL primitives—ensures both novice and expert users share the same optimized execution path. We validate across four workloads spanning three modalities (512- to 5,120-dim) and up to 2B+ vectors, achieving 95–99% recall on workloads with ground-truth labels, with 95–99% CPU reduction over brute-force baselines and ∼7,000× network reduction via codec-driven co-located execution. On the public DEEP1B benchmark (1B vectors), end-to-end BUILD+SEARCH completes in under 2.5 minutes on 200 workers with 96.6% Recall@1 against official ground truth.

PVLDB Reference Format:
Zhichen Xu, Ge Gao, Ke Wang, Aakash Deep, Junjie Qi, Jort Gemmeke, Amit Dutta, Xiaoxuan Meng, Trang Nguyen, Jiayin Yan, Xiao Du, Vivek Gaur, Kaushik Ravichandran, and Vishal Gandhi. SQL-Native Vector Search at Billion Scale in Presto. PVLDB, 19(12): 4182 - 4194, 2026.
doi:10.14778/3827998.3828025

1 INTRODUCTION

Many AI teams run into the same problem: you build a great embedding model, you get fast nearest-neighbor search working, and then someone asks for access-control filters, metadata joins, and aggregated results. Suddenly you are gluing together multiple systems. This pattern recurs across RAG pipelines, entity matching, near-duplicate detection, and video retrieval. Fast vector search exists; *composable* vector search at warehouse scale does not.

Current solutions force difficult trade-offs. Exporting embeddings to specialized vector databases (Pinecone [3], Milvus [30]) requires sync pipelines, risks inconsistency on source updates, and precludes pre-filter indexing. Cloud SQL engines (BigQuery [6], SingleStore-V [13]) offer distributed vector search but rely on pre-built indices with post-filter semantics that, at low selectivity, may return far fewer than k results, and each new filter needs a separate pre-built index. Single-node extensions (pgvector [2]) preserve SQL composability but do not scale beyond one machine. Standalone ANN libraries (FAISS [21], ScaNN [19]) offer neither SQL integration nor distributed execution, forcing client-side orchestration.

Presto Vector Search addresses these challenges by natively integrating ANN search into Presto [29], a widely deployed open-source distributed SQL engine. Our insight is that partition-based vector search is a relational workload hiding in plain sight. The system targets batch analytical workloads—large-scale offline search, hybrid analytics, and ad-hoc exploration—rather than low-latency real-time serving.

This paper makes the following contributions.

(1) Vector Search as Relational Algebra. We show that partition-based vector search decomposes into relational operators: partitioning maps to scalar functions and hash distribution, index construction maps to GROUP BY aggregation, and search maps to equi-joins. The local index type is an opaque parameter within the aggregate—FLAT, IVF_FLAT, RaBitQ [17], and HNSW [25] within the same framework. This decomposition enables pre-filter indexing, full SQL composability, and optimizer-driven distributed execution as consequences of the relational mapping, while inheriting the engine's operational infrastructure [28] (retry, spill-to-disk, memory arbitration) without bespoke fault-tolerance logic. To our knowledge, no prior system decomposes vector search into standard relational operators in this way.

(2) Zero-Shuffle Distributed Execution via Decoupled Index Architecture. We realize this distributed execution through a

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828025

decoupled index architecture that separates the global codec (cluster centroids) from local per-cluster indices—unlike monolithic ANN libraries where these are tightly coupled. The codec is a compact, broadcastable artifact (~200 MB at K=100,000); local indices remain co-located with their data. Cluster assignment serves as a dual-purpose key that simultaneously captures similarity structure and drives hash-based data distribution, eliminating index data movement entirely (~7,000× network reduction vs. full-corpus shuffle on a 700M-vector corpus).

(3) Memory-Bounded Billion-Scale Indexing. We adapt RaBitQ compression to SQL's streaming execution model—a novel challenge since standard quantization requires two passes incompatible with `GROUP BY` aggregation—and introduce sub-clustering to bound peak memory independently of cluster skew, enabling billion-scale index construction on commodity workers with 64 GB RAM (25× peak memory reduction on 700M vectors, from 1.37 TB to 54 GB).

(4) Two-Level API with Optimizer Rewriting. We design a hierarchical API in which a declarative table function is rewritten by the query optimizer into distributed SQL primitives—reducing a 68-line, 5-subquery expert pipeline to a 12-line single-subquery statement while sharing the same optimized execution path.

(5) Experimental Validation at Billion Scale. We validate the architecture through evaluations spanning four workloads across three modalities (image, text, video), scaling to queries against 2B candidates, with 95–99% recall on workloads with ground-truth labels and up to 10× speedup, demonstrating generalization across embedding types and dimensionalities (512 to 5,120). On the public DEEP1B benchmark (1B vectors), end-to-end BUILD+SEARCH completes in under 2.5 minutes on 200 workers with 96.6% Recall@1 against official ground truth.

Presto Vector Search demonstrates that the distributed SQL engine can serve as the natural platform for large-scale vector analytics, eliminating external vector silos while preserving full SQL expressiveness. Section 9 distills nine lessons from building and operating the system.

2 PROBLEM FORMULATION

We formalize the problem, survey existing approaches (Table 1), and identify four challenges that motivate our design.

2.1 Problem Statement

Given a document corpus $D = \{(v_i, id_i)\}$ of n vectors with optional metadata columns and a query set $Q = \{q_j\}$ of m vectors, the goal is to compute, for each query q_j, the top-k documents by similarity (cosine, L2, or inner product). Beyond standard ANN formulations, we impose two additional requirements:

(1) the computation must execute within SQL semantics, composing with filters (`WHERE`), joins (`JOIN`), aggregations (`GROUP BY`), and rankings (`WINDOW` functions); and
(2) execution must parallelize across workers, with data movement sublinear in corpus size.

2.2 Limitations of Existing Approaches

These two requirements—SQL composability and distributed execution—are individually addressed by existing systems but not jointly. Table 1 summarizes the landscape: no existing system supports the combination of on-the-fly pre-filter indexing, arbitrary joins and aggregations on results, and optimizer-driven distributed execution within a single SQL statement.

Vector similarity search over embeddings provides a semantic similarity primitive across text, image, and video modalities—complementing the exact-match and range predicates that SQL already supports. Integrating this primitive natively into the SQL engine closes a gap for analytical and AI workloads.

Table 1: SQL integration, distribution, and composability across vector search systems.

System	SQL	Dist.	Comp.	Workload
Pinecone [3]	No	Yes	No	RT serving
Milvus [30]	No	Yes	No	RT serving
pgvector [2]	Yes	No	Part.†	OLTP
BigQuery [6]	Yes	Yes	Part.‡	Batch
Snowflake [11]	Part.§	Yes	No	Batch
Databricks [9]	Part.§	Yes	No	Batch
SingleStore [13]	Yes	Yes	Part.‡	HTAP
Presto VS	**Yes**	**Yes**	**Yes**	**Batch**

Dist. = distributed execution. Comp. = composable: pre-filter indexing (`WHERE` before index construction), joins/aggregations on results, and on-the-fly index construction. †Joins on results but no pre-filter or on-the-fly indexing. ‡Pre-built indices with post-filter semantics only. §SQL syntax but routes vector operations to a separate engine.

2.3 Core Challenges

C1: Network. Naive distributed similarity search broadcasts every query to every shard, producing $O(m \times S)$ network messages for m queries and S shards. The challenge is achieving co-located execution where queries and their relevant index partitions reside on the same worker without shuffling index data.

C2: Memory Constraints. At 1,024 dimensions, 1B FP32 vectors require ~4.1 TB for raw embeddings alone—far exceeding a single worker's 256 GB RAM. Real-world cluster size distributions are highly skewed (some clusters contain >600× the average), causing unpredictable memory spikes during `GROUP BY` aggregation. The challenge is bounding peak memory independently of both corpus size and cluster skew.

C3: SQL Composability. Existing systems treat vector search as an opaque operation outside the relational model, preventing users from applying `WHERE` filters before indexing, joining search results with metadata, or aggregating across result sets. The challenge is implementing vector search as standard relational operators so it composes naturally with the full SQL language.

C4: API Design. Vector search users range from data scientists writing ad-hoc queries to engineers building data pipelines. A single interface cannot serve both: too simple and experts lose control; too complex and novices face a steep learning curve. The challenge is providing a layered API where both levels share the same optimized execution path.

3 ARCHITECTURE OVERVIEW

This section presents our distributed vector search architecture (Figure 1). By grouping vectors into clusters, we reduce brute-force complexity from $O(M \times N)$ to approximately $O(M \times nprobe \times N/K)$,

where K is the number of clusters and *nprobe* is the number of clusters searched per query. Since *nprobe* $\ll K$ (typically 50 out of 100K), this yields a ~2,000× work reduction.

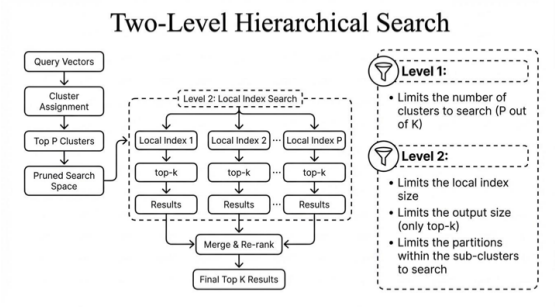


Figure 1: Two-level index architecture with broadcast global codec and co-located per-cluster local indices.

The relational decomposition. The alignment between partition-based vector search and relational algebra is structural. Vector search has three stages: (1) partition vectors by similarity, (2) build searchable structures per partition, (3) probe partitions and merge results. These map directly to relational primitives: (1) is a scalar function whose output serves as a hash-distribution key, (2) is a `GROUP BY` aggregate, and (3) is an equi-join followed by a global top-k. Because the mapping preserves computational structure, the relational engine's existing optimizations apply without modification: hash partitioning distributes clusters, predicate pushdown enables pre-filter indexing, and aggregate pipelining bounds memory.

This mapping gives cluster assignment a dual purpose: it is both a similarity proxy (vectors in the same cluster are likely similar) and a distribution key (cluster IDs drive hash partitioning). This dual role eliminates the need to broadcast queries to all workers or shuffle index data across the network. The system requires three Velox functions—a scalar, an aggregate, and a scalar—with no new plan nodes or operators. The Level 2 table function (Section 7.2) adds optimizer rewrite rules but no new execution infrastructure. Adding a new local index type requires only a builder function within the existing aggregate. Our implementation leverages Prestissimo [16], the C++ Presto worker built on Velox, enabling direct integration with native libraries such as FAISS [21] for distance computation and index operations—an integration path unavailable in the Java-based Presto worker.

Formal mapping. Let $C = \{\mu_1, \ldots, \mu_K\}$ denote the codec centroids, $\phi_C(v) = \arg\min_c \|v - \mu_c\|$ the nearest-centroid assignment, and $\phi_C^{(p)}(v)$ the top-p assignments. Using standard relational notation (π for projection, γ for grouped aggregation, $\bowtie$ for join, τ_k for top-k selection), the three stages correspond to:

(1) **Partitioning** (scalar function + distribution key):

$$D' \;=\; \pi_{v,\, id,\, \phi_C(v) \to c}\,(D)$$

Each document is augmented with a cluster label c; the optimizer uses c as the hash-distribution key. Cluster assignment computes brute-force distances between each vector and all K centroids via matrix multiplication, equivalent to IVF-FLAT centroid assignment. The computational cost is $O(N \cdot K \cdot d)$ for the full corpus, distributed across workers; for large codecs ($K \geq 50{,}000$), HNSW routing reduces this to $O(\log K)$ per vector (Section 4.3). Documents are single-probe (one cluster); queries are multi-probe (*nprobe* clusters), with a global top-k merge over shard-local results. The hash-distribution key is `cluster_id` alone; `id` is carried through for deduplication, not routing.

(2) **Index construction** (grouped aggregation):

$$I \;=\; \gamma_{c,\, \mathrm{BuildIdx}(v,\, id)}\,(D')$$

`BUILD_INDEX` is an aggregate function applied per cluster group, producing one local index per cluster. The local index type (FLAT, IVF, RaBitQ, HNSW) is a user-specified parameter within the aggregate. The local index type is an explicit parameter to `BUILD_INDEX`, specified by the user (e.g., `'FLAT'` vs. `'RaBitQ_NB8'`). The system does not auto-select; users control the accuracy–memory trade-off by changing this single parameter without modifying the query structure.

(3) **Search** (equi-join + global top-k):

$$R \;=\; \tau_k\Big(\; Q' \;\bowtie_c\; I,\;\; \mathrm{SearchIdx}(q,\, k,\, idx)\Big)$$

where $Q' = \pi_{q,\, \phi_C^{(p)}(q) \to c}(Q)$ expands each query into p rows (one per probed cluster), the equi-join on c co-locates queries with their target indices, and τ_k selects the global top-k per query.

Decoupled codec and local index. Our architecture separates the codec (global centroids) from the local index (per-cluster searchable structures), unlike monolithic ANN libraries such as FAISS [21] and SPANN [15] where these are tightly coupled. This separation has three consequences: (1) the codec is reusable—centroids encode coarse geometric structure that changes slowly, so a codec trained on one snapshot remains effective for subsequent rebuilds; (2) the local index is inherently consistent with the current table snapshot, since `BUILD_INDEX`, as a standard SQL aggregate, operates over the document rows in its enclosing query's result set—specifically, those that survive the `WHERE` clause. If the query filters to in-stock products, only in-stock product embeddings are indexed; and (3) the codec is a lightweight, broadcastable artifact (~200 MB at K=100,000), while local indices (54 GB+ at scale) remain co-located with their data.

Bounding memory and output. Real-world embeddings exhibit skewed distributions—some clusters contain >600× more vectors than others, causing OOM during joins and unbounded cross-join output. Sub-clustering (Figure 2) addresses this through capping each sub-index to available memory, top-k per sub-index (bounding output), and streaming joins (query table on build side, local indices on probe side).

This ensures peak memory is bounded by the larger of the query table and the largest local index, not their sum.

The end-to-end flow has four stages: (1) assign clusters and hash-distribute documents, building local indices per worker; (2) assign query clusters (*nprobe* per query) and hash-distribute; (3) co-located

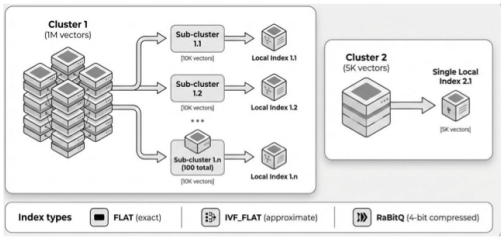


Figure 2: Sub-clustering for bounding memory and output.

join and local search—no network shuffle; (4) global merge to select final top-k per query.

Complexity analysis. We analyze the costs of this architecture (compute, memory, and communication).

Proposition 1 (Per-query work). *Under a codec with K clusters and uniform cluster sizes, each query probes p clusters of expected size N/K, performing $O(p \cdot N/K \cdot d)$ distance computations. For M queries, total work is $O(M \cdot p \cdot N \cdot d/K)$, a factor of K/p reduction over brute-force $O(M \cdot N \cdot d)$. With K=100,000 and p=50, this yields a 2,000× reduction.*

Proposition 2 (Skew bound). *Let $n_{\max}$ denote the largest cluster size. Without sub-clustering, peak per-worker memory is $\Theta(n_{\max} \cdot d \cdot b)$ where b is the per-dimension byte width. Sub-clustering with cap s reduces this to $\Theta(s \cdot d \cdot b)$, independent of $n_{\max}$. At s=10,000, d=512, b=4 (FP32): peak ≈ 20 MB per sub-index.*

Proposition 3 (Communication). *The codec (centroids) has size $O(K \cdot d)$ and is broadcast once to all S workers. Local indices remain co-located and are never transferred. Total network cost is $O(S \cdot K \cdot d + M \cdot p \cdot result\_size)$ (communication, not computational), where the second term covers query routing and result collection. For K=100,000, d=512, b=4: codec broadcast is ∼200 MB versus ∼1.4 TB for a full corpus shuffle—a ∼7,000× reduction in index-related transfer.*

Algebraic equivalence. The SQL decomposition is an algebraic refactoring of standard IVF search: the same centroids produce the same cluster assignments, the same local index type produces the same distance computations, and the same top-k merge produces the same results. The decomposition changes *how* the computation is organized across SQL operators, not *what* is computed. Recall loss comes exclusively from two user-controlled parameters—the codec (quantization) and *nprobe*—which are identical whether the search is executed monolithically in FAISS or decomposed into SQL operators. Section 8.6 validates this empirically with a recall attribution table showing zero additional recall loss from the decomposition.

The following sections detail our solutions to each challenge. We present them in dependency order: distributed execution (C1) establishes the cluster-based architecture that the memory (C2) and composability (C3) solutions build upon, and the API design (C4) synthesizes all three into a unified interface.

4 DISTRIBUTED EXECUTION (C1)

Distributed similarity search in sharded systems typically requires broadcasting every query to every node, since document IDs are orthogonal to similarity. This results in $O(m \times S)$ network messages for m queries and S shards—a dominant cost at scale (network and computational), forcing each worker to scan its entire local corpus for every query.

4.1 Cluster-Based Routing and Local Joins

Our system leverages cluster assignment as both a similarity proxy and a distribution key:

- **Efficient cluster assignment**: Computing distances to K centroids has the complexity of $O(K \times d)$, practical for moderate K (e.g., 10,000); for larger codecs ($K \geq 50{,}000$) we apply HNSW routing (Section 4.3). Cluster assignment is implemented as a Velox [28] VectorFunction.
- **Hash-based partitioning**: Documents and queries are hash-distributed by cluster ID, co-locating data and queries on the same worker.
- **Optimizer-driven query pruning**: The SQL optimizer routes each query only to nodes hosting its top-*nprobe* clusters, rather than broadcasting to all shards.

This approach enables local joins with zero index data movement. For example, with K=10,000, *nprobe*=10, S=100: each query contacts at most $\min(nprobe, S) = 10$ shards instead of all 100 (10× fan-out reduction), and searches only $nprobe/K = 0.1\%$ of the corpus—a combined ∼1,000× work reduction versus broadcast-and-scan.

4.2 Execution Plan Structure

The optimizer generates the distributed plan shown in Figure 3.

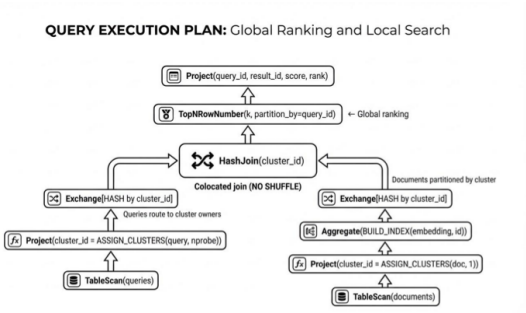


Figure 3: Execution plan structure.

Both documents and queries hash-distribute by `cluster_id`, ensuring co-location: documents in cluster c land on worker $w(c)$, and queries routed to cluster c are sent to the same worker. The join is local—zero index data movement.

Equijoin implementation. The equijoin between the query table and the index table is a standard hash join on `cluster_id`. For each matching (query, index) pair within a partition, `SEARCH_INDEX` performs a local ANN search via FAISS on the co-located index—a vectorized function call on the probe side of the join, not a nested-loop scan over raw vectors. The join operator handles data routing (matching queries to their assigned cluster's index); the ANN

computation is fully encapsulated within `SEARCH_INDEX`, which delegates to FAISS search kernels.

4.3 HNSW Routing for Large Codecs

At scale, global indices may contain several hundred thousand to millions of centroids (K). Computing distances to all K centroids becomes a bottleneck: $O(K \times d)$ per query. We address this by applying HNSW [25], a graph-based ANN index with $O(\log n)$ query time, to the global centroids. On 700K queries, HNSW routing reduces cluster assignment CPU from 23.69 days (brute-force IVF_FLAT) to 40.72 hours (14× reduction) at the cost of 4 percentage points of recall (98%→94%)—a recall-computation trade-off acceptable when K exceeds 50K. Wall-clock impact is less pronounced because execution on large clusters is I/O-bound; CPU savings primarily reduce resource consumption rather than latency.

5 MEMORY-EFFICIENT INDEXING (C2): LIMITING PEAK MEMORY

Presto's fully in-memory execution—its key strength—becomes a liability for vector search: at 1,024 dimensions (a common dimensionality), 1B FP32 vectors require ~4.1 TB for raw embeddings alone—far exceeding a single Presto worker's 256 GB RAM. We address this with two complementary techniques.

5.1 Technique 1: RaBitQ Compression

RaBitQ [17] compresses each vector from FP32 to N_B bits per dimension. The bit-width N_B controls the compression–recall trade-off: Table 2 sweeps $N_B \in \{1, 4, 8\}$ on DEEP1B [14] (100M vectors, 96-dim, L2, IVF40K, *nprobe*=50, k=100), measuring recall relative to FLAT (same codec) to isolate compression error from partitioning error.

Table 2: RaBitQ bit-width (N_B) vs. recall and compression (DEEP1B-100M subset, 96-dim, IVF40K, *nprobe*=50, k=100).

Config	R@1	R@10	R@100	B/vec	Compr.
FLAT (FP32)	1.000	1.000	1.000	384	1×
NB1 (1-bit)	0.279	0.422	0.549	20	19.2×
NB4 (4-bit)	0.841	0.882	0.912	60	6.4×
NB8 (8-bit)	0.990	0.991	0.993	100	3.8×

NB4 provides the best trade-off for most workloads (6.4× compression, 91% R@100)—on the 700M-vector corpus (512-dim), it reduces index memory from 1.37 TB to 54 GB. NB8 is near-lossless (99.3% R@100, 3.8×) when recall targets exceed 95%.

5.2 Streaming Construction: Adapting RaBitQ to SQL Semantics

Standard RaBitQ requires two passes—one for the global centroid, one for quantization—but SQL's `GROUP BY` aggregation streams vectors incrementally and cannot materialize an entire group. We adapt RaBitQ through sub-batch quantization (Figure 4): vectors are accumulated into fixed-size sub-batches (e.g., 10K); when full, a local centroid is computed, the sub-batch is quantized, and emitted as a self-contained sub-index with its own centroid and normalization parameters.

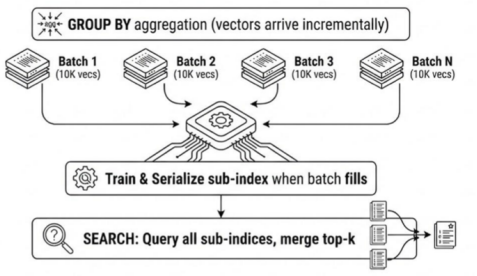


Figure 4: Streaming sub-index construction.

Local centroids approximate the global centroid well because vectors within a sub-batch share a global cluster assignment; RaBitQ's error bound [17] depends on query-vector angles relative to the centroid, which remain small within tight clusters. Concurrent work [12] independently validates this property. Our contribution is the systems challenge: making a two-pass algorithm operate within a single-pass aggregate where the engine controls batching, memory budgets, and spill-to-disk decisions.

Empirical validation: sub-batch size vs. recall. We validate the sub-batch approximation on the full DEEP1B [14] (1B vectors, 96-dim, L2) with an IVF40K codec (~25K vectors/cluster), varying sub-batch size s from 1K to 100K. Table 3 reports recall and wall-clock time on a 200-worker Presto cluster (*nprobe*=50, k=100).

Table 3: Streaming sub-batch size: recall and elapsed time (DEEP1B, 1B vectors, 96-dim, IVF40K, *nprobe*=50, k=100, 200 workers).

Config	R@100†	R@1‡	Build	Search
FLAT (baseline)	1.000	0.979	82 s	83 s
NB4 s=1K	0.893	0.788	222 s	42 s
NB4 s=5K	0.893	0.804	72 s	36 s
NB4 s=100K	0.893	0.813	66 s	48 s
NB8 s=6.4K	0.992	0.966	90 s	48 s

†R@100 vs FLAT (same IVF codec, isolates compression error). ‡R@1 vs official DEEP1B ground truth (brute-force exact search).

Table 3 confirms that **R@100 is invariant to sub-batch size** (0.893 across s=1K–100K for NB4; 0.992 for NB8), validating the streaming approximation at billion scale. R@1 improves modestly with larger s (0.788 at s=1K → 0.813 at s=100K) as local centroids better approximate the global centroid. Build time is the only computational cost: s=1K incurs 3× computational overhead from many small sub-batches, but $s \geq 5$K eliminates this penalty. Search is consistently 2× faster than FLAT because compressed indices fit in cache. Measuring end-to-end against the official DEEP1B ground truth (brute-force exact search over all 1B vectors) confirms this: FLAT itself achieves R@100 = 0.952 with *nprobe*=50, and the remaining gap for NB4 (R@100 = 0.866) and NB8 (R@100 = 0.949) is attributable entirely to the codec—not to streaming.

Validation: Distributed vs. Monolithic Execution. Table 4 validates that our distributed streaming approach preserves recall compared to monolithic single-process FAISS execution. Both configurations

use identical C++ code (FAISS IndexIVFRaBitQ), the same pre-trained centroids, and the same parameters (IVF40K, *nprobe*=50, L2); the only variable is single-process vs. 200-worker distributed execution.

Table 4: Distributed vs. monolithic recall (DEEP1B-100M subset, IVF40K, *nprobe*=50).

Config	Metric	Monolithic	Presto (200w)
NB4	R@1 vs FLAT	0.843	0.818
NB4	R@100 vs FLAT	0.911	0.893
NB8	R@1 vs FLAT	0.991	0.986
NB8	R@100 vs FLAT	0.992	0.992

NB8 achieves near-identical recall between monolithic and distributed execution (<0.5% degradation), validating that streaming sub-batch quantization preserves search quality. NB4 shows modest degradation (~2%) due to quantization sensitivity at lower bit-widths.

5.3 Technique 2: Sub-Clustering

Even with compression, skewed clusters cause memory spikes: in the 700M corpus, the largest cluster contains over 4.7M vectors—more than 600× the average—and a single FLAT sub-index at that size would consume ~9.6 GB, exceeding per-operator memory budgets on commodity workers.

Sub-clustering caps each sub-index at a fixed size s (e.g., 10K vectors). When a cluster exceeds s, the aggregate function partitions it into $\lceil n_c/s \rceil$ sub-clusters, each producing an independent sub-index. This serves a dual purpose (Proposition 2): it bounds peak memory independently of cluster skew, and it acts as a load-balancing mechanism—splitting large clusters into uniform sub-indices that hash-distribute across workers, increasing parallelism and eliminating straggler effects (up to 6.4× the average on DEEP1B, >600× on the internal corpus). We formalize the recall and peak memory trade-off below.

This mechanism also unifies with streaming RaBitQ (Section 5.2): the sub-cluster cap and the sub-batch size are the same parameter, so each sub-cluster becomes a self-contained RaBitQ quantization unit with its own local centroid. The two techniques—compression and sub-clustering—address orthogonal axes of the memory problem (vector size and cluster skew, respectively) and compose multiplicatively: at s=10,000 and d=512, peak memory per sub-index drops from ~20 MB (FLAT) to ~0.8 MB (RaBitQ), a 25× reduction.

Proposition 4 (Sub-clustering recall and computational cost). *Let cluster c contain n_c vectors and let s denote the sub-cluster cap, producing $J = \lceil n_c/s \rceil$ sub-indices. Because every query assigned to cluster c probes* all *J sub-indices and the per-query top-k is computed by merging sub-index results, the candidate set is identical to searching the full cluster:*

$$\text{top-}k\Big(\bigcup_{j=1}^{J} \text{top-}k(I_{c,j},\, q,\, \delta)\Big) \;=\; \text{top-}k(I_c,\, q,\, \delta)$$

up to tie-breaking among equidistant vectors. For exhaustive-scan indices (FLAT, RaBitQ), sub-clustering therefore reduces peak memory from $\Theta(n_{\max} \cdot d \cdot b)$ to $\Theta(s \cdot d \cdot b)$ with zero recall loss.

The computational overhead depends on the local index type:

- ***FLAT / RaBitQ.*** *Both perform a linear scan over all vectors in the cluster. Scanning J sub-indices of size s performs the same n_c distance computations as a single scan, plus an $O(J \cdot k \log k)$ merge—negligible since $k \ll s$. Sub-clustering adds no measurable computational cost.*
- ***IVF_FLAT.*** *Each sub-index maintains K_ℓ local centroids. Per sub-index query computational cost is $O(K_\ell \cdot d + p_\ell \cdot s/K_\ell)$, yielding a total of $O(J \cdot K_\ell \cdot d)$ additional centroid-lookup computational overhead compared with a single IVF index of size n_c. Since K_ℓ is typically small (a few hundred) and d is fixed, this computational overhead is modest relative to the probe computational cost.*
- ***HNSW.*** *Each sub-index builds an independent navigable small-world graph. Query computational cost per sub-index is $O(\log s)$, so total computational cost is $O(J \cdot \log s)$ versus $O(\log n_c)$ for a single graph. Because $J \cdot \log s = \lceil n_c/s \rceil \cdot \log s$ while $\log n_c = \log(J \cdot s) = \log J + \log s$, the computational overhead factor is $\lceil n_c/s \rceil / (1 + \log J/\log s)$—significant for large clusters, making HNSW the index type most sensitive to sub-clustering.*

In practice, FLAT and RaBitQ are the default local index types in our system; for these, sub-clustering is computationally cost-neutral.

Empirical validation: load balancing. Figure 5 shows the cumulative distribution of cluster sizes on two corpora. DEEP1B (IVF40K) has moderate skew: median ~22K, max ~159K (6.4× the average). The internal 700M corpus (IVF100K) exhibits extreme skew: median ~6.9K, but the largest cluster contains ~4.7M vectors (>600× the average), driven by a small number of degenerate clusters that accumulate outlier embeddings. Table 5 quantifies the effect of sub-clustering (s=10,000): it reduces the max/median ratio from 684× to 1.0× on the 700M corpus, ensuring uniform sub-index sizes that hash-distribute evenly across workers. Per-worker CPU skew on the DEEP1B search step (200 workers) confirms: the search kernel's max/p50 ratio is 1.2×, validating that uniform sub-index sizes translate to uniform per-worker execution time.

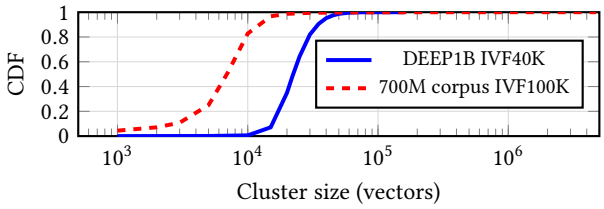


Figure 5: CDF of cluster sizes. DEEP1B IVF40K left, 700M corpus IVF100K right, log-scale x-axis.

6 SQL COMPOSABILITY (C3): AGGREGATE FUNCTIONS

SQL composability follows from the relational decomposition. Because `BUILD_INDEX` is a standard aggregate and `SEARCH_INDEX` is a scalar function within `JOIN`, vector search inherits every SQL operator that composes with aggregates and joins—`WHERE` filters

Table 5: Index size distribution with and without sub-clustering.

Dataset	Config.	Med.	p95	Max	Max/Med.
DEEP1B	No sub-clust.	22K	41K	159K	7.2×
	s=10K	≤10K	≤10K	10K	1.0×
700M corpus	No sub-clust.	6.9K	13K	4.7M	684×
	s=10K	≤10K	≤10K	10K	1.0×

before index construction, `GROUP BY` after search, `HAVING` on aggregate results—eliminating separate vector database infrastructure and enabling optimizer-driven decisions (co-located joins, predicate pushdown). External vector databases (Pinecone, Milvus, Weaviate) cannot achieve this because they implement search as an opaque API call outside the relational algebra.

6.1 Our Approach: Three SQL Primitives

The key to SQL composability is implementing vector operations as **standard SQL functions**:

- `BUILD_INDEX(embedding, id)` — aggregate, invoked via `GROUP BY cluster_id`
- `SEARCH_INDEX(query, k, index)` — scalar, within equi-join on `cluster_id`
- `ASSIGN_CLUSTERS(vec, k, codec)` — scalar, usable in any SQL expression

This design enables users to compose vector search with any SQL pattern.

6.2 Query Patterns Enabled by Aggregate Functions

Because vector search operators are standard SQL functions, they compose naturally with relational predicates. A standard `WHERE` clause filters rows before index construction—e.g., `SELECT cluster_id, BUILD_INDEX(embedding, doc_id) FROM products WHERE in_stock AND price < 50 GROUP BY cluster_id`. In Pinecone/Milvus, you must index all products first, then filter results—potentially returning fewer than k results if many are out-of-stock. These primitives compose into end-to-end hybrid analytics—pre-filter indexing, ANN search, metadata joins, and aggregation—in a single SQL statement, a pattern impossible in external vector databases or systems with pre-built indices.

6.3 Pre-Filter Analysis

Table 6: Pre-filter comparison across vector search systems.

System	Pre-filter	Post-filter	Result Guarantee
Presto VS	Yes	Yes	Exactly k results
Milvus [30]	No	Yes	May return $< k$
Pinecone [3]	No	Yes	May return $< k$
pgvector [2]	Partial†	Yes	k results (seq. scan)
BigQuery [6]	No	Yes	May return $< k$
SingleStore-V [13]	No	Yes	May return $< k$

Pre-filter: predicate applied *before* index construction. Post-filter: predicate applied *after* ANN search. †pgvector applies `WHERE` on the base table but requires a sequential scan, not an ANN index scan.

Proposition 5 (Pre-filter yield guarantee). *Let $\sigma \in (0, 1]$ be the selectivity of a predicate P on corpus D (i.e., $|P(D)| = \sigma \cdot |D|$). Post-filter indexing retrieves top-k from the full corpus, then applies P; the expected yield is $k \cdot \sigma$ when result scores are independent of P. Pre-filter indexing applies P before index construction, guaranteeing exactly $\min(k, |P(D)|)$ results per query.*

Table 7 validates this on DEEP1B (1B vectors, IVF40K, *nprobe*=50, k=10, 200 workers, 10K queries). Selectivity σ is controlled by filtering on uniformly distributed categories. At σ=1%, pre-filter achieves 95.2% Recall@10 while post-filter drops to 10.3%—a 9.2× gap. Pre-filter recall remains stable at 95–97% across all σ because the index is always built on exactly the qualifying rows. Post-filter recall degrades to approximately σ (e.g., 10.3% at σ=1%, 24.9% at σ=25%), confirming Proposition 5. To guarantee k results, post-filter must retrieve $\sim k/\sigma$ candidates—but σ depends on the predicate and is generally not known at query time, making over-retrieval impractical. At σ=100%, both are identical. The SQL decomposition makes per-predicate rebuilding a routine `GROUP BY` aggregation rather than an infrastructure burden. (The experiment uses uniformly distributed categories; correlated predicates may affect codec routing quality, though pre-filter recall is bounded below by the filtered corpus size.)

Table 7: Pre-filter vs. post-filter recall on DEEP1B (1B vectors, IVF40K, *nprobe*=50, k=10).

	FLAT		RaBitQ NB8		
σ	Pre	Post	Pre	Post	Ratio
1%	0.952	0.103	0.953	0.104	9.2×
5%	0.951	0.100	0.951	0.099	9.5×
10%	0.952	0.112	0.949	0.111	8.5×
25%	0.959	0.249	0.952	0.249	3.8×
50%	0.965	0.498	0.957	0.499	1.9×
100%	0.969	0.969	0.959	0.959	1.0×

Pre: `WHERE` before `BUILD_INDEX`. Post: search full index, then filter. Ratio: pre-filter / post-filter (FLAT).

7 HIERARCHICAL API (C4): PRIMITIVES AND SYNTACTIC SUGAR

SQL-native vector search faces a fundamental API design tension. Novice users want a simple declarative interface—specify documents, queries, and k—while expert users need full control over codec selection, cluster routing, and index construction to tune trade-offs between recall and computational cost. A single interface cannot serve both: too simple and experts lose composability; too complex and novices face a steep learning curve.

7.1 Our Solution: Two-Level API

Level 1: Primitives—Building blocks with full control. The three primitives (Section 6.1) give experts full control over cluster routing, index construction, and search. Experts compose these directly with standard SQL operators (Section 6.2).

Level 2: Syntactic Sugar—Simple one-liner for common cases. The Level 2 table function provides a declarative interface: `SELECT * FROM VECTOR_SEARCH(DOCUMENTS => TABLE(products), QUERIES => TABLE(query_vectors), TOP_K => 10)`.

7.2 Optimizer Rewrite: Sugar to Primitives

The optimizer applies three rule-based transformations (Figure 6): (1) decompose `VECTOR_SEARCH` into `ASSIGN_CLUSTERS` → `BUILD_INDEX` → `SEARCH_INDEX`; (2) inject hash-partitioning on `cluster_id` for both pipelines, with *nprobe* controlling query fan-out; and (3) assemble the final plan as a distributed `GROUP BY` with `BUILD_INDEX`, a co-located join, and a global top-k merge.

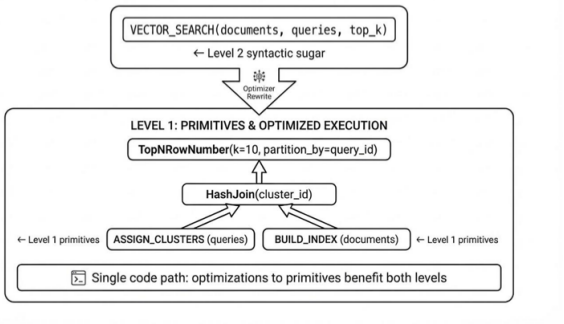


Figure 6: Optimizer rewrite `VECTOR_SEARCH` to primitives.

This rewrite ensures both API levels share the same optimized execution path. The optimizer also selects between pre-built and on-the-fly indexing based on whether `INDEX_TABLE` or `CODEC` is specified: when a pre-built index table is provided, the `BUILD_INDEX` stage is replaced with a direct table scan.

8 EVALUATION

We evaluate Presto Vector Search on an internal corpus (Embeddings-700M, 512-dim) and the public DEEP1B benchmark (1B vectors, 96-dim). Three questions arise from the decomposition: does it preserve recall? what computational overhead does it add? where does the computational cost go? Because capabilities unique to our system—on-the-fly pre-filter indexing and composable SQL functions—cannot be expressed in competing systems (Table 1), end-to-end comparisons are infeasible. We instead validate each architectural decision independently, confirm end-to-end utility across three modalities and 512- to 5,120-dimensional embeddings with 95–99% recall (Section 8.2), and directly address decomposition computational overhead via a controlled FAISS comparison (Section 8.6) and per-operator computational cost attribution (Section 8.7). On the subset of functionality shared with these systems (pre-built index search), the underlying ANN algorithms are identical (IVF + FAISS); our contribution is the relational decomposition that enables the unique capabilities.

8.1 Experimental Setup

We evaluate on two datasets:

- **Internal corpus (700M):** 700M vectors (512-dim), 700K queries, 100 workers (AMD Bergamo, 88 cores, 50 Gbps), IVF 100K codec, sampled exact ground truth.
- **DEEP1B [14]:** 1B vectors (96-dim), 10K queries, 150–200 workers (Intel Xeon AVX2, 26 cores, 25 Gbps), IVF 40K codec, official ground truth.

Both clusters use 64 GB RAM/worker, Tectonic distributed FS, and FAISS 1.14.1. Worker counts vary by experiment (noted in each table caption); proportions are hardware-independent.

Common configuration: Local index types include FLAT (exact), IVF_FLAT (hierarchical), and RaBitQ (NB4/NB8 quantization). Default *nprobe*=50. Spilling enabled on all workers. Metrics: Recall@k, wall-clock time, peak memory, network transfer.

8.2 Workload Validation

We validated Presto Vector Search across four workloads spanning three modalities (text, image, video), labeled W1 (entity matching), W2 (candidate retrieval), W3 (near-duplicate detection), and W4 (content retrieval). Table 8 summarizes the scale, performance, and recall of each workload.

Table 8: Workload scale, performance, and recall.

Use Case	Scale	Dim	Speedup	CPU Red.	Recall
W1: Entity match.	2M × 2B	1024	N/A*	N/A*	98.3%‡
W2: Cand. retrieval	100 × 21M	512	3×	95%	98.6%
W3: Near-dup. det.	1K × 69M	512	4×	97%	98.1%
W4: Content retr.	1K × 55M	5120	10×	99%	96.4%

*W1 previously timed out; now completes in ~1 hour. ‡Exact recall on 2M×2B is prohibitive (~4×10^{15} distance computations at 1024-dim). Estimated by sampling 2,931 queries against 2M candidates with a proportionally sized codec (IVF6127, *nprobe*=50); exact NN via brute-force on the sample. The sample probes 50/6,127 = 0.82% of clusters; at full scale (50/100K = 0.05%), the lower probe fraction reduces coverage but denser clusters improve routing quality—these effects work in opposite directions. 95% binomial CI: [97.8%, 98.8%] (sampling variance only; does not capture the net codec-scaling effect).

Findings.

- **W1**: A 2M × 2B entity matching query—previously timing out with even 0.25% sampling using brute-force—now completes in about 1 hour, enabling previously infeasible queries. W1 uses an independent corpus of 2 billion document vectors (1024-dim), distinct from the DEEP1B benchmark (1B vectors, 96-dim) used in Section 8.3.
- **W2**: Achieved 3× faster execution, 95% computational cost reduction, and 98.6% recall compared with brute-force search on a 21M-candidate retrieval dataset.
- **W3/W4**: Enabled interactive exploration across image and video modalities with 96–98% recall and 4–10× speedup.

All four workloads achieve over 95% recall with default parameters, validating that the approximation techniques preserve result quality across embedding modalities and dimensionalities ranging from 512 to 5,120.

8.3 Scalability & Network Efficiency (C1)

Can the relational decomposition scale to billion vectors without proportional computational cost increase? We evaluate scalability by varying the candidate corpus size from 70M to 700M vectors on the internal corpus (100 workers, 700K queries) and on the public DEEP1B benchmark at 1B scale (200 workers, 10K queries). Table 9 reports end-to-end time and index memory.

Findings.

Table 9: Scaling and execution mode comparison at *nprobe*=50. On-the-fly builds index during query; pre-built loads stored index.

Cand.	Index	Mode	Elapsed	Pk. Mem.	Spill
Internal corpus (512-dim, 100 workers, 700K queries)					
70M	FLAT	Pre-built	4.40 min	140 GB	0
70M	RaBitQ	Pre-built	3.98 min	5.51 GB	0
700M	FLAT	Pre-built	13.63 min	1.37 TB	0
700M	RaBitQ	Pre-built	13.51 min	54 GB	0
70M	FLAT	On-the-fly	7.52 min	374 GB	0
70M	RaBitQ	On-the-fly	6.73 min	373 GB	0
700M	FLAT	On-the-fly	35.2 min	1.12 TB	1.17 TB
700M	RaBitQ	On-the-fly	24.58 min	1.12 TB	1.17 TB
DEEP1B [14] (96-dim, 200 workers, 10K queries, pre-built CTE pipeline)					
1B	FLAT	Pre-built	82 s + 83 s	18.3 MB	0
1B	NB4 (s=5K)	Pre-built	70 s + 38 s	2.2 MB	0
1B	NB8 (s=6.4K)	Pre-built	89 s + 46 s	3.0 MB	0

- Presto Vector Search achieves sub-linear scaling: a 10× corpus increase (70M→700M) yields only 3.1–3.4× time increase, consistent with $O(N/K)$ per-query complexity. Search time is nearly identical across FLAT and RaBitQ (13.63 vs 13.51 min at 700M), confirming that execution is I/O-bound and compression is effectively free in terms of latency.
- On DEEP1B (1B vectors), end-to-end BUILD+SEARCH completes in under 2.5 minutes on 200 workers: NB4 takes 70 s + 38 s = 108 s and NB8 takes 89 s + 46 s = 135 s (CTE pipeline, Table 9). NB4 search is 2.2× faster than FLAT (38 s vs 83 s) and reduces peak per-worker memory by 8.3× (18.3 MB → 2.2 MB).
- Network transfer: Only 200 MB (codec) vs 1.4 TB (full corpus shuffle)—∼7,000× reduction.

CPU profiling on the 700M corpus reveals that cluster assignment dominates total compute: candidate-side assignment (top-1) accounts for 49% and query-side assignment (top-50) accounts for 50% of total CPU time, while index construction and search together consume less than 1%. This validates our design decision to optimize cluster routing via HNSW (Section 4.3) as the primary computational cost lever.

8.4 Local Index Comparison (C2)

Which local index type best balances memory and recall? We compare local index types on both the 700M internal corpus and the public DEEP1B benchmark (1B vectors) to characterize the memory–recall trade-off (Tables 10 and 11).

Note on the FLAT baseline (Table 10). The 100% recall figure is measured relative to exact brute-force within each cluster. Score-level analysis confirms that 93.7% of result differences arise from tie-breaking among vectors with identical scores (1.0), not search quality degradation—establishing FLAT as a reliable exact baseline.

Note: Recall in Table 10 is measured relative to exact (FLAT) search within each cluster (local recall). End-to-end recall (Table 9 and Figure 7a) is the product of global routing recall and local recall.

Findings.

- **Memory:** RaBitQ reduces index memory by 25× on 700M (1.37 TB → 54 GB) and 5.1× on DEEP1B (365 GB → 71 GB) with no latency overhead. IVF_FLAT does *not* reduce memory—total index size is slightly larger than FLAT (381 GB vs 365 GB on DEEP1B) due to sub-centroid memory overhead.
- **Recall:** IVF_FLAT with *nprobe*=100 achieves 98% (700M) / 96.7% (DEEP1B) local recall. Local *nprobe* has less impact than global *nprobe*; end-to-end recall is the product of global and local recall.
- **Consistency:** These trade-offs hold across scales: on DEEP1B (1B vectors), local index recall matches 700M patterns (Table 10), and NB4/NB8 achieve R@100 = 0.866/0.949 vs ground truth.

Table 10: Local index recall (global *nprobe*=50).

Local Index	nprobe	700M	DEEP1B	Use Case
FLAT	N/A	100%	100%	Max accuracy
IVF_FLAT	10	91%	92.6%	Mem-constrained
IVF_FLAT	100	98%	96.7%	Near-exact
RaBitQ NB4	N/A	89%	89.3%	Min memory
HNSW_SQ	N/A	—	OOM†	Impractical

†HNSW_SQ (M=32, 8-bit SQ) achieves 98.6% recall vs FLAT on 100M vectors but exceeds worker memory at 1B scale: the streaming `GROUP BY` must construct ∼120K graphs concurrently, and HNSW's per-graph construction buffers aggregate beyond worker capacity. RaBitQ NB8 achieves comparable recall (99.2%) without this constraint. HNSW is used effectively for centroid routing (Section 4.3), where a single graph is built once.

Table 11: Memory comparison (global *nprobe*=50).

Local Index	700M	DEEP1B	Red. 700M	Red. DEEP1B	Time
FLAT	1.37 TB	365 GB	1×	1×	13.6 min
IVF_FLAT	—	381 GB	—	0.96×*	—
RaBitQ NB4	54 GB	71 GB	25×	5.1×	13.5 min
HNSW_SQ	—	OOM†	—	—	—

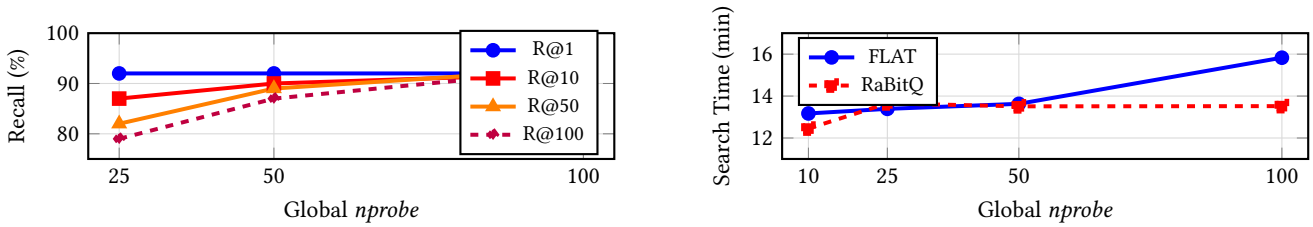
*IVF_FLAT is slightly larger than FLAT due to sub-centroid memory overhead (∼9 GB for IVF metadata).

8.5 Recall vs. Resource Trade-off

We vary the global *nprobe* parameter (the number of clusters to search per query) and report recall on a 7M-candidate corpus (Figure 7a) and resource usage on the 700M corpus (Figure 7b). **Important context:** the recall figures in this subsection (79–93%) use an intentionally over-partitioned stress-test codec—100K centroids over 7M vectors (∼70 vectors per cluster)—designed to isolate the routing layer's sensitivity to *nprobe*. With per-workload codec tuning, the evaluation workloads achieve 96–99% recall (Table 8).

Findings.

- Recall@1 remains stable at 92% across all *nprobe* values, indicating the true nearest neighbor consistently falls within the query's top cluster. Higher *nprobe* primarily improves recall at deeper ranks (Recall@50, Recall@100).
- *nprobe*=50 achieves 90% Recall@10 with minimal computational overhead. Wall-clock time is stable at 12–14 min for both FLAT and RaBitQ across *nprobe*=10–50 (Figure 7b),



(a) Recall vs. global *nprobe*.

(b) Search time vs. global *nprobe*.

Figure 7: Effect of global *nprobe* parameter. (a) Recall at $k \in \{1, 10, 50, 100\}$ on 7M-vector stress test with 100K centroids (FLAT local index, 730K queries). (b) Wall-clock search time on 700M corpus with pre-built index (200 workers, FLAT vs. RaBitQ).

making *nprobe*=50 the default sweet spot: 8 percentage-point Recall@100 gain (79%→87%) over *nprobe*=25 at no increase in wall-clock time.

- Doubling *nprobe* (50→100) yields 2–6 percentage-point recall gains depending on rank depth but increases computational cost by 11% (12.2 to 13.5 aggregate CPU-days across 100 workers). FLAT wall-clock time rises to 15.83 min at *nprobe*=100, while RaBitQ remains flat at 13.52 min—suggesting that RaBitQ's smaller index footprint avoids the memory pressure that slows FLAT at high *nprobe*.
- Wall-clock time remains stable at approximately 12–16 minutes because the search is I/O-bound, not compute-bound. CPU time reflects total compute summed across all 100 workers (e.g., 12.2 days = 100 workers × ∼2.9 hours each).

8.6 Decomposition Computational Overhead: Monolithic vs. SQL-Native

One question is whether decomposing partition-based vector search into SQL operators introduces computational overhead. Every partitioned vector search system performs the same logical steps: cluster assignment, index construction, index loading, query routing, local search, and global top-k merge. Monolithic FAISS executes all steps in a single process; standalone vector databases distribute them via custom RPC; our approach maps them to standard SQL operators (scalar functions, `GROUP BY`, equijoin, `TopN`). The computation is identical—the difference is where each step executes. Dual-engine approaches (Table 1, †) route vector operations to a separate system, requiring dedicated infrastructure and explicit index synchronization when the base table changes. The SQL-native approach executes all steps—including index construction and search—within the existing query engine, inheriting its storage, caching, and fault tolerance without additional infrastructure.

We quantify the SQL-layer computational cost by comparing against standalone FAISS on DEEP1B (100M vectors, 96-dim)—same IVF40K algorithm, same pre-trained centroids—isolating the decomposition as the only variable. Table 12 reports the results.

Recall is identical at all *nprobe* settings, confirming algebraic equivalence. **Search time is dominated by index loading** from distributed storage (24–36 s for NB8) versus FAISS's in-memory access (1.1–3.3 s). FLAT search times are higher (37–48 s) due to larger index footprint, confirming that data access—not SQL computational overhead—drives the gap. Table 14 quantifies this at

Table 12: Monolithic FAISS vs. Presto decomposed path on DEEP1B-100M.

Metric	FAISS	Presto	Δ
Build (once)	1989 s	3246 s	1.6× slower†
Search (*np*=50)	1.1 s	24 s	22×
Search (*np*=100)	2.1 s	27 s	13×
Search (*np*=200)	3.3 s	36 s	11×
R@10 (all *np*)	identical		0% loss

†NB8 build overhead from RaBitQ quantization (computational); amortized across queries. FLAT build is 1.6× faster (distributed parallelism). Presto search is dominated by index loading from distributed storage; FAISS scales linearly from in-memory indices. FLAT search times are higher (37–48 s) due to larger index footprint, confirming data access as the dominant cost (I/O-bound, not computational). Index caching can reduce Presto's loading cost (I/O) for repeated queries.

1B scale: index data access accounts for 68% of search CPU, while SQL-layer computational overhead is <1%. **Build** is slower for NB8 (quantization computational overhead, amortized across queries); FLAT build is 1.6× faster due to distributed parallelism across 150 workers.

Recall attribution. Table 13 traces recall loss through each pipeline stage. Coarse quantization accounts for 3.1 pp of loss; RaBitQ adds 1.0 pp. Sub-clustering and the SQL decomposition contribute zero additional loss, confirming Proposition 4 and algebraic equivalence.

Table 13: Recall attribution per pipeline stage (DEEP1B, 96-dim, IVF40K, *nprobe*=50).

Stage	R@10	Source of Loss
Exact brute-force	100.0%	Ground truth
IVF partitioning (*nprobe*=50, FLAT)	96.9%	Coarse quantizer (−3.1 pp)
+ RaBitQ NB8 codec	95.9%	Quantization (−1.0 pp)
+ Sub-clustering (s=10K) + merge	95.9%	None (Prop. 4)
Decomposed SQL path	95.9%	None (algebraic equiv.)

8.7 Per-Operator Computational Cost Breakdown

Table 14 breaks down CPU and elapsed time by operator on DEEP1B (1B vectors, 200 workers, 10K queries). `ASSIGN_CLUSTERS` dominates at 97.6% of total CPU—a computational cost inherent to any partition-based system, not specific to the SQL decomposition. With pre-built indices, this computational cost is paid once during index

Table 14: Operator breakdown on DEEP1B 1B vectors (200 workers, 10K queries, FLAT). CPU total across workers; bottom shows SEARCH sub-breakdown.

Operator	CPU	CPU %	Elapsed
ASSIGN_CLUSTERS	2.55 d	97.6%	3.1 min
BUILD_INDEX	1.14 h	1.8%	43 s
SEARCH_INDEX	31 min	0.8%	19 s
SEARCH sub-breakdown			
Index TableScan	—	68%†	—
FAISS search kernel	—	19%†	—
Index data transfer	—	11%†	—
SQL computational overhead (join, top-k)	—	<1%†	—
Inter-operator transfer	—	0.03%†	—

†Percentage of SEARCH CPU only (from query plan analysis, cold cache).

construction and amortized across subsequent queries (~40× per-query CPU reduction). Elapsed times are lower due to parallelism (e.g., 2.55 CPU-days → 3.1 elapsed minutes across 200 workers).

The SEARCH sub-breakdown in Table 14 drills into the search step: index data access dominates (68%), followed by the FAISS kernel (19%). SQL-layer overhead—equijoin coordination, top-k merge, and query routing—is <1% (computational), and inter-operator data transfer is 0.03%—confirming that the decomposition does not introduce significant communication cost (I/O). These figures represent cold-cache execution; index caching can amortize the dominant data access cost (I/O-bound), consistent with the index-loading overhead observed in Table 12.

8.8 Pre-built vs. On-the-Fly Indexing

When should users pre-build indices vs. build on-the-fly? Table 9 compares pre-built and on-the-fly execution on the internal corpus.

Findings.

- Pre-built indexing is consistently faster: 1.7× at 70M and 1.8–2.6× at 700M. The larger gap at 700M with FLAT reflects disk spilling during on-the-fly construction (1.17 TB spill).
- For on-the-fly indexing, RaBitQ is 1.4× faster than FLAT at 700M (24.58 vs 35.2 min) because its smaller indices reduce spill computational overhead. This makes RaBitQ the preferred choice for ad-hoc on-the-fly queries at scale.
- On-the-fly peak memory (373–374 GB at 70M, 1.12 TB at 700M) reflects the `GROUP BY` aggregation buffer—not the final index size.

In practice, on-the-fly indexing is most valuable when the predicate changes per query—for example, e-commerce search with varying attribute filters, or RAG retrieval with per-request metadata constraints. Build times remain feasible even at large scale: 7 min at 70M and 25–35 min at 700M (Table 9). If the same predicate is reused more than 2–3 times, pre-building amortizes construction computational cost and is preferable.

8.9 Summary: Validating the Four Challenges

Table 15 maps each challenge to its solution and validation.

9 LESSONS LEARNED

We distill lessons—including what we got wrong—that generalize beyond vector search to other SQL extensions.

First, build bottom-up, ship top-down. We debated two strategies: (a) start with a declarative table function and evolve the algorithm underneath, or (b) start with composable primitives and add syntactic sugar later. We chose (b), because we first needed to validate that partition-based vector search would actually work at billion scale before committing to a product-level interface. Building on primitives forced us to discover the relational decomposition; once proven, the Level 2 table function was a thin optimizer rewrite over a well-understood algebraic plan. *Primitives validate the algebra; sugar makes it accessible.*

Second, find the natural partitioning key: cluster assignment is both semantically meaningful (similarity proxy) and operationally useful (distribution key). We initially expected the architecture to mirror Presto's existing geo-index, but cluster assignment turned out to be a fundamentally different kind of key: it proxies similarity rather than spatial containment, enabling hash-based co-location that geo-indices cannot exploit. Analogous dual-purpose keys likely exist in other domains.

Third, profile before optimizing. We assumed network transfer would dominate, but early evaluation revealed OOM failures during `GROUP BY` aggregation as the binding constraint—some clusters contained over 600× the average. Sub-clustering (Section 5.3) and RaBitQ compression (Section 5.1) reduced peak memory from 1.37 TB to 54 GB.

Fourth, look for algebraic fit before building new engines. The capabilities of Presto Vector Search— pre-filter indexing, zero-shuffle execution, composable results—follow from the relational mapping, not a specialized engine. Adding a new local index type requires a single builder function within the existing aggregate, without touching distributed execution or the optimizer. We originally planned to introduce new plan nodes and operators, as Presto's geo-search had required; instead, the existing UDF infrastructure—aggregate and scalar functions—proved sufficient, requiring no changes to the planner or execution engine. In retrospect, the hardest part was not engineering a new execution engine—it was recognizing that we did not need one.

Fifth, design for debuggability from day one. When recall is low, is the codec stale, *nprobe* too small, or the embeddings poor? Exposing intermediate results—cluster assignments, per-cluster scores, sub-index hit counts—as standard SQL columns lets users diagnose quality with `GROUP BY cluster_id` queries, using skills they already have.

Sixth, codec quality is the single largest recall lever. We started with the codec—number of centroids, training corpus, and training iterations—and found it dominated recall far more than the local index algorithms and sub-clustering we added later. Our architectural separation of codec from local index lets teams invest in codec quality independently of index construction.

Seventh, spill-to-disk is a safety net, not a design target. Presto's spill-to-disk mechanism enabled on-the-fly indexing at scale (Section 8.8), allowing 700M-vector index construction despite 1.12 TB peak memory. However, spilling increased wall-clock time by 1.8–2.6× compared to in-memory execution. For recurring

Table 15: Summary: validating the four challenges.

Challenge	Solution	Metric	Result
C1: Network	Codec-driven partitioning	Network transfer	1.4 TB → 200 MB (~7,000×)
C2: Memory	RaBitQ + sub-clustering	Peak memory	1.37 TB → 54 GB (25×)
C3: SQL Composability	Aggregate functions	Query patterns	WHERE, JOIN, GROUP BY, HAVING
C4: API Design	Two-level API	SQL complexity	68 → 12 lines, 5 → 1 subquery*
Decomposition	Algebraic equivalence	Computational overhead vs. FAISS	Identical recall; SQL computational overhead <1% at 1B
Pre-filter	On-the-fly indexing	Selectivity scaling	95% recall $\forall\sigma$; 9.2× vs post-filter at σ=1%
Load balance	Sub-clustering	Max/median ratio	7.2× → 1.0× (DEEP1B); 684× → 1.0× (700M); CPU skew 1.2×

*On a representative query, Level 1 primitives require 68 lines with 5 `QuerySpecification` nodes; Level 2 expresses the same search in 12 lines with one.

workloads, we recommend pre-built indices (which fit in memory) and reserve on-the-fly indexing for ad-hoc exploration where the flexibility of pre-filter indexing justifies the computational overhead.

Eighth, correctness debugging is harder than performance debugging. Performance regressions produce obvious signals; correctness regressions in approximate search are silent. A codec version mismatch can degrade recall without being noticed. Any system that trades exactness for performance needs continuous correctness monitoring, not just performance monitoring.

Ninth, the algebra's strengths are also its limits. The same relational fit that yields composability and zero-shuffle execution bounds where it applies: inheriting Presto's batch model puts online serving and streaming index maintenance out of scope (cf. VStream [18]); the streaming `GROUP BY` excludes local indices whose build buffers do not bound (HNSW_SQ OOMs at 1B, so HNSW serves only centroid routing); and the pre-filter guarantee assumes predicates independent of codec partitioning, as correlated filters concentrate rows in few clusters and degrade routing. *Match the workload to the algebra, not the reverse.*

10 RELATED WORK

Prior work spans three classes—standalone vector databases, single-node SQL extensions, and distributed SQL with pre-built indices—but all treat vector search as a single opaque operation (service API, index scan, or table function) that hides cluster assignment, index construction, and probe from the optimizer, foreclosing predicate pushdown, hash partitioning, and per-stage spill. Presto Vector Search instead exposes these three stages as standard relational operators, from which pre-filter indexing, SQL composability, and zero-shuffle distribution follow as direct consequences.

Vector databases. Standalone systems (Pinecone [3], Milvus [30], Weaviate [1], Qdrant [4], Chroma [5], Vearch [22]) provide scalable ANN retrieval but operate outside SQL, requiring data export and client-side orchestration.

SQL extensions. Single-node extensions—pgvector [2], PASE [34], SQL Server 2025 with DiskANN [26], MyScaleDB [7]—add ANN indices to relational engines but do not distribute.

Distributed SQL. Among distributed systems, two failure modes recur. Pre-built post-filter: SingleStore-V [10, 13], BigQuery [6], and CockroachDB [8] materialize indices ahead of time then filter, risking fewer than k results. Dual-engine: AnalyticDB-V [31] and Databricks [9] dispatch vector operations to a separate runtime, precluding optimizer rewriting; Snowflake [11] offers scalar similarity but no ANN search.

ANN algorithms and distributed search. We build on FAISS [21] for IVF indexing, RaBitQ [17] for quantization, and HNSW [25] for centroid lookup, following classical database architecture principles [20]. SIEVE [23] addresses filtered search via per-predicate index collections; our `WHERE`-based pre-filter achieves arbitrary filter combinations without precomputed indices. Aguerrebere et al. [12] validate locally-adaptive quantization for streaming settings, closely related to our streaming RaBitQ adaptation (Section 5.2). Cracking Vector Search Indexes [24] proposes lazy adaptive index construction. On distributed ANN, SPANN [15] partitions posting lists for billion-scale single-node search; HARMONY [32] and Xu et al. [33] address distributed index construction but operate outside SQL semantics; VStream [18] targets streaming distributed vector search—a setting our batch-oriented architecture does not address. Pan et al. [27] survey vector database techniques comprehensively. These keep vector search as a single opaque operation.

11 CONCLUSIONS

Every partition-based vector search system performs the same six logical steps—cluster assignment, index construction, index loading, query routing, local search, and global top-k merge. Monolithic libraries execute them in a single process; standalone vector databases distribute them via custom RPC. Presto Vector Search instead maps each step to a standard SQL operator—scalar functions, `GROUP BY` aggregates, equi-joins, and `TopN`—with the local index type as a pluggable parameter within the aggregate.

From this mapping, pre-filter indexing, zero-shuffle distribution, and composable results follow—they are not features we separately engineered. The same reasoning generalizes to any retrieval that decomposes into partition-by-key, per-partition work, and a global top-k merge: the engine's algebra, optimizer, and partitioning carry over. We did not build a vector search engine; we discovered that the SQL engine already is one.

ACKNOWLEDGMENTS

We are deeply grateful to Matthijs Douze for his guidance on the RaBitQ implementation in Faiss, and to Alibek Zhakubayev for his continued RaBitQ optimization; to Zoe Ye for the Level-2 function; and to our internal collaborators who shaped and stress-tested the system through their demanding requirements.

We also thank Masha Basmanova for design reviews, Sreeni Viswanadha and Feilong Liu for performance insights, James Gill, whose geo-search work inspired several design choices, and Pankaj Singh. Finally, we thank the anonymous reviewers, whose feedback improved this paper.

REFERENCES

[1] 2019. Weaviate. https://weaviate.io/. Accessed: 2026.

[2] 2021. pgvector: Open-Source Vector Similarity Search for PostgreSQL. https://github.com/pgvector/pgvector. Accessed: 2026.

[3] 2021. Pinecone. https://www.pinecone.io/. Accessed: 2026.

[4] 2021. Qdrant. https://qdrant.tech/. Accessed: 2026.

[5] 2022. Chroma. https://www.trychroma.com/. Accessed: 2026.

[6] 2023. Google BigQuery Vector Search. https://cloud.google.com/bigquery/docs/vector-search. Accessed: 2026.

[7] 2023. MyScaleDB: An Open-Source SQL Vector Database Built on ClickHouse. https://github.com/myscale/myscaledb. Accessed: 2026.

[8] 2024. CockroachDB Vector Indexing. https://www.cockroachlabs.com/docs/stable/vector-indexes. Accessed: 2026.

[9] 2024. Databricks Vector Search. https://docs.databricks.com/en/generative-ai/vector-search.html. Accessed: 2026.

[10] 2024. SingleStore Vector Indexing. https://docs.singlestore.com/db/v8.9/reference/sql-reference/vector-functions/vector-indexing/. Accessed: 2026.

[11] 2024. Snowflake Vector Functions. https://docs.snowflake.com/en/sql-reference/functions-vector. Accessed: 2026.

[12] Cecilia Aguerrebere, Ishwar Bhatt, Mark Hildebrand, Stratos Idreos, and Theodore Willke. 2024. Locally-Adaptive Quantization for Streaming Vector Search. *arXiv preprint arXiv:2402.02044* (2024).

[13] Akif Aziz, Junichi Tatemura, Hossein Yousefi, and Zhongxian Chen. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4131–4143.

[14] Artem Babenko and Victor Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2055–2063.

[15] 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. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[16] Amit Dutta, Pedro Pedreira, Xiaoxuan Meng, Masha Basmanova, Orri Erling, Sergey Pershin, Jialiang Tan, Ge Gao, Ke Wang, Zac Wen, Krishna Pai, Xiao Du, Natasha Sehgal, Heidi Han, Artem Selishchev, Sebastiano Peluso, Chandrashekhar Singh, Sreeni Viswanandha, Aditi Pandit, Deepak Majeti, Tim Meehan, Christian Zentgraf, Aakash Deep, Rohit Jain, Sridhar Anumandla, Kaushik Ravichandran, and Vivek Gaur. 2026. Presto to Prestissimo: A Velox-Powered Modernization Journey. *Proceedings of the VLDB Endowment* 19 (2026).

[17] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[18] Shenghao Gong, Haobo Sun, Ziquan Fang, Liu Liu, Lu Chen, and Yunjun Gao. 2025. VStream: A Distributed Streaming Vector Search System. *Proceedings of the VLDB Endowment* 18, 6 (2025), 1593–1606.

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

[20] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Foundations and Trends in Databases* 1, 2 (2007), 141–259.

[21] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 2 (2021), 535–547.

[22] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Lionel M. Ni, and Ning Wang. 2018. The Design and Implementation of a Real Time Visual Search System on JD.com. In *Proceedings of Middleware Industry*. 9–16. Vearch: https://github.com/vearch/vearch.

[23] Zhaoheng Li, Silu Huang, Yongjoo Park, and Jianjun Chen. 2025. SIEVE: Effective Filtered Vector Search with Collection of Indexes. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4723–4736.

[24] Vasilis Mageirakos, Bowen Wu, and Gustavo Alonso. 2025. Cracking Vector Search Indexes. *Proceedings of the VLDB Endowment* 18, 11 (2025), 3951–3964.

[25] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 42, 4 (2020), 824–836.

[26] Microsoft. 2025. Native Vector Search with DiskANN in SQL Server 2025. https://devblogs.microsoft.com/azure-sql/sql-server-2025-rc1-faster-diskann-and-fp16-support/. Accessed: 2026.

[27] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge. *arXiv preprint arXiv:2310.11703* (2024).

[28] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3372–3384.

[29] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegber-Eravsar, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1802–1813.

[30] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, and Jun Gu. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2614–2627.

[31] Zhuhe Wei, Yinggiao Wang, Jiacheng Feng, Zhanyang Zhang, Jin Zhang, Lei Wang, Kun Yu, Chang Lin, Guanchao Zhou, Shengjun Li, Junshuang Yang, and Hai Jin. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3152–3165.

[32] Qian Xu, Feng Zhang, Chengxi Li, Lei Cao, Zheng Chen, Jidong Zhai, and Xiaoyong Du. 2025. HARMONY: A Scalable Distributed Vector Database for High-Throughput Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2506.14707* (2025).

[33] Yuming Xu, Qianxi Zhang, Qi Chen, Baotong Lu, Menghao Li, Philip Adams, Mingqin Li, Zengzhong Li, Jing Liu, Cheng Li, and Fan Yang. 2025. Scalable Distributed Vector Search via Accuracy Preserving Index Construction. *arXiv preprint arXiv:2512.17264* (2025).

[34] Meng Yang, Haoyu Li, Jianguo Li, and Gang Chen. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2241–2254.