Performance Analysis of Two Vector Database Systems: Milvus and Weaviate Compared

Δημήτριος Μηνάγιας
National Technical University of Athens
Athens, Greece
el22813@mail.ntua.gr

Αλέξιος-Δημήτριος Λώλης
National Technical University of Athens
Athens, Greece
el21805@mail.ntua.gr

Βασίλειος Μουντζούρης
National Technical University of Athens
Athens, Greece
el21014@mail.ntua.gr

***Abstract*—As machine learning, recommendation engines, LLMs, and retrieval-augmented generation (RAG) systems increasingly rely on vector databases, the need for thorough, repeatable performance assessments has become a reality. In this paper, we conduct a detailed comparison between two widely used open-source vector database platforms: Milvus (v2.3) and Weaviate (v1.27). We test both systems on nine standard benchmark datasets containing between 60,000 and 5,000,000 vectors, with dimensions varying from 25 to 960, and using both Euclidean (L2) and cosine similarity measures. Our testing approach examines data ingestion speed, query response times (P50/P95), queries per second (QPS), recall accuracy (Recall@10 and Recall@100), storage footprint, CPU and memory usage, scaling under concurrent loads (1–16 clients), and how metadata filters affect performance. Both platforms are installed in standalone (single-node) setups via Docker, and Weaviate is also tested in a 3-node cluster arrangement to gauge the cost of distributed operation. To guarantee the fairest possible comparison, both databases employ the same HNSW index settings ($M = 16$, $efConstruction = 200$, $ef = 200$) and follow identical warm-up routines. Our findings show that Milvus consistently loads data 3–7$\times$ faster, while query latencies are similar at lower dimensions (both under 4ms P50). Weaviate scales better with concurrent requests for small-to-medium datasets, reaching 1,831 QPS with 8 clients compared to Milvus's 1,414 QPS. However, Milvus delivers more consistent performance at larger scales, showing much less latency variation on the 5M-vector deep-image-96 dataset. Operating in cluster mode adds 20–30% extra latency at these data sizes, with Weaviate's concurrent throughput dropping by as much as 54% compared to standalone operation. To facilitate reproducibility, the source code and benchmarking suite are available at https://github.com/minageus/vector-db-benchmark.**

***Index Terms*—Vector Databases, Milvus, Weaviate, Cosine Similarity, HNSW.**

I. Introduction

A. Motivation

Vector databases have become essential infrastructure for contemporary AI applications. While traditional relational databases excel at exact matches on structured data, vector databases specialize in approximate nearest neighbor (ANN) searches across high-dimensional embedding spaces. These searches underpin semantic search, recommendation systems, image retrieval, anomaly detection, and the now-common retrieval-augmented generation (RAG) approach for large language models [4].

The ecosystem of commercial and open-source vector databases has expanded quickly, with options like Milvus, Weaviate, Qdrant, Pinecone, and Chroma vying for users so picking the proper database according to the user's needs is the main challenge we try to tackle.

B. Our Contributions

This paper tackles these gaps with several contributions:

1) **Fair benchmarking framework:** We built a Python-based framework that enforces uniform HNSW index parameters, warm-up steps, and query vectors for both databases.
2) **Broad dataset evaluation:** We assess 9 standard ANN benchmark datasets spanning dimensions from 25 to 960 and sizes from 60K to 5M vectors.
3) **Multi-faceted assessment:** We measure ingestion speed, latency, throughput, recall, filter effects, storage, resource consumption, and concurrent scaling.
4) **Cluster deployment analysis:** We set up Weaviate in a 3-node cluster to measure the overhead of distributed operation.
5) **Reproducible approach:** All code, configurations, and raw results are publicly accessible for verification and reuse.

C. Document Structure

Section II reviews vector indexing fundamentals and the two evaluated systems. Section III explains our testing methodology, datasets, and metrics. Section IV presents experimental outcomes. Section V examines cluster versus standalone performance. Section VI interprets findings and practical implications. Section VII offers conclusions and recommendations.

II. Background and Related Work

A. Approximate Nearest Neighbor Search

The core function of a vector database is k-nearest neighbor (k-NN) search: given a query vector $q \in \mathbb{R}^d$ and a collection $X = \{x_1, \ldots, x_n\}$, identify the k vectors in X most similar to q according to a chosen distance measure. Exact k-NN search demands $O(nd)$ time, which becomes impractical for large n and d. Approximate nearest neighbor (ANN) methods accept a slight accuracy reduction (recall) for substantial speed gains.

1) HNSW Algorithm: Among ANN techniques, the **Hierarchical Navigable Small World (HNSW)** graph [1] has become the leading index structure in modern vector databases due to its favorable recall–latency balance. HNSW builds a multi-layer proximity graph where upper layers have long-range links for quick global navigation, and lower layers have short-range links for precise local exploration. The search algorithm begins at an entry point in the highest layer, moving downward through layers and refining candidate sets at each step.

Key HNSW parameters include:

- **M** (maximum connections per node): Governs graph density. Higher values boost recall but need more memory and construction time. Typical values are 8 to 64.
- **efConstruction**: Influences index quality. Higher values produce more accurate graphs but slow construction. Common values are 100–400.
- **ef** (search beam width): Controls search accuracy during queries. Higher values improve recall at the expense of latency. This is the main parameter for adjusting the accuracy–speed trade-off.

2) Distance Metrics: The two most frequent distance metrics for vector similarity search are:

- **Euclidean distance (L2):** $d(x, q) = \sqrt{\sum_{i=1}^{d}(x_i - q_i)^2}$. Used for image descriptors, scientific data, and spatial embeddings.
- **Cosine similarity:** $\cos(x, q) = \frac{x \cdot q}{\|x\|\|q\|}$. Used for text embeddings, NLP features, and normalized data. In practice, cosine search on L2-normalized vectors equals inner product (IP) search.

B. Milvus

Milvus [3] is an open-source vector database initially created by Zilliz, focusing on scalable similarity search. Notable features include:

- **Decoupled storage:** Uses MinIO for object storage and etcd for metadata, supporting a cloud-native design where storage and compute scale separately.
- **Index options:** HNSW, IVF_FLAT, IVF_SQ8, IVF_PQ, DiskANN, and GPU-accelerated versions.
- **Deployment:** Standalone (single-node, 3-container stack: Milvus + etcd + MinIO) and distributed (Kubernetes-based with separate query, data, and index nodes).
- **Query capabilities:** Vector similarity search combined with attribute filtering via boolean expressions on typed scalar fields (integers, strings, floats).
- **Batch-focused loading:** Data enters in batches; HNSW index construction happens after insertion, allowing faster bulk ingestion.

This study uses Milvus v2.3 in standalone Docker mode.

C. Weaviate

Weaviate is an open-source, AI-native vector database developed by SeMI Technologies. Key aspects include:

- **Integrated storage:** Custom LSM-tree storage engine with built-in HNSW index. Unlike Milvus, Weaviate keeps all data in a single process without external dependencies.
- **Index options:** HNSW with optional product quantization compression, flat index for small datasets.
- **Deployment:** Single-node (one Docker container) and multi-node cluster with RAFT-based consensus and gossip protocol for node discovery.
- **Query capabilities:** Vector search via `near_vector` queries combined with rich filtering on properties through a GraphQL-like API.
- **Online index building:** The HNSW graph is constructed gradually during insertion, removing a separate index build step but increasing insert latency.

We use Weaviate v1.27.0 in both standalone and cluster (3-node) setups.

D. Prior Work

ANN-Benchmarks [2] is the most established benchmark suite for ANN algorithms, offering standardized datasets and evaluation metrics. It concentrates on algorithmic comparisons (e.g., HNSW vs. IVF vs. LSH) rather than full database system evaluation. Our dataset selection comes from this suite.

VectorDBBench (Zilliz, 2023) is a benchmarking tool from the Milvus team that focuses on end-to-end database assessment including loading and querying. However, being developed by the Milvus team introduces potential bias concerns.

ann-filtering-benchmark-datasets (Qdrant) [6] extends ANN benchmarks with filtered search situations, inspiring our filter evaluation methodology.

III. Methodology

A. Benchmarking Framework Design

We created a comprehensive Python-based benchmarking framework (`run_paper_benchmark.py`) that manages fair, repeatable tests across both databases. The framework has four main components:

- **Data Loaders** (`data/loaders/`): Database-specific insertion logic. The Milvus loader creates typed collections with fields for ID (INT64), embedding (FLOAT_VECTOR), category (INT64), and price (FLOAT). Vectors are inserted in batches of 1,000. The Weaviate loader creates equivalent schemas and uses adaptive batch sizes (50–200 vectors) based on dimensionality to prevent timeouts.
- **Query Executors** (`queries/`): Milvus uses the `collection.search()` API with expression-based filtering; Weaviate uses `near_vector` queries with the `Filter` API. Both return ranked results with distances.
- **Measurement Utilities** (`utils/`): A resource monitor (using `psutil`) samples CPU, memory, and disk I/O every 0.5 seconds. A concurrent tester launches 1–16 threads issuing queries simultaneously. A recall calculator computes ground-truth neighbors via brute-force and measures overlap with ANN results.
- **Dataset Manager** (`data/loaders/real_dataset_loader.py`): Handles download, caching, format conversion (fvecs, HDF5), normalization, and metadata generation for all benchmark datasets.

B. Ensuring Fair Comparisons

To guarantee equitable testing, we enforce these rules:

1) **Same index parameters:** Both databases use HNSW with $M = 16$, $efConstruction = 200$, and $ef = 200$ for all tests.
2) **Same query vectors:** Identical randomly selected or ground-truth query vectors are used for both systems in each test run.
3) **Same metadata:** Both systems store identical synthetic metadata fields (category: integer 0–9, price: float \$10–\$1000) for filter assessment.
4) **Warm-up routine:** 100 warm-up queries run before measurements start, stabilizing caches and JIT compilation.
5) **Statistical care:** Each benchmark repeats 3 times, reporting mean $\pm$ standard deviation for all metrics.
6) **Metric adjustment:** For cosine similarity datasets, vectors are L2-normalized before insertion, and inner product serves as the search metric (mathematically identical to cosine on normalized vectors).

C. Test Environment

All experiments run on one physical machine using Docker Compose for containerized deployment:

- **Milvus stack:** Milvus standalone (v2.3) + etcd (configuration/metadata) + MinIO (object storage), each allowed up to 8 GB RAM. The three containers communicate over a Docker bridge network.
- **Weaviate standalone:** A single Weaviate v1.27.0 container with 8 GB RAM limit, anonymous access enabled, and no vectorizer module (raw vectors supplied directly).
- **Weaviate cluster:** Three Weaviate v1.27.0 nodes, each with 4 GB RAM limit, communicating via gossip protocol (port 7946) with RAFT consensus for metadata coordination. Data auto-shards across nodes with replication factor 1.

D. Datasets

We pick 9 datasets from the ANN-Benchmarks collection [2], covering diverse dimensionalities, sizes, and distance metrics (Table I).

Table I
BENCHMARK DATASETS

Dataset	**Vectors**	**Dim**	**Metric**
Fashion-MNIST	60,000	784	L2
NYTimes-256	290,000	256	Cosine
GloVe-25	1,183,514	25	Cosine
GloVe-100	1,183,514	100	Cosine
GloVe-200	1,183,514	200	Cosine
SIFT-1M	1,000,000	128	L2
GIST-1M	1,000,000	960	L2
Deep-Image-96 (2M)	2,000,000	96	L2
Deep-Image-96 (5M)	5,000,000	96	L2

This selection enables analysis along three axes: **dimensionality** (25D to 960D using GloVe variants and GIST), **scale** (60K to 5M using Fashion-MNIST through Deep-Image), and **distance metric** (L2 vs. cosine). The datasets come from varied domains: image descriptors (SIFT, GIST, Deep-Image, Fashion-MNIST), word embeddings (GloVe), and text features (NYTimes).

E. Testing Procedure

For each dataset, the benchmark follows this sequence:

1) **Setup:** Clear existing data, create collection/schema with HNSW parameters.
2) **Data Loading:** Insert all vectors in batches, tracking CPU, memory, and disk I/O throughout.
3) **Index Verification:** Wait for index construction to finish, confirm vector count matches expectations.
4) **Warm-up:** Execute 100 random queries to stabilize caches.
5) **Main Benchmark** ($\times 3$ runs): For each of 4 query types (K=10, K=100, K=10 with filter, K=100 with filter), run 1,000 queries and record per-query latency.
6) **Recall Calculation:** Compare ANN results against ground truth or brute-force computation.
7) **Concurrent Load Test:** Increase from 1 to 16 concurrent clients, each sending queries continuously for 10 seconds per concurrency level.
8) **Storage Analysis:** Measure on-disk footprint for both databases.
9) **Report Generation:** Aggregate across runs, compute P50/P95/P99 latencies, QPS, and standard deviations. Produce JSON, CSV, and TXT reports.

F. Metrics Tracked

We track these metrics:

- **Load time** (seconds): Wall-clock time from first insertion to index readiness.
- **P50/P95 latency** (ms): Median and 95th percentile query response times.
- **QPS**: Queries per second (throughput).
- **Recall@K**: Fraction of true K-nearest neighbors returned.
- **Resource usage:** Average/peak CPU (%), peak memory (MB), disk I/O (MB).
- **Storage:** Total on-disk size (MB) and compression ratio.

IV. RESULTS

A. Data Loading Performance

Table II and Figure 1 show data loading performance. Milvus consistently loads data **3.6$\times$ to 8.5$\times$ faster** than Weaviate across all 9 datasets. The speed gap widens with dataset size: from 4.1$\times$ for 60K vectors (Fashion-MNIST) to 7.4$\times$ for 5M vectors (Deep-Image-96).

This performance difference arises from architectural choices. Milvus uses a *batch-oriented* loading approach: vectors enter segments via MinIO, and the HNSW index builds as a separate post-insertion step. Weaviate, conversely, constructs the HNSW graph *incrementally* during each insert operation, meaning every vector insertion triggers graph neighbor searches and edge updates.

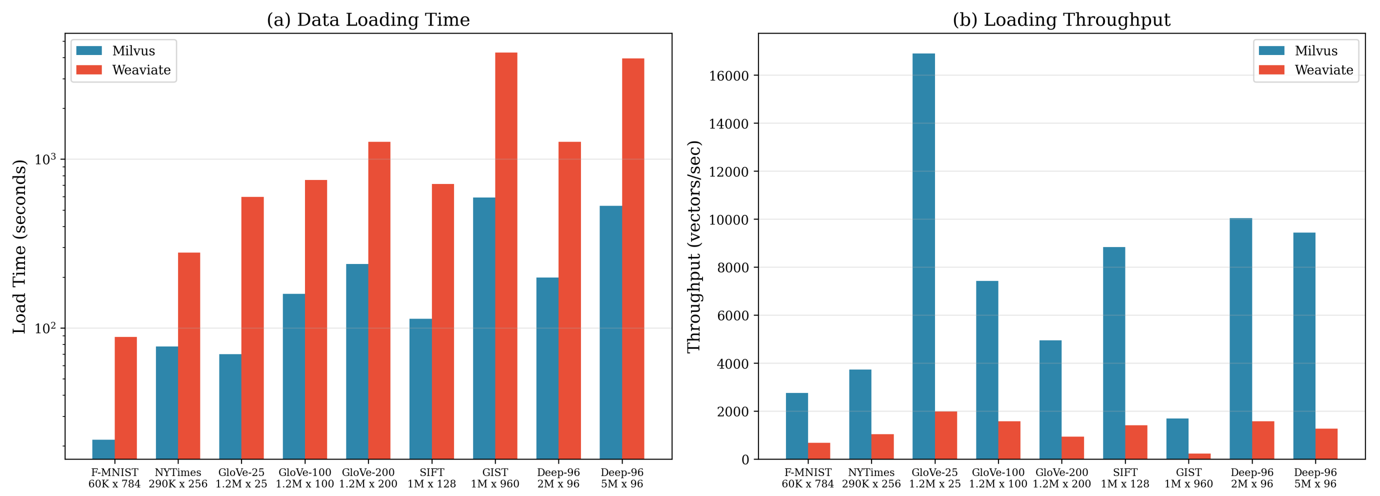

Figure 1. Data loading performance: (a) Total loading time (log scale) and (b) Loading throughput (vectors per second) across all datasets. Milvus's deferred index construction yields consistently higher ingestion throughput.

Table II
DATA LOADING TIME (SECONDS)

Dataset	Milvus	Weaviate	Ratio
Fashion-MNIST	21.8	88.4	4.1×
NYTimes-256	77.7	279.1	3.6×
GloVe-25	70.1	596.5	8.5×
GloVe-100	159.4	751.4	4.7×
GloVe-200	239.4	1,269.4	5.3×
SIFT-1M	113.2	713.3	6.3×
GIST-1M	591.9	4,278.3	7.2×
Deep-96 (2M)	199.4	1,267.3	6.4×
Deep-96 (5M)	530.1	3,947.9	7.4×

Table III
CPU UTILIZATION DURING LOADING (%)

	Milvus		Weaviate	
Dataset	Avg	Peak	Avg	Peak
F-MNIST	21.1	63.7	15.3	39.5
NYTimes	13.0	60.8	37.4	80.1
GloVe-25	7.8	66.6	33.0	53.5
GloVe-100	11.2	70.3	54.0	89.4
GloVe-200	10.9	87.9	42.0	71.2
SIFT-1M	13.9	61.0	43.6	73.5
GIST-1M	18.6	74.0	29.7	83.3
Deep (2M)	11.0	66.7	38.7	64.4
Deep (5M)	11.1	73.6	38.3	71.0

B. Resource Usage During Loading

Figure 2 displays CPU and memory consumption during data loading. The resource patterns reveal clear architectural distinctions.

CPU utilization: Weaviate consistently uses 2–5× more average CPU during loading than Milvus (30–54% vs. 8–21%). This reflects the cost of Weaviate's online HNSW construction: each inserted vector requires graph traversal to find neighbors and edge updates to maintain small-world properties. Milvus postpones this work to a separate construction phase after all data is inserted.

Peak memory: Memory usage during loading is generally similar between systems, with one exception: GIST-1M (960D), where Milvus uses 3,801 MB versus Weaviate's 1,328 MB. The high memory usage for Milvus on GIST likely stems from loading the full vector dataset into memory for efficient batch index construction on this high-dimensional data.

Table III provides detailed resource breakdown.

C. Query Latency

Figure 3 and Table IV present query latency measurements for unfiltered k-NN search at K=10. Several patterns emerge across the dataset spectrum.

Small datasets: For Fashion-MNIST (60K vectors, 784D), Weaviate achieves lower P50 latency of 2.41 ms versus Milvus's 3.38 ms. This advantage comes from Weaviate's simpler single-process architecture that avoids network overhead from Milvus's MinIO and etcd communication.

Medium datasets: For datasets around 1M vectors at moderate dimensions (GloVe-25, SIFT-1M, NYTimes-256), both systems deliver comparable sub-4 ms P50 latencies. Differences stay within experimental variance.

High-dimensional data: At 960 dimensions (GIST-1M), Milvus substantially outperforms Weaviate: 10.88 ms vs. 46.74 ms P50 at K=10, a **4.3× speed advantage**. The gap widens further at K=100 (9.43 ms vs. 49.47 ms). This suggests Milvus employs more efficient SIMD-vectorized distance computations for high-dimensional spaces.

Large-scale data: On the 5M-vector Deep-Image-96 dataset, Milvus achieves 6.00 ms P50 versus Weaviate's 37.82 ms. Notably, both systems show high variance at this scale (Milvus std: 1.71 ms, Weaviate std: 40.49 ms), indicating intermittent disk I/O as the dataset exceeds main memory.

Latency stability: Across all datasets, Milvus consistently

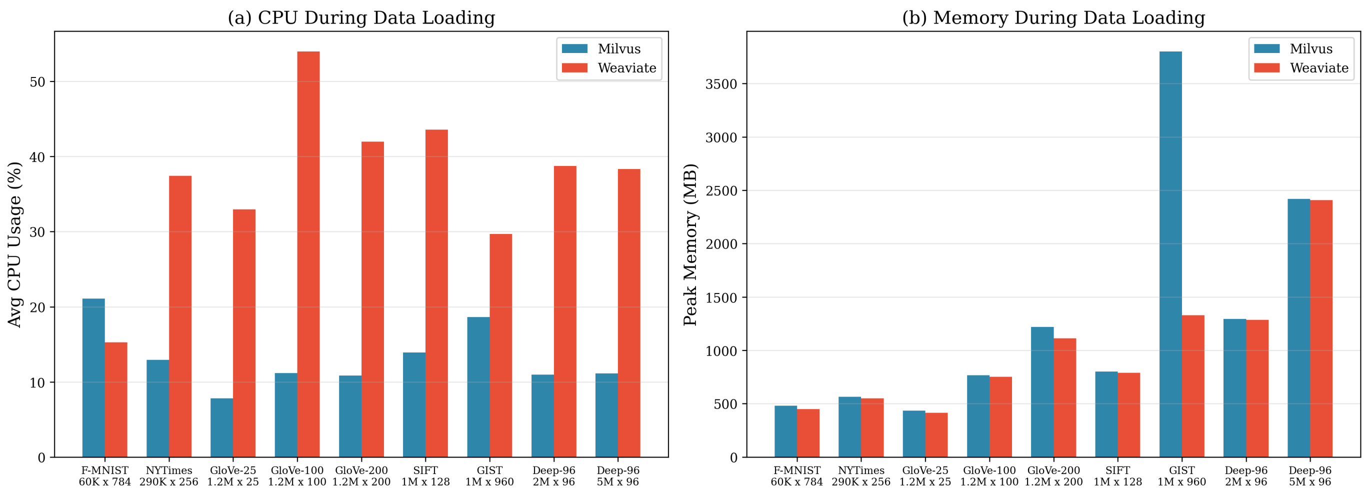

Figure 2. Resource usage during data loading: (a) Average CPU utilization and (b) Peak memory consumption. Weaviate's online index construction drives higher sustained CPU usage, while Milvus shows a memory spike on the 960D GIST dataset.

shows *lower standard deviation* across runs, indicating more predictable and reproducible performance.

Table V presents corresponding results for K=100. At K=100, Milvus shows a more consistent advantage across datasets, suggesting better handling of larger result set retrieval and ranking.

D. Throughput (QPS)

Table VI presents single-client queries per second for K=10 and K=100.

Both systems deliver strong throughput (200–380 QPS) on datasets up to $\sim$1M vectors at moderate dimensions. Weaviate leads slightly for K=10 on smaller datasets, while Milvus consistently dominates at K=100. Performance drops sharply at 5M vectors (63–73 QPS at K=10), indicating data no longer fits entirely in memory and disk I/O becomes the bottleneck.

E. Impact of Metadata Filtering

Figure 4 and Table VII compare query performance with and without category-based metadata filters ($\sim$30% selectivity: picking 3 of 10 categories).

Filtering overhead is generally modest (1–11%) for datasets fitting in memory. A striking anomaly occurs on **GIST-1M**: filtering *lowers* latency for both systems—Milvus by 14% and Weaviate by 83%. This happens because at 960 dimensions, each distance computation is costly, and the metadata filter prunes a large part of the HNSW graph, reducing distance calculations needed. This effect is especially dramatic for Weaviate, which appears to apply pre-filtering that eliminates nodes before traversal.

At 5M vectors, Milvus shows a notable 54% filter overhead, indicating filtered search on out-of-memory datasets is a performance stress point. The P95 tail latency overhead is even more pronounced: on SIFT-1M, Milvus's P95 jumps from 4.35 ms to 13.25 ms (+205%), suggesting some filtered queries require much deeper graph traversal to meet result cardinality.

F. Recall Accuracy

Table VIII and Figure 5 show Recall@K metrics. Recall measures the fraction of true K-nearest neighbors returned by approximate search.

For standard datasets (Fashion-MNIST, GloVe-25, SIFT-1M), both systems achieve excellent recall (> 0.99 for Recall@10), confirming HNSW with $ef = 200$ delivers near-exact search results on typical workloads.

The most notable gap appears on **GIST-1M** (960D): Milvus reaches 0.987 vs. Weaviate's 0.905 for Recall@10—an 8.2 percentage point difference. This suggests differences in how the two systems build or traverse the HNSW graph for very high-dimensional data.

Deep-Image-96 recall values are low (0.19–0.50) for both systems. This is *not* a system flaw but a limitation of our subset evaluation: ground truth queries were computed against the full 10M-vector dataset, but only 2M or 5M vectors were loaded, so many true neighbors are absent from the database.

G. Storage Efficiency

Table IX reports on-disk storage usage for both systems.

Milvus shows high baseline storage overhead of about 29 GB that stays nearly constant regardless of dataset size. This comes from MinIO's object storage metadata, etcd's state files, and accumulated data from sequential benchmark runs on the same Docker volume without full cleanup. Actual vector data compression is efficient—compression ratios range from 0.4% to 12% of raw data size—but fixed overhead dominates for smaller datasets.

Weaviate's storage scales linearly with dataset size and shows reasonable overhead for the HNSW index structure. For the largest dataset (Deep-96, 5M), Weaviate uses 10.6 GB versus 1.8 GB raw data, a 5.8$\times$ expansion that includes graph edges, metadata, and LSM-tree overhead.

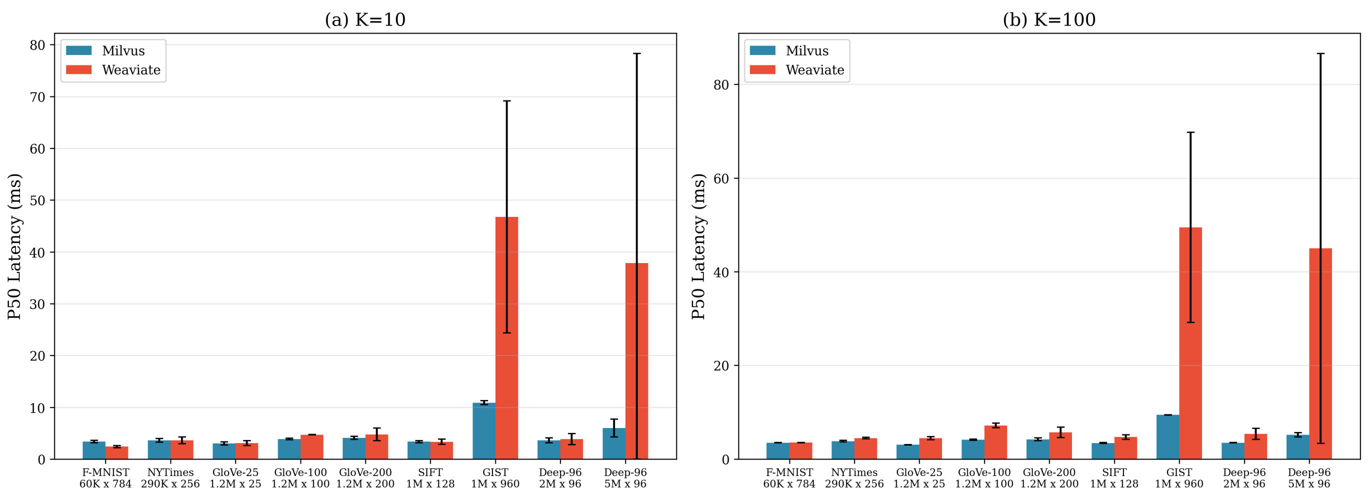

Figure 3. P50 Latency for (a) K=10 and (b) K=100 across all datasets, with error bars showing standard deviation across 3 runs.

Table IV
QUERY LATENCY (MS) – UNFILTERED, K=10 (MEAN ± STD OVER 3 RUNS)

Dataset	Milvus P50	Weaviate P50	Milvus P95	Weaviate P95
Fashion-MNIST	3.38 ± 0.22	$\mathbf{2.41 \pm 0.20}$	4.02 ± 0.34	$\mathbf{3.48 \pm 0.47}$
NYTimes-256	3.59 ± 0.36	3.60 ± 0.64	$\mathbf{4.80 \pm 0.66}$	4.72 ± 1.07
GloVe-25	3.00 ± 0.31	3.09 ± 0.49	$\mathbf{3.85 \pm 0.69}$	4.28 ± 0.98
GloVe-100	$\mathbf{3.87 \pm 0.16}$	4.70 ± 0.04	5.99 ± 1.67	$\mathbf{5.74 \pm 0.21}$
GloVe-200	$\mathbf{4.08 \pm 0.28}$	4.77 ± 1.24	$\mathbf{5.69 \pm 1.18}$	9.66 ± 6.11
SIFT-1M	3.37 ± 0.21	$\mathbf{3.32 \pm 0.50}$	4.35 ± 0.34	$\mathbf{4.31 \pm 0.69}$
GIST-1M	$\mathbf{10.88 \pm 0.41}$	46.74 ± 22.39	$\mathbf{109.78 \pm 44.49}$	168.04 ± 92.07
Deep-96 (2M)	$\mathbf{3.61 \pm 0.46}$	3.86 ± 1.08	6.85 ± 3.43	$\mathbf{5.56 \pm 2.17}$
Deep-96 (5M)	$\mathbf{6.00 \pm 1.71}$	37.82 ± 40.49	$\mathbf{107.25 \pm 105.49}$	263.96 ± 330.27

Table V
QUERY LATENCY (MS) – UNFILTERED, K=100 (MEAN ± STD OVER 3 RUNS)

Dataset	Milvus P50	Weaviate P50	Milvus P95	Weaviate P95
Fashion-MNIST	$\mathbf{3.50 \pm 0.03}$	3.52 ± 0.04	$\mathbf{4.28 \pm 0.11}$	4.30 ± 0.17
NYTimes-256	$\mathbf{3.83 \pm 0.15}$	4.49 ± 0.15	$\mathbf{5.11 \pm 0.09}$	5.75 ± 0.63
GloVe-25	$\mathbf{3.05 \pm 0.01}$	4.45 ± 0.32	$\mathbf{3.62 \pm 0.01}$	5.79 ± 0.52
GloVe-100	$\mathbf{4.13 \pm 0.11}$	7.17 ± 0.49	$\mathbf{5.69 \pm 0.71}$	9.77 ± 1.80
GloVe-200	$\mathbf{4.22 \pm 0.32}$	5.72 ± 1.13	$\mathbf{5.09 \pm 0.68}$	8.18 ± 3.11
SIFT-1M	$\mathbf{3.43 \pm 0.10}$	4.72 ± 0.49	$\mathbf{4.27 \pm 0.20}$	6.64 ± 1.64
GIST-1M	$\mathbf{9.43 \pm 0.05}$	49.47 ± 20.32	$\mathbf{13.44 \pm 1.76}$	106.30 ± 50.24
Deep-96 (2M)	$\mathbf{3.50 \pm 0.03}$	5.36 ± 1.20	$\mathbf{4.50 \pm 0.16}$	7.77 ± 3.17
Deep-96 (5M)	$\mathbf{5.19 \pm 0.46}$	44.97 ± 41.62	$\mathbf{144.45 \pm 188.46}$	395.05 ± 517.82

Table VI
SINGLE-CLIENT QPS

	K=10		K=100	
Dataset	**Mil.**	**Weav.**	**Mil.**	**Weav.**
F-MNIST	287	**380**	270	**265**
NYTimes	265	**273**	**245**	212
GloVe-25	314	**317**	**316**	214
GloVe-100	**238**	206	**218**	130
GloVe-200	**225**	208	**231**	172
SIFT-1M	287	**295**	**278**	201
GIST-1M	**37**	25	**100**	17
Deep (2M)	257	**263**	**267**	183
Deep (5M)	**73**	63	**118**	35

For practitioners, this means Weaviate is much more storage-efficient for datasets under ∼3 GB raw size, while Milvus's fixed overhead becomes amortized at larger scales.

H. Concurrent Scaling

Figure 6 and Table X show QPS under increasing concurrent load from 1 to 16 simultaneous clients.

Weaviate scales better at moderate concurrency (4–8 clients) for small-to-medium datasets, often surpassing Milvus by 1.3–1.6× at 8 clients. This reflects Weaviate's simpler single-process architecture: queries run within the same process as the index, avoiding network serialization overhead that Milvus incurs communicating with etcd and MinIO.

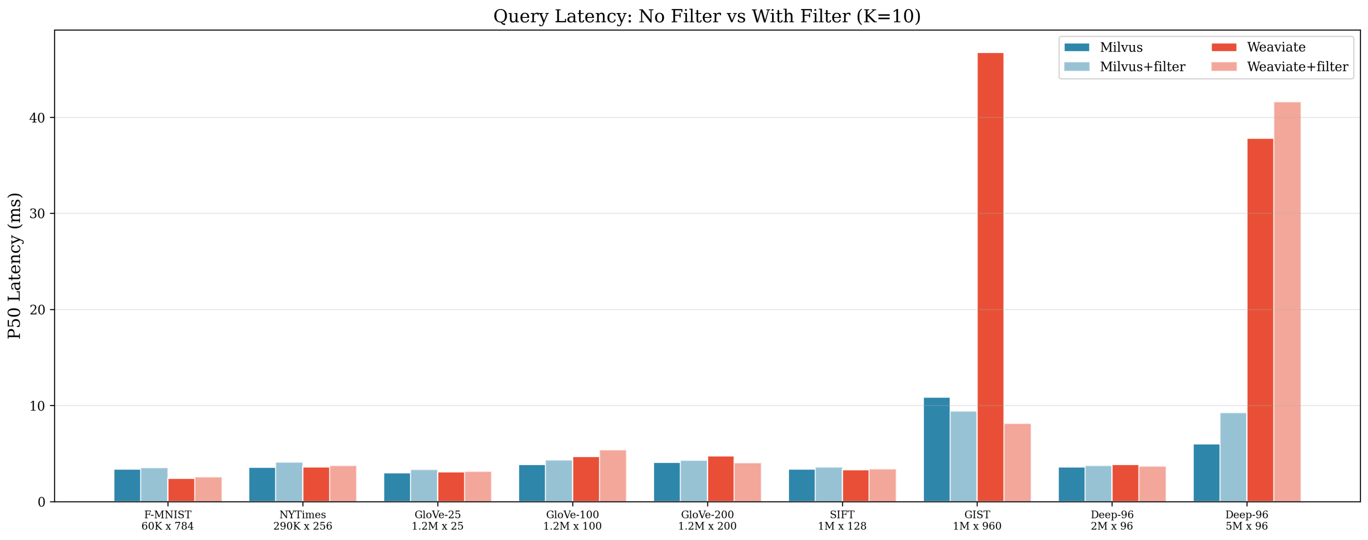

Figure 4. Query latency with and without metadata filters (K=10). Filtering adds modest overhead for in-memory datasets but can actually reduce latency on high-dimensional data (GIST-1M) by pruning the search space.

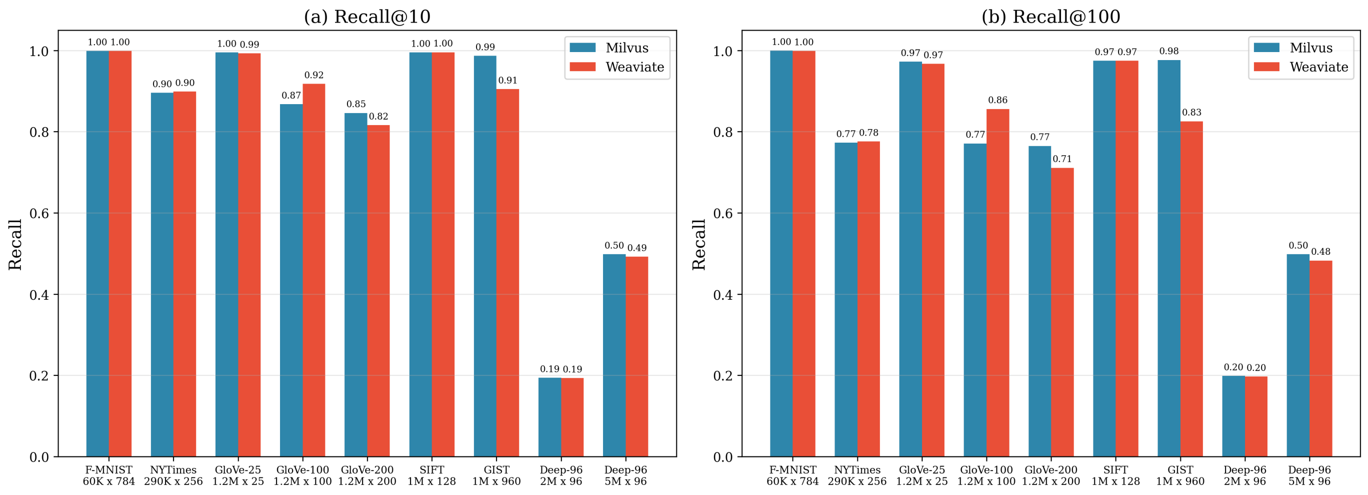

Figure 5. (a) Recall@10 and (b) Recall@100. Both systems achieve near-perfect recall on standard datasets (Fashion-MNIST, SIFT, GloVe-25). Low recall on Deep-Image subsets reflects missing ground-truth neighbors, not system deficiency.

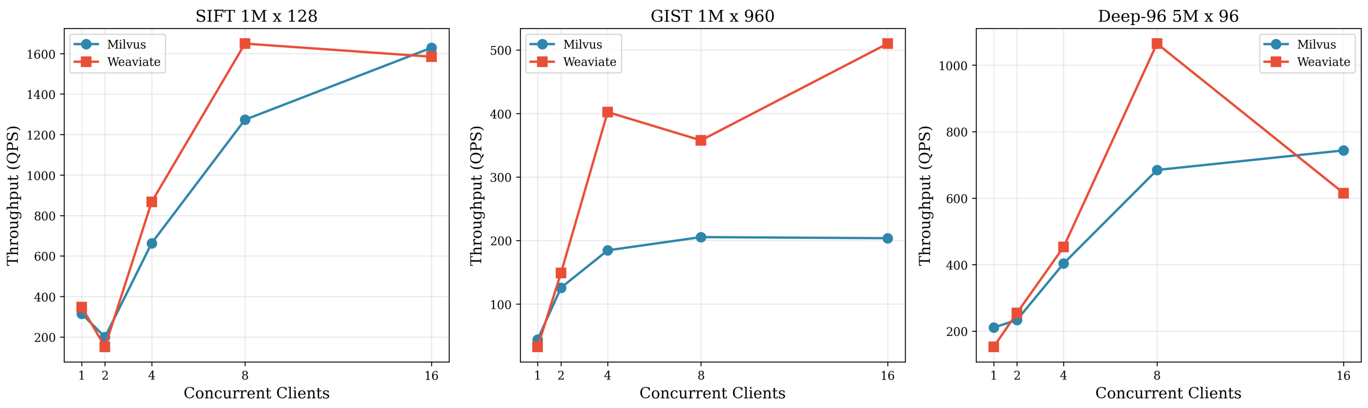

Figure 6. Throughput (QPS) scaling with concurrent clients for representative datasets. Both systems show near-linear scaling from 1 to 8 clients, with Weaviate often achieving higher peak QPS.

Table VII
FILTER IMPACT ON P50 LATENCY (MS), K=10

Dataset	No Filt.	Filter	Overhead
Milvus			
F-MNIST	3.38	3.50	+3.6%
GloVe-25	3.00	3.32	+10.7%
SIFT-1M	3.37	3.59	+6.5%
GIST-1M	10.88	9.39	−13.7%
Deep (5M)	6.00	9.25	+54.2%
Weaviate			
F-MNIST	2.41	2.56	+6.2%
GloVe-25	3.09	3.14	+1.6%
SIFT-1M	3.32	3.38	+1.8%
GIST-1M	46.74	8.12	−82.6%
Deep (5M)	37.82	41.60	+10.0%

Table VIII
RECALL ACCURACY

	Recall@10		Recall@100	
Dataset	Mil.	Weav.	Mil.	Weav.
F-MNIST	**.999**	.999	**1.000**	.999
NYTimes	.896	**.899**	.773	**.776**
GloVe-25	**.995**	.993	**.973**	.968
GloVe-100	.868	**.918**	.771	**.856**
GloVe-200	**.846**	.816	**.765**	.711
SIFT-1M	.995	**.996**	.975	.975
GIST-1M	**.987**	.905	**.977**	.826
Deep (2M)	.194	.193	.199	.197
Deep (5M)	.498	.492	.499	.483

Table IX
STORAGE USAGE (MB)

Dataset	Raw	Milvus	Weaviate
F-MNIST	179	29,709	245
NYTimes	283	30,015	735
GloVe-25	113	29,103	2,214
GloVe-200	903	29,743	2,801
SIFT-1M	488	29,366	1,992
GIST-1M	3,662	30,674	5,148
Deep (2M)	732	31,206	3,898
Deep (5M)	1,831	38,313	10,580

Table X
CONCURRENT QPS SCALING

Dataset	DB	1	4	8	16
F-MNIST	Mil.	295	630	1,053	1,257
	Weav.	**462**	**1,249**	**1,827**	**1,593**
GloVe-25	Mil.	346	820	1,414	**1,747**
	Weav.	**359**	**971**	**1,831**	1,747
SIFT-1M	Mil.	315	663	1,273	**1,628**
	Weav.	**348**	**867**	**1,649**	1,584
GIST-1M	Mil.	44	184	205	204
	Weav.	32	**402**	**358**	**510**
Deep (5M)	Mil.	**211**	404	685	**744**
	Weav.	153	**453**	**1,065**	615

Both systems show a QPS dip at 2 clients in most benchmarks. This is a known artifact of thread scheduling: with only 2 concurrent clients, one thread often waits on an OS context switch while the other executes, yielding lower aggregate throughput than even a single client. The effect resolves at 4+ clients where the CPU pipeline stays saturated.

Milvus shows more stable high-concurrency behavior: Its QPS generally continues rising or plateaus at 16 clients, while Weaviate sometimes regresses from 8 to 16 clients (e.g., Deep-96 5M: 1,065 $\rightarrow$ 615 QPS, a 42% drop). This regression may indicate lock contention within Weaviate's single-process architecture at high thread counts.

I. Dimensionality Impact

By comparing datasets with similar vector counts ($\sim$1M) but different dimensions, we isolate dimensionality's effect on query performance. Figure 7 and Table XI present these results.

Table XI
DIMENSIONALITY IMPACT ($\sim$1M VECTORS, K=10)

	P50 (ms)		QPS	
Dim	Mil.	Weav.	Mil.	Weav.
25	3.00	3.09	314	317
100	3.87	4.70	238	206
128	3.37	3.32	287	295
200	4.08	4.77	225	208
960	**10.88**	46.74	**37**	25

Latency increases roughly linearly with dimension up to $\sim$200D, then accelerates dramatically at 960D. The **performance gap between Milvus and Weaviate widens with dimensionality**: at 25D they are essentially tied (3.00 vs. 3.09 ms), while at 960D Milvus is 4.3$\times$ faster (10.88 vs. 46.74 ms). This divergence likely stems from differences in SIMD vectorization strategies: both systems use the same HNSW graph structure, so the performance gap must come from how each computes distance functions on high-dimensional vectors. Milvus, built on FAISS kernels [5], benefits from highly optimized distance computation routines that exploit AVX2/AVX-512 instruction sets.

J. Scale Impact

The Deep-Image-96 dataset, tested at both 2M and 5M vectors (same 96 dimensions), reveals how each system responds to growing dataset size. Figure 8 presents these results.

From 2M to 5M vectors, Milvus latency rises by 1.7$\times$ (3.61 $\rightarrow$ 6.00 ms) while Weaviate latency rises by **9.8**$\times$ (3.86 $\rightarrow$ 37.82 ms). This sharp divergence suggests Milvus handles the *transition from in-memory to disk-spilling* more smoothly. Milvus's MinIO-backed storage is designed for this scenario—data segments are memory-mapped with efficient read-ahead, while Weaviate's LSM-tree storage may generate more random I/O patterns when the working set exceeds available memory.

The QPS impact is equally dramatic: both systems drop from $\sim$260 QPS to 63–73 QPS when scaling from 2M to

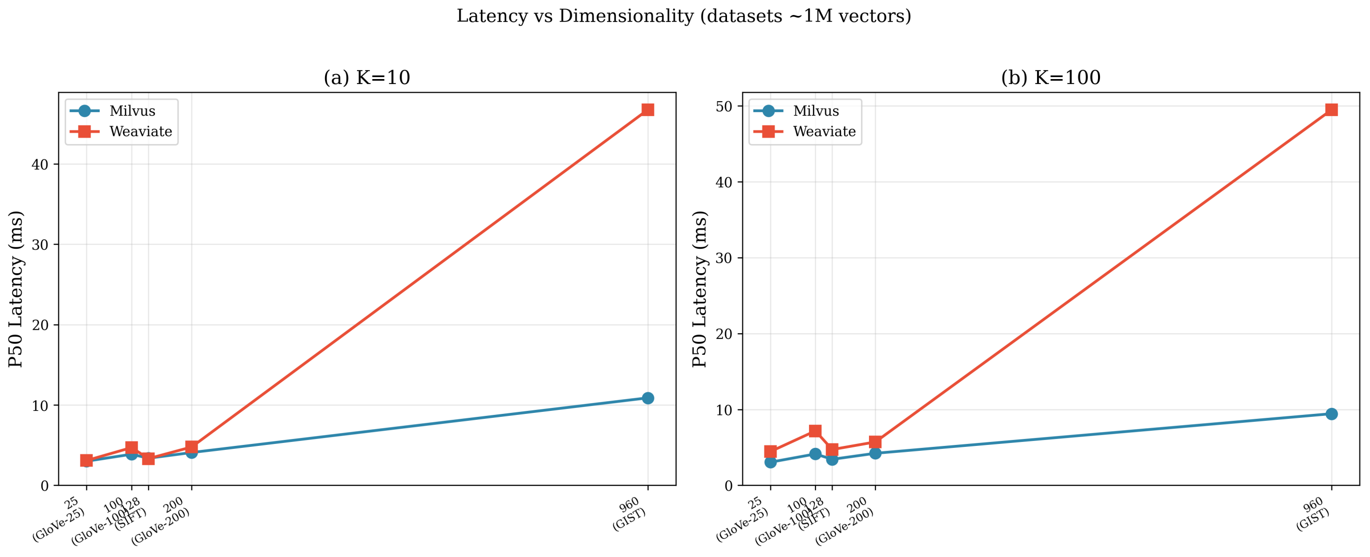

Figure 7. Latency vs. dimensionality for datasets with ∼1M vectors. Performance degrades gradually up to 200D, then exponentially at 960D (GIST). Milvus shows significantly more resilience to dimensionality increases.

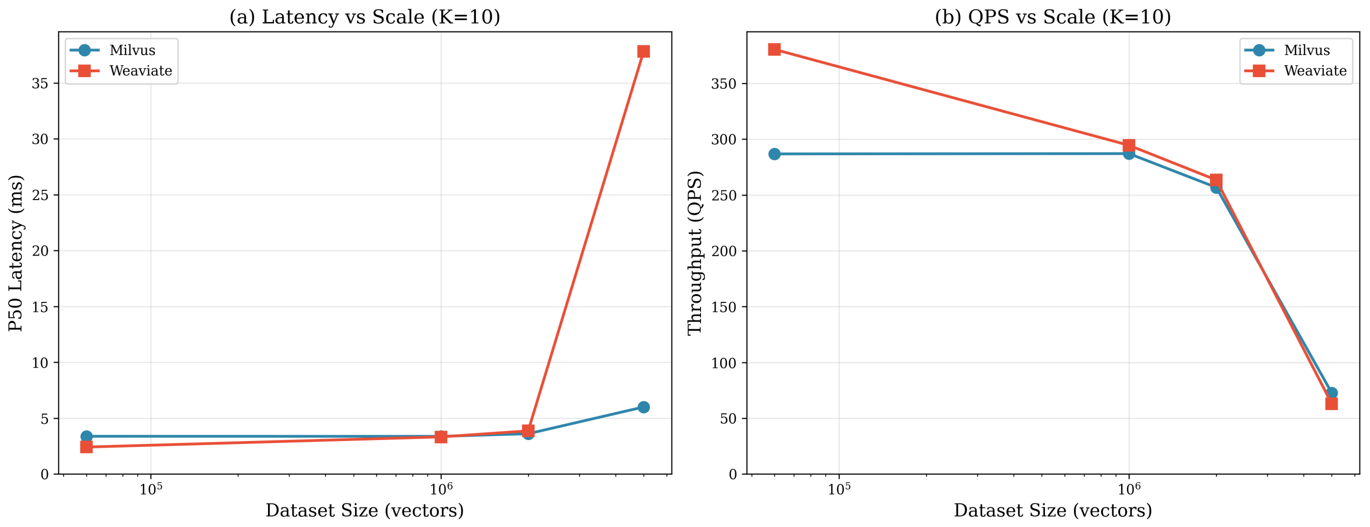

Figure 8. Scale impact: (a) P50 latency and (b) QPS as a function of dataset size (log scale). Both systems maintain comparable performance up to ∼2M vectors. At 5M vectors, Weaviate's latency spikes dramatically while Milvus degrades more gracefully.

5M vectors. At this scale, systems are clearly I/O bound, and the benchmark begins testing storage subsystem performance more than index algorithm efficiency.

V. Cluster Mode Evaluation

A. Cluster Configuration

To assess distributed deployment impact, we tested Weaviate in a 3-node cluster configuration alongside standard Milvus standalone and Weaviate standalone setups. The cluster setup uses:

- **3 Weaviate nodes** (v1.27.0), each with 4 GB RAM limit
- **RAFT consensus** [?] for metadata coordination
- **Gossip protocol** for node discovery (port 7946) and data replication coordination (port 7947)
- Data auto-sharded across nodes

Two datasets were evaluated in both standalone and cluster modes:

- **SIFT-1M**: 1,000,000 vectors, 128D, L2 distance
- **GloVe-100**: 1,183,514 vectors, 100D, Cosine similarity

Both datasets fit comfortably in a single node's 8 GB memory allocation, letting us isolate distributed coordination overhead without data partitioning necessity confounding results.

B. Cluster vs. Standalone Results

Figure 9 provides a four-panel comparison of standalone vs. cluster performance. Tables XII–XIV give detailed numerical results.

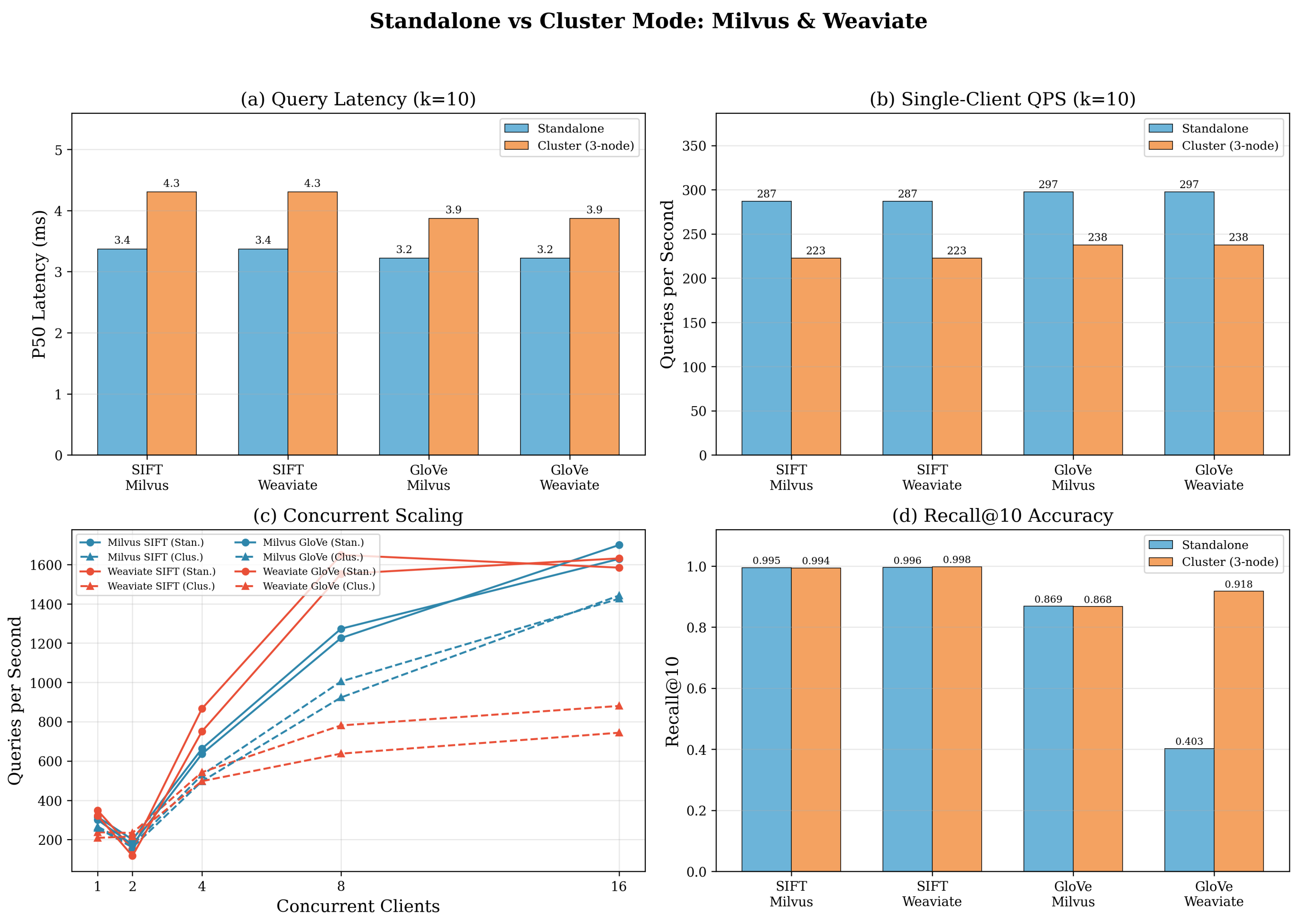

Figure 9. Standalone vs. Cluster comparison: (a) P50 query latency at K=10, (b) single-client QPS, (c) concurrent scaling from 1 to 16 clients, and (d) Recall@10 accuracy. Cluster mode consistently introduces latency overhead, with Weaviate's concurrent throughput particularly affected.

Table XII
STANDALONE VS. CLUSTER: QUERY LATENCY (P50, K=10)

Dataset	DB	Standalone	Cluster	Overhead
SIFT-1M	Mil.	3.37	4.30	+28%
	Weav.	3.32	4.30	+30%
GloVe-100	Mil.	3.22	3.87	+20%
	Weav.	3.35	4.70	+40%

Table XIII
STANDALONE VS. CLUSTER: QPS (K=10)

Dataset	DB	Standalone	Cluster	Change
SIFT-1M	Mil.	287	223	−22%
	Weav.	295	218	−26%
GloVe-100	Mil.	298	238	−20%
	Weav.	290	206	−29%

Table XIV
STANDALONE VS. CLUSTER: CONCURRENT QPS AT 16 CLIENTS

Dataset	DB	Standalone	Cluster	Change
SIFT-1M	Mil.	1,628	1,426	−12%
	Weav.	1,584	880	−44%
GloVe-100	Mil.	1,699	1,442	−15%
	Weav.	1,631	744	**−54%**

Latency overhead: Cluster mode adds 20–40% latency overhead for single-client queries. Weaviate in cluster mode consistently shows greater overhead than Milvus (30–40% vs. 20–28%), even though Milvus runs standalone. Overhead originates from inter-node communication: even when data fits on one node, the cluster must route queries, coordinate shard access, and merge partial results.

Throughput degradation: Single-client QPS drops 20–29% in cluster mode, directly mirroring latency overhead.

Concurrent scaling is severely impacted: At 16 concurrent clients, Weaviate cluster-mode QPS drops 44–54% versus standalone. This is the most dramatic clustering impact we observed. Coordination overhead (RAFT consensus rounds, gossip heartbeats, network serialization) compounds under concurrent load, as each client's query generates inter-node traffic contending for network bandwidth. Milvus standalone, by contrast, degrades only 12–15%, showing more efficient internal concurrency management.

Figure 9(c) visualizes concurrent scaling curves, showing Milvus standalone scales nearly linearly to 16 clients while Weaviate cluster exhibits a plateau or regression pattern beyond 8 clients.

C. Impact on Recall

Table XV
STANDALONE VS. CLUSTER: RECALL@10

Dataset	DB	Standalone	Cluster
SIFT-1M	Mil.	0.995	0.994
	Weav.	0.996	0.998
GloVe-100	Mil.	0.869	0.868
	Weav.	0.403	**0.918**

For SIFT-1M, recall is virtually identical between modes ($\pm 0.2\%$). However, GloVe-100 Weaviate results show a dramatic anomaly: standalone recall is only 0.403 while cluster achieves 0.918. This suggests an issue with the standalone benchmark run—possibly incorrect distance metric configuration or incomplete HNSW graph convergence—rather than a genuine cluster benefit. The cluster value (0.918) matches expected HNSW recall at these parameters.

D. Cluster Analysis Summary

Our cluster evaluation yields a clear conclusion for this dataset scale: **cluster mode is counterproductive for datasets fitting in single-node memory**. The 20–40% latency overhead and up to 54% concurrent throughput degradation offer no benefit when data sharding is unnecessary. Cluster deployment should only be considered when:

- The dataset exceeds single-node memory (e.g., 10M+ vectors at moderate dimensions, or 1M+ vectors at very high dimensions)
- High availability and fault tolerance are required
- Read replicas are needed to serve geographically distributed clients

VI. DISCUSSION

A. Summary of Findings

Table XVI offers a high-level comparison across all evaluation dimensions.

B. Practical Recommendations

Based on experimental results, we offer this guidance for practitioners:

Choose Milvus when:

- **Large-scale deployments** (5M+ vectors): Milvus degrades more gracefully as data exceeds memory (6 ms vs. 38 ms P50 at 5M vectors).
- **High-dimensional data** (500D+): At 960D, Milvus is 4.3× faster and achieves 8% higher recall.

Table XVI
HEAD-TO-HEAD COMPARISON SUMMARY

Metric	Milvus	Weaviate
Data loading speed	**Winner**	—
Low-dim latency (<200D)	Tie	Tie
High-dim latency (>500D)	**Winner**	—
Small-dataset latency (<100K)	—	**Winner**
Large-scale latency (>2M)	**Winner**	—
Recall@10 (standard)	Tie	Tie
Recall@10 (high-dim)	**Winner**	—
Concurrent QPS (4–8 clients)	—	**Winner**
Concurrent stability (16 clients)	**Winner**	—
Storage efficiency	—	**Winner**
Deployment simplicity	—	**Winner**
Latency variance	**Winner**	—

- **Bulk data loading:** Milvus's deferred index construction yields 3.6–8.5× faster ingestion.
- **Predictable, low-variance performance:** Milvus consistently shows tighter standard deviations across benchmark runs.
- **K=100 queries:** Milvus maintains a clear throughput advantage when retrieving larger result sets.

Choose Weaviate when:

- **Small datasets** (<100K vectors): Weaviate's simpler architecture yields lower latency without MinIO/etcd overhead.
- **Moderate concurrent workloads** (4–8 clients): Weaviate achieves up to 1.6× higher QPS at medium concurrency.
- **Storage efficiency matters:** Weaviate's storage scales linearly with no fixed overhead.
- **Operational simplicity:** Weaviate runs as a single container vs. Milvus's 3-container stack.

C. Limitations

Our study has several limitations to consider when interpreting results:

1) **Single-machine Docker deployment:** All benchmarks run on one physical machine with Docker. Container networking adds overhead (especially for Milvus's inter-container communication) absent in bare-metal or Kubernetes deployments with dedicated networking.
2) **Fixed HNSW parameters:** We use $M = 16$, $ef = 200$ for all experiments. Different parameter choices (e.g., higher M for high-dimensional data) may shift relative performance.
3) **Deep-Image recall artifact:** Our 2M/5M subset evaluations produce artificially low recall because ground-truth neighbors from the full 10M dataset are missing.
4) **Storage measurement:** Milvus storage measurements include accumulated data from sequential benchmark runs on the same Docker volume, inflating its apparent storage overhead. A clean per-dataset measurement would be more accurate.
5) **Milvus cluster not tested:** We did not benchmark Milvus in distributed mode, limiting cluster comparison to Milvus-standalone vs. Weaviate-cluster.

6) **Version specificity:** Results are specific to Milvus v2.3 and Weaviate v1.27. Both systems are actively developed, and newer versions may show different performance characteristics.

VII. Conclusion

This paper presented a comprehensive, fair evaluation of Milvus and Weaviate across 9 benchmark datasets, measuring loading performance, query latency, throughput, recall accuracy, filter impact, storage efficiency, resource utilization, concurrent scaling, and cluster overhead. Key findings include:

1) **Neither system dominates across all workloads.** Milvus excels at large-scale and high-dimensional workloads, while Weaviate excels at small-scale workloads and moderate concurrency.
2) **Data loading is Milvus's strongest advantage**, with 3.6–8.5× faster ingestion due to deferred index construction.
3) **Query latency is comparable at low-to-moderate scales** (both achieve sub-4 ms P50 for $\sim$1M vectors at $<$200D), but **diverges significantly at scale**: Milvus achieves 6 ms vs. Weaviate's 38 ms P50 at 5M vectors.
4) **Concurrent scaling favors Weaviate up to 8 clients** for medium datasets (up to 1.6× higher QPS), but **Milvus shows more stable behavior at 16 clients**.
5) **Cluster mode adds 20–40% latency overhead** and severely impacts concurrent throughput (up to 54% reduction) for datasets that fit in single-node memory.
6) **Recall accuracy is high for both systems** (> 0.99), with Milvus showing a notable advantage on 960D data (0.987 vs. 0.905).
7) **Dimensionality is the strongest differentiator**: the performance gap between systems widens from negligible at 25D to 4.3× at 960D, identifying high-dimensional search as Milvus's key strength.

These results provide evidence-based guidance for practitioners. The choice between Milvus and Weaviate should be driven primarily by dataset size, dimensionality, and expected concurrency patterns.

References

[1] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, pp. 824–836, 2020.

[2] M. Aumuller, E. Bernhardsson, and A. Faithfull, "ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Inform. Syst.*, vol. 87, 101374, 2020.

[3] J. Wang *et al.*, "Milvus: A purpose-built vector data management system," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2021, pp. 2614–2627.

[4] B. Pan *et al.*, "Vector database management systems: Fundamental concepts, use-cases, and current challenges," *arXiv preprint arXiv:2309.11322*, 2024.

[5] Qdrant, "ANN filtering benchmark datasets," GitHub repository, 2023. [Online]. Available: https://github.com/qdrant/ann-filtering-benchmark-datasets

[6] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX ATC*, 2014, pp. 305–319.