VecBench: A Controllable Benchmark for Filtered Vector Search: [Experiments & Analysis]

XIANG ZHANG, Renmin University of China, China
CHAO ZHANG, Renmin University of China, China
JU FAN*, Renmin University of China, China
GUOLIANG LI, Tsinghua University, China
XIAOYONG DU, Renmin University of China, China

As vector databases become increasingly critical to supporting large language models (LLMs), efficient and effective vector data retrieval has become imperative. *Filtered vector search*, which retrieves relevant vectors under scalar filter constraints, has recently attracted significant attention. However, existing benchmarks for evaluating filtered vector search have three major limitations. First, they rely on simple, low-dimensional datasets that fail to reflect real-world scenarios involving thousands of dimensions and millions of records. Second, they use random filter values, resulting in trivial query plans that do not adequately challenge modern vector database optimizers. Third, they lack a unified evaluation metric to comprehensively quantify end-to-end performance.

In this work, we propose VecBench, a controllable benchmark for holistic evaluation of filtered vector search. Designing such a benchmark poses three key challenges: (1) generating high-dimensional and large-scale vector data while preserving the original similarity distribution, (2) constructing representative benchmark queries that capture diverse filtered search strategies, and (3) developing a unified metric that enables fair and comprehensive comparisons under both static and dynamic scenarios. VecBench addresses these challenges through *controllable data generation*, *adjustable query synthesis*, and *end-to-end holistic benchmarking*. First, it can generate massive vector data with flexible dimensionality and scale while preserving the original distribution with a provable error bound. Second, it supports fine-grained workload control by varying selectivity and filter correlation, enabling systematic stress testing of different filtered search strategies. Third, it introduces a unified evaluation framework covering six phases, including concurrent query processing and dynamic update scenarios. To demonstrate the effectiveness of VecBench and to shed light on the strengths and weaknesses of different filtered vector search methods, we conduct extensive experiments on ten representative approaches across four popular vector databases.

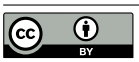
CCS Concepts: • **Information systems → DBMS engine architectures**; **Information retrieval query processing**; • **General and reference → Evaluation**.

Additional Key Words and Phrases: Vector Databases; Filtered Vector Search; Benchmarking.

ACM Reference Format:
Xiang Zhang, Chao Zhang, Ju Fan, Guoliang Li, and Xiaoyong Du. 2026. VecBench: A Controllable Benchmark for Filtered Vector Search: [Experiments & Analysis]. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 248 (June 2026), 27 pages. https://doi.org/10.1145/3802125

*Ju Fan is the corresponding author.

Authors' Contact Information: Xiang Zhang, Renmin University of China, Beijing, China, xiangzhang@ruc.edu.cn; Chao Zhang, Renmin University of China, Beijing, China, cycchao@ruc.edu.cn; Ju Fan, Renmin University of China, Beijing, China, fanj@ruc.edu.cn; Guoliang Li, Tsinghua University, Beijing, China, liguoliang@tsinghua.edu.cn; Xiaoyong Du, Renmin University of China, Beijing, China, duyong@ruc.edu.cn.



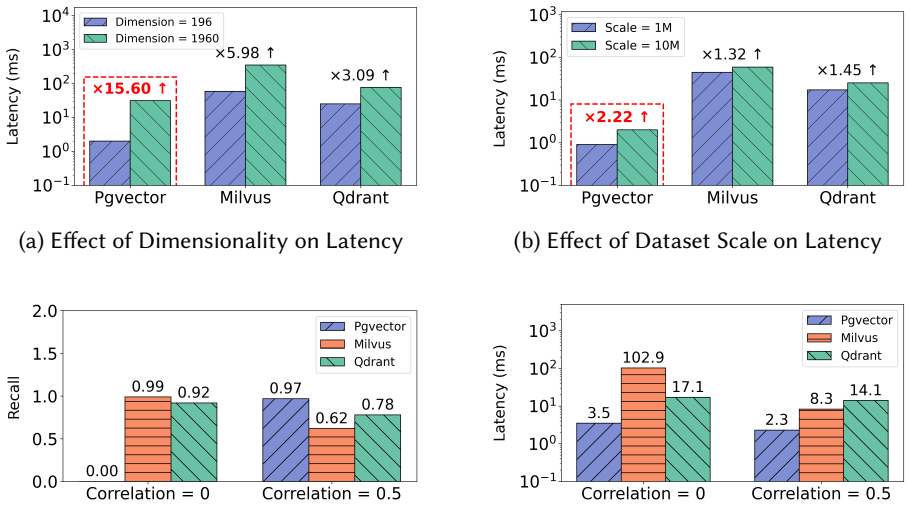
ACM 2836-6573/2026/6-ART248
https://doi.org/10.1145/3802125

1 Introduction

Efficient and high-recall vector search has become a critical component for large language models (LLMs), as it underpins retrieval-augmented generation (RAG) and effectively mitigates hallucination issues [7, 10, 33, 50, 51]. Vector databases serve as external memory to support complex inference processes in LLMs [9, 28, 32, 35]. Beyond plain similarity search, many real-world applications require filtered vector search [4, 38, 41], where similarity search over embeddings is combined with structured attribute constraints (e.g., equality or range filters). For example, an e-commerce user may specify size and price filters when retrieving similar product images [44, 46], or a marketing team may search internal documents within a specific time period. To meet such demands, mainstream vector databases have incorporated filtered vector search as a core functionality, adopting different indexing methods (e.g., HNSW [29, 30] and IVFFLAT [8]) and search strategies (e.g., pre-filtering, post-filtering, and in-filtering) [4, 35, 44]. However, the diversity of these techniques calls for a tailored and comprehensive benchmark to evaluate the performance of filtered vector search methods.

Limitations of Existing Benchmarks. Lately, several benchmarks have been proposed for filtered vector search, including Big-ANN [40], VectorDBBench [53], and BigVectorBench [22]. While the benchmarks provide data and workloads for evaluating filtered vector search, they remain insufficient due to three limitations:

- **Limited data realism.** Existing datasets have fixed, relatively low dimensionality (typically below 1,000), making them inadequate to reflect realistic high-dimensional, large-scale scenarios.
- **Simplistic workload design.** The workloads of existing benchmarks typically use only a single type of filter (equality or range) with randomly sampled values from a small data domain, offering little challenge for modern query optimization where filters are often diverse and correlated with the vector distribution.
- **Lack of dynamic evaluation.** Most benchmarks focus solely on static query workloads, overlooking dynamic update scenarios that are common in real-world applications.



(a) Effect of Dimensionality on Latency

(b) Effect of Dataset Scale on Latency

(c) Effect of Correlation on Recall

(d) Effect of Correlation on Latency

Fig. 1. Comparison of average query latency in (a, b, d) and recall in (c) across three vector databases on the YFCC dataset [31].

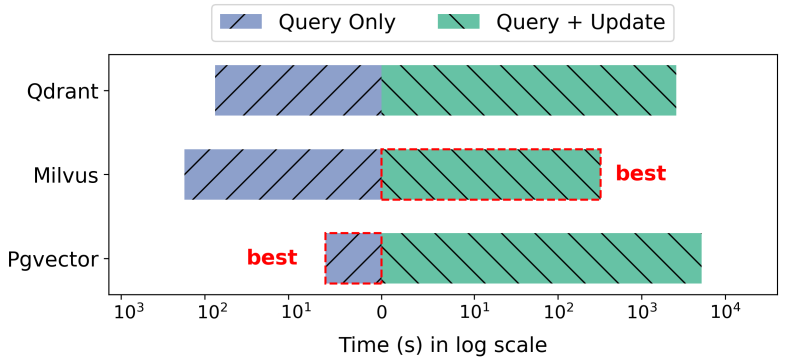


Fig. 2. Comparison of query-only and query+update execution times for three vector databases. Left values represent query-only times, while right values represent the combined query+update times.

These limitations hinder a comprehensive understanding of vector database performance under realistic, heterogeneous workloads.

Motivations, Challenges, and Solutions. To address the limitations, we propose VecBench, a controllable benchmark for holistic evaluation of filtered vector search. Table 1 compares existing benchmarks with VecBench, highlighting six key features. Unlike prior work that primarily targets approximate nearest neighbor (ANN) search with simple filters over static data, VecBench supports controllable data generation, adjustable selectivity, steerable filter correlation, multi-type predicates, and dynamic update scenarios. Specifically, VecBench addresses three key challenges.

C1. Controllable vector data generation with realistic distribution. Existing benchmarks typically rely on datasets with fixed dimensionality and scale, lacking controllability to adapt to diverse benchmarking scenarios. However, in practice, variations in these factors can significantly affect system performance [47]. As illustrated in Figure 1a and 1b, increasing dimensionality and scale by 10x leads to substantial latency increases in Pgvector, Milvus, and Qdrant (e.g., 15x and 2x higher latency for Pgvector). This observation underscores the need to evaluate systems under diverse data distribution for a comprehensive performance understanding.

However, generating vector data that is both controllable in dimension and scale and can closely approximate realistic distributions is non-trivial. Naive methods, such as randomly dropping or selecting columns, may distort the original distribution, while deep generative models (e.g., Generative Adversarial Network [14], Variational Autoencoder [23], and diffusion models [18]) may incur prohibitive training and generation costs at a large scale. To address this challenge, VecBench introduces a controllable data generation method that preserves the essential structure of the original distribution while allowing flexible control over dimensionality and scale. Specifically, it applies random linear projection based on the Johnson–Lindenstrauss lemma [24], ensuring both efficiency and a provable error bound.

C2. Filter vector search query synthesis with adjustable selectivity and filter correlation. Existing benchmarks typically adopt simple filter types and randomly sample filter values, failing to reflect realistic query patterns where filters often include equality, range, or contain predicates, and their values may correlate with the distribution of the underlying vectors. For example, when the correlation between filter values and vector distribution is zero, the nearest neighbors are unlikely to satisfy the filter conditions. As shown in Figure 1c and 1d, varying filter correlation significantly impacts system behavior: Milvus achieves the highest recall when correlation = 0, whereas Pgvector performs best when correlation = 0.5. In terms of latency, Milvus exhibits an

Table 1. Comparison of Existing Benchmarks on Filtered Vector Search.

Features	ANN-Benchmarks [2]	Big-ANN [40]	Vector DBBench [53]	BigVector Bench [22]	VecBench
Approximate Nearest Neighbor Search	✓	✓	✓	✓	✓
Controllable Data Generation	×	×	×	×	✓
Adjustable Selectivity	×	×	×	×	✓
Steerable Filter Correlation	×	×	×	×	✓
Multi-type predicates that include equal, range and in filters.	×	×	×	✓	✓
Dynamic test cases that include insert, update, and delete	×	×	×	✓	✓

order-of-magnitude increase when correlation = 0, highlighting that query characteristics can drastically affect performance.

However, generating queries with a target criterion (such as selectivity and correlation) is challenging due to the complex joint distribution between structured and vector features. To address this, VecBench introduces a distribution-aware query synthesis method that enables fine-grained control over the correlation between vector features and filter values, enabling precise evaluation and better explainability of filtered vector search performance.

C3. Unified metric for holistic comparison of filtered vector search. Existing benchmarks typically report isolated metrics such as latency, recall, or throughput, which fail to capture the inherent system-level trade-offs. For example, fast index construction may come at the cost of slow updates. Focusing on a single dimension (e.g., query latency) risks overlooking critical bottlenecks in indexing or maintenance. As shown in Figure 2, Pgvector and Milvus achieve the lowest and highest latency in query-only scenarios, respectively. However, we observe the opposite results when updates are introduced, allowing Pgvector and Milvus to achieve the worst and best overall latency. This observation highlights the necessity of evaluating both static and dynamic workloads.

However, defining a unified metric for holistic comparison across systems is non-trivial due to the large exploration space and diverse performance dimensions [48]. To address the challenge, VecBench proposes an end-to-end evaluation framework covering six execution phases, considering concurrent query processing and dynamic updates. We further introduce a unified ranking-based metric that aggregates multi-phase performance into a single comparable score, enabling holistic benchmarking of filtered vector search systems.

In summary, we make the following contributions:

- We propose VecBench, a controllable benchmark for holistic evaluation of filtered vector search.
- For understanding of vector database performance under realistic, heterogeneous workloads, we develop novel techniques for controllable data generation, adjustable query synthesis, and end-to-end holistic benchmarking.
- We demonstrate the effectiveness of VecBench by evaluating four widely used vector databases with ten representative filtered vector search techniques. The results show that VecBench enables a more principled and comprehensive evaluation of filtered vector search, providing clearer insights into system trade-offs and performance characteristics. The source code of VecBench is publicly available at Github[1], along with a leaderboard for tracking benchmark results.

2 Preliminary

In this section, we formally define the research problem of filtered vector search, along with two essential concepts, selectivity and filter correlation. We then introduce representative vector storage formats and index types, followed by a detailed review of the principles of various filtered search

[1]https://github.com/ruc-datalab/VecBench

strategies as well as their respective advantages and challenges. Finally, we introduce the CRUD mechanisms and concurrent query processing in vector databases.

2.1 Definitions

*Definition 2.1 (**Filtered Search Query** Q).* A filtered search query is a triple $Q = (\mathbf{q}, P, K)$, where $\mathbf{q} \in \mathbb{R}^d$ is the query vector; $P(\cdot)$ is a logical predicate set over scalar attributes, possibly involving multiple columns; and K is the number of nearest neighbors to retrieve. Given a dataset $\mathcal{D} = \{(\mathbf{v}_i, \mathbf{S}_i) \mid i = 1, 2, \ldots, N\}$, where $\mathbf{v}_i \in \mathbb{R}^d$ denotes the vector embedding and $\mathbf{S}_i = (s_{i1}, s_{i2}, \ldots, s_{im})$ the multi-column scalar attributes, the predicate-filtered subset is

$$\mathcal{D}_P = \{(\mathbf{v}_i, \mathbf{S}_i) \in \mathcal{D} \mid P(\mathbf{s}_i) = \text{true}\}.$$

The execution of query Q over dataset $\mathcal{D}$ yields the top-K nearest vectors to $\mathbf{q}$ within the filtered subset $\mathcal{D}_P$:

$$\mathrm{R}(Q) = \text{Top-}K_{\mathbf{v}_i \in \mathcal{D}_P} \, \mathrm{dist}(\mathbf{q}, \mathbf{v}_i), \tag{1}$$

where $\mathrm{dist}(\cdot, \cdot)$ is a vector distance function (e.g., Euclidean or cosine). In other words, Q specifies both the filtering predicate and similarity criterion that jointly determine the final result $\mathrm{R}(Q)$. Our work evaluates a system's ability to execute filtered vector search under diverse benchmarking tasks, specifically investigating how it minimizes latency while maintaining high recall under different data dimension, scale, and filtering conditions.

A filtered vector search query can be expressed as follows:

```
SELECT * FROM T
WHERE col1 = tag AND col2 > value
ORDER BY distance(q, vector_col)
LIMIT K;
```

The above query retrieves the top-K nearest vectors to $\mathbf{q}$ from table **T**, e.g., vector_col, restricted to the subset of records satisfying the predicate $P(\cdot)$ over the target columns e.g., col1 and col2.

*Definition 2.2 (**Filter Correlation** $c_f^{(k)}$).* The *filter correlation* quantifies the local dependency between the scalar predicate $P(\cdot)$ and the spatial distribution of vectors in the embedding space. Unlike the selectivity s_f, which reflects overall selectivity, $c_f^{(k)}$ characterizes how strongly the predicate is correlated with the local vector neighborhood around a query.

Formally, for a query vector $\mathbf{q}$, let $\mathcal{N}_k(\mathbf{q}) \subseteq \mathcal{D}$ denote its k nearest neighbors. We define the *filter correlation* $c_f^{(k)}$ for predicate $P(\cdot)$ over query $\mathbf{q}$ within this neighborhood as

$$c_f^{(k)}(\mathbf{q}) = \frac{|\mathcal{N}_k(\mathbf{q}) \cap \mathcal{D}_P|}{|\mathcal{N}_k(\mathbf{q})|}. \tag{2}$$

where $c_f^{(k)}(\mathbf{q})$ reflects how densely the predicate is satisfied within the local neighborhood of $\mathbf{q}$: a higher $c_f^{(k)}(\mathbf{q})$ indicates that predicate-satisfying records are concentrated near $\mathbf{q}$, while a lower value suggests sparsity.

EXAMPLE 1. *Considering the same SQL above. If the query* $\mathbf{q}$ *lies in a region where* 80 *of its* $k = 100$ *nearest neighbors satisfy the predicate, then* $c_f^{(100)}(\mathbf{q}) = 80/100 = 0.8$*, indicating strong filter correlation. Conversely, if none of the* 100 *neighbors satisfy the predicate (*$c_f^{(100)}(\mathbf{q}) = 0$*), reflecting weak filter correlation.*

2.2 Vector Storage and Indexes

Existing systems can be broadly categorized into two categories [34]: ***relational-extended databases*** that embed vector types into traditional engines (e.g., PostgreSQL+pgvector, AnalyticDB-V [44], ClickHouse [39]), and ***native vector databases*** that adopt purpose-built architectures (e.g., Milvus and Qdrant). Their design ideas lead to fundamental differences in storage layout and indexing behavior, which in turn influence performance characteristics between different databases.

Storage Models. Relational-extended systems inherit the *page-based* storage model of traditional databases, where vectors are stored as column values within fixed-size pages. This design benefits from mature transaction and recovery mechanisms but suffers from limited spatial locality and coarse-grained I/O. Moreover, page-level locking and fixed-size allocation restrict concurrent or asynchronous index construction, and queries typically traverse entire tables or indexes within a unified execution plan. In contrast, native vector databases employ *segment-based* data layouts, where data are partitioned into independently managed segments. Each segment can be indexed, searched, or compacted asynchronously, allowing batch index construction and concurrent queries. During search, multiple segments are searched in parallel, and partial results are merged to form the final top-K, yielding better scalability and recall. Although such systems often relax strict ACID guarantees, their lightweight design significantly improves indexing and query efficiency at scale [1, 27].

Index Structures. Existing vector databases primarily adopt either ***table-based*** or ***graph-based*** structures, each offering distinct trade-offs between retrieval precision, update efficiency, and memory overhead. Table-based indexes, such as IVFFLAT and IVFADC, partitions the vector space into clusters (e.g., via k-means [3] and restricts search to a subset of clusters. It supports efficient updates and controllable recall–latency trade-offs but is sensitive to clustering quality and the number of probed partitions. Graph-based indexes such as HNSW [30], NSG [11], and KGraph [6] construct the navigable small-world graph where each node represents a vector connected to its neighbors by similarity. However, they typically incur higher memory overhead and maintenance costs. Besides, efficient online updates on them remain non-trivial. Recently, several *hybrid indexes* (e.g., NHQ [43], Filtered-DiskANN [13], ACORN [36], SeRF [54], UNIFY [26]) have been proposed to integrate vector similarity with scalar predicates for filtered search. While some of these designs have begun to be integrated into production systems (e.g., DiskANN in Milvus and ACORN in Weaviate), broad adoption remains challenging as many existing hybrid indexes still struggle to handle complex query predicates, high-concurrency processing, and frequent dynamic updates.

2.3 Filtered Search Strategies

Filtered search strategies in vector databases can be categorized into four approaches: *pre-filtering*, *post-filtering*, *in-filtering*, and *expanded-filtering*, depending on when scalar predicates are checked relative to approximate nearest neighbor search (ANNS), e.g. before, during, or after the ANNS process and whether scalar attributes are integrated into the index structure itself [34] [4]. As shown in Figure 3, pre-filtering and post-filtering refer to sequentially performing ANNS and scalar filtering operations in different orders, in-filtering refers to performing filtering operations while conducting ANNS, and expanded-filtering achieves filtered search on an expanded index structure by modifying the original index structure to integrate scalar information.

(1) Pre-filtering. It first applies scalar indices like B+-tree [15] to obtain a subset of data satisfying the filtering predicates, followed by a brute-force vector search within this subset since vector indices constructed based on the entire dataset cannot be reused on filtered subsets. Such method performs well in scenarios with low selectivity, as the reduced candidate set lowers the computational cost. However, it becomes prohibitively expensive when selectivity is high because the time overhead is

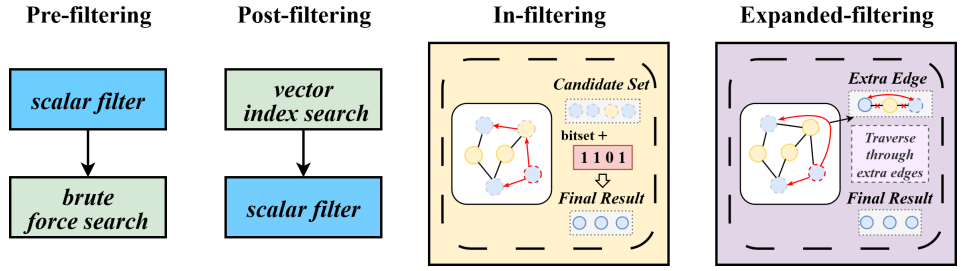


Fig. 3. Comparison of four mainstream filter search strategies in vector databases: pre-filtering, post-filtering, in-filtering, and expanded-filtering.

extremely high due to the need to perform brute-force searches on a large number of candidate data points.

(2) Post-filtering. This strategy first performs ANNS using vector indexes to retrieve a candidate set, and then applies scalar filters to produce final results. Post-filtering is efficient when the selectivity is high, since vector search dominates the cost. However, the candidate set may contain too few items that satisfy the predicate conditions with low selectivity, risking insufficient results. To compensate, relational systems like *Pgvector* employ *iterative post-filtering*, which performs iterative index scans to fetch candidates in batches until the required number of results is met. However, this approach still faces performance challenges as it is hard to tune the specific expansion ratio due to the influence of data and query distribution.

(3) In-filtering. This strategy integrates predicate checks during the ANNS process, e.g., by verifying each visited candidate directly or using a precomputed bitset. While in-filtering mitigates the "insufficient results" issue of post-filtering, its performance can still be compromised in complex data distributions since the underlying index structure remains unchanged. To address this, recent advancements like NHQ guide the search using a hybrid distance metric that fuses vector similarity with scalar constraints, while UNIFY adaptively selects between skip-lists and joint-filtering based on query selectivity.

(4) Expanded-filtering. This strategy integrates scalar information directly into the vector index structure and adapts the search logic accordingly. Representative hybrid indexes leverage this by modifying the index topology or traversal rules. For example, iRangeGraph and SeRF organize subgraphs into hierarchies or use attribute-sorted insertion to encode constraints into the graph. To handle diverse distributions, RWalks employs attribute-aware random walks to diffuse constraint signals, steering the search towards nodes satisfying the predicates. This design achieves high efficiency and accuracy by tightly coupling filtering and similarity computation. Its main drawbacks lie in increased index construction complexity, higher storage overhead, and potentially higher query cost due to more elaborate index structures.

Dynamic Strategy Selection. Existing vector databases tend to employ multiple of these strategies and choose among them dynamically to optimize filtered search performance. Overall, relational systems make cost-driven decisions at query planning time, while native systems employ lightweight runtime heuristics to balance efficiency and recall across varying query conditions. Relationally-extended systems such as *PostgreSQL with Pgvector* typically rely on a cost-based optimizer, which estimates the computational cost of different query plans based on factors such as filter selectivity, index statistics, and data size. During query planning, the optimizer evaluates whether it is more efficient to perform pre-filtering or post-filtering. In contrast, native vector databases such as *Milvus* and *Qdrant* typically employ threshold-based heuristics to decide whether to utilize vector indexes

or fall back to full scans when filters are applied. When the estimated number of filtered data points exceeds this threshold, the system executes an indexed ANNS (e.g., IVF or HNSW) and applies the filter condition either during or after the vector search.

2.4 CRUD Mechanisms

CRUD operations—*Create, Read, Update, and Delete*—form the foundation of vector database management can significantly influence index maintenance overhead, recall and latency [5, 17, 21, 49].

Specially, in relational-extended systems, inserts and updates are executed synchronously through SQL operations, with corresponding vector index entries updated in place. This design provides immediate query visibility and strong consistency but incurs higher write latency and CPU overhead due to transactional coupling between data and indexes. Native systems, by contrast, adopt asynchronous ingestion pipelines in which new vectors are first written to in-memory buffers or write-ahead logs and indexed lazily during background compaction. Such decoupling achieves high ingestion throughput and scalability but delays index availability until the next merge cycle. For read operations, relational systems typically employ multi-version concurrency control (MVCC) [45] and cost-based optimization to ensure snapshot consistency and unified query planning, whereas native systems rely on lightweight snapshot isolation over immutable segment files, enabling highly concurrent and low-latency retrieval. Updates and deletions in relational systems trigger synchronous index maintenance to preserve correctness, while native systems use lazy invalidation—marking outdated entries as inactive and periodically rebuilding indexes—to avoid online disruption. This deferred maintenance enhances availability and write efficiency but introduces transient staleness and additional storage overhead.

2.5 Concurrent Query Processing

Concurrent query processing in vector databases requires balancing throughput, isolation, and latency under multiple read requests in parallel. Relational-extended systems, such as PostgreSQL with Pgvector, execute concurrent hybrid queries within the SQL engine under snapshot isolation. Each query plan is optimized independently using a cost-based model, enabling efficient scheduling and resource utilization even under high connection concurrency. Although these systems carry transactional overhead, their sophisticated execution pipelines and caching mechanisms often yield stable performance for concurrent workloads. Native systems adopt a more data-parallel model where queries are decomposed into subtasks (e.g., candidate generation and distance computation) executed concurrently across threads or segments. This design achieves high throughput through shared, read-optimized indexes but lacks global query optimization, which can lead to less efficient resource sharing under many simultaneous queries. Overall, while native systems emphasize throughput-oriented parallelism, relational systems can exhibit superior performance in pure read concurrency due to optimized planning and efficient I/O scheduling.

3 Benchmark Details

In this section, we provide an overview of `VecBench` and introduce three key components within the framework. Based on this foundation, we then discuss the detailed contributions of `VecBench`, including JL-based data synthesis, distribution-aware query generation, and a unified metric.

3.1 Overview

To enable a comprehensive evaluation of filtered vector search, we develop an end-to-end framework. As shown in Figure 4, `VecBench` is a modular benchmarking framework and composed of three key components: a *data synthesizer*, a *filtered query generator* and a *workload executor*.

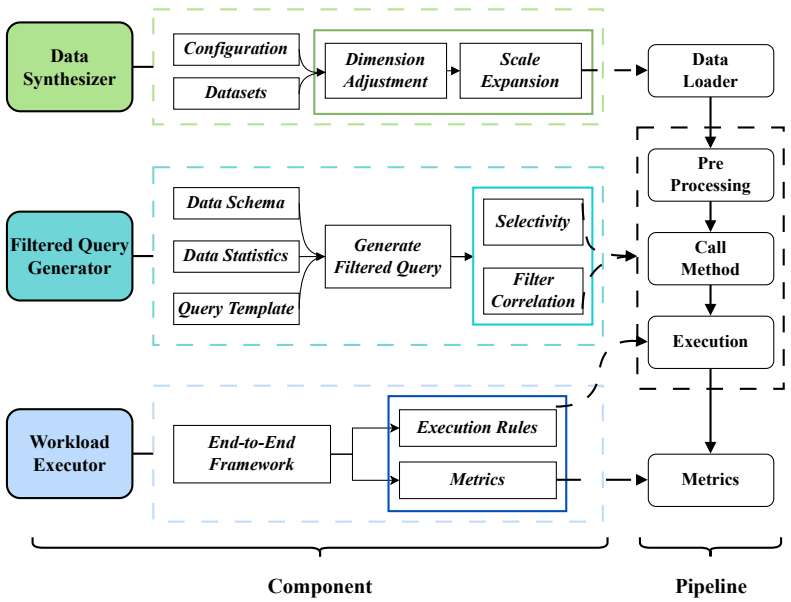


Fig. 4. The VecBench Framework, which consists of three key components: a *data synthesizer*, a *filtered query generator* and a *workload executor*.

The entire evaluation process is controlled based on a configuration file. By integrating heterogeneous datasets and utilizing modular components, it achieves fully automated execution of filtered search testing workflows for vector databases. Within this framework, the data synthesizer flexibly adjusts the dimensions and scale of raw data to generate test data meeting diverse testing needs. The filtered query generator automatically constructs filtering query tasks with varying selectivity and filter correlation. Specifically, it generates queries with different filtering conditions based on the existing query set by modifying the structured predicates on scalar attributes. The workload executor supports typical filtered vector search tasks and dynamic data update operations. It coordinates two types of workloads, query and update, according to a set of predefined execution rules, while simultaneously measuring various key metrics and ranking databases based on these metrics for comparison.

3.2 Data Synthesizer

By systematically adjusting vector dimension and data scale, we can generate datasets that not only preserve realistic statistical properties but also challenge vector database systems under increasingly complex conditions.

3.2.1 Datasets. In our benchmark, we adopt the YFCC [31] dataset, a large-scale collection derived from Flickr, covering a wide range of real-world scenes along with rich textual metadata. The version used in our study corresponds to the subset aligned with the Big-ANN benchmark. We observed that most studies use traditional vector datasets such as SIFT [20] and GIST [19]. Since these datasets lack metadata, they often employ simple random scalar generation methods to support filtered search. This leads to minimal variance in scalar column values, posing little challenge to filtered search in vector databases. Therefore, we choose the YFCC dataset that incorporates 192-dimensional

encoded image vectors and associated metadata. Since its tags support IN or contain operations, this metadata exhibits extremely high cardinality, reflecting the inherent complexity of scalar information in real-world data. To further enrich scalar attributes and support diverse filtering operations, we additionally generate two synthetic scalar columns based on the Zipf distribution tailored for equality and range queries.

3.2.2 Controllable Data Generation with Varied Dimension. Dimensionality adjustment aims to investigate how varying embedding dimensions affect retrieval behavior while preserving the distribution of the overall dataset for high fidelity. A naive approach is to duplicate vectors, which exactly preserves pairwise distances but can only scale dimensions by integer multiples and fails to introduce any structural variation. Alternatively, truncating or sampling dimensions leads to severe local distortions and low fidelity. To achieve smooth and globally consistent transformation, we adopt a linear mapping of the form $f(\mathbf{x}) = \mathbf{M}\mathbf{x}$, where $\mathbf{x} \in \mathbb{R}^d$ denotes the original feature vector and $\mathbf{M} \in \mathbb{R}^{k\times d}$ is a projection matrix with entries that maps the data from the d-dimensional source space to a k-dimensional target space. The choice of $\mathbf{M}$ critically determines the geometric fidelity of the transformed data. Fixed projection matrices (e.g, $m_{ij} = c$) often lack robustness across datasets, whereas random linear projection ((e.g, $m_{ij} \sim \mathcal{N}(0, 1/k)$) provides a statistically stable and theoretically grounded mechanism for adjusting dimensionality. We define this approach as Johnson–Lindenstrauss (JL)–based data synthesis, as it leverages the JL lemma to project data into a target space while bounding distortion. It introduces slight random perturbations while approximately preserving pairwise relationships [42], enabling controlled and generalizable evaluation of dimensionality effects. Figure 5 intuitively shows the effects of two different methods. It can be found that compared with random selection, the linear projection based on JL can keep the points near the same query point basically unchanged.

The Johnson–Lindenstrauss (JL) lemma [24] provides the theoretical foundation for the linear projection we use. It states that for any $0 < \varepsilon < 1$ and any set X of N points in $\mathbb{R}^d$, there exists a linear mapping $f : \mathbb{R}^d \to \mathbb{R}^k$ such that

$$(1-\varepsilon)\,\|u-v\|_2^2 \;\le\; \|f(u)-f(v)\|_2^2 \;\le\; (1+\varepsilon)\,\|u-v\|_2^2, \tag{3}$$

for all $u, v \in X$, provided that

$$k = O\left(\frac{\log N}{\varepsilon^2}\right). \tag{4}$$

This theorem guarantees that random linear projection approximately preserves pairwise distances when reducing the dimensionality of data. Although the JL lemma is conventionally applied to dimensionality reduction ($k < d$), the same random projection principle can be extended to dimensionality expansion ($k > d$), since the concentration property of Gaussian projections does not depend on the relative magnitudes of k and d. We further derive the theoretical error of random linear projection under arbitrary dimensionality transformations.

Theorem 3.1 (Expected Error Bound). *Let $X = \{x_1, \ldots, x_N\} \subset \mathbb{R}^d$ and $M \in \mathbb{R}^{k\times d}$ be a random Gaussian projection matrix with entries $m_{ij} \sim \mathcal{N}(0, 1/k)$. Then, with probability at least δ, all pairwise distances in the transformed space satisfy*

$$\left|\|f(u)-f(v)\|_2^2 - \|u-v\|_2^2\right| \;\le\; \varepsilon\,\|u-v\|_2^2, \tag{5}$$

for all $u, v \in X$, where the expected relative error bound is

$$\varepsilon(N,k,\delta) = \sqrt{\frac{4}{k}\left(2\ln N + \ln\frac{1}{\delta}\right)}. \tag{6}$$

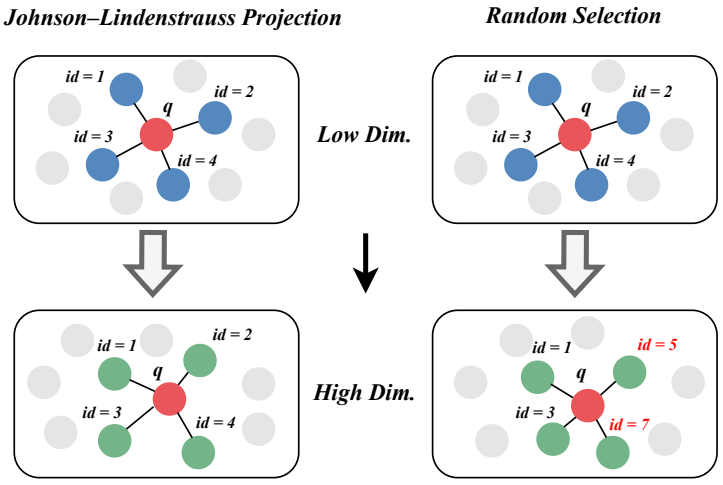


Fig. 5. Comparison between JL-based Projection and Random Selection during dimension upscaling (from low to high dim.). Note that the JL-based method maintains the same nearest neighbors (id 1-4).

Proof (Sketch). For any fixed vector $v = u - v' \in \mathbb{R}^d$, we have $\|Rv\|_2^2 = \|v\|_2^2 \cdot \frac{1}{k}\sum_{i=1}^{k} Z_i$, where $Z_i \sim \chi^2(1)$ are i.i.d. random variables with $\mathbb{E}[Z_i] = 1$. By the standard concentration bound of the chi-square distribution,

$$\Pr\left(\left|\frac{1}{k}\sum_{i=1}^{k} Z_i - 1\right| \geq \varepsilon\right) \leq 2\exp\left(-\frac{k\varepsilon^2}{4}\right).$$

Applying the union bound over all $\binom{N}{2}$ pairs [25] and setting the total failure probability to δ yields

$$2\binom{N}{2}\exp\left(-\frac{k\varepsilon^2}{4}\right) \leq \delta,$$

which gives

$$\varepsilon(N,k,\delta) = \sqrt{\frac{4}{k}\ln\frac{2\binom{N}{2}}{\delta}} \approx \sqrt{\frac{4}{k}\left(2\ln N + \ln\frac{1}{\delta}\right)}. \tag{7}$$

This establishes the claimed probabilistic error bound. Note that a larger δ relaxes the distance preservation guarantee, thereby allowing for larger deviations with higher probability. This holds for both dimensionality reduction and expansion since the concentration property depends only on k, not on d. As a sufficient condition for neighborhood consistency, the top-K structure is guaranteed to remain unchanged if

$$\frac{d_{K+1}}{d_i} > \sqrt{\frac{1+\varepsilon}{1-\varepsilon}}, \quad \forall i \leq K, \tag{8}$$

where d_i denotes the i-th nearest neighbor distance. Next, we explain this theory through an example.

EXAMPLE 2. *Consider $N = 10^6$ points projected to $k = 2000$ dimensions with a failure probability $\delta = 0.01$. Using Theorem 3.1, the expected relative error is $\varepsilon(N,k,\delta) \approx 0.254$. This means that, for any*

pair of points u, v, the squared distance after projection satisfies the following equation:

$$\|f(u) - f(v)\|_2^2 \in [(1-\varepsilon)\|u-v\|_2^2, (1+\varepsilon)\|u-v\|_2^2],$$

i.e., the distance changes by at most roughly 25% in squared magnitude. Taking square roots to interpret in terms of actual distances gives an approximate deviation of $\sqrt{1-\varepsilon} \approx 0.87$ to $\sqrt{1+\varepsilon} \approx 1.13$, so pairwise distances are preserved within about ±13% in length.

Consequently, the ordering of the top-K nearest neighbors remains unchanged as long as the ratio between the $(K+1)$-th distance d_{K+1} and the i-th distance d_i satisfies the following equation:

$$\frac{d_{K+1}}{d_i} > \sqrt{\frac{1+\varepsilon}{1-\varepsilon}} \approx 1.30.$$

In other words, in the worst case, as long as the distance of the $(K+1)$-th point is at least 30% larger than the i-th nearest neighbor distance, their relative order is guaranteed to remain unchanged after projection.

3.2.3 Controllable Data Generation with Varied Scale. Controlled scaling is crucial for benchmarking how indexing, filtering and retrieval performance of different database systems varies with data size. To analyze different approaches, we formally define *scale generation* as the process of increasing the size of a heterogeneous dataset containing both vector and scalar attributes, while preserving the joint distribution between vectors and scalars. A straightforward approach is to directly duplicate the dataset, but this is not feasible because modern databases can easily perform a *distinct* operation to automatically remove duplicate entries. Alternatively, copying vector features while reassigning scalar attributes can avoid exact duplicates but destroys the conditional relationships between vector and scalar. To address these issues, we adopt a strategy of copying the data while incorporating small perturbations to the vectors, named *DisturbGen* approach. Such a approach generates non-duplicate samples while maintaining the overall relationships between vector and scalar, effectively preserving the joint distribution in the expanded dataset.

To quantitatively assess the preservation of conditional distributions, we introduce a new metric, *Average Maximum Mean Discrepancy (AMMD)*, defined as follows:

*Definition 3.2 (**Average Maximum Mean Discrepancy** AMMD).* To evaluate conditional distribution alignment, we group samples by scalar values and compute the squared Maximum Mean Discrepancy [16] [12] (MMD) within each group, we can assess whether the conditional distributions are preserved. The metric is defined as:

$$\text{AMMD} = \frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} \text{MMD}^2(X_r, Y_r), \tag{9}$$

where $\mathcal{R}$ denotes the set of scalar attribute values that appear in both datasets, and X_r, Y_r denote subsets of samples corresponding to scalar condition r.

3.3 Filtered Query Generator

To understand the behavior of filtered vector search in practice, we conducted a large-scale statistical analysis over 3000 filtered queries executed on `Pgvector`. These queries were synthetically generated based on the YFCC10M dataset: we randomly sampled 1,000 base vectors and, for each, produced three distinct scalar predicates to cover a diverse range of query characteristics. Specifically, the selectivity of these queries range from 33.87% to 0.01%, and the filter correlations span from 18.57% to 0.01%, providing a representative sample of real-world complexity. As shown in Figure 6, the recall and latency of different queries exhibit substantial variation, revealing that search performance is highly sensitive to query characteristics. Two major factors are particularly

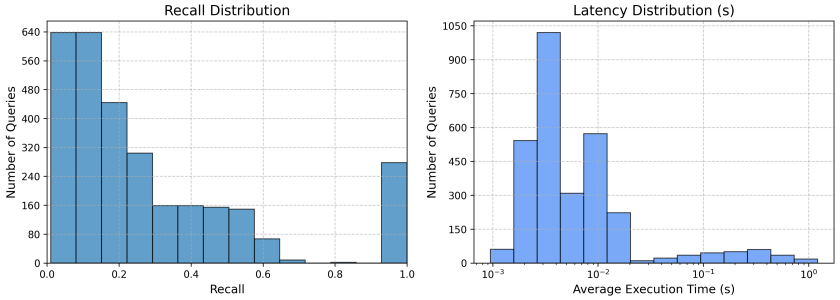


Fig. 6. Distribution of recall and latency in Pgvector.

important: the *selectivity*, which determines the proportion of data remaining after scalar filtering, and the *filter correlation*, which reflects the dependency between scalar predicates and vector similarity (See definition 2.2). To systematically control these factors, we design a **Filtered Query Generator** that enables distribution-aware query generation. It constructs parameterized filtered queries by adapting filtering conditions to match specified selectivity and correlation levels. This design provides a flexible mechanism for synthesizing diverse filtered search workloads, supporting the controlled evaluation of vector databases under varying hybrid search conditions. Below, we describe methods for controlling selectivity and filter correlation, respectively.

3.3.1 Controllable Selectivity. Our workload employs three types of predicates (equality, range, and containment) to construct filtered queries. For the simplicity of benchmarking, we select one type of predicate for each query. We control the selectivity by varying the filter parameters based on the distribution by the following three steps. First, we perform a statistical analysis of the dataset's distribution to get the value counts and selectivity of all scalar attributes. Second, given a target selectivity s_f, we randomly select a scalar attribute A, and gradually constitute the target rate by selecting its values in a descending order based on each value's selectivity. Concerning equality and containment predicates, we combine multiple values using OR semantics in filtered vector search to reach a target rate. Concerning range predicate, our approach finds the range upper bound ub by accumulating the value counts in an ascending order until the target selectivity is reached, then it produces the range predicate $A < ub$. Or we can accumulate the value counts in a descending order to produce lower bound $A > lb$. While exact cardinality is feasible for small datasets, we employ cardinality estimators as a scalable optimization for large-scale datasets with complex joint distributions, where exact pre-computation for all predicate combinations is computationally expensive. This allows our generator to efficiently approximate the target selectivity s_f with minimal overhead. Suppose the dataset contains a scalar attribute country with the distribution USA: 60%, UK: 20%, France: 10%, Others: 10%. If the goal is to construct a query with a low selectivity (e.g., 10%), the generator may select a predicate with country = 'France', which keeps only a small portion of the dataset. Conversely, if the target is a high selectivity (e.g., 70%), the generator may select country = 'USA' or 'UK', thus retaining most of the dataset. This ensures that queries can systematically span different levels of selectivity.

3.3.2 Controllable Filter Correlation. We develop two methods for controlling the filter correlation, the *selection* method and the *generation* method. The former one handles the cases where the input dataset already contains the vector and structured data, and the latter one tackles the cases where only the vector data is available.

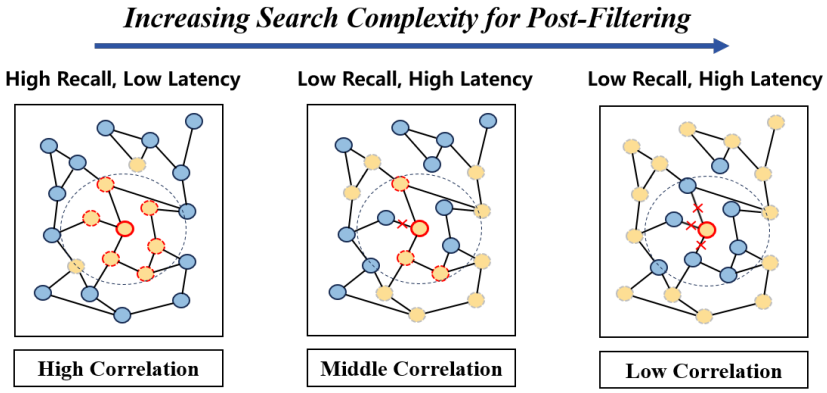


Fig. 7. Impact of Data Distribution on Search Difficulty with Respect to the Varied Correlation.

Given a target correlation $c_f^{(k)}$, the *selection* method first randomly selects a query vector q, then calculates its top-K' (K'>K) results. The generator then obtain the value counts and corresponding ratio. Among the top-K' results, we can also accumulate the ratio to reach the $c_f^{(k)}$ using OR semantics for equality or containment or range filters. Consider a query vector representing a romantic movie. If its top-100 nearest neighbors contain 80% romantic movies with genre = 'Romance', while other scalar attributes (e.g., 'Horror') account for only 20%. We can directly select the 'Romance' as its scalar value for a correlation of 0.8, or selecting 'Horror' as the scalar value for a correlation of 0.2.

Next, we present the *generation* method. Our goal is to generate queries with different levels of correlation between vectors and scalar predicates (*low*, *medium*, and *high*), given only a vector dataset as input. To achieve this, we construct a joint vector–scalar distribution in two steps. The first step partitions the vector dataset into K clusters using K-means, and set $K = 3$ to correspond to the above three correlation levels, which ensures that each cluster contains a sufficiently large number of vectors (approximately $N/3$). Then, we assign scalar values to vectors according as follows: (1) vectors in the first cluster are assigned scalar label "A", (2) vectors in the second cluster are assigned label "B", and vectors in the third cluster are assigned labels "C" and "D" uniformly at random. The second step generates queries by selecting cluster centroids as query vectors and pairing them with specific scalar values. Specifically, we generate three types of queries as follows: (1) *High correlation*: selecting the first cluster centroid with scalar "A", so that the data vectors near the query are highly likely to satisfy the scalar predicate; (2) *Low correlation*: selecting the first cluster centroid with scalar "B", so that the data vectors near the query are unlikely to satisfy the scalar predicate; (3) *Medium correlation*: selecting the third cluster centroid with scalar "C" or "D", where approximately half of the nearest neighbors satisfy the predicate.

3.4 Workload Executor

3.4.1 End-to-End Framework. We present the end-to-end benchmarking workflow designed to evaluate vector databases under different workloads. As illustrated in Figure 8, the framework consists of six consecutive phases—*Initialization*, *Query Execution*, *Concurrent Execution*, *Incremental Load*, *Update*, and *Deletion*. These phases are grouped into two major scenarios: *Filtered Search*, which assesses query quality, latency, and scalability; and *Update Scenario*, which evaluates index construction and maintenance under dynamic data. Each phase is associated with specific

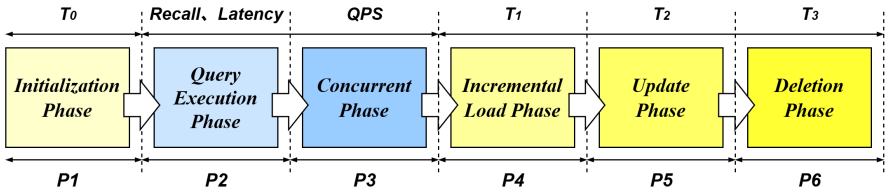


Fig. 8. Execution Phases and Associated Metrics.

performance metrics (e.g., latency, recall, throughput) to provide a comprehensive view of system behavior. We next detail the execution rules of each phase and the metrics used in our evaluation.

3.4.2 Execution Rules and Metrics. The end-to-end workflow enforces a sequential execution of six phases to emulate realistic database operations under both query and update workloads. And the evaluation metrics are designed to capture both query effectiveness and system efficiency under diverse scenarios.

- **Initialization Phase.** Insert the initial a proportion of the dataset and build the vector index. The index construction latency is recorded as T_0.
- **Query Execution Phase.** Perform filtered search queries over the indexed dataset. We record the average query latency (Latency) and measure average query accuracy in terms of recall (Recall).
- **Concurrent Phase.** Concurrently execute multiple filtered search queries to evaluate the system's concurrency capabilities. The throughput of the system is measured in queries per second (QPS).
- **Incremental Load Phase.** Insert the remaining $(1 - a)$ proportion of the dataset and trigger index maintenance. The sum of data insertion and index maintenance time is measured as T_1.
- **Update Phase.** Update a proportion b of data by modifying both vectors and scalar attributes, requiring maintenance of both vector and scalar indexes. The latency is recorded as T_2.
- **Deletion Phase.** Delete a proportion c of the dataset and measure the delay of this operation, reported as T_3.

Consequently, we propose a unified metric, named V_{rank}, which employs the average relative ranking to quantify the overall end-to-end performance of different vector databases:

$$V_{\text{rank}}(\text{VDB}) = \sum_{i=1}^{6} RelativeRank(P_i, \text{SUT})/6 \tag{10}$$

where VDB denotes a tested vector database, P_i represents the i-th stage in the E2E workflow, and SUT refers to the systems under test. The function $RelativeRank(P_i, \text{SUT})$ computes the relative ranking of the system with respect to all evaluation metrics at stage P_i.

4 Experiments

Experimental Environment. We deploy VecBench on a server with an Intel(R) Xeon(R) Gold 6330 CPU (28 cores, 56 threads, 3.10 GHz, 42MB cache). The server is equipped with 251 GiB of system memory and 8 GiB of swap space.

Benchmark Configuration. We conduct our benchmark experiments on four well-known vector databases, including **PostgreSQL 16.1** with *Pgvector 0.8.1*, *Milvus 2.5.0*, *Qdrant 1.13.6* and *Weaviate 1.35.1*. All systems are deployed in a standalone mode. The benchmark dataset is based on YFCC, originally containing 100K 192-dimensional vectors, and expanded by our methods to 1M and 10M

samples with 1920 dimensions. The query workload is randomly generated from the raw data with 3,000 queries.

Index Configuration. Our dataset consists of scalar and vector columns, where scalar columns cover three representative data types: *categorical (equality)*, *numerical (range)*, and *set-based (containment)* attributes. Accordingly, we employed different index mechanisms suited to each database: **B+-tree** index for scalar data in Pgvector, **inverted indexes** [37] in Milvus and Weaviate, and **payload indexes** in Qdrant. For vector indexing, we evaluated two representative types: **HNSW** and **IVFFLAT**. The HNSW index was configured with parameters $M = 16$, *ef_construction* = 64, and query-time *ef_search* = 100. For IVFFLAT, the number of clusters (*nlist*) was determined adaptively: *rows / 1000* for datasets up to 1M entries and $\sqrt{rows}$ for larger ones. The corresponding query parameter *nprobe* was set to 1% of *nlist*.

Systems Under Test (SUT). We evaluate four vector databases with various techniques as follows:

(1) Pgvector. Pgvector is a PostgreSQL extension that introduces a native vector data type and similarity operators. Vectors are stored as regular table columns in PostgreSQL's row-oriented storage and can be directly combined with relational predicates. It supports well-known ANN indexes such as IVFFLAT and HNSW, managed by PostgreSQL's storage engine and buffer/cache. For filtered queries, Pgvector relies on PostgreSQL's cost-based optimizer to choose between pre-filtering and post-filtering.

(2) Milvus. Milvus is a native vector database designed for large-scale similarity search, organized around a distributed microservice architecture. Data are partitioned into segments, each storing its own vectors and indexes. Milvus supports a wide range of index types (FLAT, IVF, PQ, HNSW, DiskANN, etc.). For filtered search, Milvus performs cardinality estimation to predict predicate selectivity, and dynamically switches between pre-filtering and in-filtering to balance efficiency and accuracy.

(3) Qdrant. Qdrant is a Rust-based vector search engine optimized for low-latency retrieval, adopting a segment-oriented storage model. Its primary index is HNSW, complemented by payload indexes for scalar attributes, supporting both in-memory and persistent modes. For filtered queries, Qdrant also leverages cardinality estimation and adaptively selects the search strategy between pre-filtering and expanded-filtering.

(4) Weaviate. Weaviate is a Go-based vector database that stores both objects and vectors, allowing for the combination of vector search with structured filtering. It has implemented the hybrid index of ACORN [36] supporting expanded-filtering.

We selected these four SUTs to represent the two primary architectures: (i) relational-extended and (ii) native vector databases, ensuring a comprehensive evaluation of the design trade-offs discussed in Section 2. First, *Pgvector* represents the page-based, synchronous CRUD model of relational engines, while *Milvus* and *Qdrant* represent the segment-oriented, asynchronous native model. Second, these four systems collectively cover all four mainstream filtered search strategies: *pre-filtering* and *post-filtering* (Pgvector), *in-filtering* (Milvus), and *expanded-filtering* (Qdrant). We further include Weaviate with ACORN as a representative hybrid indexing approach to complement expanded-filtering, improving coverage of hybrid designs. Together, they provide a comprehensive and representative baseline for filtered vector search.

4.1 Overall Performance

Table 2 presents the end-to-end experimental results on YFCC 10M dataset and their corresponding rankings across seven evaluation dimensions. The workload comprises 3,000 queries (1,000 points per Equality, Range, and Containment predicate) to measure Recall, Latency, and QPS. Dynamic updates and deletions (P4-P6) are performed on a 2M-record scale. The compared methods include four vectors databases, each tested representative vector index types. The metrics (T_0, Recall,

Table 2. End-to-End Benchmarking Results of Six Filtered Vector Search Methods.

Method	T_0 (P1)		Recall (P2)		Latency (P2)		QPS (P3)		T_1 (P4)		T_2 (P5)		T_3 (P6)		V_{rank}
	Value	Rank	Value	Rank	Value	Rank	Value	Rank	Value	Rank	Value	Rank	Value	Rank	
Milvus-HNSW	**229.99**	**1**	0.993	2	0.0586	7	19.41	9	**146.29**	1	171.36	3	7.75	2	4.2
Milvus-IVFFLAT	425.39	2	0.970	5	0.0327	5	21.26	8	157.04	2	**132.96**	1	7.94	4	4.5
Pgvector-IVFFLAT	562.10	3	0.960	6	0.0828	8	101.83	3	495.09	4	622.06	6	**7.00**	**1**	5.2
Pgvector-IVFFLAT-ite	562.10	3	0.975	4	0.0842	9	87.65	4	495.09	4	622.06	6	7.00	1	5.2
Pgvector-HNSW-ite	6727.25	5	0.897	7	0.0159	2	455.19	2	5207.49	8	5547.22	8	7.82	3	5.8
Pgvector-HNSW	6727.25	5	0.257	10	**0.0020**	**1**	**1963.25**	**1**	5207.49	8	5547.22	8	7.82	3	6.0
Qdrant-HNSW*	—	5	**0.995**	**1**	0.0250	4	47.78	7	2523.77	6	785.43	7	10.12	6	6.0
Milvus-DiskANN	978.38	4	0.988	3	0.5869	10	7.05	10	186.92	3	145.90	2	8.92	5	6.2
Qdrant-HNSW	—	5	0.455	9	0.0200	3	51.17	5	640.64	5	441.96	4	11.12	7	6.3
Weaviate-ACORN	—	5	0.704	8	0.0473	6	50.24	6	3060.54	7	516.50	5	182.29	8	7.5

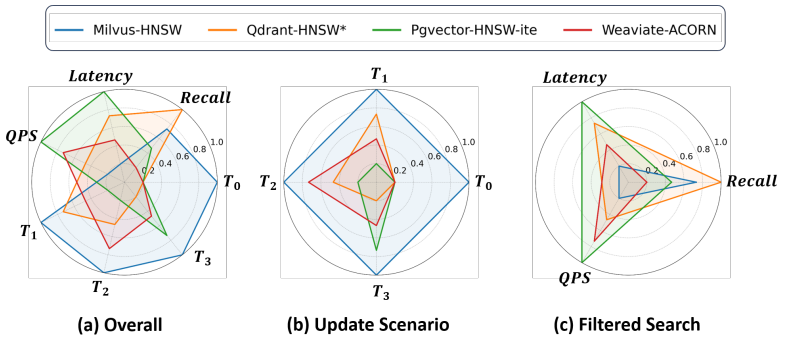


Fig. 9. Radar charts for representative methods of various systems in overall, update and filtered search scenario.

Latency, QPS, T_1, T_2, T_3) jointly reflect the systems' search efficiency, filtering capability, and scalability under hybrid workloads. Note that (1) Qdrant-HNSW* refers to HNSW with the expanded filtering that has additional shortcut edges and (2) we set NULL T_0 for Qdrant as it loads the data and creates the index at the same time, and we cannot record its index construction time.

Overall, **Milvus-HNSW** achieves the best comprehensive performance with the lowest dynamic data maintenance overhead and superior filtered search capability, resulting in the top rank with $V_{rank} = 4.2$. Its better performance reflects Milvus' incremental delta management for index maintenance and data updates. Surprisingly, **Milvus-HNSW** has the fast index construction time that even exceeds Milvus-IVFFLAT. In principle, IVFFLAT should have lower construction overhead than HNSW. According to its official documentation, we speculate that Milvus-HNSW may have employed parallel computing like SIMD [1] to speed up the construction process. Regarding Milvus-IVFFLAT, it has also good performance on update cases (e.g., P4 and P5). And Pgvector-IVFFLAT also excels at DML operations at P6 and P1. Besides, it has superior throughput, which suggests that its cost model prioritize low-latency execution. Moreover, its matured concurrent query processing engine gives the highest QPS. Unfortunately, Pgvector-HNSW has the lowest recall, indicating that its limitation on searching for target remote neighbors. However, increasing the iteration depth helps mitigate this (from 0.257 to 0.897) by broadening the search area. Moreover, it has the largest T_0 values of 6727.25s for constructing the HNSW index. Interestingly, Qdrant-HNSW* has the best recall in P2 phase because of its expanded filtering strategy. However, this leads to highest cost in index maintenance and data-update handling, leading to the lowest ranks. Furthermore, Qdrant-HNSW has low recall of 0.455 without the expanded filtering.

Figure 9 provides three radar charts to investigate the trade-offs of both static and dynamic cases with the end-to-end framework of VecBench. Specifically, it is clearly visible that **Milvus-HNSW**

Table 3. Filtered Search Results under Different Selectivity across Multiple Datasets.

Dataset	Method	High Selectivity						Middle Selectivity						Low Selectivity					
		Recall			Latency			Recall			Latency			Recall			Latency		
		E	R	C	E	R	C	E	R	C	E	R	C	E	R	C	E	R	C
YFCC 100K 192	Pgvector-HNSW	0.5800	0.5100	0.5100	0.0010	0.0011	0.0008	0.0900	0.1600	0.1500	0.0011	0.0010	0.0010	1.0000	1.0000	1.0000	0.0003	0.0003	0.0243
	Milvus-HNSW	1.0000	0.9800	1.0000	0.0072	0.0067	0.0068	1.0000	1.0000	1.0000	0.0101	0.0101	0.0103	1.0000	1.0000	1.0000	0.1281	0.1230	0.1546
	Qdrant-HNSW	1.0000	1.0000	1.0000	0.0158	0.0165	0.0158	1.0000	1.0000	1.0000	0.0120	0.0136	0.0133	1.0000	1.0000	1.0000	0.0095	0.0085	0.0108
	Qdrant-HNSW*	1.0000	1.0000	1.0000	0.0130	0.0161	0.0144	1.0000	1.0000	1.0000	0.0112	0.0131	0.0124	1.0000	1.0000	1.0000	0.0093	0.0086	0.0100
	Pgvector-IVFFLAT	0.5600	0.3800	0.5200	0.0019	0.0012	0.0012	0.4900	0.0800	0.5700	0.0025	0.0005	0.0018	1.0000	1.0000	1.0000	0.0004	0.0004	0.0288
	Milvus-IVFFLAT	0.6300	0.4700	0.6400	0.0063	0.0058	0.0059	0.5800	0.6800	0.7300	0.0067	0.0071	0.0072	1.0000	1.0000	1.0000	0.0839	0.0838	0.0977
YFCC 100K 1920	Pgvector-HNSW	1.0000	1.0000	1.0000	0.2982	0.2732	0.2639	1.0000	1.0000	1.0000	0.0812	0.0792	0.0735	1.0000	1.0000	1.0000	0.0011	0.0009	0.0187
	Milvus-HNSW	1.0000	1.0000	1.0000	0.0096	0.0101	0.0090	1.0000	1.0000	1.0000	0.0163	0.0155	0.0158	1.0000	1.0000	1.0000	0.0644	0.0647	0.0682
	Qdrant-HNSW	1.0000	0.9900	0.9900	0.0210	0.0197	0.0184	0.7900	0.4200	0.5100	0.0208	0.0196	0.0187	1.0000	1.0000	1.0000	0.0161	0.0152	0.0161
	Qdrant-HNSW*	1.0000	1.0000	1.0000	0.0255	0.0307	0.0232	1.0000	1.0000	1.0000	0.0258	0.0249	0.0224	1.0000	1.0000	1.0000	0.0195	0.0162	0.0169
	Pgvector-IVFFLAT	0.4100	0.2900	0.4500	0.0025	0.0021	0.0032	1.0000	1.0000	1.0000	0.0804	0.0789	0.0943	1.0000	1.0000	1.0000	0.0011	0.0011	0.0277
	Milvus-IVFFLAT	0.8300	0.6400	0.8200	0.0072	0.0069	0.0068	0.5200	0.5200	0.7000	0.0085	0.0099	0.0102	1.0000	1.0000	1.0000	0.1309	0.1295	0.1447
YFCC 1M 192	Pgvector-HNSW	0.5500	0.5700	0.5100	0.0043	0.0016	0.0016	0.1700	0.1100	0.0700	0.0030	0.0030	0.0023	1.0000	0.0100	1.0000	0.0025	0.0025	0.2723
	Milvus-HNSW	0.9900	0.9800	0.9800	0.0088	0.0072	0.0073	0.9700	0.9800	0.9800	0.0114	0.0125	0.0114	1.0000	1.0000	1.0000	0.1795	0.1743	0.1100
	Qdrant-HNSW	0.9700	0.9700	0.9700	0.0116	0.0134	0.0137	0.4500	0.3000	0.4600	0.0199	0.0216	0.0202	1.0000	1.0000	1.0000	0.0115	0.0135	0.0153
	Qdrant-HNSW*	1.0000	1.0000	1.0000	0.0137	0.0167	0.0179	1.0000	1.0000	1.0000	0.0191	0.0221	0.0224	1.0000	1.0000	1.0000	0.0136	0.0143	0.0157
	Pgvector-IVFFLAT	0.3300	0.3200	0.3500	0.0010	0.0008	0.0008	0.2700	0.1000	0.6800	0.0013	0.0007	0.0023	1.0000	0.0200	1.0000	0.0016	0.0024	0.2060
	Milvus-IVFFLAT	0.6100	0.6200	0.6400	0.0072	0.0066	0.0065	0.5100	0.5000	0.6200	0.0076	0.0081	0.0078	0.5300	0.7900	0.6200	0.0953	0.0898	0.0899
YFCC 1M 1920	Pgvector-HNSW	0.5500	0.5500	0.5200	0.0073	0.0041	0.0031	0.1300	0.1000	0.0700	0.0039	0.0039	0.0030	1.0000	1.0000	1.0000	0.0135	0.0133	0.1138
	Milvus-HNSW	1.0000	1.0000	1.0000	0.0389	0.0355	0.0375	1.0000	1.0000	1.0000	0.0567	0.0598	0.0702	1.0000	1.0000	1.0000	0.2138	0.2445	0.2358
	Qdrant-HNSW	0.9800	0.9800	0.9900	0.0249	0.0230	0.0240	0.4200	0.3400	0.3800	0.0200	0.0196	0.0220	1.0000	1.0000	1.0000	0.0174	0.0183	0.0215
	Qdrant-HNSW*	1.0000	1.0000	1.0000	0.0360	0.0360	0.0376	1.0000	1.0000	1.0000	0.0333	0.0326	0.0326	1.0000	1.0000	1.0000	0.0214	0.0193	0.0202
	Pgvector-IVFFLAT	0.2500	0.2400	0.2600	0.0047	0.0075	0.0109	0.3700	0.0900	0.2000	0.0068	0.0066	0.0065	1.0000	1.0000	1.0000	0.0168	0.0207	0.1371
	Milvus-IVFFLAT	0.6400	0.5800	0.7000	0.0136	0.0085	0.0106	0.8400	0.7000	0.9200	0.0153	0.0127	0.0151	1.0000	1.0000	1.0000	0.1919	0.1931	0.2949
YFCC 10M 192	Pgvector-HNSW	0.6100	0.5600	0.8200	0.0012	0.0010	0.0010	0.1500	0.1600	0.1100	0.0014	0.0007	0.0009	1.0000	0.0100	0.0200	0.0102	0.0009	0.0012
	Milvus-HNSW	1.0000	1.0000	1.0000	0.0089	0.0078	0.0089	1.0000	1.0000	1.0000	0.0127	0.0112	0.0153	1.0000	1.0000	1.0000	0.2884	0.2379	0.3918
	Qdrant-HNSW	0.9700	0.9900	1.0000	0.0143	0.0142	0.0153	0.2400	0.4300	0.2500	0.0122	0.0149	0.0124	1.0000	1.0000	1.0000	0.0130	0.0137	0.0171
	Qdrant-HNSW*	1.0000	1.0000	1.0000	0.0322	0.0283	0.0340	1.0000	1.0000	1.0000	0.0207	0.0223	0.0309	1.0000	1.0000	1.0000	0.0161	0.0176	0.0203
	Pgvector-IVFFLAT	0.9100	0.9400	0.9600	0.0128	0.0085	0.0083	0.6400	0.7600	0.8300	0.0076	0.0057	0.0088	1.0000	0.0800	0.0300	0.0184	0.0146	0.0182
	Milvus-IVFFLAT	0.7300	0.7100	0.7600	0.0106	0.0098	0.0095	0.7700	0.9900	0.7900	0.0110	0.0115	0.0124	0.9700	0.9700	0.9200	0.1135	0.1256	0.2228
YFCC 10M 1920	Pgvector-HNSW	0.2600	0.2200	0.3200	0.0086	0.0018	0.0018	0.0500	0.1000	0.0600	0.0083	0.0028	0.0012	1.0000	0.0100	1.0000	0.4077	0.0040	1.1028
	Milvus-HNSW	1.0000	1.0000	1.0000	0.2046	0.1592	0.1499	1.0000	1.0000	1.0000	0.4378	0.3538	0.3902	1.0000	1.0000	1.0000	1.4654	1.3972	1.4551
	Qdrant-HNSW	0.9800	0.9800	1.0000	0.0307	0.0341	0.0328	0.1500	0.2500	0.1500	0.0290	0.0307	0.0291	1.0000	0.9500	1.0000	0.0454	0.0270	0.0445
	Qdrant-HNSW*	1.0000	1.0000	1.0000	0.5312	0.2588	0.2031	0.9400	1.0000	1.0000	0.1404	0.1641	0.1278	1.0000	0.9500	1.0000	0.0670	0.0608	0.0615
	Pgvector-IVFFLAT	0.9700	0.9200	0.9900	0.0790	0.0444	0.0427	0.5300	0.8000	0.8000	0.0646	0.0445	0.0525	1.0000	0.0900	1.0000	0.4240	0.0810	1.1193
	Milvus-IVFFLAT	0.9100	0.9300	0.9200	0.0438	0.0385	0.0390	0.9800	1.0000	0.9900	0.0454	0.0440	0.0525	1.0000	1.0000	1.0000	0.8853	0.9759	1.1716

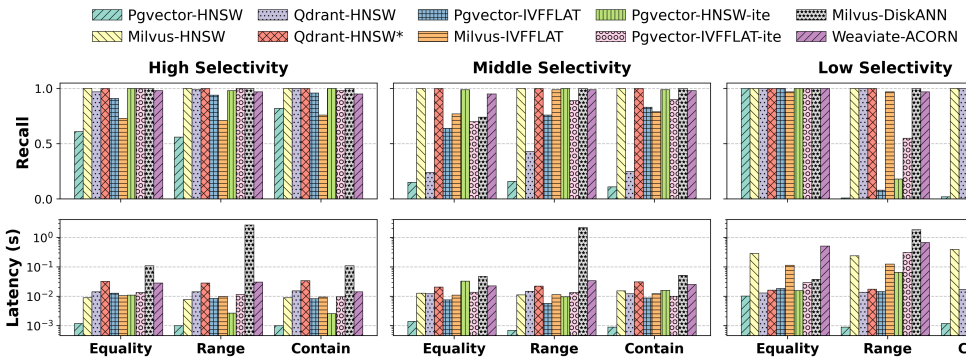


Fig. 10. Search Performance under Different Selectivity (based on YFCC-10M-192).

covers the largest area in sub-graph (a), the same result is shown in (b), while the area is extremely small in (c). Such observation indicates that since Milvus-HNSW performs extremely well in the update scenario, it achieves the overall best results even though it is not prominent in the filtered search. Moreover, we observe that in sub-graph (c), Pgvector-HNSW.ite and Qdrant-HNSW* both have large areas but focus on different aspects. **Pgvector-HNSW.ite** achieves the best latency and QPS but it suffers from low recall, showing its cost model prioritizes execution speed over retrieval quality. While Qdrant-HNSW* achieves highest recall with balanced throughput, reflecting its design for accuracy and search stability. In summary, the overall performance indicate that Milvus best balances recall and maintenance overhead, Pgvector excels in throughput-oriented workloads, and Qdrant offers flexible filtering at the expense of dynamic efficiency.

Table 4. Search Performance under Varied Filter Correlation.

Dataset	Method	Low		Middle		High	
		Recall	Latency	Recall	Latency	Recall	Latency
YFCC 100K 192	Pgvector-HNSW	0.0000	0.0012	0.4900	0.0008	0.9000	0.0012
	Milvus-HNSW	0.9400	0.0215	1.0000	0.0073	0.9900	0.0081
	Qdrant-HNSW	1.0000	0.0166	1.0000	0.0136	1.0000	0.0159
	Qdrant-HNSW*	1.0000	0.0174	1.0000	0.0143	1.0000	0.0180
	Pgvector-IVFFLAT	0.0000	0.0021	0.9700	0.0022	0.3900	0.0012
	Milvus-IVFFLAT	0.1100	0.0158	0.9600	0.0069	0.4500	0.0068
YFCC 1M 192	Pgvector-HNSW	0.0000	0.0037	0.5000	0.0031	0.9600	0.0034
	Milvus-HNSW	1.0000	0.0403	0.9900	0.0094	1.0000	0.0072
	Qdrant-HNSW	0.8600	0.0125	0.5400	0.0113	1.0000	0.0141
	Qdrant-HNSW*	1.0000	0.0156	1.0000	0.0167	1.0000	0.0226
	Pgvector-IVFFLAT	0.1000	0.0173	0.9900	0.0172	0.7800	0.0066
	Milvus-IVFFLAT	0.3000	0.0245	0.9500	0.0105	0.8100	0.0096
YFCC 10M 192	Pgvector-HNSW	0.0000	0.0035	0.9700	0.0023	0.9300	0.0032
	Milvus-HNSW	0.9900	0.1029	0.6200	0.0083	0.9900	0.0100
	Qdrant-HNSW	0.8800	0.0171	0.4000	0.0141	0.9600	0.0190
	Qdrant-HNSW*	0.9200	0.0309	0.7800	0.0187	1.0000	0.0288
	Pgvector-IVFFLAT	0.1000	0.1809	1.0000	0.1613	0.7100	0.0557
	Milvus-IVFFLAT	0.2700	0.0361	0.9900	0.0116	0.6200	0.0114

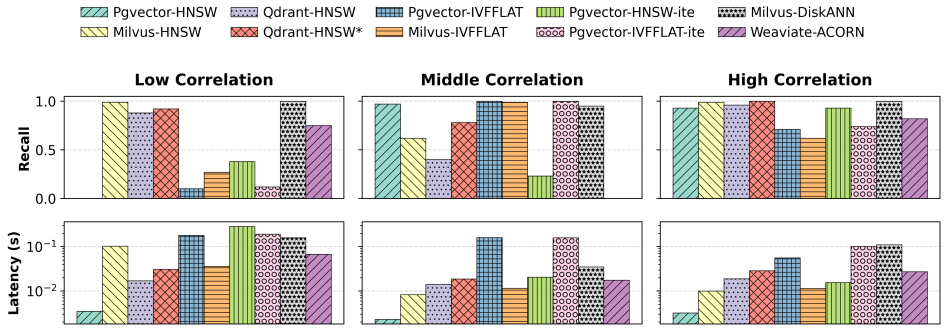


Fig. 11. Search Performance under Different Filter Correlation (based on YFCC-10M-192).

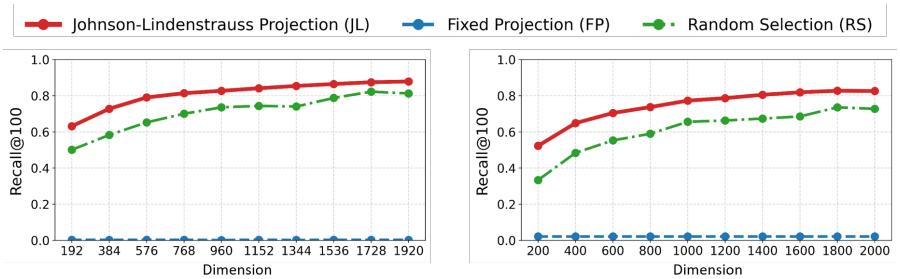
4.2 Evaluation of Filtered Search Strategies

4.2.1 Varying Selectivity. Table 3 summarizes the recall and latency results across multiple datasets with different scales and vector dimensions. We define three levels of selectivity, namely **High** ($> 50\%$), **Middle** ($1\% < X < 50\%$), and **Low** ($< 1\%$), corresponding to different selectivity levels of scalar filtering. Three representative filtering operations are adopted: **E**, **R**, and **C**, standing for *equality*, *range*, and *contain* conditions, respectively.

We observe that when the selectivity is low, all databases achieve nearly perfect recall (close to 1.0) while experiencing the highest latency. This is because they all employ pre-filtering in this case. An exception is the Milvus-IVFFLAT's bad performance on YFCC-1M-192. The main reason is it leverages threshold-based method to obtain the wrong size of intermediate results, hence the selection of post-filtering strategy. As the selectivity increases from middle to high, the recall is lower and latency decreases because of the utilization of vector indices. We also notice that Qdrant-HNSW* always achieves the best recall and relatively-low latency with its expanded filtering. Besides, Milvus-HNSW also maintains a consistently high recall rate across all filtering thresholds at the cost of higher latency. Pgvector exhibits low latency with its cost-based strategy.

Table 5. Evaluation of Data Generation with two datasets.

Dataset	Method	DCR	AMMD	Overhead (s)
YFCC	RandomGen	2.278	4.1×10^{-3}	0.36
	FrequencyGen	2.278	4.1×10^{-3}	0.38
	DisturbGen	2.279	7.6×10^{-4}	3.59
IMDB	RandomGen	5.25×10^{-2}	1.7×10^{-2}	1.02
	FrequencyGen	5.25×10^{-2}	1.7×10^{-2}	2.03
	DisturbGen	5.88×10^{-2}	1.5×10^{-3}	8.14



(a) Recall at 100K YFCC **(b) Recall at 100K IMDB**

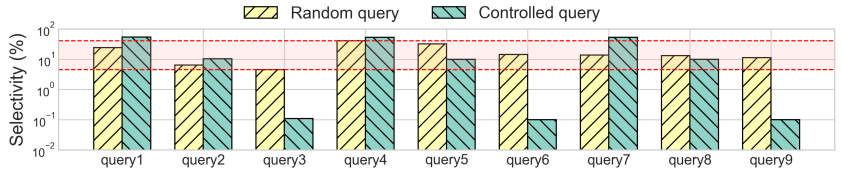
Fig. 12. Evaluation of Dimension Scaling Methods.

However, it lacks the consideration of recall optimization. Another observation is that the overall trend indicates that VecBench can challenge the performance of existing vector databases with larger datasets and higher dimensionality. Particularly, they leads to higher latency (up to 10x). As for different filter types, we find that *contain* filter poses the largest challenge for the vector databases, especially for low selectivity.

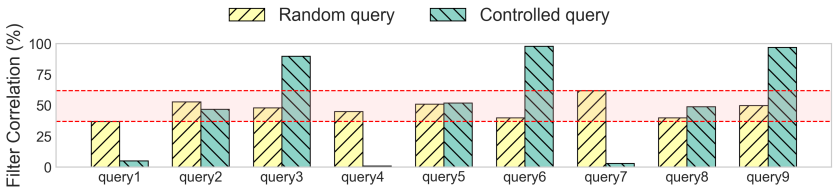
4.2.2 Varying Filter Correlation. Table 4 summarizes the recall and latency results under various filter correlation (i.e.,, we define three levels for better illustration, namely, **High**= 1, **Middle** = 0.5, **Low** = 0). Overall, the results show that vector databases perform disparately regarding different filter correlations. For instance, Pgvector has extremely low recall (0 or 0.1) on low correlation due to its limited filtered search. Milvus-HNSW can improve the recall with in-filtering search on low correlation but may have trouble to achieve high recall in middle correlation. A side observation is that Milvus-HNSW has also the highest latency concerning low correlation. The main reason is that in-filtering search spends more time exploring irrelevant candidates. Qdrant-HNSW displays a significant drop in recall under middle correlation because post filtering tends to break graph traversal paths when intermediate nodes fail to meet scalar predicates, preventing the search from reaching relevant neighbors. Nevertheless, Qdrant-HNSW* achieves 100% recall in nearly all cases with its expanded filtering. Another observation is that larger datasets amplify the performance gap under different correlation, where recall fluctuates more sharply and latency increases substantially. For example, when YFCC 1M is augmented to 10M, Milvus-HNSW's recall decreases from 0.99 to 0.62, and Pgvector-IVFFLAT increases the latency by 10x.

4.3 Ablation Study of VecBench

(1) Data Generation. We study the data generation methods of VecBench in two aspects. For dimension scaling, we compare our JL-based projection method with two baselines, Random Selection (RS) and Fixed Projection (FP). Unlike JL or RS, FP employs a fixed projection matrix with



(a) Query Selectivity



(b) Query Filter Correlation

Fig. 13. Evaluation of Query Selectivity and Filter Correlation.

constant values for all samples to achieve mapping. For size scaling, we contrast our disturbance-based generation method (**DisturbGen**) against two baselines, RandomGen, and FrequencyGen. We employ three metrics, Distance to Closest Record (DCR) for quantifying vector distribution quality, AMMD (See definition 3.2) for evaluating filtered vector search, and the generation overhead in seconds. Note that trivially copying the data for augmentation will not work for the purpose of benchmarking as it can be easily recognized as duplication and be bypassed by existing vector databases.

Table 5 demonstrates that DisturbGen achieves the best performance. By introducing small random perturbations to vectors while keeping scalar attributes fixed, it yields highest DCR and lowest AMMD. Although this results in a slight increase in overhead due to additional vector-level computation operations (e.g., noise injection and normalization) compared to the simple scalar shuffling, the cost remains within a few seconds for millions of points. In contrast, RandomGen and FrequencyGen, which reuse identical vectors with shuffled scalars, break the vector–scalar dependency, leading to smaller DCR and larger AMMD. Therefore, the controlled-noise strategy of DisturbGen provides the best trade-off between statistical fidelity and diversity, making it a simple yet effective technique for large-scale vector data augmentation. Figure 12 presents the evaluation results of dimension scaling methods, indicating that our JL method achieves better performance than the other two baselines. As the dimension increases, the recall of the queries' top-K also increases, meaning that our method is also applicable to high-dimensional vectors.

(2) Query Generation. We compare the query generation methods of VecBench with a randomized method [36] that randomly selects scalar parameters for each filter. As defined in Section 4.2, we categorize selectivity and filter correlation into three levels (High, Middle, and Low). Figure 13a depicts the distributions of the nine specific query instances used for our benchmark experiments. It is clearly visible that the random queries have a narrow range (between 5% and 33%), covering only scenarios with middle selectivity. While the controlled queries in VecBench are specifically chosen to span the predefined levels, covering a wide range from 0.1% to over 55%. Figure 13b illustrates the distribution of filter correlation. Random queries are clustered between 37% and 62%, while the nine instances in VecBench are distributed across the High, Middle, and Low correlation levels (ranging from 1% to 98%), posing much higher search complexity. It is worth noting that while we use these representative instances for benchmarking, VecBench is capable of generating queries with even wider or tighter ranges to satisfy diverse evaluation needs.

Table 6. Benchmark Results on IMDB Datasets.

Dataset	Method	Recall	Latency	QPS
IMDB 100K 2000	Pgvector-HNSW	0.7080	0.0018	556.35
	Milvus-HNSW	0.9670	0.0130	76.91
	Qdrant-HNSW	0.9510	0.0481	20.78
	Qdrant-HNSW*	0.9920	0.0502	19.92
	Pgvector-IVFFLAT	0.8360	0.0035	288.42
	Milvus-IVFFLAT	0.7930	0.0311	32.18
IMDB 1M 2000	Pgvector-HNSW	0.7400	0.0130	76.44
	Milvus-HNSW	0.9930	0.0129	77.83
	Qdrant-HNSW	0.8649	0.0557	17.95
	Qdrant-HNSW*	0.9900	0.0775	12.90
	Pgvector-IVFFLAT	0.8920	0.0141	71.03
	Milvus-IVFFLAT	0.8454	0.0396	25.22

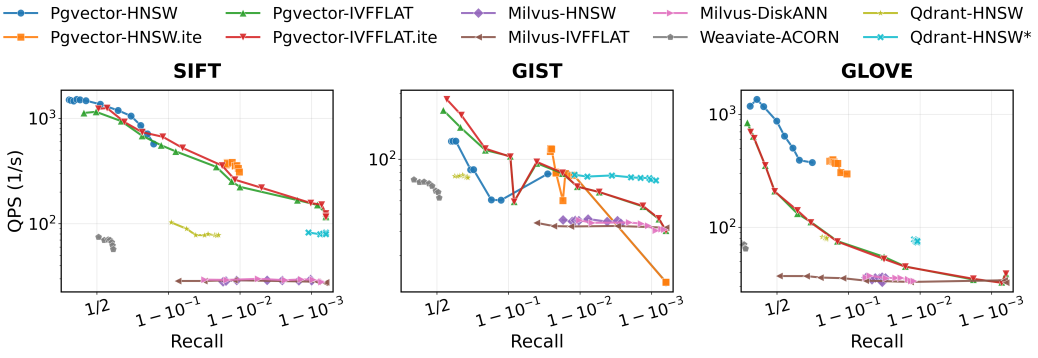
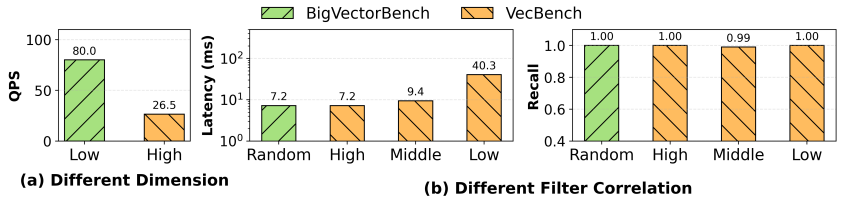


Fig. 14. QPS vs. Recall performance across different datasets.

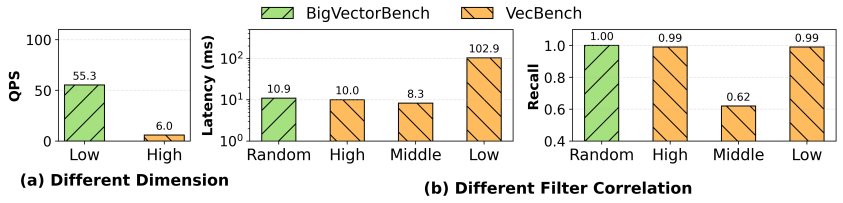
(3) Dataset Diversity. In addition to YFCC datasets, we extended our evaluation to **IMDB, SIFT, GloVe, and GIST**, covering diverse distributions and dimensionalities. As shown in Table 6, we find that **Milvus-HNSW** shows the best overall performance on IMDB datasets, producing the consistent results with YFCC datasets. Moreover, Figure 14 shows that the performance trends and rankings are highly sensitive to data characteristics. While Milvus-HNSW remains the most stable across all datasets, we observe that Pgvector-HNSW performs best on the simpler SIFT dataset but exhibits significant QPS fluctuations. This is because Pgvector's cost-based optimizer dynamically adjusts execution plans based on parameters, whereas native databases like Milvus and Qdrant utilize heuristic-based strategies that are less sensitive to search parameter tuning. On the more challenging GloVe and GIST datasets, Qdrant-HNSW with payload indexing shows superior efficiency.

4.4 Comparison with Existing Benchmarks

To verify the effectiveness of VecBench, we compare VecBench with *BigVectorBench* [22]. We employ datasets of the same scale (1M and 10M, corresponding to `cc_news` and `amazon_books` in *BigVectorBench*). We evaluate Milvus-HNSW by varying data dimensionality based on YFCC with matched scales and filter correlation. As illustrated in Figure 15, as dimensionality increases, Milvus-HNSW's throughput in VecBench drops from 26.5 to 6.0 QPS, a significantly decrease compared to the 80.0 to 55.3 QPS transition in *BigVectorBench*. This demonstrates that VecBench can exert significantly higher empirical stress by exploring extreme data characteristics. Furthermore,



(a) Scale = 1M



(b) Scale = 10M

Fig. 15. Performance comparison between VecBench and BigVectorBench across different data scales.

at the 1M scale, VecBench exposes a significant latency increase (7.2 ms to 40.3 ms) under low correlation, which remains obscured under the random filters used by existing benchmarks. Notably, VecBench also captures a severe recall degradation (0.99 to 0.62) in middle correlation settings as scale grows, demonstrating that filtered search complexity scales non-linearly. These findings highlight VecBench's comprehensive ability to evaluate system behaviors that existing benchmarks cannot cover.

5 Related Work

Vector database benchmarking efforts have recently evolved from evaluating approximate nearest neighbor (ANN) algorithms [52] to assessing filtered vector search on heterogeneous data.

ANN-Benchmarks [2] are widely used for comparing ANN algorithms under common metrics, but it focuses solely on pure vector retrieval. Although the Big-ANN Challenge [40] (based on ANN-Benchmarks) has expanded its evaluation scope to include filtered search, it only introduces simple filtered vector search without involving any update cases. Furthermore, it focuses on algorithms rather than conducting end-to-end evaluation from a system-level perspective. More recently, *VectorDBBench* [53] introduced system-level evaluations covering indexing, querying, and concurrent processing. However, they fall short in covering complex filtered search tasks and lack micro-level analysis for filtered search operations. *BigVectorBench* [22] is a recent advancement that further emphasizes the necessity of benchmarking vector databases with a variety of datasets and workloads. The designed *filtering query* task specifically evaluates filtered search performance under scalar constraints (such as timestamps or ratings), revealing how filtering mechanisms alter the recall-latency trade-off across different index types. Its design follows the approach of *ANN-Benchmarks*, meaning that it cannot control the data generation and query generation in a fine-grained way. In summary, existing benchmarks are not enough to make a holistic evaluation of filtered vector search.

6 Takeaways

We synthesize our findings into insights for system design and selection under filtered vector search workloads.

(1) Database architecture comparison: Relationally extended systems such as Pgvector achieve higher throughput, benefiting from cost-based query optimization and efficient memory management. Native vector systems such as Milvus provide a more balanced trade-off between performance and recall, owing to their segment-based storage design and parallel query execution.

(2) Hybrid execution strategy: Different query selectivity and filter correlations have a significant impact on search performance. Specifically, both pre-filtering and in-filtering perform poorly under low selectivity due to the large candidate set and interrupted traversal paths. Post-filtering becomes ineffective when filter correlation is low, as the resulting candidate set may be insufficient. Extended filtering is generally more robust across settings, but incurs higher index construction and query-time overhead. Overall, our evaluation indicates that selecting an optimal strategy across varying selectivity and correlations remains an open problem.

(3) Dynamic update performance: Vector database performance is strongly influenced by how systems handle dynamic updates. In-place update mechanisms (e.g., Pgvector) are more efficient in read-dominated workloads with infrequent modifications, whereas append-based designs (e.g., Milvus) offer more stable performance under high update rates by decoupling writes from index maintenance through asynchronous processing.

(4) The impact of data scaling: Different vector index types exhibit different performance characteristics as data scale increases. IVFFlat performs well on small-scale datasets with moderate dimensionality, benefiting from efficient space partitioning. However, as dataset size or dimensionality grows, its performance degrades due to reduced clustering accuracy and increasingly expensive linear scans within inverted lists. In contrast, HNSW scales more effectively to large, high-dimensional datasets by enabling efficient graph-based search, at the cost of higher memory consumption and longer indexing time.

(5) The comparison of libraries vs. systems: Compared with vector database systems with integrated support for filtered vector search, vector search libraries have three key limitations. First, they do not natively support filter predicates, requiring applications to handle filtering logic externally. Second, they lack the ability to dynamically adapt execution strategies under different query characteristics. Third, they provide no system-level memory management or optimized storage layouts, which leads to higher overhead under concurrent workloads.

7 Conclusion

In this work, we propose a controllable benchmark, named VecBench. We make three contributions. First, we propose a controllable data generation method for providing dataset with diverse dimension and scale as well as a theoretical error bound. Second, we propose a distribution-aware query generation method with varied selectivity and filtered correlation. Third, we propose a unified metric to quantify the overall performance. Experimental results over four representative vector databases verify the effectiveness of VecBench on benchmarking filtered vector search.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (Grant Nos. 62441230, 62436010, 62502520), the National Key Laboratory of Data Space Technology and System (Grant No. QZQC2026041), and the Scientific Research Innovation Capability Support Project for Young Faculty (Grant No. SRICSPYF-ZY2025001). Guoliang Li was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (62525202, 62232009), Shenzhen Project (CJGJZD20230724093403007), China Railway Science Research Institute Group Co., Ltd, Zhongguancun Lab. The authors would like to thank Mingyang Yi for his insightful guidance and support regarding the theoretical aspects of JL–based data synthesis. Ju Fan is the corresponding author of this paper.

References

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

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

[3] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1999. When is "nearest neighbor" meaningful?. In *International conference on database theory*. Springer, 217–235.

[4] Yannis Chronis, Helena Caminal, Yannis Papakonstantinou, Fatma Özcan, and Anastasia Ailamaki. 2025. Filtered Vector Search: State-of-the-Art and Research Opportunities. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5488–5492.

[5] Haowen Dong, Chao Zhang, Guoliang Li, and Huanchen Zhang. 2024. Cloud-native databases: A survey. *IEEE Transactions on Knowledge and Data Engineering* 36, 12 (2024), 7772–7791.

[6] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*. 577–586.

[7] Karima Echihabi, Themis Palpanas, and Kostas Zoumpatianos. 2021. New Trends in High-D Vector Similarity Search: AI-driven, Progressive, and Distributed. *Proc. VLDB Endow.* 14, 12 (2021), 3198–3201.

[8] FAISS. 2025. IVFFLAT: Inverted file with exact post-verification. https://github.com/facebookresearch/faiss/wiki/Faiss-indexes. 2025/10/18.

[9] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining small language models and large language models for zero-shot NL2SQL. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2750–2763.

[10] Meihao Fan, Ju Fan, Nan Tang, Lei Cao, Guoliang Li, and Xiaoyong Du. 2025. AutoPrep: Natural Language Question-Aware Data Preparation with a Multi-Agent Framework. *Proc. VLDB Endow.* 18, 10 (June 2025), 3504–3517. doi:10.14778/3748191.3748211

[11] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143* (2017).

[12] Hanjia Gao and Xiaofeng Shao. 2023. Two sample testing in high dimension via maximum mean discrepancy. *Journal of Machine Learning Research* 24, 304 (2023), 1–33.

[13] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. 2023. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*. 3406–3416.

[14] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. *Advances in neural information processing systems* 27 (2014).

[15] Goetz Graefe et al. 2011. Modern B-tree techniques. *Foundations and Trends® in Databases* 3, 4 (2011), 203–402.

[16] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. 2012. A kernel two-sample test. *The journal of machine learning research* 13, 1 (2012), 723–773.

[17] Joseph M Hellerstein, Michael Stonebraker, James Hamilton, et al. 2007. Architecture of a database system. *Foundations and Trends® in Databases* 1, 2 (2007), 141–259.

[18] Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. *Advances in neural information processing systems* 33 (2020), 6840–6851.

[19] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[20] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 861–864.

[21] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[22] Guoxin Kang, Zhongxin Ge, Jingpei Hu, Xueya Zhang, Lei Wang, and Jianfeng Zhan. 2025. BigVectorBench: Heterogeneous Data Embedding and Compound Queries are Essential in Evaluating Vector Databases. *Proceedings of the VLDB Endowment* 18, 5 (2025), 1536–1550.

[23] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[24] Kasper Green Larsen and Jelani Nelson. 2016. The Johnson-Lindenstrauss Lemma Is Optimal for Linear Dimensionality Reduction. *Leibniz International Proceedings in Informatics* 55 (2016), 82–1.

[25] Beatrice Laurent and Pascal Massart. 2000. Adaptive estimation of a quadratic functional by model selection. *Annals of statistics* (2000), 1302–1338.

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

[27] Jiayi Liu, Yunan Zhang, Chenzhe Jin, Aditya Gupta, Shige Liu, and Jianguo Wang. 2026. Fast Vector Search in PostgreSQL: A Decoupled Approach. In *Conference on Innovative Data Systems Research (CIDR)*.

[28] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025. A Survey of Text-to-SQL in the Era of LLMs: Where are we, and where are we going? *IEEE Transactions on Knowledge and Data Engineering* (2025).

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

[30] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[31] Karl Ni, Roger Pearce, Kofi Boakye, Brian Van Essen, Damian Borth, Barry Chen, and Eric Wang. 2015. Large-scale deep learning on the YFCC100M dataset. *arXiv preprint arXiv:1502.03409* (2015).

[32] Openai. 2025. ChatGPT Retrieval Plugin. https://github.com/openai/chatgpt-retrieval-plugin/. 2025/10/18.

[33] James Pan and Guoliang Li. 2025. Database Perspective on LLM Inference Systems. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5504–5507.

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

[35] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector database management techniques and systems. In *Companion of the 2024 International Conference on Management of Data*. 597–604.

[36] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. Acorn: Performant and predicate-agnostic search over vector embeddings and structured data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[37] Giulio Ermanno Pibiri and Rossano Venturini. 2020. Techniques for inverted index compression. *ACM Computing Surveys (CSUR)* 53, 6 (2020), 1–36.

[38] Jianbin Qin, Wei Wang, Chuan Xiao, and Ying Zhang. 2020. Similarity query processing for high-dimensional data. *Proceedings of the VLDB Endowment* (2020).

[39] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. Clickhouse-lightning fast analytics for everyone. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3731–3744.

[40] Harsha Vardhan Simhadri, Martin Aumüller, Amir Ingber, Matthijs Douze, George Williams, Magdalen Dobson Manohar, Dmitry Baranchuk, Edo Liberty, Frank Liu, Ben Landrum, et al. 2024. Results of the Big ANN: NeurIPS'23 competition. *arXiv preprint arXiv:2409.17424* (2024).

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

[42] Santosh S Vempala. 2005. *The random projection method*. Vol. 65. American Mathematical Soc.

[43] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2023. An efficient and robust framework for approximate nearest neighbor search with attribute constraint. *Advances in Neural Information Processing Systems* 36 (2023), 15738–15751.

[44] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3152–3165.

[45] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.

[46] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2241–2253.

[47] Chao Zhang et al. 2018. Parameter Curation and Data Generation for Benchmarking Multi-model Queries. *PhD@ VLDB* (2018).

[48] Chao Zhang, Guoliang Li, Leyao Liu, Tao Lv, and Ju Fan. 2025. CloudyBench: A testbed for a comprehensive evaluation of cloud-native databases. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 1–13.

[49] Chao Zhang, Guoliang Li, Jintao Zhang, Xinning Zhang, and Jianhua Feng. 2024. Htap databases: A survey. *IEEE Transactions on Knowledge and Data Engineering* 36, 11 (2024), 6410–6429.

[50] Shaolei Zhang, Ju Fan, Meihao Fan, Guoliang Li, and Xiaoyong Du. 2025. Deepanalyze: Agentic large language models for autonomous data science. *arXiv preprint arXiv:2510.16872* (2025).

[51] Yuxin Zhang, Meihao Fan, Ju Fan, Mingyang Yi, Yuyu Luo, Jian Tan, and Guoliang Li. 2025. Reward-sql: Boosting text-to-sql via stepwise reasoning and process-supervised rewards. *arXiv preprint arXiv:2505.04671* (2025).

[52] Yunan Zhang, Shige Liu, and Jianguo Wang. 2024. Are there fundamental limitations in supporting vector data management in relational databases? A case study of PostgreSQL. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3640–3653.

[53] Zilliz. 2025. VectorDBBench: A Benchmark Tool for VectorDB. https://github.com/zilliztech/VectorDBBench. 2025/10/12.

[54] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: segment graph for range-filtering approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

Received October 2025; revised January 2026; accepted February 2026