“Performance Evaluation of Milvus and Qdrant: A Comparative Study of Vector Databases for High-Dimensional Similarity Search”

Stefanos Giannakopoulos
(Στέφανος Γιαννακόπουλος
el20829) Team Id:08
School of Electrical & Computer Engineering
NTUA
Athens,Greece
stefanos.yian@gmail.com

Nikolaos Papakonstantopoulos
(Νικόλαος Παπακωνσταντόπουλος
el20069) Team Id:08 School of Electrical & Computer Engineering
NTUA
Athens,Greece
nikolaospapa3@gmail.com

Abstract—This paper presents a comprehensive performance comparison of two open-source vector databases, Milvus and Qdrant, focusing on single-node installation, configuration, data loading, query generation, and performance benchmarking. We ingest sizable synthetic and real-world vector datasets—spanning various dimensions and data types—into both systems to evaluate scalability and storage efficiency. We then execute a suite of similarity search queries, measuring key metrics such as query latency and throughput under different distance metrics (e.g., Euclidean, Cosine) and filtering conditions. By leveraging an existing benchmarking framework, we systematically analyze and highlight each database's strengths, weaknesses, and practical considerations for large-scale similarity search applications. Our findings offer valuable insights to practitioners seeking robust single-node vector database solutions, guiding them on how to best address high-dimensional data management and query processing challenges.

Index Terms—Vector databases, Milvus, Qdrant, similarity search, data loading, single-node deployment, performance benchmarking.

I. Introduction

With the rise of machine learning, artificial intelligence, and large-scale data analytics, the need for efficient high-dimensional similarity search has become more critical than ever. Traditional relational databases are not optimized for handling vector-based data, which is essential for applications such as image retrieval, recommendation systems, natural language processing, and anomaly detection. To address this challenge, specialized vector databases have emerged, providing scalable and efficient indexing mechanisms for managing high-dimensional vectors.

Among the leading open-source vector database solutions, Milvus [1] and Qdrant [2] have gained significant adoption due to their optimized indexing structures and search capabilities. Both databases support Approximate Nearest Neighbor (ANN) search [3], enabling fast similarity queries even on large datasets. However, they differ in architecture, indexing strategies, filtering capabilities, and system performance. Understanding these differences is crucial for selecting the most suitable database for a given application.

This paper presents a comprehensive comparison of Milvus and Qdrant in a single-node setup, evaluating key performance metrics such as query latency, throughput and storage efficiency. We assess their installation and configuration process, data loading efficiency, and performance under various similarity metrics, including Euclidean distance (L2) , Cosine Similarity and Inner Product [4], [5]. By leveraging existing benchmarking frameworks, we provide an in-depth analysis of their strengths and weaknesses, helping practitioners make informed decisions about vector database selection.

The remainder of this paper is structured as follows: Section II provides background on vector databases and similarity search algorithms. Section III outlines the installation and setup process. Section IV discusses data discovery and loading, while Section V details the query generation methodology. Section VI explores indexing and search configuration. Section VII presents the measurement of performance metrics, and Section VIII evaluates the the strengths and limitations of Qdrant and Milvus.

II. Background

A. Vector DBs 101

Vector databases are specialized systems designed for storing, indexing, and retrieving high-dimensional vector embeddings that encode semantic relationships between data points. Unlike traditional relational databases that store structured data in tables, vector databases manage numerical representations generated by AI models, making them essential for applications such as semantic search, recommendation systems, and anomaly detection. These databases follow a structured workflow to optimize storage, indexing, retrieval, and scalability [6].

Initially, vector databases ingest, process, and store vector embeddings efficiently. These embeddings, generated by AI models such as Word2Vec, BERT, and CNNs [7]-[9], transform raw data into multi-dimensional numeric representations. The embeddings encode semantic meaning, ensuring that related entities are positioned closer in the vector space. Data ingestion occurs through batch processing or real-time streaming, often accompanied by metadata (e.g., category, timestamp, or user preferences) to facilitate advanced filtering.

Efficient retrieval in vector databases requires specialized indexing mechanisms that enable rapid similarity searches. Unlike traditional databases that use B-Trees or Hash Indexes, vector databases rely on Approximate Nearest Neighbor (ANN) [3] algorithms for efficient indexing. Common strategies include Hierarchical Navigable Small World (HNSW) [10] , a graph-based index enabling fast and scalable searches; Inverted File Index (IVF) [11], a clustering-based method that organizes vectors into partitions for faster lookups; and Locality-Sensitive Hashing (LSH) [12], a hashing technique that maps similar vectors into the same bucket, improving search speed (we will focus on HNSW indexing). As data evolves, indexes are periodically rebuilt or optimized to maintain high query performance, ensuring that newly added vectors are properly indexed and searchable. Once indexed, vector databases perform high-speed similarity searches based on user-provided query vectors, such as images, text, or products. The nearest neighbors are determined using similarity metrics including Euclidean Distance (L2 norm) [4], which measures straight-line distance between vectors and cosine similarity [4], which calculates the angle between vectors and is widely used in NLP [13] and document retrieval.

Additionally, modern vector databases support metadata filtering to refine search results based on attributes such as category, price, or brand. Hybrid search techniques further enhance accuracy by combining vector similarity with keyword-based searches, facilitating multi-modal retrieval tasks.

To manage large-scale datasets and high query loads, vector databases employ scalability techniques designed for single-node deployments. Performance optimization strategies such as efficient memory management, indexing enhancements, and query parallelization allow single-node vector databases to handle significant workloads. Techniques like Product Quantization (PQ) [14] and optimized caching mechanisms reduce memory footprint while maintaining fast search speeds. Additionally, hardware acceleration using GPUs or specialized vector processing units (VPUs) can further enhance performance by accelerating similarity calculations. As vector databases must continuously adapt to evolving data and user preferences, new vectors are incorporated, outdated entries are removed, and indexes are restructured to maintain real-time performance. Re-ranking techniques dynamically adjust search results based on user interactions or AI-driven heuristics, further improving relevance and accuracy.

Vector databases operate through a structured workflow that includes data ingestion, indexing, query processing, scaling, and continuous optimization. By leveraging advanced indexing techniques, ANN-based similarity search, and scalable architectures, these databases power AI-driven applications such as semantic search, personalized recommendations, and anomaly detection. Their ability to efficiently manage high-dimensional vector data establishes them as critical components within modern AI and machine learning ecosystems.

B. Similarity search & Indexing

Similarity search is performed by ANN (Approximate Nearest Neighbours) [3] Algorithm. Specifically, Approximate Nearest Neighbor (ANN) search is a fundamental technique in vector databases that enables efficient similarity search over high-dimensional data. Unlike exact nearest neighbor search, which is computationally expensive and scales poorly with increasing dataset size, ANN algorithms provide an optimized trade-off between accuracy and retrieval speed. These algorithms are designed to return near-optimal results while significantly reducing computational complexity, making them suitable for large-scale applications such as semantic search, recommendation systems, and anomaly detection.

Vector databases employ specialized ANN [3] indexing techniques to optimize search performance. A Common ANN method is Hierarchical Navigable Small World (HNSW) [10], which constructs a multi-layered graph structure to enable fast, approximate searches with logarithmic complexity. This indexing strategy significantly reduces query times by limiting the number of distance calculations required during search operations. The performance of ANN-based retrieval depends on the chosen similarity metric. Common distance functions include Euclidean Distance (L2 norm) [4] and cosine similarity [4].

To ensure scalability, vector databases continuously optimize ANN indexes through dynamic reordering and periodic reindexing, allowing for efficient adaptation to newly inserted or deleted data. Additionally, hybrid search techniques combine ANN-based retrieval with metadata filtering, enhancing accuracy for applications requiring both vector similarity and structured search constraints.

ANN search [3] is a cornerstone of modern vector databases, providing the computational efficiency necessary for real-time AI-driven applications. By leveraging advanced indexing methods, optimized distance metrics, and scalable architectures, ANN techniques enable vector databases to support high-speed, high-accuracy similarity search across massive datasets, making them indispensable in artificial intelligence and machine learning ecosystems.

Hierarchical Navigable Small World (HNSW) [10] graphs are among the most efficient indexing methods for approximate nearest neighbor (ANN) search. HNSW builds upon the principles of proximity graphs, skip lists, and navigable small world (NSW) graphs to enable fast and scalable vector search.

The construction of an HNSW index involves iteratively inserting vectors into a multi-layered graph structure. Each inserted vector is assigned a maximum layer based on a probability function governed by the level multiplier , which is typically set to to balance search efficiency and recall. The insertion process follows these steps:

1. Layer Assignment: A new vector is assigned to a layer with probability , ensuring that higher layers contain fewer nodes with long-range links.
2. Initial Search: The search for the insertion point starts at the topmost layer and proceeds downward, following a greedy routing mechanism to locate the closest existing node.
3. Neighbor Selection: At each layer, the closest neighbors of the inserted vector are identified. In layer 0, this number increases to M_0, improving connectivity.
4. Edge Creation: Each node maintains a maximum of M links, while layer 0 nodes have a limit of M_0. Connections are formed by selecting the nearest neighbors from the candidate set.
5. Layer Transition: The process repeats for each lower layer until layer 0 is reached, ensuring a well-connected graph across multiple levels.

The search operation in HNSW leverages the hierarchical structure to optimize traversal speed and accuracy. It involves the following steps:

1. Entry Point Selection: The search begins at the topmost layer, where a pre-defined entry point is used.
2. Greedy Routing: The algorithm navigates the graph by moving to the closest neighbor in each step, reducing the search space logarithmically.
3. Layer Transition: When a local minimum is found, the search continues in the next lower layer, refining the results with more granular connections.
4. Final Retrieval: At layer 0, the search refines the nearest neighbors using a best-first search strategy, ensuring high recall.

Key parameters influence the performance of HNSW indexing:

- M defines the maximum number of links per node. Higher values improve recall but increase memory consumption.
- efSearch controls the number of explored neighbors during search. Larger values yield higher recall but increase query time.
- efConstruction determines the number of neighbors considered during insertion, impacting graph quality and indexing speed.
- mL is the level multiplier affecting the hierarchical distribution of nodes, balancing search depth and efficiency.

HNSW indexing achieves sub-linear search complexity, typically for large datasets, due to its hierarchical structure and efficient greedy routing mechanism. The memory overhead is primarily influenced by and the number of layers, making parameter tuning crucial for optimal performance.

III. Installation & Setup

To conduct a comprehensive performance comparison of Milvus[1] and Qdrant[2], we utilized the open-source benchmarking repository qdrant/vector-db-benchmark [15], which provides a standardized framework for evaluating vector database performance. This section outlines the installation and configuration procedures for both the database servers and the Python clients. For additional commands and detailed instructions, please refer to the README.md file in the repository [19].

A. Prerequisites

Prior to deploying the database servers, the following prerequisites must be installed:

- Docker Desktop: Required for containerizing and managing the vector database servers [16].
- Python3: Version 3.10.12 was used in our experiments to ensure compatibility with the project's dependencies [17].

B. Deploying Servers with Docker

Deploying both Milvus [1] and Qdrant [2] in Docker is streamlined using configurations provided in the engine/servers directory of the repository. Specifically, deployment scripts are located in the qdrant-single-node and milvus-single-node directories.

1. Milvus Deployment:

When deploying Milvus [1], a multi-container Docker application is initiated, comprising the following key components:

- milvus-standalone: The core component responsible for handling data ingestion, indexing, and query processing.
- etcd: A distributed key-value store used to manage Milvus metadata efficiently.
- minio: A high-performance object storage service responsible for data persistence.

2. Qdrant Deployment:

In contrast, Qdrant [2] deployment involves a simpler single-container setup. The Qdrant server exposes:

- REST API on port 6333, facilitating standard HTTP-based interactions.
- gRPC API on port 6334, optimized for high-performance communication in distributed systems.

C. Python Client Execution

To interface with the deployed vector databases, we utilized Python clients with dependencies managed via the poetry tool , which supports Python versions 3.8 and above. For consistency, Python 3.10.12 was employed in all experiments.

The benchmark execution is initiated through the run.py script, which requires the following parameters:

- engines: Specifies the target database engine. The parameter must correspond to an engine name defined in the JSON configuration files located in the experiments/configurations directory.
- datasets: Determines the dataset to be used for data ingestion and query evaluation. The dataset name must match an entry in datasets/datasets.json.

This setup ensures a controlled environment for evaluating performance metrics, allowing for reproducible and consistent benchmarking across different vector database systems.

IV. Data Discovery and Loading

This project utilizes datasets declared in a JSON file within the benchmarking repository. The file, located at datasets/datasets.json, contains configuration details for all available datasets used for testing. New datasets can be added by appending a dictionary with the appropriate key-value pairs to this JSON file.

A. *Dataset Configuration Fields*

- name: A string representing the dataset's name, which can be used as a value for the --datasets parameter when executing run.py.
- vector_size: An integer indicating the number of dimensions for each vector in the dataset.
- distance: A string specifying the distance metric for similarity-based queries. Supported values include: cosine and l2
- type: A string denoting the file type of the dataset (e.g., h5, tar).
- path: A string indicating the directory path within the datasets/ folder where the dataset files are stored.
- link: A URL string pointing to the source from which the dataset can be downloaded.

B. *Tested Datasets*

- glove-100-angular:
  This dataset consists of pre-trained word embeddings from the GloVe model with 100-dimensional vectors. It is designed for evaluating natural language processing applications where semantic similarity between words is important.
  - Size: 462 MB
  - File Type: HDF5
  - Distance Metric: cosine
- random-match-keyword-100-angular-no-filters:
  This synthetic dataset is generated with random vectors and associated keywords. It is used to test basic similarity search performance without any metadata filters.
  - Size: 404 MB
  - Files: vectors.npy, tests.jsonl
  - Distance Metric: cosine
- random-match-keyword-100-angular-filters:
  Similar to the no-filters version, this dataset includes additional metadata payloads for testing the effectiveness of filtered searches alongside vector similarity queries.
  - Size: 434 MB
  - Files: vectors.npy, tests.jsonl, payloads.jsonl
  - Distance Metric: cosine
- h-and-m-2048-angular-no-filters:
  Derived from the H&M fashion dataset, this dataset contains high-dimensional image feature vectors (2048 dimensions) used to evaluate image retrieval performance without metadata filters.
  - Size: 1.23 GB
  - Files: vectors.npy, tests.jsonl
  - Distance Metric: cosine
- h-and-m-2048-angular-filters:
  This version of the H&M dataset includes metadata for filtered searches, allowing performance benchmarking of hybrid queries that combine vector similarity with structured filters.
  - Size: 1.29 GB
  - Files: vectors.npy, tests.jsonl, payloads.jsonl, filters.jsonl
  - Distance Metric: cosine
- gist-960-euclidean:
  The GIST dataset contains image descriptors with 960 dimensions, optimized for evaluating similarity search tasks in computer vision applications. It uses Euclidean distance as the similarity metric.
  - Size: 3.58 GB
  - File Type: HDF5
  - Distance Metric: l2

C. *Dataset Selection Criteria*

The datasets used in this study were carefully chosen to meet several key objectives essential for robust performance evaluation. First, they include large datasets, such as gist-960-euclidean (3.58 GB) and h-and-m-2048-angular-filters (1.29 GB), which exceed the size of main memory. This ensures that the benchmarks reflect real-world scenarios where data cannot be entirely cached, thereby placing stress on both the I/O subsystem and the indexing efficiency of each database.

Second, the datasets vary in dimensionality, ranging from 100-dimensional vectors in glove-100-angular to 2048-dimensional vectors in the H&M datasets. This variation allows for an assessment of how each database handles different data complexities and vector sizes.

Lastly, the same datasets are loaded into both Milvus and Qdrant [1],[2] under identical conditions, eliminating dataset-specific factors that could otherwise bias the comparison. This approach ensures a comprehensive and unbiased performance analysis across diverse use cases, providing a fair assessment of each system's scalability, memory usage, and query efficiency.

D. Disk Usage and Memory Constraints

During the upload and indexing phase, vector databases often require substantial disk space, sometimes exceeding 10 GB or more, to store and organize the dataset. This surge in disk requirements arises from the nature of the indexing process, which may generate multiple copies of the underlying data to facilitate efficient searching. Specifically, the system must store not only the raw vectors but also the associated index structures such as tree structures, inverted indices, or hash tables that optimize similarity searches. These indexing structures can be memory-intensive and may, in fact, require more space than the raw dataset itself.

Moreover, some databases leverage disk-based storage for intermediate computations during index construction, further increasing overall disk utilization. As dataset sizes exceed available RAM, the system may swap data between disk and memory, thus exerting a significant performance penalty. This issue becomes especially critical for large-scale datasets, where the entire dataset does not fit into RAM, leading to frequent disk I/O operations. Consequently, the combination of large index structures, disk-based intermediate processing, and memory constraints results in higher disk space usage during indexing, often surpassing the available RAM capacity and stressing the underlying hardware.

E. Storage

1. Milvus

In standalone mode, Milvus consolidates services typically separated in the cluster version. Instead of relying on external systems (such as etcd for metadata or Pulsar for logs), the standalone deployment merges these functionalities into a single process. This design choice lowers the operational overhead and simplifies local development and smaller-scale production scenarios.

Metadata and Log Management

- Embedded Metadata:
  Rather than storing schemas and other metadata in a separate system like etcd, the standalone version writes these snapshots directly into a local storage engine. This approach reduces complexity but places all metadata operations within the same process.
- RocksDB as Log Broker:
  In cluster mode, Milvus leverages Apache Pulsar for its publish-subscribe (pub-sub) log mechanism. By contrast, standalone mode uses RocksDB to store and replay incremental data changes. When Milvus restarts, it replays operations from RocksDB, allowing the system to recover states accurately.

This integrated logging and metadata system makes standalone Milvus easier to install and maintain, at the cost of forgoing the external high-availability features provided by etcd and Pulsar in cluster mode.

Object Storage Considerations

By default, standalone Milvus uses local disks to hold large files (e.g., data snapshots, index files). Nonetheless, users can optionally configure remote object storage (e.g., Amazon S3 or MinIO) if they require greater durability and elasticity. In practice, local disk is sufficient for many small-scale or development workloads, while production environments may prefer scalable remote storage.

Performance and Limitations

1. Simplicity and Reduced Overhead:
   - Single-process deployment eases setup and is well-suited for proof-of-concept or testing environments.
   - Eliminates external services like etcd and Pulsar, reducing the infrastructure footprint.
2. Single Point of Failure and Scale Constraints:
   - Standalone mode operates as a single process; if the node fails, services go offline until a restart.
   - Scaling horizontally (adding more standalone nodes) is not feasible without transitioning to the full cluster architecture.
3. Local Resource Bottlenecks:
   - Depending on hardware resources, local disk usage for logs and snapshots may limit throughput under heavy workloads.

Despite these constraints, standalone mode remains an attractive option for users who need basic vector indexing and searching without the overhead of a distributed setup [20].

2. Qdrant

This section describes how Qdrant organizes and manages data, including the use of segments, different options for vector storage, payload storage, and the versioning mechanism that ensures data integrity.

Segments and Their Role

All data within a Qdrant collection is divided into segments. Each segment contains its own vector storage, payload storage, indexes, and an ID mapper that tracks the relationship between internal and external point IDs. Typically, segments do not overlap; however, duplicate points present in multiple segments are automatically deduplicated during searches.

Segments can be either appendable (supporting insertion, deletion, and querying) or non-appendable (supporting only reads and deletions). By design, each collection must contain at least one appendable segment to allow for ongoing data insertions.

Vector Storage Options

Qdrant offers two primary storage configurations for vectors, balancing performance requirements and available system resources:

1. In-Memory Storage
   - Vectors reside fully in RAM, offering the highest query throughput and lowest latency.
   - Disk access is only required to persist data; all active operations remain in memory.
   - This setup demands sufficient RAM to store the entire dataset.
2. Memmap (On-Disk) Storage
   - Uses memory-mapped files, which rely on the operating system's page cache rather than loading all data into RAM at startup.
   - Provides nearly the same performance as in-memory when ample RAM is available but also gracefully adapts to memory constraints.
   - Enables storage of large collections with reduced memory overhead, assuming fast and reliable disks.

Qdrant supports two main approaches to configuring memmap storage for vectors:

- Setting `on_disk` to `true`: Immediately places all vectors in memmap storage at collection creation.
- Using `memmap_threshold`: Automatically converts a segment from in-memory to memmap storage once it grows beyond a specified threshold. The threshold can be set globally in Qdrant's configuration or per collection via `optimizers_config`.

In addition, Qdrant can store its HNSW index on disk by configuring `hnsw_config.on_disk = true`. This setting can reduce memory usage further for large-scale datasets at the cost of increased disk I/O.

Payload Storage Options

Payload storage refers to how Qdrant handles non-vector data, such as text fields or arbitrary user-provided attributes:

1. In-Memory Payload Storage
   - Loads all payload information into RAM at startup.
   - Offers the fastest read performance but requires ample memory, especially if payloads are large (e.g., containing images or lengthy text).
2. On-Disk Payload Storage

- Stores payloads in RocksDB, minimizing RAM usage.
- May incur greater latency when filtering queries rely on payload values.
- To mitigate this overhead, Qdrant allows creating field-specific indexes that keep frequently used payload values in memory, improving filter query speed.

Whether payloads reside in RAM or on disk can be configured in Qdrant's global settings or at the time of collection creation (e.g., by setting `on_disk_payload`).

Versioning and Data Integrity

Qdrant guarantees data consistency and safety via a two-stage commit process:

1. Write-Ahead Log (WAL):
   - Every operation (insertion, deletion, update) is appended to the WAL in a strictly increasing order.
   - Ensures no data is lost in the event of a system crash or power failure; the WAL can be replayed for recovery.
2. Segment Updates and Point Versions:
   - When changes are flushed from the WAL to segments, each point version is compared to the segment's stored version.
   - If the incoming update's version is older than the point's current version, it is discarded (i.e., out-of-order updates do not overwrite newer data).
   - Each segment, and each point within that segment, keeps track of the latest version of the data it contains, streamlining rollback or recovery procedures.

These capabilities—segmentation, flexible vector and payload storage types, and robust versioning—collectively make Qdrant well-suited for large-scale vector similarity search applications with varying performance and resource requirements [21].

V. Query Generation

To evaluate the performance of the two vector databases, similarity queries are pre-generated and included within each downloadable dataset. These queries, along with their expected results, including the closest vector IDs and cosine/euclidean similarity scores, are integral to benchmarking the databases. Some datasets also include payloads and filter conditions, which are used to assess the databases' ability to handle filtered searches alongside similarity queries.

A. *Dataset Structure*

Each dataset generated by the qdrant/ann-filtering-benchmark-datasets repository [18] consists of the following files:

- vectors.npy: A Numpy matrix of vectors with the shape (number of vectors, dimension) representing the data to be uploaded into the vector database.
- payloads.jsonl: A JSONL file containing the metadata or payloads associated with each vector. The number of lines in this file matches the number of vectors.
- tests.jsonl: A collection of queries with associated filter conditions and expected results. Each query entry in the file includes the following fields:
    - query: The vector to be used for similarity search.
    - conditions: Filtering conditions, which can be of three types: match, range, and geo.
    - closest_ids: The expected IDs of the vectors that should be found as the closest matches based on the query vector.
    - closest_scores: The expected cosine similarity scores for the closest vectors.

For datasets in HDF5 format, such as those containing test data and expected results, the structure might include the following:

- test: A dataset of query vectors.
- neighbors: A list of IDs of the closest vectors for each query.
- distances: The cosine similarity or euclidean distance scores corresponding to the closest vectors.

B. *Query Generation Process*

The process for generating queries, expected results, and filter conditions involves the following steps:

1. Random Selection of Query Vector:

A random query vector is selected from the dataset's vectors (from vectors.npy for Numpy datasets or test in HDF5 datasets).The selected vector is used for the similarity search against the database.

2. Random Selection of Filter Condition:

A random filter condition is selected. The filter condition restricts the query to only those vectors whose payloads meet the filter criteria. The possible filter types include:

- Match: A filter based on a specific keyword or value.
- Range: A filter based on numerical ranges.
- Geo: A spatial filter based on geographic locations.

Only match filters were used for the experiments.

3. Filter Application:

The selected filter is applied to the dataset's payloads (from payloads.jsonl for Numpy datasets or filters.jsonl for HDF5 datasets). Only vectors whose payloads satisfy the filter condition are retained for further evaluation.

4. Finding the Closest Vectors:

After applying the filter (if present), the remaining vectors are compared to the selected query vector using cosine similarity or l2 distance. The similarity score is computed using a linear algebra library to measure the closeness of vectors and the top-N closest vectors are identified based on their cosine similarity scores.

5. Returning Results:

The IDs of the top-N closest vectors (closest_ids) and their corresponding cosine similarity or l2 distance scores (closest_scores) are returned as the expected results for the query. These results along with query vectors and filter conditions are stored in tests.jsonl for the dataset.

C. *Filtered and Unfiltered Queries*

Unfiltered Queries: These queries do not apply any filters. All vectors in the dataset are considered for the similarity

Filtered Queries: Queries with filters apply the selected filter conditions to restrict the vectors considered for the similarity Search and results are return beased purely on the similarity measure.

VI. Indexing and Search Configuration

The results of the experiments comparing the two databases are stored in the results directory of the repository [19], which was forked from [15]. The exact configurations used in the experiments are explicitly defined in the repository results/README.md [19]. This section provides an overview of key parameters used in both Milvus and Qdrant for HNSW-based vector indexing and search.

A. *Indexing and Upload Parameters*

For the upload and indexing phase, the parallel parameter specifies the number of concurrent operations—either for indexing or searching. A higher parallel value can improve throughput by utilizing more CPU threads but may lead to contention in multi-tenant environments. In all experiments, this parameter is set to 16 for both databases. Concerning Index Parameters:

- M (m in Qdrant): Defines the maximum number of bidirectional connections (neighbors) each node can have in the HNSW graph. Higher values increase recall by creating more connections but require additional memory and longer indexing times.

- efConstruction (ef_construct in Qdrant): Controls the candidate list size during vector insertion. A larger value improves recall at the cost of slower indexing performance.

The evaluated configurations include: (M, efConstruction): (16,128), (32,128), (32,256), (32,512), (64,256), (64,512).

In addition to standard HNSW parameters, Qdrant introduces memory management optimizations by defining memmap_threshold. According to Qdrant Documentation [2], this parameter specifies the Maximum size (in KiloBytes) of vectors to store in-memory per segment and consequently segments larger than this threshold will be stored as read-only memmaped file. In all experiments, this parameter was set to 10,000,000, which increases RAM usage by keeping more data in memory while allowing larger segments to be stored as read-only memory-mapped files.

B. Search Parameters

The following search parameters were used in different experiments with varying values:

- ef (hnsw_ef in Qdrant): Determines how many candidate neighbors each query explores during a search. Higher values enhance recall but increase search latency and CPU usage. We tested values of 128, 256, and 512 for both databases, with an additional setting of 64 exclusively for Qdrant.
- Parallel (search): Defines the number of concurrent search operations. Higher values increase throughput but can impact performance under heavy system load. Experiments were conducted using parallel = 1 and parallel = 100.

VII. MEASUREMENT OF PERFORMANCE METRICS

All measurements are available in our GitHub repository: https://github.com/ntua-el20069/vector-db-benchmark [19]. Specifically, the data analysis is documented in the file analysis.ipynb, located in the directory https://github.com/ntua-el20069/vector-db-benchmark/tree/ADIS/results.

Additionally, all visual representations and diagrams can be found in the https://github.com/ntua-el20069/vector-db-benchmark/tree/ADIS/results/plots-html directory. These diagrams can be viewed using the Live Server extension in Visual Studio Code for better accessibility and interaction.

A. Upload and Index Time

This section compares the upload time and index time for Qdrant[2] and Milvus[1] under identical configurations (m and ef_construct values). A subset of the tested datasets is examined.

Observations:

1. glove-100-angular: Milvus requires approximately 15% more upload time and double the index time compared to Qdrant in most configurations.
2. random-match-keyword-100-angular-no-filters: Milvus requires 35% more upload time and double the index time compared to Qdrant in most configurations.
3. h-and-m-2048-angular-no-filters: Milvus requires 25% more upload time and up to 50% more index time compared to Qdrant in most configurations.
4. gist-960-euclidean: Milvus requires 35% more upload time and double or more index time compared to Qdrant in most configurations.

Summary:

- Upload Time: Milvus requires 20-30% more time for upload than Qdrant in most configurations.
- Index Time: Milvus requires at least double the index time compared to Qdrant in most cases.
- As m and ef values increase, the differences in upload_time and index_time between the two databases become more pronounced.
- Higher values of m and ef lead to slower indexing performance.

B. Search Results

This section compares the search latency (total_time, mean_time, p95_time), throughput (rps), and precision (mean_precisions) for Qdrant[2] and Milvus[1] under identical configurations (m, ef_construct, and ef_search values). A subset of the tested datasets is examined for a specific value of parallel search threads (here 100).

Observations:

1. glove-100-angular: For small ef_search (e.g., 128), Qdrant requires 50-80% more search time and performs fewer rps compared to Milvus. For larger ef_search values (e.g., 512), Qdrant requires double the search time and performs half the rps compared to Milvus. Qdrant achieves 5-10% higher mean precisions.
2. random-match-keyword-100-angular-no-filters: Qdrant requires 40-80% more search time and performs fewer rps compared to Milvus. Qdrant achieves 5-10% higher mean precisions.
3. h-and-m-2048-angular-no-filters: Qdrant is slower by a factor of 1.5 in search time compared to Milvus, and Milvus serves more rps. Both databases achieve high mean precisions for this dataset.
4. gist-960-euclidean: Both databases have similar search time and rps for most configurations. Qdrant shows unexpectedly high latency in some configurations (m-32-ef-256 and m-64-ef-256).

Mean precisions are high (95-99%) and similar for both databases, with a slight advantage for Qdrant.

Summary:

- Search Time: Qdrant requires 40-80% more search time compared to Milvus.
- Throughput: Milvus performs up to 50% or more rps than Qdrant.
- Precision: Qdrant achieves 5-10% higher mean precisions for low m, ef_construct, and ef_search values. For high values, mean precisions are similar (95-99%) for both databases.
- Sensitivity: Milvus is more sensitive to changes in ef_search and shows significant improvements in mean precisions with each increase.

C. Precision Based Comparison

This section compares engine configurations achieving the same mean_precisions_rounded concerning latency and rps metrics. Additionally, upload and index time for the corresponding engines that achieved the same precision are examined.

Observations:

1. total_time: High precision requires more total query time for both databases.
   - glove-100-angular: Milvus requires total query time similar to the fastest Qdrant configuration.
   - random-match-keyword-100-angular-no-filters: Milvus requires less query time than Qdrant in most precisions.
   - h-and-m-2048-angular-no-filters: Milvus is faster in query time. Both databases achieve 100% precision for most configurations.
   - gist-960-euclidean: Milvus and Qdrant best configurations achieve similar total query time.
2. rps:
   - glove-100-angular: Milvus achieves slightly higher rps than Qdrant.
   - random-match-keyword-100-angular-no-filters: Milvus achieves higher rps than Qdrant.
   - h-and-m-2048-angular-no-filters: Milvus achieves higher rps due to 1.0 precision for all configurations.
   - gist-960-euclidean: Qdrant achieves higher rps.
3. import_time:
   - glove-100-angular: Milvus requires at least double the import time compared to Qdrant.
   - random-match-keyword-100-angular-no-filters: Milvus is slower in import time.
   - h-and-m-2048-angular-no-filters: Milvus is slower in import time.
   - gist-960-euclidean: Milvus has some fast configurations but is still slower than Qdrant.

Summary:

- total_time: Qdrant requires similar or up to 20-40% more query time compared to Milvus.
- rps: Milvus achieves a higher number of requests per second (RPS) when processing cosine-distance datasets (**Milvus does not natively support cosine distance as a similarity metric. Instead, when cosine distance is specified in our benchmark, Milvus utilizes inner product (IP) [5] as an alternative. This is because cosine similarity is mathematically equivalent to the inner product of normalized vectors**), whereas Qdrant demonstrates superior RPS performance when utilizing Euclidean-distance dataset (gist-960-euclidean).
- import_time: Milvus requires approximately double the import time compared to Qdrant for the same precision level.

D. Filtered Search Benchmark

This section compares regular search and filter search for two datasets (h-and-m-2048-angular and random-match-keyword-100-angular) concerning latency, rps, and import time.

Observations:

1. h-and-m-2048-angular:
   - mean_precisions: Unaffected for both databases (mean precisions = 1.0).
   - total_time: Regular search is faster for both databases, with Milvus being faster.
   - rps: Milvus serves more rps in regular search.
   - import_time: Import time is at least double for datasets with filters, with Qdrant being more affected.
2. random-match-keyword-100-angular:
   - mean_precisions: Qdrant achieves 1.0 precisions in filter search, while Milvus achieves worse precisions (0.6-0.7).
   - total_time: Filtered search is faster for both databases, with Qdrant being faster.
   - rps: Qdrant serves more rps in filtered search compared to Milvus.
   - import_time: Milvus requires more time for datasets with filters, while Qdrant is less affected.

Summary:

- The behavior of the two databases in filter search depends on factors such as payloads and filters size, the number of vectors affected by a filter, and the specific construction and search configurations of the engine.

- Qdrant exhibits higher precision in filtered search, whereas Milvus does not always show the same behavior. The impact of filtering on index time, query time, and requests per second (RPS) varies depending on the specific filter applied, making it difficult to draw a generalized conclusion regarding these metrics.
- Milvus is faster when precision is not affected by the filter.
- Qdrant exhibits faster query times when the filter affects significantly the examined vectors.

This comprehensive comparison highlights the trade-offs between Qdrant and Milvus in terms of upload time, index time, search latency, throughput, and precision, providing valuable insights for selecting the appropriate database based on specific application requirements [19].

VIII. Evaluating Qdrant and Milvus: Strengths and Limitations

In general, Milvus[1] exhibits a significantly higher index time than Qdrant[2], resulting in longer data ingestion durations and consequently making Qdrant noticeably faster in terms of total import time. Furthermore, Qdrant consistently demonstrates higher precision, particularly for small to medium-sized datasets, where the gap compared to Milvus can be substantial. In larger datasets (exceeding 1GB), both systems exhibit comparable accuracy, with Qdrant still achieving slightly higher precision overall.

Conversely, Milvus delivers higher throughput (requests per second) and lower query latency, positioning it as an attractive choice for applications that demand rapid query responses at scale. Examples include real-time recommendation engines, which must handle large query volumes with minimal latency, and enterprise search systems where query throughput is paramount.

In contrast, Qdrant's faster ingestion process and robust precision make it suitable for scenarios where data is frequently updated and high recall quality is essential. These use cases might include A/B testing environments that perform frequent index rebuilds, or applications that prioritize result quality such as semantic content filtering or specialized research tasks over peak query speed.

Overall, organizations prioritizing rapid data indexing and high precision may favor Qdrant, while those seeking minimal query latency and elevated throughput might opt for Milvus. The ultimate selection hinges on the specific operational requirements, including data size, update frequency, precision needs, and peak throughput demands.

References

[1] Milvus Open Source Documentation , available: https://milvus.io/docs/overview.md

[2] Qdrant Open Source Documentation, available: https://qdrant.tech/documentation/overview/

[3] Approximate Nearest Neighbors, MongoDB Open Source Documentation, available: https://www.mongodb.com/resources/basics/ann-search

[4] Cosine and L2, Zilliz Documentation, available: https://zilliz.com/blog/similarity-metrics-for-vector-search

[5] Inner Product Similarities , Zilliz Documentation, available: https://zilliz.com/blog/similarity-metrics-for-vector-search

[6] Vector Databases Introduction , IBM Official Documentation, available: https://www.ibm.com/think/topics/vector-database

[7] Word2Vec, TensorFlow official Documentation, available: https://www.tensorflow.org/text/tutorials/word2vec

[8] BERT Article, available: https://www.techtarget.com/searchenterpriseai/definition/BERT-language-model

[9] Convolutional Neural Networks, article available: https://www.datacamp.com/tutorial/introduction-to-convolutional-neural-networks-cnns

[10] HNSW (Hierarchical Navigable Small Worlds), Pinecone Documentation, available: https://www.pinecone.io/learn/series/faiss/hnsw/

[11] Inverted File Index , Medium Article available https://medium.com/@Jawabreh0/inverted-file-indexing-ivf-in-faiss-a-comprehensive-guide-c183fe979d20

[12] Locality Sensitive Hashing , Pinecone Documentation available https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/

[13] NLP (Natural Language Processing) ,IBM official documentation , available: https://www.ibm.com/think/topics/natural-language-processing

[14] Product quantization, Pinecone Documentation available : https://www.pinecone.io/learn/series/faiss/product-quantization/

[15] Benchmarking Repository, available: https://github.com/qdrant/vector-db-benchmark

[16] Docker Desktop , available: https://www.docker.com/products/docker-desktop/

[17] Python3 , available : https://www.python.org/

[18] Query Generation Benchmark, available: https://github.com/qdrant/ann-filtering-benchmark-datasets

[19] Repository Forked from [15], including results of the experiments, available: https://github.com/ntua-el20069/vector-db-benchmark

[20] Milvus storage architecture, available: https://milvus.io/docs/four_layers.md

[21] Qdrant storage architecture, available: https://qdrant.tech/documentation/concepts/storage/https://qdrant.tech/documentation/concepts/storage/