SingleStore-V: An Integrated Vector Database System in SingleStore [Industry]

Cheng Chen, Chenzhe Jin[†], Yunan Zhang[†], Sasha Podolsky, Chun Wu , Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, Jianguo Wang[†]

SingleStore   Purdue University[†]

{cchen; sasha; chunwu; szupo; hanson; zhou; rob}@singlestore.com
{jin467; zhan4404; csjgwang}@purdue.edu[†]

ABSTRACT

Vector databases have recently gained significant attention due to the emergence of large language models that can produce vector embeddings for text. Existing vector databases can be broadly categorized into two types: specialized and generalized. Specialized vector databases are explicitly designed and optimized for managing vector data, while generalized vector databases support vector data management within a general purpose database (e.g., a relational database). While the development of specialized vector databases is interesting, there is a substantial customer base interested in generalized vector databases for various reasons, e.g., a reluctance to move data out of relational databases to reduce data silos and operational costs, the desire to use SQL, and the need for more sophisticated query processing involving both vector and non-vector data. However, there are two main challenges with generalized vector databases. The first challenge is performance. The second challenge is the interoperability of vector search with the existing SQL ecosystem, such as combining vector search with predicate filters, range filters, joins, or even fulltext search.

In this paper, we present the design and implementation of SingleStore-V, a full-fledged generalized vector database integrated into SingleStore, a modern distributed relational database optimized for both OLAP and OLTP workloads. SingleStore-V achieves both high performance and high interoperability via a suite of optimizations. Experiments on standard vector benchmarks and datasets show that SingleStore-V achieves performance comparable to that of Milvus, a highly-optimized specialized vector database. Furthermore, SingleStore-V significantly improves upon pgvector, a popular generalized vector database implemented in PostgreSQL. We believe that this paper will shed light on integrating vector search into relational databases in general, as many of the design concepts and optimizations are relevant to other relational databases.

1 INTRODUCTION

Vector databases are becoming increasingly important due to the rising demand for efficient processing of vector-similarity queries. This increased demand was triggered by the introduction of a new wave of generative AI systems led by OpenAI's ChatGPT, and the ability of their large language models (LLMs) to produce vector embeddings that accurately represent the meaning of text. Other deep neural network models are available for transforming different types of complex objects such as images of faces and objects into vector embeddings.

Vector databases are designed and optimized for processing and storing these kinds of high-dimensional vectors. This makes them suitable for supporting various applications, including semantic search, retrieval augmentation generation (RAG) [35, 53], recommendation systems [15, 19, 33], face recognition and image search.

Currently, there are two main categories of vector databases: specialized and generalized vector databases. Specialized vector databases are designed from scratch to explicitly manage vector data. Examples include Milvus [49] and Pinecone [17]. Generalized vector databases, on the other hand, are designed to manage vector data inside an existing database (mostly a relational database), following a one-size-fits-all [30] design philosophy. Examples include pgvector [16] and AnalyticDB-V [51].

SingleStore-V falls into the category of generalized vector databases. It supports vector search in SingleStore, a modern high performance distributed relational database. We believe that vectors and vector search are a data type and query processing approach, and they do not require a fundamentally different way of processing data. However, using a specialized vector database combined with other data management tools can lead to many problems: redundant data, excessive data movement, extra labor expense for specialized skills, extra licensing costs, limited query language power, programmability and extensibility, and poor data integrity and availability compared with a true DBMS.

Therefore, instead of using a specialized vector database, we believe that application developers using vector similarity search will be better served by building their applications on a modern relational database (like SingleStore [44]) that meets all their database requirements.

However, there are two main challenges in building a generalized vector database: high performance and high interoperability.

Challenge #1: High Performance. An often-cited criticism of generalized vector databases is their low performance (compared to specialized vector databases), because they are not natively designed to support vector data. For example, a recent paper [55] shows that there is a significant performance gap (which can exceed 10X) between certain specialized and generalized vector databases.

Challenge #2: High Interoperability. If vector search becomes an important operator in relational databases, it must be seamlessly compatible with other SQL operators such as filters and joins, as a single SQL query may involve both vector and non-vector data.

The importance of interoperability is two-fold. First, the introduction of vector search into relational databases should not disrupt existing SQL semantics. For instance, when a user incorporates vector search into a standard SQL query alongside filters, it is expected that the results will only include vectors meeting the filter criteria. Second, and importantly, many real-world applications, such as

Table 1: Comparing with Other Vector Databases

Vector Databases	General-purpose	Transactions	Full SQL Support	Distributed	Filtered Vector Search	Vector Range Search	Vector Range Join	Vector with Fulltext Search	Auto Index	Index Algorithms
PASE [52]	✓	✓	✓	✗	✓	✗	✗	✗	✗	IVF_FLAT, HNSW
pgvector [16]	✓	✓	✓	✗	✓	✓	✗	✗	✗	IVF_FLAT, HNSW
ElasticSearch [4]	✓	✗	✗	✓	✓	✗	✗	✓	✗	HNSW
AnalyticDB-V [51]	✓	✓	✓	✓	✓	✓	✗	✓	✗	VGPQ
ClickHouse [2]	✓	✓	✓	✓	✓	✓	✗	✗	✗	Annoy, HNSW
Rockset [10]	✓	✗	✓	✓	✓	✗	✗	✗	✗	IVF
MongoDB [14]	✓	✓	✗	✓	✓	✓	✗	✓	✗	HNSW
Vespa [22]	✗	✗	✗	✓	✓	✗	✗	✓	✗	HNSW
Milvus [49]	✗	✗	✗	✓	✓	✓	✗	✗	✓	Pluggable (IVF, HNSW)
Qdrant [18]	✗	✗	✗	✓	✓	✓	✗	✓	✗	HNSW
Weaviate [23]	✗	✗	✗	✓	✓	✓	✗	✗	✗	HNSW
Pinecone [17]	✗	✗	✗	✓	✓	✗	✗	✗	✗	Not disclosed
SingleStore-V	✓	✓	✓	✓	✓	✓	✓	✓	✓	Pluggable (IVF, HNSW)

e-commerce and recommendations, demand more than just vector data. They often additionally require filtering based on specific attributes (e.g., price) or joining with other data tables. This is exactly what relational databases excel at. By integrating vector search into relational databases, one can naturally support hybrid query processing in a single SQL query that includes both vector and non-vector data. This integration provides advanced vector data analytics that are crucial for real-world applications.

Motivation. We are motivated to build SingleStore-V by the technology opportunity, competitive challenge, and known customer needs. We have analyzed proof-of-concept requirements and real vector search queries from our customers. For example, Lumana.AI [11] is developing a video security application that uses vector search to identify objects alongside complex SQL filters on properties like time range. Some top financial institutions are building real-time RAG applications that combine semantic search, traditional fulltext search, knowledge graph traversal, and SQL joins. Most of them employ a mixture of SQL operations such as filters, joins, common table expressions, and subqueries, integrated with vector search, not just pure vector search.

Although there are a few relational databases that already support vector search, e.g., pgvector [16] (based on PostgreSQL), PASE [52] (based on PostgreSQL), and AnalyticDB-V [51] (based on AnalyticDB), they have not addressed the performance and interoperability challenges effectively. For example, the performance of the above three databases is significantly worse than that of specialized vector databases like Milvus, as shown in [55]. Furthermore, they do not integrate vector search well with the existing SQL ecosystem. For instance, they do not support vector search with range filters, joins, or fulltext search. Note that although Oracle and Azure SQL have announced their support for vector search [24, 26], few technical details have been disclosed. Table 1 shows a comparison of SingleStore-V with other vector databases.

Overview of SingleStore-V. In this paper, we introduce SingleStore-V, a generalized vector database integrated into SingleStore, a modern distributed relational database. It addresses the performance and interoperability challenges via a suite of optimizations.

To achieve high performance, SingleStore-V introduces a new physical operator called `Top()` that pushes down the top-k vector search to table scan in the SQL execution. It also introduces a new concept of "per-segment vector index" to achieve high scalability in both parallel and distributed environments. It supports pluggable vector indexes to embrace the best implementations of vector indexes in the community. It also provides auto indexing to choose an appropriate index and parameters based on users' requirements to achieve high performance.

To achieve high interoperability, SingleStore-V seamlessly integrates vector search with existing SQL queries by efficiently supporting hybrid queries that involve both vector and non-vector data, including combining vector search with predicate filters, range filters, joins, and fulltext search.

Experimental Overview. Experiments on VectorDBBench [21] show that SingleStore-V achieves comparable performance to Milvus, a leading specialized vector database, for both quantization-based and graph-based indexes. Moreover, SingleStore-V significantly outperforms pgvector, a popular generalized vector database implemented in PostgreSQL.

Contributions. The overall contribution of this paper is the design and implementation of SingleStore-V, a full-fledged generalized vector database system implemented within SingleStore, a modern distributed relational database. SingleStore-V achieves high performance comparable to that of a highly optimized specialized vector database and supports high interoperability with the existing SQL ecosystem, efficiently querying both vector and non-vector data together to provide real-time response with transactional consistency.

This paper discusses the key architectural decisions that enables SingleStore-V to achieve state-of-the-art vector search capabilities within a relational database. Considering that generalized vector databases support hybrid queries involving both vector and non-vector data far more effectively than specialized vector databases through powerful SQL queries, we believe that generalized vector databases serve the needs of user applications better than specialized systems. We hope that the insights from this paper would serve as the basis to the design of other relational database systems that incorporate vector functionalities.

2 BACKGROUND AND RELATED WORK

In this section, we introduce the background of SingleStore and review the related work on vector databases.

2.1 SingleStore

2.1.1 SingleStore Overview

SingleStore [44] is a horizontally-partitioned, distributed shared-nothing database system that optimizes performance for both operational and analytical workloads. It separates storage and compute by utilizing cloud storage as a shared disk, which provides benefits such as allowing data storage to exceed local disk space and fast provisioning of new compute resources. At the same time, it also utilizes local memory and disk storage to enable low-latency read and write operations. SingleStore's columnstore [45] supports high-speed data ingestion, massive scalability, and the ability to perform complex analytical queries directly on encoded data using SIMD instructions [41].

2.1.2 Distributed Cluster Architecture

A cluster of SingleStore consists of two types of nodes: aggregators and leaves. The aggregator nodes are used to interface with clients and manage distributed coordination. They accept user queries, build an optimized query plan, and split the plan into sub-queries suitable for distributed execution, passing these sub-queries down to leaf nodes. The leaf nodes hold partitions of data in a table and perform most of the computations.

Tables are distributed into different partitions according to a shard key. Write operations in SingleStore are executed in ACID transactions, committed atomically across database partitions. Changes in a transaction become visible as soon as the transaction is committed. Within each database partition on leaf nodes, write operations are persisted to a log file on the local disk before committing. Replaying these persisted log files recovers the state of the database in the event of a node restart. Log files and data blobs are asynchronously uploaded to cloud storage to support the separation of compute and storage, without incurring the latency of cloud storage as part of the write transaction. The durability of data is ensured within the cluster by replicating database partitions across multiple leaf nodes and considering a transaction as committed only after it has been replicated. This replication guarantees that the loss of a single node will not cause data loss. In the event of a node becoming unavailable, a replica can be quickly promoted as the master copy of a database partition to ensure high availability.

2.1.3 LSM-based Table Storage

SingleStore utilizes unified table storage, which internally combines an in-memory rowstore based on a lock-free skip list and a disk-based columnstore LSM tree. This supports both transactional and analytical workloads without duplicating data into different layouts, and it also serves as the basis of the vector index design described in the later sections. The columnstore LSM tree storage consists of a collection of immutable segments each storing a disjoint subset of rows, in the form of data blobs compressed separately for each column. Writing data in large immutable data blobs works well with hierarchical storage, and per-column compression allows for efficient scans in OLAP queries using SIMD instructions. At the same time, seekable compression encodings are used to allow fast reads at a specific row offset, which is a common operation in OLTP and search workloads. Users can optionally specify a sort key on each table, which defines the sort order of rows within each segment, as well as the order to maintain across segments by the LSM tree. Each segment stores the min and max value of each column in its metadata, so that a table scan can skip segments that have no overlap with filter conditions, as is common when filtering using sort key columns.

The in-memory rowstore portion of the table persists recently modified rows from small write transactions (large write operations create columnstore segments directly). A background flusher process periodically converts those rows into columnstore segments. Deletions from the columnstore segments are represented in segment metadata by marking rows as deleted in a bit vector, without rewriting compressed data blobs. Representing deleted rows as a bit vector avoids the cost of storing logical delete or row overwrite records in the LSM tree, as well as the cost of performing merge-based reconciliation to apply those records during a scan. A background merger process periodically merges segments to maintain the sort order in the LSM tree and to combine segments that are either too small or have too many deleted rows.

All foreground and background operations on unified storage tables use MVCC to support transaction rollback and consistent reads. Each table read sees a consistent snapshot of on-disk columnstore and in-memory rowstore data, so that internal data movement between the two is invisible to the user. Update and delete operations lock only modified rows, rather than entire segments, so that concurrent write transactions do not block each other as long as they do not modify the same row.

2.1.4 Query Execution

Scanning a SingleStore table has three main stages. First, we compute a list of relevant segments, skipping ones which can be eliminated by using min/max segment metadata or global secondary index structures. Second, we execute filters to find the offsets of the rows to output. Third, we decode the columnstore segments for the relevant columns selectively at those offsets. We also scan the rowstore at the same snapshot version. Both cases produce a batch of data with an in-memory column-oriented representation suitable for the execution of later operations like joins and group-bys.

The flexibility of unified table storage allows multiple data access methods to be used for the same computation. For instance, a filter clause can either utilize a secondary index or encoded execution. For hybrid workloads, it is necessary to choose the access method for each operation and apply them in the optimal order to get good performance. Since the performance of the data access methods is heavily influenced by query parameters and data encodings, SingleStore employs adaptive query execution to perform these decisions dynamically at runtime. In the context of filtering operations, we organize the clauses of the filter condition as a filter tree and time each filter clause on a small sample of data from the beginning of each segment to estimate its per-row evaluation cost. The combination of the estimated cost and selectivity determines the evaluation method to use for each clause, as well as the optimal order in which to execute them.

2.1.5 Query Planning

SingleStore employs a cost-based query optimizer [27] to perform logical rewrites and plan physical operations in the query. Due to the distributed nature of SingleStore, one important goal of the optimizer is planning the broadcast and reshuffle operations and minimizing the data movement. The optimizer runs on the aggregator node to which a given user query is submitted, and it produces a query plan which includes sub-queries to send to leaf nodes, with additional annotations to specify join orders and join strategies. SingleStore implements full-query code generation using LLVM [36] to generate efficient machine code to improve query efficiency.

2.2 Vector Similarity Search

Vector similarity search (a.k.a high-dimensional nearest neighbor search) has been explored comprehensively in the past [7, 32, 34, 37, 47] because of its fundamental importance in many real-world applications. Given a query vector $q \in \mathbb{R}^d$ with d dimensions and a vector set S, the problem is to find the top-k vectors from S that are closest to q based on a similarity (or distance) function, e.g., Euclidean distance or cosine distance. As d and n are usually large, e.g., d can be 10s to 1000s of dimensions and n can be millions to even billions of vectors, vector similarity search is computationally expensive.

Therefore, high-dimensional indexes are required for supporting efficient vector search. In the literature, there are mainly four types of high-dimensional indexes: quantization-based (e.g., IVF_FLAT [7, 34], IVF_PQ [34], IVF_PQFS [8], IVF_SQ8 [7], SPANN [28]), graph-based (e.g., HNSW [40], NSG [31], Rand-NSG [47], DiskANN [47]), LSH-based [32, 39, 56], and tree-based [38] indexes. Existing vector databases mostly use quantization-based indexes (for space efficiency) and graph-based indexes (for high performance and accuracy).

2.3 Vector Databases

Vector databases are designed to efficiently store and search high-dimensional indexes and address system-related challenges, see [42, 43] for recent surveys. Broadly speaking, vector databases are categorized into specialized and generalized vector databases, depending on whether they support vector search within an existing database (mostly relational databases).

There are a couple of specialized vector databases developed in the past, e.g., Faiss [5], Milvus [49], and Pinecone [17], following the design principle of one-size-not-fits-all [46]. As those systems are specialized for vector data, they have full control to implement high-dimensional indexes in the most efficient way to speed up vector similarity search. Among them, Milvus is a leading specialized vector databases. It is an in-memory system developed based on Faiss [5]. It leverages the internal functions of Faiss while supporting CPU-GPU heterogeneous hardware and distributed environments. Milvus inherits the indexes implemented in Faiss and introduces additional indexes optimized for GPUs.

Typical generalized vector databases include pgvector [16], PASE [52], AnalyticDB-V [51].[1] As mentioned in Sec. 1, those systems did not fully address the performance and compatibility challenges. There are other vector databases that are implemented inside ElasticSearch and MongoDB, but SingleStore-V is implemented within SingleStore, a relational database.

[1]Note that we do not discuss Oracle [24] and Azure SQL [26] because the technical details of their support for vector search are not publicly available.

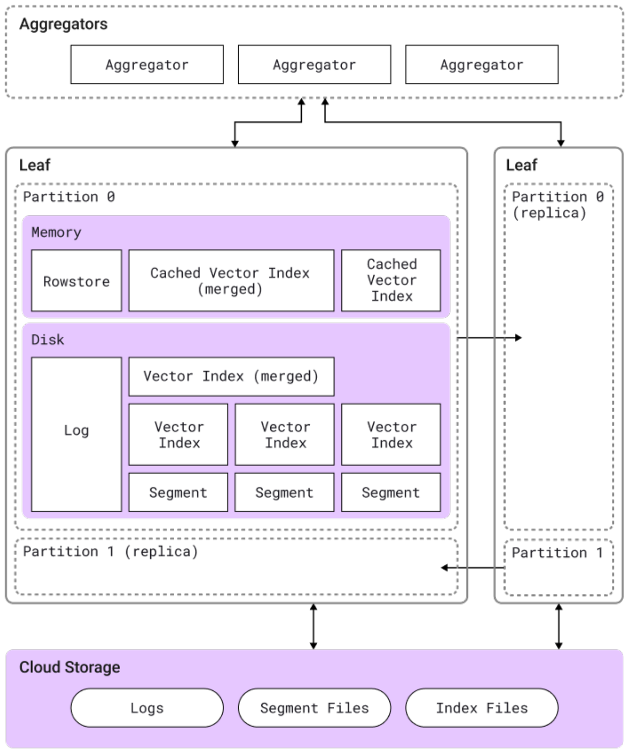


Figure 1: SingleStore-V Architecture

Furthermore, a prior work relevant to this paper is [55], which analyzes the root causes of the performance gap between specialized and generalized vector databases. However, that work is an experimental study without implementing a generalized vector database. In contrast, this paper builds a full-fledged generalized vector database inside SingleStore.

3 SYSTEM OVERVIEW

Figure 1 shows the system architecture of SingleStore-V, which is a distributed vector database based on SingleStore that includes aggregator nodes to coordinate SQL queries and leaf nodes to store shards of data.

The architecture of SingleStore makes it uniquely suitable for achieving the goals of high performance and high interoperability as a generalized vector database. In particular, the immutable segments in SingleStore's LSM-based table storage support fast search performance on per-segment vector index structures, while the seekable column encodings enable efficient retrieval of vector data from those segments (Sec. 4). Using in-memory rowstore for recently modified data ensures that this data is visible for vector search in real-time. The adaptive query execution pipeline models index use as an alternative access method for the filter operation, which allows vector search to be modeled as a filter to enable interoperability with other SQL operations (Sec. 5). Building on these ideas, the design of SingleStore-V includes the following features:

Extended SQL Semantics for Vector Search. SingleStore-V introduces necessary SQL statements to operate on vector data by extending the SQL syntax. It implements a new data type, `VECTOR`, to represent vectors. Users can thus create a table in SingleStore-V that specifies a column as `VECTOR`, which allows them to load and insert vector data. SingleStore-V also implements `ADD INDEX` to create a vector index and specify index parameters. Furthermore, SingleStore-V extends the SQL `SELECT` statement to search both vector and non-vector data (Sec. 5).

Persisted Vector Index with In-memory Caching. As shown in Figure 1, SingleStore-V builds vector index structures over the segments in the LSM-based table storage (Sec. 4.1). Recall that each segment stores a large immutable subset of rows. By making vector indexes part of the immutable segments, table storage features such as data persistence, replication, offloading to cloud storage, etc. all work seamlessly on vector indexes. In additional to the persisted storage, SingleStore-V also caches the vector indexes in memory for better search performance.

Real-time Update and Transaction Support. SingleStore-V offers full update, delete, and ACID transaction support for the vector index, just like the rest of the table storage. As a relational database, this capability is an important requirement for interoperability. Since row deletions simply mark the row as deleted in the segment metadata, update and delete operations are performed efficiently without modifying the vector index structures. Vector search execution skips the rows marked as deleted in segment metadata and also incorporates rows from the in-memory rowstore to ensure a transactional consistent view, including real-time changes.

Scalable Vector Search. As a distributed shared-nothing system, SingleStore-V scales horizontally by executing vector search in parallel across database partitions on different leaf nodes and combining the results (Sec. 5.1). Within each database partition, the vector index merger improves scalability by building larger indexes covering multiple segments (Sec. 4.2).

Flexible Vector Index Algorithms. Building vector indexes on immutable segments imposes no constraint on the indexing algorithm used, which allows SingleStore-V to plug in different vector index algorithms (Sec. 4.3) and pick suitable index algorithms automatically (Sec. 4.4).

Interoperability with Other SQL Operations. During query execution, SingleStore-V pushes down vector search into the table scan to perform efficient search using the vector index. The search operation is modeled as a new `Top()` filter to enable filtered vector search (Sec. 5.2) and hybrid search, which combines vector search with with fulltext index search (Sec. 5.3). Vector range search and vector range join support (Sec. 5.4) further extends SingleStore-V's ability to incorporate vector search into existing SQL queries.

4 VECTOR INDEX DESIGN

In this section, we present the design of the vector index in SingleStore-V. Key ideas of the design includes per-segment vector index (Sec. 4.1), vector index merger (Sec. 4.2), pluggable vector index (Sec. 4.3), and auto vector index (Sec. 4.4).

4.1 Per-Segment Vector Index

As SingleStore is a distributed shared-nothing database, it shards a table into multiple partitions, and each partition contains multiple segments (each of size 512MB by default) that are managed by an LSM-based storage engine (Sec. 2.1). Therefore, there are two approaches for building a vector index: (1) *per-partition vector index* that builds a vector index for each partition, and (2) *per-segment vector index* that builds a vector index for each segment.

SingleStore-V does not choose the per-partition vector index. This is because the data in a partition is a *changing* dataset, which includes not only immutable on-disk columnstore segments but also an in-memory rowstore that is continuously modified by write transactions. Besides that, new segments can be created and inserted into a partition for new data, and old segments can be deleted when data is deleted or updated, or when segments are merged. Thus, partitions are inherently mutable. However, all major vector index algorithms, such as quantization-based [7, 8, 34] and graph-based indexes [31, 40, 47], are extremely inefficient in supporting dynamic data. For example, quantization-based indexes require an initial training stage during which the distribution of vectors must be known. While graph-based indexes like HNSW do not require an initial training stage, the frequent adding and removing of vectors significantly impacts the quality of the index. When the distribution of vectors undergoes significant changes or the quality of the index degrades, the per-partition vector index must be retrained and rebuilt. This process can be both costly and time-consuming, and the index may become temporarily unavailable or inaccurate during this period.

SingleStore-V's Design Choice on Vector Index. SingleStore-V chooses the per-segment vector index approach. Specifically, it builds a vector index for each on-disk immutable segment. The in-memory segment is not indexed because it is typically small and changes frequently. Building an index for the in-memory segment would be computationally expensive and not significantly improve query performance. Therefore, SingleStore-V simply performs a full scan on the in-memory segment during vector search.

There are several advantages to this design. First, since the content of each on-disk segment is immutable, the per-segment vector index is built once when the segment is created and remains unchanged. As a result, it eliminates the need for global index rebuilding and scales well as additional vectors are added. Second, building a per-segment vector index on immutable segments allows us to treat the vector index algorithm as a black box, enabling easy integration with different vector index algorithms and also implementing powerful automatic vector index algorithm selection and parameter tuning. Further details will be delved into in Sec. 4.3 and Sec. 4.4. Third, the per-segment vector index avoids the cost of performing an LSM-tree lookup per matched row to find the row in the primary LSM-tree. This is because the per-segment vector index stores row offsets rather than the primary key as in the per-partition index. This advantage becomes particularly significant when the vector index produces many output rows.

In SingleStore-V, a vector index can be created on an empty table as part of the `CREATE TABLE` command. Alternatively, a vector index can be added to an existing table using the `ALTER TABLE` command. With `ALTER TABLE`, a per-segment vector index will be built for all existing segments.

SingleStore-V supports inserts, deletes, and updates in a similar way to other types of data in SingleStore as described in Sec. 2.1.3.

When vectors are inserted into a table, small write transactions initially persist the data in the in-memory rowstore. When the background flusher process converts a batch of in-memory rows into a columnstore segment, it also builds a per-segment vector index for that segment. Deletes and updates to a row in a columnstore segment modify segment metadata to mark the row as deleted, leaving the corresponding per-segment vector index unchanged. In other words, each per-segment vector index is immutable once created. At search time, the deleted bit vector is used as a filter to exclude deleted rows. Since update operations also insert the modified rows into the in-memory rowstore, the search query sees the updated version of the row as it scans the in-memory rowstore. Larger write transactions bypass the in-memory rowstore to directly write to columnstore segment, which build the vector index as part of the write transaction and keep the size of the in-memory rowstore small. The background merger process also builds vector index on the newly merged segments it produces, keeping the per-segment vector indexes consistent with the segments.

The vector index in SingleStore-V is similar to Milvus, but it is integrated seamlessly into SingleStore's LSM-based storage engine as a secondary index, rather than building a dedicated LSM-based index as in Milvus. It also differs from those in other generalized vector databases such as pgvector [16] and PASE [52] because the latter are based on PostgreSQL, which relies on a Btree-based storage engine. In contrast, SingleStore-V utilizes an LSM-based storage engine, enabling the construction of vector indexes on immutable data segments for improved performance. Moreover, SingleStore-V's vector index is built per-segment, which is suitable for both parallel and distributed environments. However, pgvector and PASE do not support a distributed vector index.

4.2 Vector Index Merger

The challenge of the per-segment index is performance. This is because vector indexes typically have sub-linear search complexity. For example, IVF-based algorithms achieve a search time of $O(\sqrt{n})$, while graph-based algorithms take a search time of $O(\log n)$ under typical parameter settings. Therefore, searching multiple per-segment vector indexes is slower compared to searching a single per-partition vector index. Also, for HNSW in particular, we observe cases where base layer graph search significantly dominates search in higher levels, suggesting high per-index constant factors.

Increasing segment size would allow SingleStore-V to reduce the number of per-segment indexes, but this approach has drawbacks as well. The segment merger in SingleStore-V materializes output segments in memory, column by column, so larger segments would result in significant temporary increases in memory use. Segment elimination during filter execution would also be less effective, as the likelihood of a segment being entirely eliminated would decrease. Performance in the case of clustered deletions would also decrease, as SingleStore-V would be less likely to delete whole segments at a time.

Instead of increasing segment size, SingleStore-V decouples index size from segment size with a **two-level vector index structure integrated with the LSM-based storage engine**. In this design, SingleStore-V builds per-segment indexes and a background vector index merger constructs additional vector indexes spanning multiple segments. Both the per-segment indexes and the cross-segment index are built on static datasets, as segments are immutable. During vector search, the larger cross-segment vector index can be leveraged to improve search efficiency.

Specifically, starting with an individual vector index for each segment, the index merger in SingleStore-V combines segments into groups. For each group, it creates a vector index covering all segments in the group. The merger will repeat this process iteratively with groups of two, four, eight segments, and so on, following a tree structure. The original per-segment input indexes are evicted from memory, but remain available on disk. To minimize the compute cost of merging, in cases like HNSW index, this process avoids rebuilding indexes from scratch, but rather makes a copy of one input index and grows it by appending vector data from the other segments with which it is grouped.

Large cross-segment indexes have seemingly similar drawbacks to large segments. However, as they are built on a subset of segment data, they allow SingleStore-V to strike a different balance and take advantage of extra optimizations. Index merging, like segment merging, has memory use proportional to the index size. But tuning the segment size around the segment merger requires accounting for the worst-case memory use from the largest columns in a segment, while tuning for index merging requires accounting only for the index sizes. For indexes based on PQ, this memory use is expected to be especially small compared to the data size. Likewise, the smaller size makes it tractable for SingleStore-V to store per-segment indexes. These per-segment indexes allow SingleStore-V to reduce performance degradation related to clustered deletions and segment elimination. Namely, SingleStore-V bounds the performance degradation in these cases by falling back on per-segment indexes from disk when it detects that deletions or filters would allow it to skip searching through enough per-segment indexes.

Although we can always fall back to per-segment indexes, the effort put into merging indexes is wasted if the merged index frequently contains deleted segments. This commonly occurs when the segment merger decides to merge a segment that is already part of a cross-segment vector index. To reduce the likelihood of such scenarios, the index merger only merges vector indexes of segments within a sorted run of the LSM tree. It gives priority to merging vector indexes from longer sorted runs first, as these are expected to have a longer lifespan and, hence, extend the lifespan of the cross-segment indexes.

4.3 Pluggable Vector Index

There are many different types of vector indexes, each offering unique design trade-offs. For example, IVF_PQFS has a faster index build time and a smaller memory footprint compared to HNSW, but its recall and search performance are not as good. Moreover, many new algorithms continue to be developed each year, and even for the same algorithm, multiple implementations are available. For example, Faiss, hnswlib, and Knowhere [13] all provide (different) implementations of the same set of vector indexes.

With this in mind, we believe it is crucial to design the system to easily integrate with a wide range of vector indexes and implementations. This approach not only enables the system to be optimized to work well under different constraints, but also ensures its future-proofing, allowing for seamless adoption of cutting-edge algorithms as they become available.

As a result, SingleStore-V supports pluggable vector indexes. This is possible because both per-segment and cross-segment vector indexes are built on immutable data, allowing SingleStore-V to treat the vector index as a black box and utilize only a very generic interface. Currently, SingleStore-V uses state-of-the-art vector index libraries, including Faiss and hnswlib, and supports various popular in-memory vector indexes, such as IVF_FLAT, IVF_PQ, IVF_PQFS, HNSW_FLAT, and HNSW_PQ. Users can select a vector index algorithm along with its parameters at index creation time. One example of this would be:

```
ALTER TABLE t
ADD VECTOR INDEX (v)
INDEX_OPTIONS '{"index_type":"IVF_PQFS",
               "metric_type":"EUCLIDEAN_DISTANCE",
               "m":250, "nlist":512, "nprobe":8}';
```

Internally, we **decouple the vector index algorithm from its implementation** to allow the use of multiple implementations for the same algorithm. This enables a seamless upgrade of a vector index algorithm to utilize a newer and improved implementation. We achieve this by associating each vector index with its implementation ID.

The pluggable vector index architecture is highly flexible and extends beyond in-memory vector index algorithms to include on-disk vector index algorithms as well such as DiskANN [47] and SPANN [28]. Furthermore, this decoupling allows the vector index to be built in an external service equipped with GPUs, thereby improving resource utilization and significantly enhancing the index build time compared to running on regular database nodes.

4.4 Auto Vector Index

Existing vector databases are difficult for customers to use in terms of index selection and index parameter tuning, because they require users to manually select the index and configure complicated index parameters. Arbitrarily choosing values can result in high cost, low performance, or low accuracy. This places a considerable burden on customers, and in many cases, customers have a limited understanding of the impact of each parameter. Furthermore, as data distribution evolves over time, it may not be optimal to utilize the same index or parameters all the time.

To address this issue, SingleStore-V supports the `AUTO` index, which automatically selects a suitable vector index algorithm and tunes its parameters for each vector index, both per-segment and cross-segment, individually. This approach is feasible because the vector index in SingleStore-V is built on immutable data, since all the vectors have already been seen before the index is built.

Users can specify their requirements for the `AUTO` index. For example, a requirement could be that the index size needs to be small but the recall has to be above 90%. The engine then generates a shortlist of candidate algorithms and index options using rule-based heuristics based on Faiss guidelines [9] and autofaiss [1]. For each candidate, SingleStore-V builds the vector index and then determines the best search options through a grid search based on Faiss autotune [6]. SingleStore-V selects several choices on the Pareto frontier and allows users to choose based on their search performance and recall requirements.

To expedite the above process of index tuning, SingleStore-V introduces two heuristics. First, since building the vector index is the most intensive operation, the `AUTO` index will try to use the last index algorithm and its options for the new index, assuming that the data distribution has not changed. If satisfactory, we avoid exploring other candidates. Second, for each algorithm, we define the options such that larger values lead to better recall but worse search time and index size. This property helps to prune the search space when conducting a grid search for index and search options.

Auto-tuning the vector index can be a time-consuming process, and we aim to prevent it from blocking the write path in SingleStore-V. One optimization in SingleStore-V is to initially build a quick but acceptable index, such as IVF_PQFS, when building a new per-segment or cross-segment vector index. This process can be usually done within 1 second per segment. Automatic vector index selection and parameter tuning are then carried out in the background or within a dedicated service. Once the more optimal indexes are ready, they replace the original index. This approach allows SingleStore-V to maintain fast write speeds and ensures that new data becomes searchable quickly, while still achieving reasonable recall rates.

We are not aware of any other vector databases that support auto vector index, except for Milvus. However, this feature is only available in Zilliz Cloud, with technical details undisclosed [12]. It is also unclear how Milvus's auto index affects the write path, since finding a suitable index is usually time-consuming. SingleStore-V addresses this issue by quickly building IVF_PQFS and fine-tuning the index in the background.

5 VECTOR SEARCH DESIGN

In this section, we present how SingleStore-V executes vector search queries using the vector index. The key idea is the introduction of a new physical operator, termed the `Top()` filter, enabling seamless integration with the existing query execution engine. Supported queries include pure vector search (Sec. 5.1), filtered vector search (Sec. 5.2), vector search with fulltext search (Sec. 5.3), vector range search and vector range join (Sec. 5.4).

5.1 Vector Search

In SingleStore-V, a vector search query that returns top-k similar vectors to a query vector can be expressed in SQL as follows:

```
SELECT *
FROM t
ORDER BY t.v <-> @vector
LIMIT k;
```

where `<->` is the infix operator for Euclidean distance, `@vector` is a user-defined query vector, and `t.v` is a vector column `v` in the table `t`.

Query Optimization and Execution. As SingleStore-V is a distributed vector database, it includes aggregator nodes that coordinate SQL queries and leaf nodes that store shards of data (Figure 2). Given the above SQL query of vector search, an aggregator node pushes down the `ORDER BY ... LIMIT` to the leaf nodes. It sends the same SQL query to each partition of `t` to gather the top-k rows from each partition, then merge-sorts those sorted rows and outputs the top-k globally.

Within each partition, when there is no matching vector index, SingleStore-V treats it as a kNN (k-Nearest Neighbor) query. The leaf node conducts a full table scan, computes the distance to `@vector` for each row, and uses a heap to output the top-k rows for the current partition.

When there is a matching vector index, specifically a vector index on column `t.v`, SingleStore-V treats it as an ANN (Approximate Nearest Neighbor) query. The leaf node pushes down the `ORDER BY ... LIMIT` clause to the table scan as a filter **`Top(m, t.v <-> @vector)`**. This is a new physical operator introduced in SingleStore-V that conducts a vector index scan to select the top-`m` candidates for the current partition based on vector indexes. This process is similar to how SingleStore utilizes other secondary indexes (e.g., hash indexes) in table scans, treating them as an alternative data access method with a filter clause, as described in Sec. 2.1.4. The physical plan of the corresponding SQL query on a leaf node is:

```
Project [t.v <-> @vector]
TopSort limit:k [t.v <-> @vector]
ColumnStoreFilter [Top(m, t.v <-> @vector) index]
ColumnStoreScan t
```

Note that `m`, the number of candidates the vector index scan outputs, may be larger than `k`, the number of vectors the user desires. This is because when PQ [34, 50] is employed, vectors are compressed within the vector indexes, and the distances computed from these indexes are approximate. Therefore, it becomes necessary to select more candidates and then refine them using exact distances. `TopSort` accomplishes this refinement by reading the candidates from disk, computing the exact distance to `@vector`, and then using a heap to output the top-k for the current partition. The ratio `m/k` depends on the vector indexes being used but can be adjusted by the user.

To efficiently execute the table scan with the `Top()` filter, SingleStore-V will first find a set of vector indexes that exactly cover all the segments within the current partition. These vector indexes can be either per-segment or cross-segment, but a larger vector index is preferred for faster search times. Next, it will find the top-`m` rows from each vector index and then perform a merge sort to obtain the top-`m` rows for the current partition. When conducting a vector search for each vector index, the deleted bits for the segments covered by the vector index will be passed to the vector index library to filter out deleted rows, ensuring all `m` rows obtained from each index are present. A full scan is also conducted on the in-memory rowstore segment to include unflushed rows. The data flow is shown in Figure 2, where `<filter>` is empty.

The vector search in SingleStore-V differs from other generalized vector databases such as pgvector [16] and PASE [52], as they do not support `Top()` filter pushdown. While AnalyticDB-V [51] supports pushdown, there are differences in implementations, such as SingleStore-V returning `m` vectors (instead of `k`) per partition. Also, AnalyticDB-V does not discuss distributed query processing.

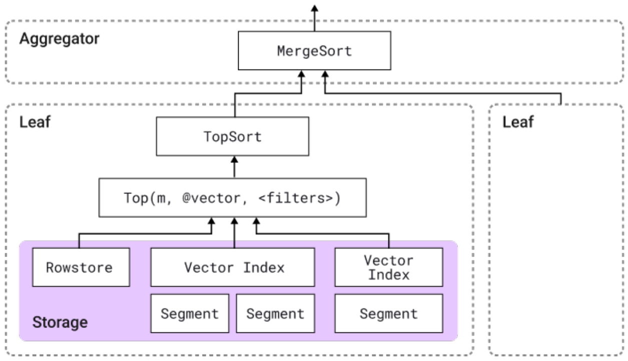


Figure 2: Vector Search in SingleStore-V

5.2 Filtered Vector Search

Many real-world applications require filtered vector search, which combines vector search with filters to find the top-k rows while satisfying a given filtering condition. This can be expressed in SingleStore-V in SQL as follows:

```
SELECT *
FROM t
WHERE <filters>
ORDER BY t.v <-> @vector
LIMIT k;
```

Query Optimization and Execution. Optimizing filtered vector search in SQL is non-trivial. If we conduct the vector search first to select the top-k rows and then apply the filters, it is possible to produce fewer than k rows, which is not desirable. One approach is to predict the selectivity of the filters and compensate for it by selecting more candidates from the vector index scan. However, in practice, it is very difficult to accurately predict the number of candidates needed.

SingleStore-V addresses this issue by incorporating the filters into the vector index scan. Specifically, we extend the `Top()` filter mentioned in Sec. 5.1 to include `<filters>`, resulting in `Top(m, t.v <-> @vector, <filters>)`. This operator selects the top-`m` rows that pass `<filters>`. Consequently, the leaf node pushes down the `ORDER BY ... LIMIT` clause to the table scan as `Top(m, t.v <-> @vector, <filters>)`. The physical query plan on a leaf node is:

```
Project [t.v <-> @vector]
TopSort limit:k [t.v <-> @vector]
ColumnStoreFilter [
  Top(m, t.v <-> @vector, <filters>) index
]
ColumnStoreScan t
```

During query execution, SingleStore-V will first perform segment elimination using `<filters>`. Next, it will find a set of per-segment or cross-segment vector indexes that exactly cover those segments with potential matches. It then creates an approximate sorted iterator for each individual vector index. This iterator exposes a `Next()` interface, which returns the next approximate nearest neighbor. While some vector index algorithms already support this interface [54], a generic iterator is used for those that do not.

Algorithm 1: Generic Approximate Sorted Iterator

```
$\mathcal{V} \leftarrow \emptyset$;
while $k < threshold$ do
    $\mathcal{R} \leftarrow VectorIndex.\texttt{VectorSearch}(k, @vector)$;
    for $(distance, id)$ in $\mathcal{R}$ do
        if $id$ not in $\mathcal{V}$ then
            emit $(distance, id)$;
            $\mathcal{V} \leftarrow \mathcal{V} \cup \{id\}$;
    $k \leftarrow k \times 2$;
fall back to full table scan;
```

The generic iterator wraps the typical top-k interface of the vector index algorithms. It employs a retry strategy, starting with approximate nearest neighbors with k and double k if additional rows are required. An internal hash table $\mathcal{V}$ is utilized to track already outputted rows. If k is doubled beyond a certain threshold where the vector index scan is worse than full scan, the iterator falls back to full scan. Algorithm 1 shows the procedure.

Next, a merged iterator is used to merge the approximate sorted rows from each individual iterator. Internally, it uses a heap to perform a merge sort and exposes a `Next()` method that returns the next approximate sorted row for all segments. It also takes into account the in-memory rowstore segment by performing a full scan on it.

When executing the `Top()` filter, a batch of candidate rows is read from the merged iterator using the `Next()` interface. Subsequently, these candidates are sorted by segments, and then the filters are evaluated one segment at a time. Conducting filter evaluation in batches, instead of row by row, enables SingleStore-V to prefetch and cache the filter column within a segment.

The initial batch size is chosen to be $c * m$ where $1/c$ is the estimated selectivity of the filter. The execution terminates when there are enough rows after filtering for the current batch. Otherwise, the batch size is doubled, and we continue with the next batch.

Note that our approach differs from the pre-filtered approach [23, 49, 51], which involves first filtering data, then storing the results in a bitmap, and subsequently performing a vector search while checking if the candidate is present in the bitmap. However, the pre-filtered approach evaluates the filter for every row, whereas our approach only needs to evaluate the filter on a small number of candidate rows. We utilize the pre-filtered approach for the deleted bits filter and rely on our approach for all other filters because, in these cases, the filter column is on disk and the filter can be arbitrarily complicated.

5.3 Combining Fulltext and Vector Search

In many applications, each entity is associated with many attributes. During the search, users may want to assign a score to each attribute and then sort by a combined score. One common example is the hybrid search combining vector search and fulltext search. The hybrid query can be expressed in SingleStore-V in SQL as follows:

```
SELECT
  *,
  MATCH(t.s) AGAINST ('pattern') AS score1,
  t.v <-> @vector AS score2
FROM t
ORDER BY w1 * score1 + w2 * score2
LIMIT k;
```

Query Optimization and Execution. There are several solutions for combining individual scores. For instance, Elasticsearch utilizes a weighted sum approach, while Azure AI employs Reciprocal Rank Fusion [25]. Currently, Elasticsearch and Azure implement the query as a full-outer join. They execute each individual query separately to obtain top-k candidates from each, then apply a full-outer join to retrieve all candidates. Next, they compute the combined score for each candidate and sort the results by the combined score. However, this approach does not fully adhere to the semantics of the SQL query above because it assumes that only the top-k rows have non-zero scores. Although it is possible to instruct each individual query to output more candidates, this encounters the same difficulty as filtered vector search in predicting how many candidates to produce in practice.

SingleStore-V addresses this issue by using a `Top()` filter that selects the top-k rows with the highest combined score. During execution, this `Top()` filter employs an iterative merging algorithm, similar to what Milvus does for their multi-vector query [49]. The iterative merging algorithm proves effective when the function combining individual scores is monotonic and relies solely on the generic top-k interface provided by the vector index algorithms. The physical query plan is described as follows:

```
Project [*, score1, score2]
TopSort limit:k [w1 * score1 + w2 * score2]
ColumnStoreFilter [
  Top(m, w1 * score1 + w2 * score2) index
]
ColumnStoreScan t
```

Comments on the `Top()` Filter. We believe the `Top()` filter represents a generic approach that extends beyond vector search to facilitate the pushdown of `ORDER BY ... LIMIT`. Its generic form, `Top(k, <expr>, <filters>)`, selects the top-k rows according to `<expr>` while satisfying `<filters>`.

Aside from `ORDER BY ... LIMIT` pushdown, the `Top()` filter can be used by itself as a regular leaf node (index filter) within the filter tree. Multiple instances of `Top()` filters can coexist in the filter tree, each with its own `<expr>` and `<filters>`. Filter reordering can occur within each `<filters>` and throughout the filter tree independently.

5.4 Vector Range Search and Vector Range Join

SingleStore-V also supports vector search with range filters that return all vectors whose distance is within a certain threshold to the target vector. This is useful in many applications, such as plagiarism detection for documents. This query can be expressed in SingleStore-V in SQL as follows:

```
SELECT *
FROM t
WHERE (t.v <-> @vector) < @threshold;
```

For query execution, SingleStore-V executes vector range search filters by scanning the vector index to find all rows within that threshold, producing a list of (segment id, row offset) pairs for the matching rows. A vector range search filter functions as a regular SQL filter that can be combined and reordered with other filters, without the additional semantic considerations required for `Top()` filters as described in Sec. 5.2.

Besides that, SingleStore-V also supports vector range joins, which join the vector columns from two tables to return pairs of vectors if their distance falls within a certain threshold. Vector range queries are useful in applications such as document auto-tagging [29, 48], which assigns one or multiple labels to each unseen document by finding the label embeddings closest to a document embedding. The SQL query can be expressed in SingleStore-V as follows:

```
SELECT *
FROM s JOIN t
ON (s.v <-> t.v) < @threshold
WHERE <filters>(s);
```

SingleStore-V executes the query with a vector index by performing a nested loop join. It loops through all rows from table `s` and filters by `<filters>`, then applies a vector range search on `t` to find `t.v` that are within `@threshold` of `s.v`. The physical query plan is described as follows:

```
Project [*]
NestedLoopJoin
  ColumnStoreFilter [<filters>(s)]
  ColumnStoreScan s
ColumnStoreFilter [(s.v <-> t.v) < @threshold index]
ColumnStoreScan t
```

We are not aware of other vector databases that support vector range join, except for VBase [54]. However, unlike VBase, the implementation in SingleStore-V does not assume an iterator interface from the vector index algorithm, making it compatible with a wider range of algorithms and better integrated into SingleStore.

6 EXPERIMENTS

6.1 Experiment Design

Experimental Environment. All experiments (except the scalability experiment in Sec. 6.3) used identically provisioned r6a.8xlarge EC2 instances (AMD EPYC 7R13 Processor, 32 vCPUs, 256GB memory, 64MB L3 cache, AVX2) with gp2 storage (maximum 16KB IOPS per volume, maximum 150MB/s throughput per volume). We followed public self-administration guides, including recommended Linux tuning, to launch single-host clusters colocated with our benchmarking client. We set the default concurrency of 16 and enabled SIMD with AVX2 for all experiments.

For the scalability experiment, we used SingleStore's managed service to provision multi-host clusters consisting of 10 r6a.4xlarge EC2 instances (16 vCPUs, 128GB memory, AVX2) with the same gp2 storage. We ran the benchmark scripts on a separate host.

Datasets. We ran benchmark workloads using VectorDBBench [21], an open-source benchmarking tool for testing vector database performance. We utilized three well-known datasets: SIFT in different sizes (SIFT1M, SIFT10M, SIFT100M, SIFT1B) and GIST1M, in addition to the Cohere10M dataset chosen by the benchmark authors. The Cohere dataset consists of vector embeddings of Wikipedia articles in various languages, each divided into passages. Each passage is assigned a 768-dimensional embedding vector. Table 2 shows a summary of the datasets.

Table 2: Statistics of Real-world Datasets

Dataset	# Dimensions	# Vectors	# Queries
SIFT1M [20]	128	1,000,000	10,000
SIFT10M [20]	128	10,000,000	10,000
SIFT100M [20]	128	100,000,000	10,000
SIFT1B [20]	128	1,000,000,000	10,000
GIST1M [20]	960	1,000,000	1,000
Cohere10M [3]	768	10,000,000	1,000

Competitors. We compare SingleStore-V with both specialized and generalized vector databases. For specialized vector databases, we choose Milvus [49] as it is a leading specialized vector database in this area. For generalized vector databases, we choose pgvector [16], as it is a popular implementation based on PostgreSQL. We believe they are representative in the vector database space. Using them as comparisons provides a reliable indication of our system's performance. Since pgvector [16] is very slow in VectorDBBench during index construction and search phases, only a limited number of experiments are conducted.

We do not compare SingleStore-V with AnalyticDB-V [51] because AnalyticDB-V is significantly slower than both pgvector and Milvus, as demonstrated in [55] and [49]. We also do not compare SingleStore-V with PASE [52], due to the considerably slower performance of PASE compared to specialized vector databases like Milvus, as indicated in [55].

As Milvus is an in-memory vector database that stores everything in memory, to ensure a fair comparison, we allocate enough buffer memory in both pgvector and SingleStore-V to cache the entire vector data and vector index in memory.

Indexes. For vector indexes, we evaluate both quantization-based and graph-based indexes in SingleStore-V. Specifically, we choose IVF_PQFS as it is the fastest quantization-based index. We also choose HNSW because it is a widely used graph-based index in many vector databases due to its high search performance and recall.

Parameters. We largely follow the terminology for each database or platform to introduce the parameters and set default values, as shown in Table 3. Unless otherwise noted, we use the default parameters for experiments.

Evaluation Metrics. We adopt commonly used metrics for measuring vector search, which include throughput (queries per second (QPS)), recall, and index construction time. We utilize recall to evaluate the accuracy of the returned top-k results. Specifically, we compare the set G of ground truths with the set R of top-k results returned by the system. Recall is then calculated as $|R \cap G|/|G|$, representing the percentage of correctly identified items in the ground truths.

Table 3: Parameters and Default Values

Parameters	Meaning and Default Value
k	The number of results selected from vector similarity search **Default Value**: 100
reorder_k	The number of results selected for each segment in IVF_PQFS **Default Value**: 1000
nlist	The number of centroids in IVF_FLAT and IVF_PQFS **Default Value**: 1000 in GIST1M, 3162 in SIFT10M and Cohere10M, 10000 in SIFT100M, 31622 in SIFT1B
n_{probe}	The number of selected centroids in IVF_FLAT and IVF_PQFS **Default Value**: 20
qr	Quantization ratio, dimension/qr is the number of subvectors into which each vector will be divided in IVF_PQFS. **Default Value**: 4
M	The number of neighbors for a vector node in HNSW **Default Value**: 16
efc	The queue length in HNSW build **Default Value**: 128
efs	The queue length in HNSW search **Default Value**: 200

6.2 Comparing with Other Vector Databases

In this subsection, we compare our SingleStore-V with Milvus and pgvector in terms of search performance and index construction time on IVF_PQFS and HNSW in different datasets.

6.2.1 Search Performance

With indexes built using the same parameters, we present the throughput (QPS) for each system under different recalls. Figure 3a, 4a and 5a show the the search performance of SingleStore-V comparing with Milvus and pgvector in different datasets.

SingleStore-V achieves **comparable performance** comparing to Milvus in different datasets. For HNSW, SingleStore-V can reach **81.8%** ~ **94.7%** QPS of Milvus and **1.7**× ~ **2.6**× faster than pgvector among different datasets. In terms of IVF_PQFS, the QPS of SingleStore-V is **78.7%** ~ **98.9%** of Milvus and **47**× ~ **110**× better than the IVF_FLAT in pgvector among different datasets under different recalls. The minor performance gap is due to some constant O(1) overhead in SingleStore-V's SQL pipeline as a generalized database.

6.2.2 Index Construction Performance

Figure 3b, 4b, and 5b show the index construction times on different datasets. These figures indicate that SingleStore-V exhibits the fastest index build performance among the systems compared. In terms of the HNSW index, pgvector requires **40.3**× ~ **74.8**× **more time** than SingleStore-V on GIST1M and SIFT10M, and Milvus requires **1.6**× ~ **2.7**× **more time** than SingleStore-V on various datasets. For the quantization-based index, despite IVF_PQFS incorporating a PQ calculation process that IVF_FLAT lacks, pgvector still requires **4.6**× **more time** on SIFT1M and **4.5**× **more time** on GIST1M for IVF_FLAT than SingleStore-V's IVF_PQFS. Milvus's IVF_PQFS also takes **5.2**× ~ **6.0**× **more time** than SingleStore-V on different datasets. The higher performance of SingleStore-V compared to Milvus is attributed to SingleStore-V having an optimized API for loading data into the table before index building, with the

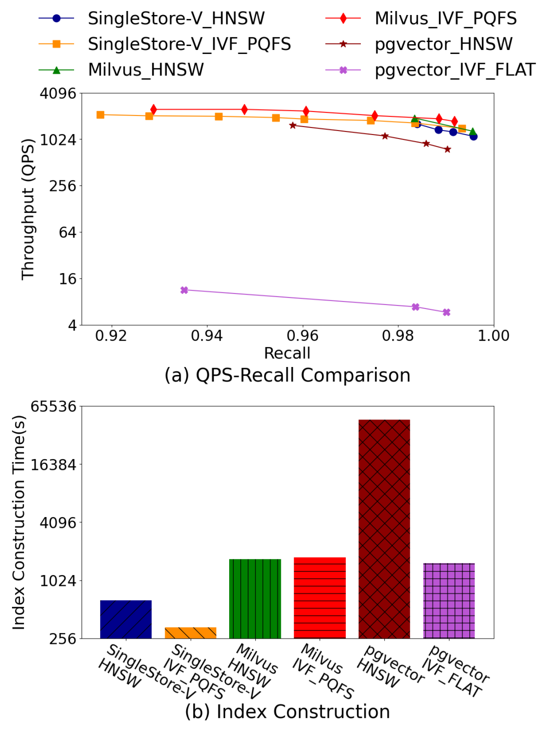


Figure 3: Comparing Vector Databases on SIFT10M

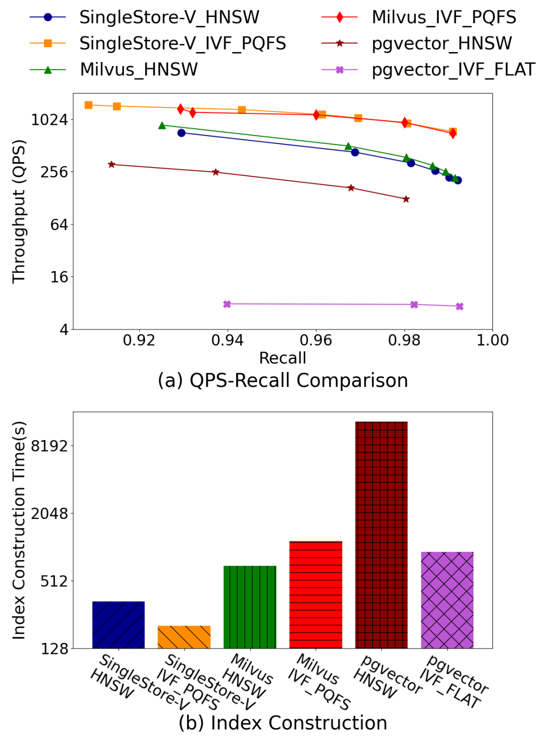
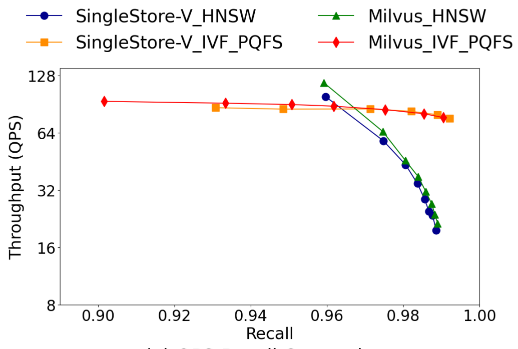
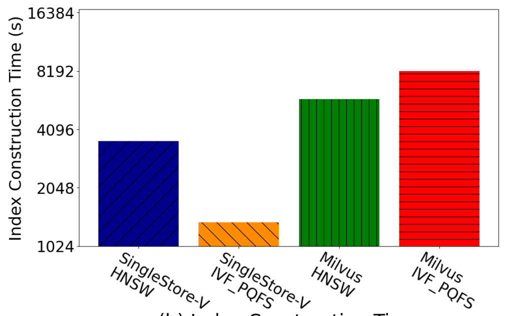


Figure 4: Comparing Vector Databases on GIST1M



(a) QPS-Recall Comparison



(b) Index Construction Time

Figure 5: Comparing Vector Databases on Cohere10M

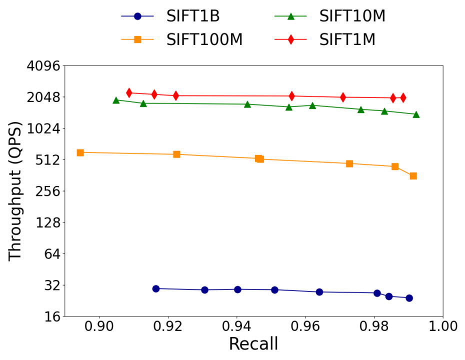
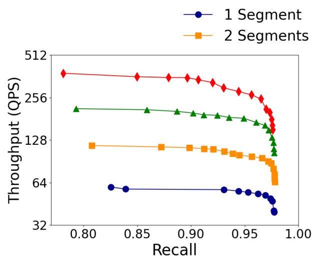


Figure 6: Evaluating Scalability of SingleStore-V on 1M, 10M, 100M, 1B Vectors

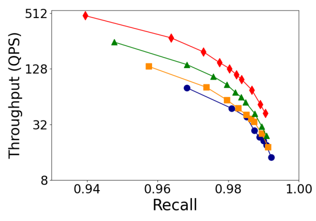
time spent on index construction being mostly the same in both systems.

6.3 Evaluation of Scalability

In this experiment, we evaluate the scalability of SingleStore-V with respect to data size. We use the SIFT1M, SIFT10M, SIFT100M, and SIFT1B datasets, which range from 1 million vectors to 1 billion vectors. In this experiment, we use the IVF_PQFS index in SingleStore-V. We conduct experiments on the CPU instance of r6i.4xlarge (128 vCPUs and 1TB memory) and gp2 storage with 32 concurrency, also using the default parameters as outlined in Table 3. Figure 6 demonstrates that both QPS and recall decrease proportionally with the increase in data sizes.



(a) IVF_PQFS Recall-QPS Comparison



(b) HNSW Recall-QPS Comparison

Figure 7: Evaluating Cross-Segment Index Merger

6.4 Evaluation of Vector Index Merger

In this experiment, we evaluate the performance of the index merger in SingleStore-V with regard to a varying number of segments per vector index on the Cohere10M dataset. We used the same type of instance as in the search performance experiment, r6a.8xlarge. We chose the default values shown in table 3, and used different search parameters to measure throughput at various recall levels. The per-segment size is limited to at most 80,000 to achieve 8 segments in each of the 16 partitions. We chose 80,000 instead of the exact average of 78,125 to accommodate any small variances that partitioning might cause. We conducted experiments on the per-segment index as a baseline and evaluated the performance boost from the index merger when it creates cross-segment indexes for 2, 4, and 8 segments, respectively. Figure 7 shows that the throughput scales well as the index merger creates larger cross-segment indexes.

7 CONCLUSION

In this work, we have presented SingleStore-V, a generalized vector database designed to achieve both high performance and high interoperability. By employing per-segment vector indexes and vector index merger, SingleStore-V ensures scalability and efficiency on vector search queries. The pluggable vector index architecture allows SingleStore-V to leverage a wide range of algorithms and libraries, while also providing a user-friendly interface through auto vector indexing. Moreover, the `Top()` filter introduced in this paper facilitates the pushdown of `ORDER BY ... LIMIT` to the table scan, enabling seamless integration with other SQL constructs. Through expressive SQL semantics, SingleStore-V empowers complicated analytic queries that combine vector similarity, filters, fulltext, and joins to provide insights in real-time.

For future work, it would be interesting to explore ways of indexing different kinds of vector data, such as sparse vectors instead of dense vectors as discussed in this paper. Additionally, further investigation into the applications of vector search in SQL and alternative methods for performing analytics on both vector and non-vector data would be valuable. Furthermore, investigating broader uses of the `Top()` filter beyond `ORDER BY ... LIMIT` pushdown presents an interesting opportunity.

8 ACKNOWLEDGMENTS

We extend our gratitude to Eric (Yongzhu) Li, Viacheslav Ocheretnyi, Eeshwar Gurushankar, Sofiia Melnyk, Alex Moore, and Kristin Tufte for their work on SingleStore-V.

REFERENCES

[1] [n. d.]. AutoFaiss (https://github.com/criteo/autofaiss).

[2] [n. d.]. ClickHouse Approximate Nearest Neighbor Search Indexes [Experimental] (https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/annindexes).

[3] [n. d.]. Cohere (https://huggingface.co/datasets/Cohere/wikipedia-22-12/tree/main/en/).

[4] [n. d.]. ElasticSearch Github (https://github.com/elastic/elasticsearch).

[5] [n. d.]. Facebook Faiss. https://github.com/facebookresearch/faiss

[6] [n. d.]. Faiss AutoTune (https://github.com/facebookresearch/faiss/wiki/Index-IO,-cloning-and-hyper-parameter-tuning#auto-tuning-the-runtime-parameters).

[7] [n. d.]. Faiss Indexes. https://github.com/facebookresearch/faiss/wiki/Faiss-indexes

[8] [n. d.]. Fast Accumulation of PQ and AQ Codes (FastScan) (https://github.com/facebookresearch/faiss/wiki/Fast-accumulation-of-PQ-and-AQ-codes-(FastScan)).

[9] [n. d.]. Guidelines to Choose an Index in Faiss (https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index).

[10] [n. d.]. Introducing Vector Search on Rockset: How to Run Semantic Search with OpenAI and Rockset (https://rockset.com/blog/introducing-vector-search-on-rockset/).

[11] [n. d.]. Lumana (https://www.lumana.ai/).

[12] [n. d.]. Milvus AUTOINDEX Explained (https://docs.zilliz.com/docs/autoindex-explained).

[13] [n. d.]. Milvus Knowhere (https://milvus.io/docs/knowhere.md).

[14] [n. d.]. MongoDB (https://www.mongodb.com).

[15] [n. d.]. MongoDB Vector Search (https://www.mongodb.com/products/platform/atlas-vector-search).

[16] [n. d.]. pgvector (https://github.com/pgvector/pgvector).

[17] [n. d.]. Pinecone (https://www.pinecone.io/).

[18] [n. d.]. Qdrant (https://qdrant.tech/).

[19] [n. d.]. Recommender System in Milvus (https://milvus.io/docs/recommendation_system.md).

[20] [n. d.]. SIFTData (http://corpus-texmex.irisa.fr/).

[21] [n. d.]. VectorDBBench: A Benchmark Tool for VectorDB (https://github.com/zilliztech/VectorDBBench).

[22] [n. d.]. Vespa (https://vespa.ai/).

[23] [n. d.]. Weaviate (https://weaviate.io/).

[24] 2023. Oracle Introduces Integrated Vector Database to Augment Generative AI and Dramatically Increase Developer Productivity (https://www.oracle.com/news/announcement/ocw-integrated-vector-database-augments-generative-ai-2023-09-19/).

[25] 2023. Relevance Scoring in Hybrid Search using Reciprocal Rank Fusion (RRF) (https://learn.microsoft.com/en-us/azure/search/hybrid-search-ranking).

[26] 2023. Vector Search with Azure SQL Database (https://devblogs.microsoft.com/azure-sql/vector-search-with-azure-sql-database/).

[27] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. 2016. The MemSQL Query Optimizer: A Modern Optimizer for Real-time Analytics in a Distributed Database. *Proceedings of the VLDB Endowment (PVLDB)* 9, 13 (2016), 1401–1412.

[28] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 5199–5212.

[29] Sheng Chen, Akshay Soni, Aasish Pappu, and Yashar Mehdad. 2017. Doc-Tag2Vec: An Embedding Based Multi-label Learning Approach for Document Tagging. In *Proceedings of the 2nd Workshop on Representation Learning for NLP, Rep4NLP@ACL 2017*. 111–120.

[30] Jens Dittrich and Alekh Jindal. 2011. Towards a One Size Fits All Database Architecture. In *Conference on Innovative Data Systems Research (CIDR)*. 195–198.

[31] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proceedings of the VLDB Endowment (PVLDB)* 12, 5 (2019), 461–474.

[32] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *International Conference on Very Large Data Bases (VLDB)*. 518–529.

[33] Samuel Leonardo Gracio. 2023. Reinvent your recommender system using Vector Database and Opinion Mining (https://medium.com/dailymotion/reinvent-your-recommender-system-using-vector-database-and-opinion-mining-a4fadf97d020).

[34] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 33, 1 (2011), 117–128.

[35] Madhukar Kumar, Yaroslav Demenskyi, and Pranav Aurora. 2024. How We Built a Real-Time RAG Application for Free With SingleStore and Vercel (https://www.singlestore.com/blog/real-time-rag-app-with-singlestore-and-vercel/).

[36] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*. 75–88.

[37] Yuliang Li, Jianguo Wang, Benjamin S. Pullman, Nuno Bandeira, and Yannis Papakonstantinou. 2019. Index-Based, High-Dimensional, Cosine Threshold Querying with Optimality Guarantees. In *International Conference on Database Theory (ICDT)*, Vol. 127. 11:1–11:20.

[38] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. 2020. VHP: Approximate Nearest Neighbor Search via Virtual Hypersphere Partitioning. *Proceedings of the VLDB Endowment (PVLDB)* 13, 9 (2020), 1443–1455.

[39] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2017. Intelligent Probing for Locality Sensitive Hashing: Multi-Probe LSH and Beyond. *Proceedings of the VLDB Endowment (PVLDB)* 10, 12 (2017), 2021–2024.

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

[41] Michal Nowakiewicz, Eric Boutin, Eric N. Hanson, Robert Walzer, and Akash Katipally. 2018. BIPie: Fast Selection and Aggregation on Encoded Data using Operator Specialization. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1447–1459.

[42] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of Vector Database Management Systems. *CoRR* abs/2310.14021 (2023).

[43] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.

[44] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 2340–2352.

[45] Alex Skidanov, Anders J. Papito, and Adam Prout. 2016. A Column Store Engine for Real-time Streaming Analytics. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1287–1297.

[46] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 2–11.

[47] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 13748–13758.

[48] Yukihiro Tagami. 2017. AnnexML: Approximate Nearest Neighbor Search for Extreme Multi-label Classification. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 455–464.

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

[50] Runhui Wang and Dong Deng. 2020. DeltaPQ: Lossless Product Quantization Code Compression for High Dimensional Similarity Search. *Proceedings of the VLDB Endowment (PVLDB)* 13, 13 (2020), 3603–3616.

[51] 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 (PVLDB)* 13 (2020), 3152–3165.

[52] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 2241–2253. https://github.com/alipay/PASE

[53] Alan Zeichick. 2023. What Is Retrieval-Augmented Generation (RAG)? (https://www.oracle.com/artificial-intelligence/generative-ai/retrieval-augmented-generation-rag/).

[54] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 377–395.

[55] 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 *International Conference on Data Engineering (ICDE)*.

[56] Bolong Zheng, Xi Zhao, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S. Jensen. 2020. PM-LSH: A Fast and Accurate LSH Framework for High-Dimensional Approximate NN Search. *Proceedings of the VLDB Endowment (PVLDB)* 13 (2020), 643–655.