TigerVector: Supporting Vector Search in Graph Databases for Advanced RAGs

Shige Liu
Purdue University
liu3529@purdue.edu

Zhifang Zeng
TigerGraph
michael.zeng@tigergraph.com

Li Chen
TigerGraph
li.chen@tigergraph.com

Adil Ainihaer
TigerGraph
adil.ainihaer@tigergraph.com

Arun Ramasami
TigerGraph
arun.ramasami@tigergraph.com

Songting Chen
TigerGraph
songting.chen@tigergraph.com

Yu Xu
TigerGraph
yu@tigergraph.com

Mingxi Wu
TigerGraph
mingxi.wu@tigergraph.com

Jianguo Wang
Purdue University
csjgwang@purdue.edu

ABSTRACT

In this paper, we introduce TigerVector, a system that integrates vector search and graph query within TigerGraph, a Massively Parallel Processing (MPP) native graph database. We extend the vertex attribute type with the `embedding` type. To support fast vector search, we devise an MPP index framework that interoperates efficiently with the graph engine. The graph query language GSQL is enhanced to support vector type expressions and enable query compositions between vector search results and graph query blocks. These advancements elevate the expressive power and analytical capabilities of graph databases, enabling seamless fusion of unstructured and structured data in ways previously unattainable. Through extensive experiments, we demonstrate TigerVector's hybrid search capability, scalability, and superior performance compared to other graph databases (including Neo4j and Amazon Neptune) and a highly optimized specialized vector database (Milvus). TigerVector was integrated into **TigerGraph v4.2**, the latest release of TigerGraph, in December 2024.

1 INTRODUCTION

Retrieval-augmented generation (RAG) is an emerging technology that grounds LLMs to enhance accuracy and reliability by retrieving relevant context from external data sources [16]. Today, most RAGs are based on vector databases [10, 36], which store the semantic embeddings of underlying data. However, vector-based RAGs fall short in many complex applications, as they cannot capture the accurate relationships between underlying data objects. Thus, vector-based RAGs frequently suffer from poor prompt hit rates [17, 31], particularly in scenarios where the retrieved context does not fully align with user queries. This can result in multiple iterative LLM API calls to refine the response, increasing both latency and costs.

To address these challenges, a new concept called GraphRAG was recently proposed [14, 15, 28]. Instead of using vector databases, GraphRAG leverages graph databases (e.g., Neo4j [15] or TigerGraph [12]) to store knowledge graphs of underlying data. This approach is useful for answering questions that may require understanding information across multiple documents, such as finding all positive reviews written by a specific customer or summarizing the impact of COVID-19 on the global economy.

In this paper, we advocate for a hybrid RAG, *VectorGraphRAG*, which combines vector-based and graph-based RAGs to leverage the best of both worlds. This approach will enable many new possibilities for grounding LLMs by leveraging both vector search and graph search [15]. For example, one could perform vector search and graph search separately to retrieve different candidate sets and merge them to obtain a comprehensive result set. One could also use vector search to identify a smaller set of results first and then apply graph traversal to expand it for more relevant context.

In particular, we aim to support VectorGraphRAG by developing TigerVector, which integrates vector search seamlessly into TigerGraph, a distributed graph database system.

While the straightforward way to support VectorGraphRAG is to use two separate databases (i.e., a vector database and a graph database), we argue that it is highly desirable to have a *unified system* that natively supports both vector data and graph data. First, it reduces data movement and minimizes data silos by keeping both vector data and graph data within the same system. Second, it maintains data consistency by directly linking the vector embeddings with their source data. Third, it facilitates hybrid searches of vector and graph data using a unified query language, which is GSQL [13] in our case. Fourth, it supports efficient data governance by providing a single set of access controls (e.g., role-based access control) for both vector data and graph data.

Challenges. It is challenging to efficiently support vector search within graph database systems. (1) Since vector data is not a first-class citizen in graph databases, it is non-trivial to achieve high query performance compared to specialized vector database systems like Milvus [36]. (2) Since graph databases utilize a graph query language, it remains unclear how to represent different types of vector search queries (such as pure vector search, filtered vector search, and hybrid vector and graph search) and execute them efficiently. (3) It is crucial to support updates that are both transactionally consistent and efficient, ensuring that updates to vector data do not negatively impact other graph data. (4) It is challenging to efficiently manage different types of vector embeddings (e.g., text embeddings or image embeddings) within graph databases.

Limitations of Prior Work. Although a few graph databases (e.g., Neo4j [23] and Amazon Neptune [25]) have started to support vector search, they address the above challenges only partially and have many limitations. For example, Neo4j and Neptune lack high performance, fail to support advanced vector search (e.g., filtered vector search and hybrid search), do not provide efficient and atomic updates, and lack support for multiple embedding types.

Overview of TigerVector. In this paper, we present TigerVector, a new system that adds efficient vector search support within TigerGraph, a distributed graph database system. It introduces vector embeddings as a new attribute of existing graph nodes, similar to other attributes within the graph nodes. Moreover, it supports multiple types of vector embeddings within the same graph node. The vector index is based on HNSW [20], which is widely used in modern vector databases. TigerVector introduces a suite of techniques to address the above challenges as described below.

To address the performance challenge, TigerVector fully leverages TigerGraph's MPP architecture, allowing multiple compute cores or machines to process queries in parallel. Moreover, TigerVector decouples the storage of vector embeddings from other graph attributes to best utilize the native vector index implementation.

To address the usability challenge, TigerVector integrates vector search into GSQL [13], TigerGraph's SQL-like declarative graph query language. In particular, it supports both pure vector search and advanced vector search, including filtered vector search, vector search on graph patterns, and vector similarity join on graph patterns. Moreover, TigerVector supports flexible vector search functions that seamlessly interact with other GSQL features such as vertex set variables and graph algorithms.

To address the update challenge, TigerVector supports both efficient and atomic updates. Specifically, the decoupling of vector embeddings from other attributes in graph nodes allows TigerVector to handle incremental vector index updates effectively. This design also facilitates graph updates and reduces update overhead. Moreover, updates involving both graph attributes and vector attributes are performed atomically.

For efficient vector embedding management, TigerVector introduces a new data type called `embedding` to handle vector embeddings decoupled from other graph attributes. More importantly, the `embedding` type not only manages important metadata, such as dimensionality and distance metric, but also enables the definition of multiple embedding types.

Experimental Overview. Experiments on SIFT100M and Deep100M show that TigerVector significantly outperforms both Neo4j and Amazon Neptune in vector search. Specifically, TigerVector achieves **3.77**× to **5.19**× higher throughput and **23%** to **26%** higher recall rate compared to Neo4j. Moreover, TigerVector achieves **1.93**× to **2.7**× higher throughput at significantly lower hardware costs (**22.42**× less) compared to Amazon Neptune while maintaining a similar recall rate. Besides that, TigerVector also achieves comparable, and sometimes even higher, throughput compared to Milvus.

Contributions. The main contribution of this work is TigerVector, a novel system that efficiently supports vector search within TigerGraph, a distributed graph database system. It introduces a suite of new techniques (see Sec. 3) to address the challenges in performance and usability. Extensive experiments show that TigerVector can achieve comparable and even higher performance than Milvus, a highly optimized specialized vector database. TigerVector also significantly outperforms other graph databases, such as Neo4j and Amazon Neptune, in vector search performance. Moreover, TigerVector supports advanced vector search efficiently. TigerVector was integrated into **TigerGraph v4.2** in December 2024. The community edition can be downloaded via https://dl.tigergraph.com/.

Although TigerVector is designed based on TigerGraph, many of the proposed design decisions and techniques are applicable to other graph databases. Further discussion can be found in Sec. 7.

2 BACKGROUND AND RELATED WORK

2.1 TigerGraph

Graph databases have emerged as a new database category. Prominent commercial graph databases include TigerGraph [12], Neo4j [22], and Amazon Neptune [24]. A native graph database is purpose-built for managing graph data, with nodes, edges, and properties directly represented in its storage architecture. Relationships (edges) are treated as first-class entities, typically implemented as direct pointers or links between nodes, enabling highly efficient traversal operations. Beyond traditional data compression techniques, native graph databases also employ graph-specific compression methods optimized for their structures.

Graph databases excel in handling graph-shaped query workloads, offering significant performance advantages over other database types. Benchmark suites like LDBC-SNB, LDBC-FinBench, and LDBC-GraphAnalytics measure performance for such workloads [1]. The key performance advantage of graph databases lies in their design: common joins are materialized as edges during data ingestion. At runtime, the query engine traverses these edges to access related data directly, often eliminating the need for expensive join operations.

TigerGraph employs the property graph model [7, 34], representing data through vertex and edge types. Its graph model allows both directed and undirected edges, and allows multiple edges (of the same type or different types) between two nodes.

TigerGraph is a distributed, native graph database designed for scalability and high performance. It supports deployment on clusters, with data automatically partitioned and distributed across nodes. In the storage layer, data is organized into units called *segments*, each containing a fixed number of vertices. These segments serve as the fundamental units for parallel and distributed computing. Outgoing edges are stored within the source vertex's segment. The compute layer leverages a massively parallel processing architecture to enable hybrid transactional and analytical processing. Two key parallel primitives, *VertexAction* and *EdgeAction*, allow user-defined functions to operate across segments in parallel.

TigerGraph's query language, GSQL [13], is a Turing-complete graph database query language. One of its key advantages over other graph query languages is its support for accumulators [13], which can be either global or vertex-local. It also includes flow control primitives such as `WHILE`, `IF-ELSE`, `FOREACH`, and others to support graph algorithms. GSQL enables query block composition using vertex set variables and accumulators. A query is composed of a sequence of query blocks (`SELECT-FROM-WHERE`), each producing a vertex set variable. Query blocks in the `FROM` clause can utilize vertex set variables from prior blocks, thus facilitating query block composition. Additionally, GSQL supports `UNION`, `INTERSECT`, and `MINUS` binary operators between two vertex set variables.

Accumulators can be used as another compositional tool in GSQL. They are runtime variables that are mutable throughout the query life cycle. Global accumulators (prefixed by `@@`) can be read and written within a query block or across different query blocks. Vertex-local accumulators (prefixed by `@`) can be read and written within a query block when their associated vertex is accessible. Since vertices can be activated in different query blocks, query composition is facilitated through their attached runtime vertex accumulators.

2.2 Vector Databases

Vector databases [10, 26, 27, 35, 36] have recently gained significant attention due to their important role in grounding LLMs through vector-based RAGs.

There are two types of vector databases in the literature: specialized vector databases and integrated vector databases. Specialized vector databases are designed explicitly to manage vector data, achieving high performance and scalability. Examples include Milvus [36], Pinecone [30], and Weaviate [4]. Integrated vector databases aim to incorporate vector search into existing data systems, such as relational databases [39] and graph databases [23]. Examples include pgvector [29], PASE [38], and SingleStore-V [10].

This work follows the integrated approach because real-world applications often involve more than just vector data. A unified system can address various issues, such as data consistency, data silos, and advanced vector search, as discussed in Sec. 1.

2.3 Supporting Vector Search in Graph Databases

To the best of our knowledge, only a few graph databases, including Neo4j [23], Amazon Neptune [25], and DGraph [11], support vector search. However, their support is rudimentary and comes with many limitations. For example, Neo4j implements vector search using Lucene's index [3]. However, it does not support index parameter tuning, which is crucial in vector databases to achieve high performance. It also lacks support for advanced vector search, declarative vector search using a graph query language, and the ability to accommodate multiple embedding types. Our system addresses all of these limitations.

Amazon Neptune [25] and DGraph [11] share the limitations mentioned above. In addition, Neptune explicitly states that updates to the vector index are not atomic [25], whereas TigerVector supports transactional updates. Moreover, Neptune builds a single vector index for the entire graph, and this vector index is not distributed, which significantly limits its scalability for vector search [25]. In contrast, TigerVector's vector index is distributed.

3 SYSTEM DESIGN OVERVIEW

In this section, we present the system architecture of TigerVector, as illustrated in Figure 1. Our system is built inside TigerGraph, a distributed graph database, which partitions vertices into segments and evenly distributes these segments across multiple machines to fully utilize all compute resources. TigerVector manages embeddings using a vertex-centric approach, storing all embeddings related to the same vertex segment together and building a vector index per segment, thereby unifying the management of vector data and graph data.

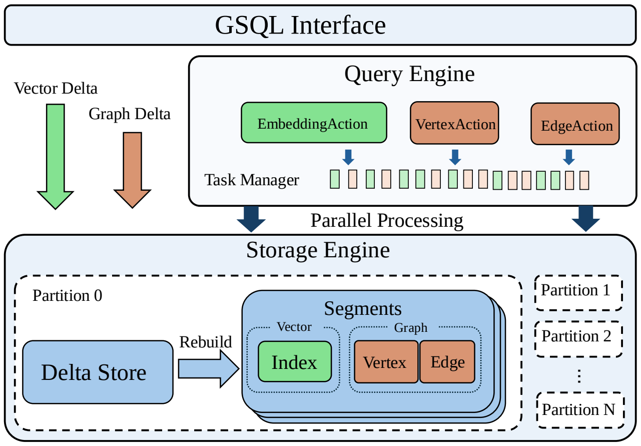


Figure 1: System Overview

Since TigerGraph is a property graph database, where each node contains multiple key-value pairs as attribute properties, TigerVector introduces vector embeddings as a new attribute for existing graph nodes, similar to other attributes. Moreover, TigerVector supports multiple embedding types within the same graph node, as some applications may require different vectors for various types of data (e.g., document data or image data).

Next, we highlight the key features of TigerVector.

Embedding Type. Vectors are (float) data arrays generated by machine learning models that can capture the semantic meaning of the underlying data. Instead of managing vectors using data type `LIST<float>` as if they were simple lists of floats, we introduce a new `embedding` data type to efficiently manage vectors and associated metadata, such as dimension, data type, and model information. We also introduce the concept of `embedding space`, which defines multiple `embedding` types from a specific machine learning model. More details can be found in Sec. 4.1.

Decoupled Storage for Vectors. Vector embeddings associated with the same vertex segments are stored as a single embedding segment and are stored separately from other attributes, allowing them to be managed independently of their vertex segments. Vector indexes are built for each embedding segment and are incrementally updated as new vector deltas are incorporated into embedding segments. Background vacuum processes are deployed for efficient updates while ensuring transaction support. The number of index update threads is dynamically tuned to strike a balance between efficiency and responsiveness for other queries.

MPP Design. We fully leverage the MPP (Massively Parallel Processing) architecture of TigerGraph to achieve high performance. In this architecture, graphs and vectors are partitioned into vertex segments and embedding segments. Queries on large datasets are distributed as multiple tasks across these vertex and embedding segments, enabling parallel processing and distributed processing. Sec. 5.1 presents more details.

GSQL-integrated Declarative Vector Search. We integrate vector search into GSQL [13] to support declarative vector search by extending the ORDER BY...LIMIT syntax. More advanced queries, such as filtered vector search, vector search on graph patterns, and vector similarity join on graph patterns, are also supported. Detailed descriptions are provided in Sec. 5.

Flexible Vector Search Function. Our system also supports a powerful vector search function – VectorSearch(), enabling flexible vector searches across multiple vertex types. It seamlessly integrates into GSQL queries by accepting a vertex set variable from prior query blocks for filtered searches and returning a vertex set variable for further composition. Optional parameters include an accumulator that returns top-k distances and corresponding vertices, and an index search parameter that adjusts search accuracy. By using vertex set variables as input and output, the function supports easy integration with graph algorithms, unlocking new avenues for advanced vector search applications.

4 VECTOR INDEX DESIGN

4.1 Embedding Type

In TigerVector, we introduce a new data type called embedding to manage the vector attributes within graph vertices. It specifies metadata for the vectors, including dimensionality, the machine learning model used to generate the embedding, the vector index, the vector data type, and the similarity metric. Furthermore, each graph vertex can have one or more embedding attributes alongside other attributes, which provides finer granularity and more flexibility.

Consider an example from the LDBC SNB (LDBC Social Network Benchmark) [33], a standard benchmark for graph databases that models a social network application. It includes entities such as people, posts, comments, and places, as well as their relationships and activities, including creates, replies, and likes. We can augment each Post node with a vector embedding representing its content, as follows.

```
-- embedding type example
CREATE VERTEX Post (
    id INT PRIMARY KEY,
    author STRING,
    content STRING
);

ALTER VERTEX Post
ADD EMBEDDING ATTRIBUTE content_emb (
    DIMENSION = 1024,
    MODEL = GPT4,
    INDEX = HNSW,
    DATATYPE = FLOAT,
    METRIC = COSINE
);
```

Defining the embedding type has many advantages compared to relying solely on existing data types such as LIST<FLOAT>. First, vectors are not merely simple arrays of floats. The metadata associated with vectors, such as dimensions, data types, and models, is also important and should be explicitly managed. For example, vector search across multiple vertex types is enabled using the embedding type, which allows searching multiple vector attributes generated by the same model. The key challenge in this search is to ensure compatibility, as vectors with different dimensions or generated by different models must be handled differently. We address this challenge through static analysis of the embedding data involved during query compilation, leveraging the metadata from embedding types. If all aspects of the vector metadata, except for the index type, are identical, the query is allowed. Otherwise, the query is rejected and a semantic error is returned.

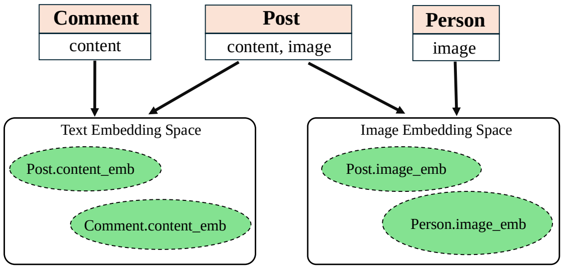


Figure 2: Example of Embedding Space

Second, it simplifies the data loading process because vectors and other graph attributes often originate from different data sources, with vectors typically generated by a machine learning model. For example, we can easily load data from two separate files (f1 and f2) to populate vectors and graph attributes using the following script, where vector values are assumed to be separated by ":". Otherwise, this process would be much more complex.

```
-- data loading example
CREATE loading job j1 FOR graph g1{
  LOAD f1 TO VERTEX Post VALUES (id, author, content);
  LOAD f2 TO EMBEDDING ATTRIBUTE content_emb
    ON VERTEX Post VALUES (id, split(content_emb, ":"))
        ;
 }
```

Third, as we will physically separate the storage of vector embeddings from other graph attributes (as discussed in Sec. 4.2), introducing the embedding type facilitates this process.

Next, we introduce a new embedding space type, which defines the vector attributes for multiple vertex types. This is for cases where users want a unified schema for all embeddings generated from the same model. For example, in Figure 2, if both the Post and Comment types have a content vector attribute generated by the GPT4 model, then we can define the schema as follows:

```
-- embedding space example
CREATE EMBEDDING SPACE GPT4_emb_space (
  DIMENSION = 1024,
  MODEL = GPT4,
  INDEX = HNSW,
  DATATYPE = FLOAT,
  METRIC = COSINE
);

ALTER VERTEX Post
ADD EMBEDDING ATTRIBUTE content_emb
IN EMBEDDING SPACE GPT4_emb_space;
```

4.2 Decoupled Storage for Vectors

In TigerVector, the storage of vectors is separated from other graph attributes because vectors typically have a much larger data volume. For example, an embedding generated by OpenAI can have

1536 dimensions, occupying at least 6144 bytes, which is significantly larger than typical attributes of the INT or STRING type. We separate vector storage from other graph attributes to reduce data duplication, as vectors are already stored in the vector indices. Furthermore, TigerGraph uses multiple versions of vertex segments during delta rebuilding, a.k.a., vacuum, applying deltas to old segments to produce new segments before switching to them. Storing embedding values alongside normal graph attributes in these vertex segments not only delays the rebuilding process but also causes a large amount of disk I/O. Therefore, we manage vector storage separately through an embedding service module.

To leverage the MPP architecture for parallel and distributed processing, TigerVector stores vectors per vertex segment. Specifically, TigerGraph uses a vertex-centric method to partition vertices into segments. The vectors follow the same partition scheme but are stored separately as *embedding segments*, as shown in Figure 3. Each vector attribute on the vertices has its own embedding segments. TigerVector builds a vector index for each embedding segment, limiting the index size to the maximum size of the vertex segment. When sufficient memory is available, all vertex segments, including vertex data and edge data, are loaded into memory. However, for embedding segments, only the vector indexes are loaded for vector search to reduce memory consumption.

We choose to partition the vector embeddings and build a separate vector index for each segment, rather than constructing a single large HNSW index for all vectors and then partitioning the graph index. We observe that querying a partitioned graph index distributed across a cluster incurs significant network overhead during traversal. In contrast, our approach only requires sending the query to different servers; each server performs a local search and sends the results back for a global merge, which minimizes network communication.

While it is possible to build cross-segment indexes to search across multiple segments, as demonstrated by SingleStore-V [10], we choose to build a segment-level index using vertex-centric partitioning to achieve better elasticity and horizontal scaling. For instance, vector search and graph traversal are often combined within a single query (Sec. 5). Ensuring that the embedding segments and vertex segments reside on the same node enables local computation, minimizing network communication.

Unlike SingleStore-V [10], segments in TigerGraph are always mutable. Having vertex-centric embedding segments ensures that updates or failures are limited to the segment level and makes fault-tolerance easier to implement. In contrast, updating or recovering cross-segment indexes from failures can be more expensive and complex. Furthermore, ensuring high availability is simplified with embedding segment replicas distributed across the cluster.

4.3 Incremental Update

We next discuss how to efficiently handle vector updates, which include insertions, updates, and deletions of vectors, while ensuring transaction support in TigerVector.

ACID in TigerGraph. TigerGraph supports ACID properties for database transactions. In particular, it employs an MVCC (multi-version concurrency control)-based scheme. Each committed transaction is assigned a transaction ID (TID). A background vacuum process periodically cleans up the accumulated deltas and builds a snapshot up to a particular TID. Queries with a specific TID are processed by combining deltas and snapshots. Distributed and replicated write-ahead log (WAL) is used for durability. Finally, a transaction becomes visible only after it has been committed. TigerVector reuses many of the existing transaction processing techniques in TigerGraph, such as write-ahead logging and the atomic commit protocol, while employing a different algorithm for MVCC-based vector updates, as described below.

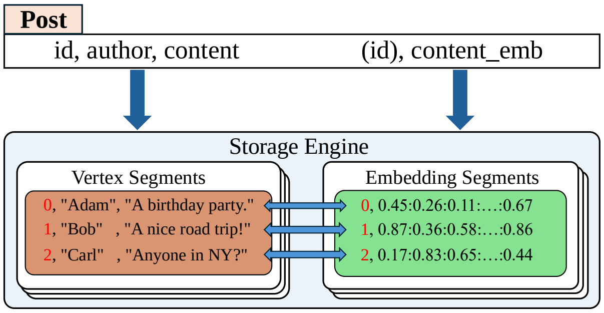


Figure 3: Decoupled Storage. Vectors within a vertex segment (left) are stored separately in another embedding segment (right), while keeping the same ids.

MVCC-based Vector Deltas. TigerVector also employs an MVCC scheme for vector updates. Each committed vector update is assigned a transaction ID (TID). These updates are accumulated as vector deltas in an in-memory delta store. The schema of vector deltas has four fields: Action Flag (Upsert/Delete), ID, TID, and Vector Value. The management of vector deltas is handled by the vacuum processes.

Vacuum Processes for Vector Deltas. The vacuum for vector updates in TigerVector consists of two steps: flushing deltas from the in-memory store to disk and updating vector indexes with deltas. However, the second step is much slower than the first one. Our tests show that 1 million 128-dimensional vectors can be flushed into a file within one second, but building a vector index based on these vectors takes more than 30 seconds, even with parallel optimization. Therefore, we decouple the vacuum into two processes: one delta merge process vacuums the in-memory delta store into disk delta files, and another index merge process vacuums the delta files into vector indexes.

As shown in Figure 4, the delta merge vacuum process incrementally creates a new file containing all the vector deltas up to a particular TID since the last build (the right side of the figure). The index merge process incrementally merges those delta files into the vector index, also up to a particular TID (the left side of the figure). After a new vector index snapshot is built, the engine will switch to the new snapshot. The old index snapshot and delta files are deleted only after the new index snapshot is visible to all running transactions. Vector search queries combine index snapshot search results with brute-force search results over vector deltas.

In TigerVector, we use multiple threads to merge delta files into one index snapshot in parallel. To minimize the impact on foreground query processing, we monitor the CPU utilization and dynamically tune the number of threads for parallel index updates

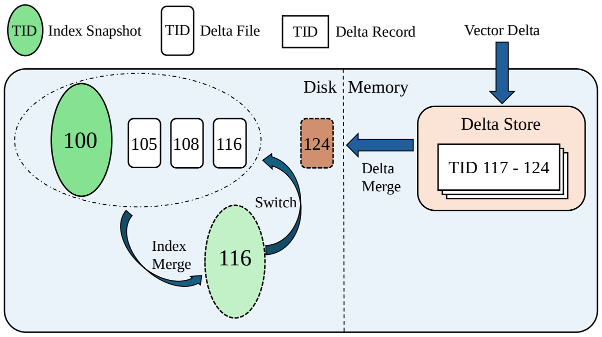


Figure 4: Incremental Vector Vacuum Processes. The delta merge process (right) flushes delta records into a new delta file. The index merge process (left) updates the index snapshot with a sequence of delta files.

to strike a balance between efficiency and responsiveness for other queries.

4.4 Vector Index Choice

Since vector embeddings are decoupled from other graph attributes in TigerVector, the implementation of vector indexes becomes highly flexible. This decoupling enables us to utilize existing native implementations of vector indexes for high performance.

In TigerVector, we currently support HNSW [20], since it is the most widely used vector index in modern vector databases such as Milvus [36] and SingleStore [10], offering high search performance and accuracy. However, other vector indexes (such as quantization-based indexes [18, 19]) can be easily integrated into TigerVector.

Specifically, we use an open-source HNSW library. There are four generic functions that we implement: `GetEmbedding`, `TopKSearch`, `RangeSearch`, and `UpdateItems`. `GetEmbedding` and `TopKSearch` are typically provided by the vector indexes libraries. Since HNSW does not provide a range search function, we adapt the DiskANN [32] approach to implement `RangeSearch` by performing multiple `TopKSearch` operations until the given threshold is smaller than the median of all distances.

We also implement parallel index building in `UpdateItems`, which incrementally updates the vector indexes using records from delta files. Specifically, each update thread works on a subset of ids to maintain record order. Additionally, we enhance the indexes to report relevant statistics for measuring its performance. With these functions in place, integrating additional vector indexes into TigerVector becomes straightforward.

5 VECTOR SEARCH DESIGN

TigerVector supports both declarative vector search as well as procedural vector search by integrating it into the GSQL [13] query language to enhance usability. In this section, we present the syntax and query optimizations.

5.1 Vector Search

In TigerVector, we extend GSQL to support top-k vector search by adopting the `ORDER BY...LIMIT` structure from SQL/GSQL syntax and using the keyword `VECTOR_DIST` to represent the similarity distance.

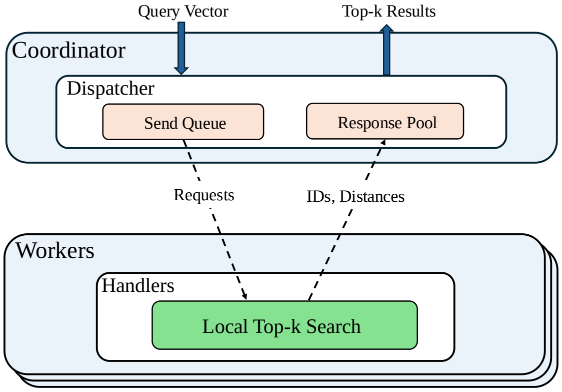


Figure 5: Distributed Query Processing. The coordinator prepares top-k vector search requests in the send queue and dispatches requests to worker servers. Each worker conducts top-k search locally and sends IDs and distances as results back to the response pool in the coordinator.

For example, finding the top-k posts similar to a query sentence (`query_vector`) can be expressed with the following query, where `content_emb` represents the embedding attribute of the `Post` node.

```
SELECT s
FROM (s:Post)
ORDER BY VECTOR_DIST(s.content_emb, query_vector)
LIMIT k;
```

Query Optimization. Since the vectors are partitioned into segments, each of which has its own index, TigerVector executes the query by performing a top-k search on each segment and then merging the local results to generate the global top-k results. We term this process **EmbeddingAction** in TigerVector. Specifically, a thread pool is employed to perform efficient parallel vector searches across multiple embedding segments. The query plan for this query is shown below:

```
EmbeddingAction[Top k, {s.content_emb}, query_vector]
```

TigerVector ensures that all vector results retrieved from the vector indexes are valid, excluding deleted or unauthorized vectors. This is achieved by passing a filter function, based on a bitmap (marking all deleted and unauthorized vectors as invalid), to the vector indexes. During the vector index search, each retrieved vector is evaluated using the filter function and added to the result list only if it is valid. Consequently, a single call to the vector index returns the valid top-k vectors for each segment.

There are a few optimizations in TigerVector to further improve performance. First, a threshold is set for the number of valid points in the bitmap. If the number of valid points is below the threshold, a brute-force search is conducted. This is because when there are too few valid points, the index must search through more vectors near the query vector to fetch enough valid results, which may take longer time than directly computing distances using the brute-force approach. Second, instead of generating a new bitmap, TigerVector reuses a global vertex status structure in TigerGraph and wraps it as a bitmap. This is particularly beneficial for pure vector searches, where no bitmap is passed from previous predicate evaluations. Generating a new bitmap can be computationally expensive, especially for large-scale datasets.

Distributed Vector Search. Since TigerVector is integrated with TigerGraph, a distributed graph database, it natively supports distributed vector search by operating on segments that are distributed across multiple machines. TigerVector designates one server as the coordinator, which distributes query tasks to all other servers (the work servers). Note that a coordinator can also function as a work server simultaneously. Once each segment retrieves its top-k results, the IDs and distances are sent back to the coordinator for a final merge, as illustrated in Figure 5.

Range Search. TigerVector also supports range search using the `WHERE` clause syntax. For example, finding all posts similar to a query sentence with a distance less than a given threshold can be expressed with the following query.

```
SELECT s
FROM (s:Post)
WHERE VECTOR_DIST(s.content_emb, query_vector)
      < threshold;
```

Similar to the top-k search described above, segment-level range search is performed first, and the results are then merged into the final output. Specifically, TigerVector creates an `EmbeddingAction` for the range search, which calls the `RangeSearch` API of the vector index during query execution. As outlined in Sec. 4.4, TigerVector implements `RangeSearch` using the approach from DiskANN [32].

5.2 Filtered Vector Search

Filtered vector search is an important operation in vector databases because, in real-world applications, vectors are often combined with other attributes to provide a filtered top-k search. For example, if we want to find the top-k English posts similar to a query sentence, the query can be expressed as follows:

```
SELECT s
FROM (s:Post)
WHERE s.language = "English"
ORDER BY VECTOR_DIST(s.content_emb, query_vector)
LIMIT k;
```

During query execution, TigerVector first processes the filter and generates a bitmap representing the qualified results, which is then passed to the vector search. As the vector search progresses, each candidate vector is evaluated against the bitmap to determine if it meets the filtering condition. This method is referred to as the *pre-filter* approach, where the filter is applied before the vector search. Alternatively, in the post-filter approach, the vector search is performed first, and the filtering condition is applied afterward [37]. Some approaches (e.g., [21]) combine two stages to improve performance by pruning vector partitions based on graph attributes. However, they introduce inflexibility by creating a tight coupling between graph and vector attributes, making them less favorable.

TigerVector uses the pre-filter approach for filtered vector search for several reasons. First, this approach is generic for any filters, as all predicates can be processed by the graph engine, and only the qualified items (represented as a bitmap) are passed to the vector search. Second, it achieves high query performance because only one vector search call is required to ensure there are k results. In contrast, the post-filter approach may produce less than k results, necessitating additional rounds of vector search with an enlarged k, which incurs high computational overhead under low selective filtering conditions. Third, we can only use the pre-filter approach when the graph query involves accumulators, a unique feature in TigerGraph (mentioned in Sec. 2.1), as detailed in Sec. 5.3.

The query plan for the example above is illustrated below, with execution proceeding from the bottom up.

```
EmbeddingAction[Top k, {s.content_emb}, query_vector]
VertexAction[Post:s {s.language = "English"}]
```

5.3 Vector Search on Graph Patterns

TigerVector supports hybrid query processing of vector search and graph pattern matching. It performs vector search while satisfying a specific graph pattern, which is important for advanced RAG applications. Current vector databases or relational databases with vector features cannot support this type of query because they lack graph query capabilities. Although Neo4j and Amazon Neptune provide vector search capabilities, they cannot perform vector search on a specific vertex set filtered by a graph pattern.

TigerVector seamlessly integrates vector search and graph queries. It allows vector search on any vertex alias within a graph pattern, and executes hybrid queries with great efficiency, even for multi-hop graph queries. For example, to identify the top-k long posts created by individuals connected to Alice that are most relevant to a given query sentence, we can express the query as follows:

```
SELECT t
FROM (s:Person) - [:knows] -> (:Person)
    <- [:hasCreator] - (t:Post)
WHERE s.firstName = "Alice" AND t.length > 1000
ORDER BY VECTOR_DIST(t.content_emb, query_vector)
LIMIT k;
```

During query execution, TigerVector treats the graph pattern as a filter and applies filtered vector search (explained in Sec. 5.2). The query plan for the this query is illustrated as follows:

```
EmbeddingAction[Top k, {t.content_emb}, query_vector]
EdgeAction[Person, <hasCreator, Post:t {t.length >
    1000}]
EdgeAction[Person:s, knows>, Person]
VertexAction[Person:s {s.firstName = "Alice"}]
```

Note that in graph pattern matching scenarios, post-filtering is further limited, making pre-filtering the preferred approach. First, graph pattern filters tend to have low selectivity, as they narrow down qualified vectors in addition to attribute filters. Second, evaluating the graph pattern filters requires extra effort to identify valid bindings, which makes repeated vector search rounds costly. Third, if the query includes accumulators, the system should collect data across all valid bindings of the pattern. However, post-filtering may reduce the number of valid bindings, potentially leading to incorrect query results.

5.4 Vector Similarity Join on Graph Patterns

We support vector similarity joins on graph patterns, enabling the identification of the top-k most similar (source, target) pairs from all node pairs defined by a specified graph pattern. This functionality applies to any pair of vertex sets within the pattern. For example, to find the most similar Comment pairs created by Alice and her friends, we can express this query as a 3-hop vector similarity join:

```
SELECT s, t
FROM (s:Comment) - [:hasCreator] -> (u:Person)
- [:knows] -> (v:Person) <- [:hasCreator] - (t:Comment)
WHERE u.firstName = "Alice"
ORDER BY VECTOR_DIST(s.content_emb, t.content_emb)
LIMIT k;
```

For query execution, we enumerate all possible matched paths. Then, for each node pair in the paths, we compute their similarity scores and return the top-k closest pairs. Specifically, we use a global heap accumulator to store the top-k vector similarity scores for each matched node pair during MPP computation. A brute-force approach is employed to compute the similarity scores of all matched node pairs, because the matched paths are typically sparse. The query plan is as follows.

```
EdgeAction[Person:v, <hasCreator, Comment:t,
           @@heapAcc += (s, t,
                 dist(s.content_emb,t.content_emb))]
EdgeAction[Person:u, knows>, Person:v]
EdgeAction[Comment:s,
    hasCreator>, Person:u {u.firstName = "Alice"}]
VertexAction[Comment:s]
```

This query type has many real-life use cases. Examples include:

- **Case Law Similarity**. Identify similar cases for legal research or argument preparation by finding top-k case pairs (source, target) connected by Case → Cites → Statute → Cites → Case, where the embedding of each `Case` represents the text of legal arguments or case summaries.
- **Similar Patient Pathways**. Identify patients with similar healthcare journeys by finding top-k patient pairs (source, target) connected by Patient → Diagnosis → Treatment → Diagnosis → Patient, where the embedding of each `Patient` represents patient medical histories.
- **Vendor Similarity**. Identify suppliers with similar capabilities by finding top-k supplier pairs (source, target) connected by Supplier → Product → Buyer → Product → Supplier, where the embedding of each `Supplier` encodes supplier profiles.

5.5 Flexible Vector Search Function

Sec. 5.1 to Sec. 5.4 discusses single-query block (`SELECT-FROM-WHERE`) searches. GSQL supports query procedures in which the procedure body comprises a sequence of query blocks executed in a top-down fashion, with query blocks interconnected through vertex set variables. To integrate with query block composition, TigerVector introduces a versatile vector search function called `VectorSearch()` that seamlessly fits into the GSQL query composition framework.

The `VectorSearch()` function accepts the following parameters:

(1) **VectorAttributes**. A list of one or more compatible embedding attributes from one or more multiple vertex types.
(2) **QueryVector**. A query vector compatible with the specified `VectorAttributes`.
(3) **K**. A positive integer defining the number of top results to be returned.
(4) **Optional Parameters**.
   - A vertex set variable serving as a candidate filter.
   - A Map container to hold the top-k distances and corresponding vertices.
   - An index search parameter (*ef*) to adjust search accuracy.

As illustrated below, the function returns a value assignable to a vertex set variable, which will hold the top-k vertices.

```
topK = VectorSearch({VectorAttributes}, QueryVector,
                k, optionalParam)
```

This flexible vector search function enables a wide range of query patterns, a few of which are described below.

Vector Search Across Multiple Vertex Types. The function supports vector search across multiple vertex types specified in the `VectorAttributes` argument, as long as the attributes pass the compatibility check. For instance, to find the top-k comments or posts related to a specific topic, you can convert the topic into a vector and execute the following query:

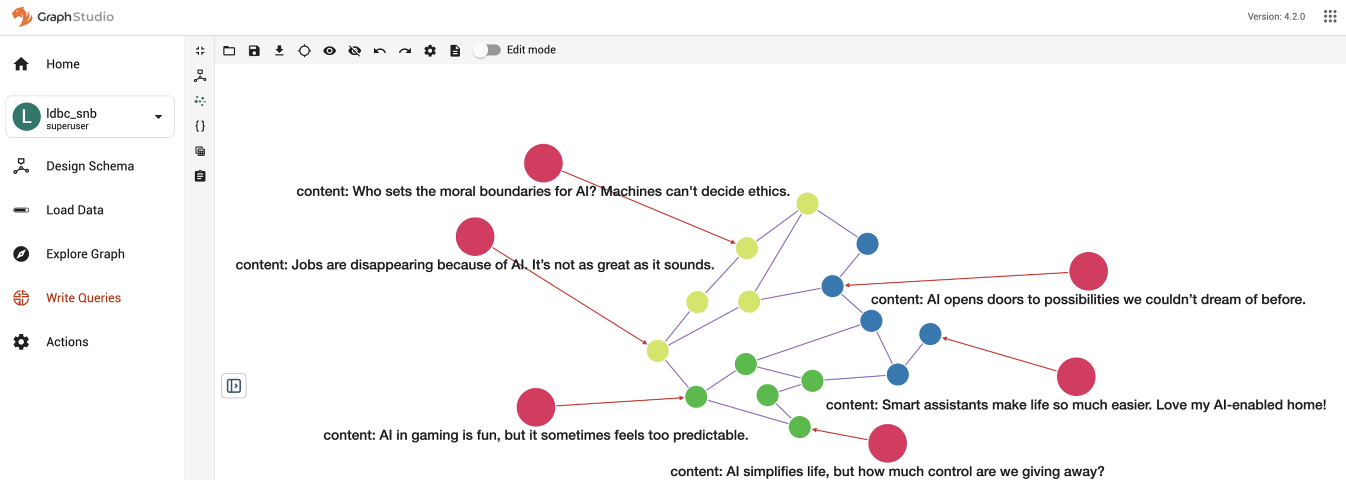
```
CREATE QUERY Q1(List<FLOAT> topic_emb, INT k) {
  -- top-k similar messages (comments or posts)
  Msgs = VectorSearch
        ({Comment.content_emb, Post.content_emb},
         topic_emb, k);
  Print Msgs;
}
```

Vector Search Function in Query Composition. Query composition allows the output of one query block to serve as input for another. As discussed earlier, in GSQL, a query procedure consists of a sequence of query blocks, each defined by a `SELECT-FROM-WHERE` clause. These blocks can produce either a vertex set or a table. When a query block outputs a vertex set, the result can be assigned to a vertex set variable. Subsequent query blocks can reference this variable in their `FROM` clause, enabling the composition of queries using vertex set variables.

Since the `VectorSearch` function returns a vertex set, it can participate in query composition. For example, query Q2 below uses the `VectorSearch` function to retrieve the top-k Comments or Posts closest to a given query. The results are stored in the vertex set variable `TopKMessages`. In the subsequent query block, this variable is used as the starting point to find the creators of the top-k messages through a 1-hop graph pattern match.

```
CREATE QUERY Q2(List<FLOAT> topic_emb, INT k) {
  -- top-k similar messages (comments or posts)
  TopKMessages =
      VectorSearch
          ({Comment.content_emb, Post.content_emb},
            topic_emb, k);

  -- find authors via TopKMessages composition
  Authors =
      SELECT p
      FROM (:TopKMessages)-[:hasCreator]->(p:Person);
```



content: Who sets the moral boundaries for AI? Machines can't decide ethics.

content: Jobs are disappearing because of AI. It's not as great as it sounds.

content: AI opens doors to possibilities we couldn't dream of before.

content: Smart assistants make life so much easier. Love my AI-enabled home!

content: AI in gaming is fun, but it sometimes feels too predictable.

content: AI simplifies life, but how much control are we giving away?

Figure 6: Demonstration of Combing Community Detection and Vector Search. The Person vertices are partitioned into three communities, colored green, blue, and yellow. The top-k Posts from each community are colored red.

```
  PRINT Authors;
}
```

Conversely, a graph query block can generate a candidate vertex set that is passed to the vector search function as a filter. In the example query Q3 below, the first query block selects comments located in the United States, storing the results in the USComments vertex set. This set is then passed as a filter to the VectorSearch function, achieving query composition.

```
CREATE QUERY Q3(List<FLOAT> topic_emb, INT k) {
    -- a global map accumulator
    Map<VERTEX, FLOAT> @@disMap;

    -- find U.S. comments from a graph query block
    USComments =
        SELECT t
        FROM (c:Country)<-[:LOCATED_IN]-(t:Comment)
        WHERE c.name = "United States";

    -- find top-k comments via USComments composition
    TopKComments =
        VectorSearch
            ({Comment.content_emb},
             topic_emb, k,
             {filter: USComments, ef: 200,
              distanceMap: @@disMap});

    -- output top-k comments and their distances
    PRINT TopKComments;
    PRINT @@disMap;
}
```

An interesting variation of Q3 is query Q4, where a Louvain community detection algorithm [9] is first used to tag each person with a community ID. The top-k vector search function is then applied within each community's vertex set, demonstrating the flexibility of combining vector search with advanced graph analytics.

```
CREATE QUERY Q4(List<FLOAT> topic_emb, INT k) {
    -- detect communities on vertex Person and edge
         knows
    -- write community id into Person.cid
    C_num = tg_louvain(["Person"], ["knows"]);

    -- loop each community to do top-k search
    FOREACH i IN RANGE[0, C_num] DO
        -- select posts from each community
        CommunityPosts =
            SELECT t
            FROM (s:Person)<-[e:hasCreator]-(t:Post)
            WHERE s.cid = i;

        -- do top-k search for each community
        TopKPosts =
            VectorSearch
              ({Post.content_emb},
               topic_emb, k, {filter: CommunityPosts});

        PRINT TopKPosts;
    END;
}
```

Demonstration. Figure 6 shows the visual result of the query Q4, in which Person nodes are partitioned into three communities using the Louvain algorithm [9]. For each community, we conduct a top-2 vector search on Posts created by people in this community and present the content of the Posts to analyze the attitudes of different communities toward AI development.

6 EXPERIMENTS

In this section, we present experimental results to evaluate TigerVector. Our experiments include four parts. The first part is vector search performance evaluation, which aims to test the capability of our system to conduct vector search with high performance and accuracy, compared with the state-of-the-art specialized vector database Milvus [36] and two popular graph databases, Neo4j [22] and Amazon Neptune [24], with the vector search feature (Sec. 6.2). The second part is scalability evaluation, which aims to test how our system can scale with more machines and larger datasets (Sec. 6.3). The third part is update evaluation, which aims to test the time our system needs to prepare databases from scratch and update existing vectors (Sec. 6.4). The fourth part is the hybrid vector search and graph search evaluation, in which we prepare a hybrid dataset based on LDBC-SNB (Sec. 6.5).

Table 1: Statistics of Datasets

Dataset	# Dimensions	# Vectors	# Queries
SIFT100M	128	100,000,000	10,000
SIFT1B	128	1,000,000,000	10,000
Deep100M	96	100,000,000	10,000
Deep1B	96	1,000,000,000	10,000

6.1 Experiment Setting

Experimental Platform. By default, we perform the experiments using n2d-standard-32 instances (AMD EPYC 7B13, 32 vCPUs, 128GB memory, 64MB L3 cache, SSE) with SCSI disks on Google Cloud.

Datasets. For the vector search performance evaluation, we use the standard SIFT100M [2] and Deep100M [8] datasets, each containing 100 million vectors. Additionally, we use SIFT1B [2] and Deep1B [8] datasets, which contain 1 billion vectors, for scalability evaluation. Table 1 summarizes these datasets.

To evaluate hybrid vector and graph search, we create a new dataset by incorporating vectors into the LDBC SNB benchmark [33], as no existing graph datasets with vectors are available. Specifically, vectors are added as content embeddings to the Comment and Post vertices. To generate queries, we modify queries from [13], which simulate a variety of complex read queries for analyzing social network data, and incorporate vector search into them.

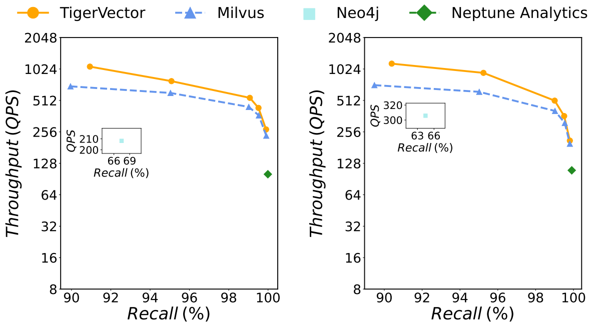
Competitors. To evaluate the vector search performance of TigerVector, we compare it with Neo4j [22] and Amazon Neptune [24], two popular graph databases that support vector search. Additionally, we compare TigerVector with Milvus [36], a highly optimized specialized vector database system. For a fair comparison, all systems build the same HNSW index using the same parameters (e.g., $M = 16$ and $efb = 128$ as recommended in [10]).

Since Neo4j and Amazon Neptune do not allow parameter tuning, we present only a single data point in Figure 7 and Figure 8. We use HTTP requests to send vector search queries for all systems except Milvus, as the performance of its HTTP port is extremely slow. Instead, we test Milvus using its gRPC port. For Amazon Neptune, a fully managed cloud service on AWS, we run it using 1024 m-NCU (Neptune Memory Capacity Units) [6], the largest Neptune instance available. This hardware is actually more powerful than the Google Cloud hardware used by TigerVector, ensuring no unfair advantage.

All experiments are conducted in-memory, as a single server has sufficient capacity to store the entire dataset (SIFT100M or Deep100M). For SIFT1B and Deep1B, the datasets are distributed across the memory of multiple servers during the scalability experiment (in Sec. 6.3).

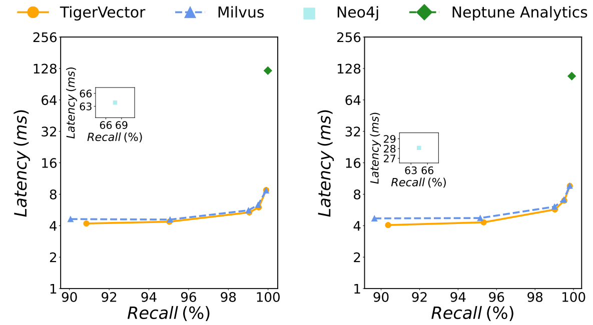
6.2 Evaluating Vector Search

Figure 7 shows the throughput results on SIFT100M and Deep100M, using 16 threads to send queries. The results demonstrate that TigerVector significantly outperforms both Neo4j and Amazon Neptune in vector search performance. Specifically, on SIFT100M, TigerVector simultaneously delivers **5.19×** higher QPS and **23%** higher recall **(90.94%, 1079.00)** compared to Neo4j **(67.50%, 208.15)**. On Deep100M, TigerVector achieves **3.77×** higher QPS and **26%** higher recall **(90.39%, 1151.92)** compared to Neo4j **(64.46%, 305.77)**. Moreover, TigerVector achieves **1.93×** to **2.7×** higher throughput compared to Amazon Neptune while maintaining a similar recall rate (**99.9%**), even though Amazon Neptune uses better hardware. From a cost perspective, TigerVector runs on an n2d GCP instance costing $1.37 per hour, whereas Amazon Neptune uses 1024 m-NCUs costing $30.72 per hour, which is **22.42×** more expensive.



(a) SIFT100M (b) Deep100M

Figure 7: Throughput Evaluation on SIFT100M and Deep100M



(a) SIFT100M (b) Deep100M

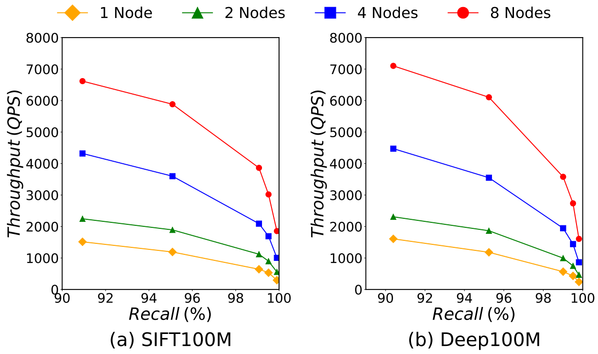
Figure 8: Latency Evaluation on SIFT100M and Deep100M

Interestingly, when compared to Milvus, a highly optimized specialized vector database, TigerVector remains competitive, achieving **1.07×** to **1.61×** higher throughput. This can be attributed to the more effective use of multi-core parallelism, as TigerGraph is an MPP database. Another factor is the difference in programming languages, as Milvus is written in Go, whereas TigerVector is written in C++.

Figure 8 shows the latency results obtained with a single thread. We observe a trend similar to that in Figure 7. The results show that TigerVector significantly outperforms Neo4j and Amazon Neptune, being up to **15×** and **13.9×** faster, respectively. Furthermore, TigerVector is slightly faster than Milvus, achieving up to **1.16×** lower latency.

6.3 Evaluating Scalability

In this experiment, we evaluate the scalability of TigerVector in terms of the number of nodes and data size. To minimize the impact of the network port, we use an additional sender machine that evenly distributes requests across all machines. The sender machine utilizes a popular HTTP benchmarking tool, wrk2 [5]. In this



(a) SIFT100M

(b) Deep100M

Figure 9: Node Scalability

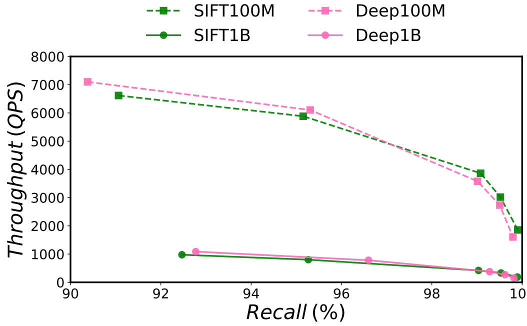


Figure 10: Data Size Scalability

benchmark, it maintains 320 connections and forks 16 threads in total. Each thread prepares payloads with randomly selected query vectors to minimize the impact of CPU caches caused by identical payloads. The number of requests sent per second is sufficiently high to saturate the throughput of our system.

Figure 9 illustrates the node scalability of TigerVector. The results show that TigerVector achieves excellent scalability. For instance, when the recall rate is 99.9%, TigerVector achieves a throughput performance gain of **1.84**× to **1.91**× when the number of machines is doubled. Even with a recall rate of 90%, where network communication overhead constitutes a larger proportion compared to the high recall rate scenario, it achieves a **1.5**× performance gain when the number of machines is doubled.

Figure 10 shows the scalability of TigerVector with respect to dataset size, where we scale the dataset size from 100M to 1B while keeping the same search parameters. We use 8 machines to run this experiment and utilize wrk2 for benchmarking throughput, maintaining the same configuration as in Figure 9.

The results in Figure 10 show that TigerVector achieves great scalability, with throughput decreasing roughly proportionally as the dataset size increases. The number of segments for the 1B dataset is exactly 10× that of the 100M dataset. For performance with $ef = 12$ (the lowest recall rate point), when the dataset scales from SIFT100M to SIFT1B, TigerVector still maintains **14.75%** of the original QPS despite a 10× increase in workload. This is because the proportion of computation increases, improving CPU utilization from 60% to 80%. For higher recall rate points, the QPS decreases proportionally to **10%** when the dataset size increases by **10**×.

Table 2: Index Building Time

Dataset	Measure	TigerVector	Milvus	Neo4j
SIFT100M	End to End	**3,977s**	8,577s	20,679s
	Data Load	202s	4,554s	133s
	Index Build	3,775s	4,023s	20,546s
Deep100M	End to End	**3,462s**	6,430s	23,560s
	Data Load	338s	3,258s	559s
	Index Build	3,124s	3,172s	23,001s

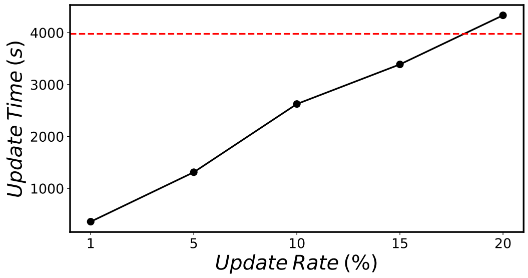


Figure 11: Index Update Evaluation on SIFT100M

6.4 Evaluating Index Update

In this experiment, we evaluate the vector index building and update performance of TigerVector. We compare the end-to-end index building time across different datasets using a single machine. TigerVector and Neo4j both load data from CSV files using their respective loading tools, while Milvus loads data directly from raw vector files. The HNSW vector index is built after data loading.

Table 2 shows that TigerVector achieves **5.2**× to **6.8**× shorter index building time compared to Neo4j. Additionally, TigerVector is **1.86**× to **2.16**× faster than Milvus for index building. This improvement is due to TigerVector's optimized data loading tool, which outperforms Milvus. Although Neo4j has a similar data loading time as TigerVector, it exhibits slower index building performance.

Figure 11 evaluates the incremental index update performance of TigerVector. It shows that as the update ratio increases, the update time also increases. The red line indicates the time required to completely rebuild the HNSW index in TigerVector. The results suggest that if more than 20% of the vectors are updated, it is more efficient to rebuild the index rather than updating it incrementally.

6.5 Evaluating Hybrid Search

In this experiment, we evaluate the performance of hybrid queries by modifying the interactive complex (IC) family of queries from the LDBC SNB benchmark [33] to incorporate top-k vector search. Specifically, we select IC queries involving the KNOWS edge type and vary the number of repetitions of KNOWS to generate queries with different path lengths. A global accumulator collects all the Message vertices (either Post or Comment) matched by the IC queries. Subsequently, a top-k vector search is performed on the collected Message set. This design ensures that each graph-shaped IC query touches a different number of segments, allowing us to study the behavior of our top-k index.

The benchmark is conducted at scale factors of 10 and 30, with each Message vertex augmented by an embedding attribute sampled

Table 3: Evaluating Hybrid Search for SF10

#Hops	Measure	IC3	IC5	IC6	IC9	IC11
2	End to End	0.64s	2.36s	1.22s	0.88s	0.67s
	#candidate	0	5,266,252	1,261	20	11,242
	Vector Search	0	**1.18ms**	**0.31ms**	**0.15ms**	**0.85ms**
3	End to End	1.15s	3.19s	2.44s	2.62s	0.91s
	#candidate	15	6,598,427	2,724	20	49,475
	Vector Search	**0.13ms**	**1.52ms**	**0.45ms**	**0.16ms**	**2.32ms**
4	End to End	1.56s	3.37s	2.66s	2.88s	1.13s
	#candidate	23	6,601,992	2,726	20	50,036
	Vector Search	**0.15ms**	**1.60ms**	**0.40ms**	**0.17ms**	**2.83ms**

Table 4: Evaluating Hybrid Search for SF30

#Hops	Measure	IC3	IC5	IC6	IC9	IC11
2	End to End	1.10s	4.78s	1.44s	1.83s	0.78s
	#candidate	3	11,679,980	1,603	20	21,519
	Vector Search	**0.09ms**	**2.32ms**	**0.36ms**	**0.23ms**	**1.32ms**
3	End to End	3.30s	10.67s	8.31s	6.94s	1.59s
	#candidate	71	17,543,581	8,348	20	118,893
	Vector Search	**0.20ms**	**2.23ms**	**0.96ms**	**0.17ms**	**5.44ms**
4	End to End	3.65s	11.26s	9.00s	7.78s	2.12s
	#candidate	71	17,567,224	8,364	20	122,459
	Vector Search	**0.21ms**	**3.92ms**	**0.86ms**	**0.22ms**	**4.94 ms**

from the SIFT100M dataset. Tables 3 and 4 summarize the hybrid search performance of TigerVector.

We measure the end-to-end query execution times, the number of collected Message candidates, and the top-k vector search time. The queries have different query shapes and path lengths (hops), which result in varying sizes of the Message set used for the top-k vector search. For example, IC5 collects the largest number (5M to 17M) of Messages due to its broader node traversal, whereas IC6 and IC11 collect a moderate number of nodes, and IC3 and IC9 operate on fewer than 100 Messages.

The results show that the end-to-end execution time increases with the number of hops, typically scaling linearly or sublinearly. Notably, vector search demonstrates excellent performance and scalability, often completing within a few milliseconds. Interestingly, the proportion of time spent on top-k vector search does not always scale directly with the size of the candidate Message set. For instance, while 3-hop-IC5 has a larger candidate set than 3-hop-IC11 on SF10 (6,598,427 vs. 49,475), it spends less time on vector search (1.52ms vs. 2.32ms). We find that IC5 touched 32 segments while IC11 touched 64 segments. We also note that even though IC11 performed brute-force search on its segments while IC5 invoked HNSW index search, the time spent on each segment is similar, which explains the shorter vector search time for IC5.

7 LESSONS AND DISCUSSION

In this section, we discuss the key lessons learned from integrating vector search within TigerGraph.

Decoupling vector embeddings from other graph attributes. There are several advantages to separating vector embeddings from other graph attributes: (1) We can leverage fast, native vector index implementations to support vector search, achieving performance comparable to highly optimized specialized vector databases. (2) We can conveniently manage different types of vector embeddings using the embedding type. (3) We can support efficient vector updates without negatively impacting existing graph database updates. (4) We can reduce vector data movement during updates to other attributes in graph databases.

MPP architecture. Many graph databases already support parallel or distributed graph processing via MPP. When integrating vector search into such graph databases, we can leverage MPP to support parallel and distributed vector search by building an index for each segment.

Query composition. Query composition is a powerful tool for expressing complex queries. It seamlessly integrates declarative vector search, the vector search API, and other graph algorithms to support advanced RAGs.

Unified design. We show that it is possible to support vector search within a graph database while achieving high performance. Therefore, we advocate for a unified system that simultaneously supports both vector search and graph search. Such a unified design offers additional advantages, including enabling hybrid vector and graph searches for advanced RAGs and addressing issues related to consistency, data silos, and data movement.

Extension to other graph databases. We believe that the above lessons are applicable not only to TigerGraph but also to other graph databases, such as Neo4j and Amazon Neptune.

8 CONCLUSION

In this work, we present TigerVector, a new system that seamlessly integrates vector search into TigerGraph. By unifying the management of graph and vector data, TigerVector can support advanced RAGs and can serve as a data retrieval infrastructure for LLMs. TigerVector introduces a suite of techniques to address challenges in performance and usability. Extensive experiments show that TigerVector achieves performance on par with or better than Milvus and significantly outperforms Neo4j and Amazon Neptune in vector search.

REFERENCES

[1] [n. d.]. LDBC (https://ldbcouncil.org/benchmarks/overview/).

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

[3] [n. d.]. Vector Indexes of Neo4j (https://neo4j.com/docs/cypher-manual/current/indexes/semantic-indexes/vector-indexes/).

[4] [n. d.]. Weaviate: An Open Source Vector Database (https://github.com/weaviate/weaviate).

[5] [n. d.]. wrk2 (https://github.com/giltene/wrk2).

[6] 2024. Amazon Neptune Analytics now introduces new smaller capacity units (https://aws.amazon.com/about-aws/whats-new/2024/07/amazon-neptune-analytics-smaller-capacity-units/).

[7] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. 2023. PG-Schema: Schemas for Property Graphs. *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 2 (2023), 198:1–198:25.

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

[9] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (2008), P10008.

[10] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proceedings of the VLDB Endowment (PVLDB)* 17, 12 (2024), 3772–3785.

[11] Raphael Derbier. 2023. Dgraph and Vector Database - The Best of Two Worlds (https://dgraph.io/blog/post/20230628-vectordb/).

[12] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248 (2019).

[13] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 377–392.

[14] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. 2024. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. *CoRR* abs/2404.16130 (2024).

[15] Emil Eifrem. 2024. GraphRAG: The Marriage of Knowledge Graphs and RAG (https://www.youtube.com/watch?v=knDDGYHnnSI).

[16] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*. 6491–6501.

[17] Gartner5188263 2024. How to Calculate Business Value and Cost for Generative AI Use Cases (https://www.gartner.com/en/documents/5188263).

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

[19] 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.

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

[21] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F. Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-Throughput Vector Similarity Search in Knowledge Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 197:1–197:25.

[22] Neo4j [n. d.]. Neo4j (https://neo4j.com/).

[23] Neo4jVec [n. d.]. Neo4j Vector Index and Search (https://neo4j.com/labs/genai-ecosystem/vector-search/).

[24] Neptune [n. d.]. Neptune (https://aws.amazon.com/neptune/).

[25] NeptuneVec [n. d.]. Vector Indexing in Neptune Analytics (https://docs.aws.amazon.com/neptune-analytics/latest/userguide/vector-index.html).

[26] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of Vector Database Management Systems. *VLDB Journal* 33, 5 (2024), 1591–1615.

[27] 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)*. 597–604.

[28] Boci Peng, Yun Zhu, Yongchao Liu, Xiaohe Bo, Haizhou Shi, Chuntao Hong, Yan Zhang, and Siliang Tang. 2024. Graph Retrieval-Augmented Generation: A Survey. *CoRR* abs/2408.08921 (2024).

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

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

[31] Amber Roberts. 2024. RAG Evaluation (https://arize.com/blog-course/rag-evaluation/).

[32] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, 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.

[33] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proceedings of the VLDB Endowment (PVLDB)* 16, 4 (2022), 877–890.

[34] Yuanyuan Tian. 2022. The World of Graph Databases from An Industry Perspective. *SIGMOD Record* 51, 4 (2022), 60–67.

[35] Jianguo Wang, Shasank Chavan, Guoliang Li, Yannis Papakonstantinou, and Charles Xie. 2024. Vector Databases: What's Really New and What's Next? *Proceedings of the VLDB Endowment (PVLDB)* 17, 12 (2024), 4505–4506.

[36] 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.

[37] 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, 12 (2020), 3152–3165.

[38] 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.

[39] 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 *Proceedings of the International Conference on Data Engineering (ICDE)*. 2835–2849.