GaussDB-Vector: A Large-Scale Persistent Real-Time Vector Database for LLM Applications

Ji Sun
Tsinghua University
sunji@tsinghua.edu.cn

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

James Pan
Tsinghua University
jamesjpan@tsinghua.edu.cn

Jiang Wang
Huawei
wangjiang16@huawei.com

Yongqing Xie
Huawei
xieyongqing1@huawei.com

Ruicheng Liu
Huawei
liuruicheng1@huawei.com

Wen Nie
Huawei
niewen2@huawei.com

ABSTRACT

Vector databases are widely used as a fundamental tool for addressing the weaknesses of large language model (LLM) applications, specifically hallucinations and the high cost of inference. However, existing vector databases either cater to niche applications with low-latency in-memory search, or offer sophisticated data management capabilities but at the cost of low performance.

To address these limitations, we propose GaussDB-Vector, a high-performance, real-time persistent vector database that excels in low-latency scalable search, real-time inserts and deletes, high availability, large-scale distributed search, and hybrid scalar-vector filtered search capabilities. These features are primarily achieved through an innovative storage architecture designed for a graph-based vector index, optimized for I/O operations and adaptable across various dataset sizes and dimensions, complemented by novel buffering strategies to further reduce I/O burdens. GaussDB-Vector supports product quantization, parallel search, and hardware acceleration via SIMD, GPUs, and NPUs in order to further accelerate queries. Experimental results show that GaussDB-Vector outperforms competitive baselines by a factor of 1 to 5 times.

PVLDB Reference Format:
Ji Sun, Guoliang Li, James Pan, Jiang Wang, Yongqing Xie, Ruicheng Liu, and Wen Nie. GaussDB-Vector: A Large-Scale Persistent Real-Time Vector Database for LLM Applications. PVLDB, 18(12): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Large language models (LLM) are the foundation for modern AI applications. For example, intelligent Q&A systems such as ChatGPT use LLMs to generate answers according to user prompts, and intelligent agents use LLMs to generate actions that can be taken to accomplish certain tasks. But LLM-based tools face two main challenges, hallucinations and expensive inference cost. To address these challenges, a popular approach is to use a vector database to serve as "long-term memory". For example, retrieval-augmented generation (RAG) [26] can reduce hallucinations by using a vector database populated with knowledge embeddings to make LLM answers more controllable and explainable. To reduce inference cost, semantic memory [12, 16] caches various stages of inference in embedding form inside a vector database to be retrieved later for future queries without invoking the LLM. To support these use cases, the vector database must efficiently store and search up to billions of embeddings while supporting features like frequent updates, high user concurrency, distributed search, high availability, user isolation, and hybrid scalar-vector filtered search.

This need for a high-performance feature-rich vector database has led to a number of commercial vector database systems, such as Milvus [35], PGVector[6] and ElasticSearch[2]. Milvus is designed for large-scale web applications. Data is organized inside segments of an LSM tree [30], and a search index is built for each segment. Milvus can persist segments onto the file system and load them into memory during search. However, Milvus suffers from (1) high query latency as it must search every segment and then merge the results; (2) expensive updates due to index rebuild when merging and splitting segments; (3) slow attribute filtering because it only supports pre-filtering (first predicates then vectors); (4) low data freshness as new data is not inserted into indexes immediately. On the other hand, ElasticSearch is designed for text searching, and it integrates vector search ability to support semantic search. Similar to Milvus, ElasticSearch also caches inserted data into in-memory segments, storing data onto disk when the size of a segment exceeds a threshold. Data on disk is also organized into segments, leading to similar drawbacks. PGVector is a vector extension built on PostgreSQL. Similar vector engines include AlloyDB (Google) and PASE [37] (Alibaba). PGVector supports IVFFLAT[3] and HNSW [28], and it works by building index structures over data pages in PostgreSQL. However, it also has drawbacks: (1) HNSW consumes more I/Os for search on disk-based systems, and its performance is worse than algorithms like NSG [21] and DiskANN [32]; (2) only single-machine search is supported, making it unable to scale to datasets with billions of vectors.

To address these limitations, we propose a persistent real-time vector database system GaussDB-Vector for LLM applications,

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:XX.XX/XXX.XX

which is competitive in terms of high performance, high availability, high scalability, and zero-delay data freshness, along with hybrid scalar-vector (attribute filter + vector) search capability.

High Performance. GaussDB-Vector has extremely low latency (< 50 ms) and high recall (> 95%) for datasets with over 1 billion vectors on a single machine, and can scale to tens of billions of vectors on multiple machines with high throughput of over 100,000 QPS. GaussDB-Vector also supports high performance vector updates. To reduce I/O cost of graph-based search, GaussDB-Vector uses an approximate nearest neighbor (ANN) index based on Vamana (DiskANN) which reduces the radius of ANN search by controlling the out-degrees of each vector while using a soft edge-pruning method to retain shortcuts. To speed up index construction, GaussDB-Vector introduces a new edge pruning heuristic that reduces the complexity of neighborhood construction without sacrificing search quality. From a systems perspective, GaussDB-Vector also extends DiskANN to support tailored storage structures that optimize I/O efficiency over datasets with different sizes and dimensionalities, in addition to introducing a novel buffering strategy based on hot nodes to further relieve I/O pressure. For updates, GaussDB-Vector designs an asynchronous deletion mechanism to reduce deletion latency. GaussDB-Vector supports product quantization [24] (PQ), multi-threaded parallel computing, and hardware acceleration via SIMD, GPUs, and NPUs to accelerate queries.

High Availability. GaussDB-Vector adds new log types for operations on vectors, and GaussDB-Vector uses data replicas and primary node selection techniques to guarantee the consistency of data and availability of the service.

High Scalability. GaussDB-Vector can support tens of billions of vector data by using only 10 commodity machines, and it can guarantee the performance due to sophisticated query routing. GaussDB-Vector also designs a vector data sharding strategy along with drift detection and automatic redistribution mechanisms to maintain the performance of distributed queries.

Data Freshness. GaussDB-Vector guarantees zero-delay data freshness. In GaussDB-Vector, newly inserted vectors are visible immediately upon commit because it supports incremental index updates on disk. The transactional management engine in GaussDB-Vector also guarantees ACID transactions.

Hybrid Search. GaussDB-Vector supports efficient scalar-vector hybrid search, regardless of the scalar condition and selectivity. We design a novel hybrid index for hybrid search, and the evaluations show that GaussDB-Vector outperforms baselines by 70%-300%.

In summary, we make the following contributions.

(1) We propose a novel vector database architecture GaussDB-Vector to support semantic retrieval (Section 2).

(2) We propose real-time index update algorithms to enhance data freshness, introduce hot node buffering to speed up search, design graph index pruning algorithms to improve insertion performance, and optimize indexes to reduce storage space and memory access costs (Section 3).

(3) We propose a novel hybrid search method to support scalar-vector hybrid search. We design a balanced tree structure to support efficient hybrid queries, and propose cardinality-based sub-index selection algorithms to improve the stability of query and update latency across different scalar selectivity rates (Section 4).

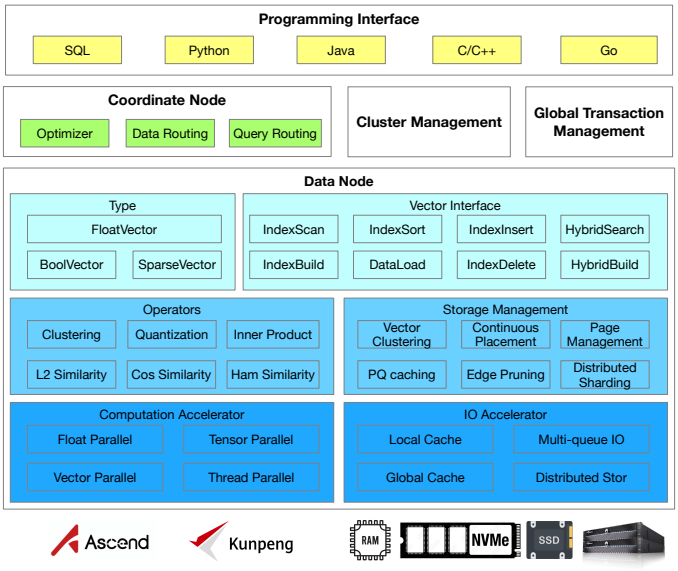


Figure 1: Architecture of GaussDB-Vector

(4) We design vector sharding and query scheduling techniques for distributed GaussDB-Vector that can support tens of billions of vectors (Section 5) and propose hardware acceleration techniques for GPU, NPU and SIMD (Section 6). The proposed method simultaneously supports sharding of both scalar and vector data, enables query scheduling that takes into account data distribution, and allows for online fine-tuning of data distribution to maintain query performance and accuracy.

(5) We conduct thorough experiments on datasets with varying sizes and dimensions and show the superiority of GaussDB-Vector over baselines (Section 7).

2 SYSTEM OVERVIEW

2.1 Architecture

To achieve high performance, high availability, high scalability, and data freshness, GaussDB-Vector is designed as a two-tiered structured distributed database, as shown in Figure 1. Coordinate nodes (CNs) are responsible for accepting client connections via the programming interface, parsing SQL, and planning the execution path. Data nodes (DNs) are responsible for plan execution, including operator execution and data manipulation, and contain modules for storage management and hardware-accelerated execution. Aside from these nodes, a cluster manager orchestrates execution over database clusters, and a global transaction manager keeps track of global transaction IDs for distributed transactions. The CN-DN design can facilitate load balancing, parallel vector search, and high scalability. This architecture is capable of supporting large-scale cloud-based applications with billions of connections. Specifically, GaussDB-Vector serves as the backend vector search database for XiaoYi Search, a cloud service providing intelligent assistant features on Huawei mobile phones (similar to Siri on the iPhone). Additionally, GaussDB-Vector supports trillions of code generation queries within Huawei's internal R&D Support System.

Programming Interface. GaussDB-Vector inherits the advantages of SQL for relational databases, and thus can process complex

queries, including hybrid search and nested queries. We integrate new vector types into SQL and expose vector manipulation commands for ease of use. For example, users can create a table with both vector and ordinary data by using `CREATE TABLE`; build a vector index by using `CREATE INDEX`; and conduct a top-k search using `SELECT··· ORDER BY··· LIMIT K`. With the help of the SQL engine, `GaussDB-Vector` is able to apply rewrite rules and cost-based optimization to accelerate query processing. For example, the optimizer selects sequence scan if the selectivity of a predicate is large; otherwise index scan is used. On top of the standard SQL interface, `GaussDB-Vector` also supports SDKs for other programming languages that can be used to develop database clients.

Storage Management. In order to reduce data fragmentation and increase the efficiency of concurrent I/O, `GaussDB-Vector` adopts page-based storage management. Vector data is a special type of fixed length array, and it is stored in a row-based storage format with other relational columns. `GaussDB-Vector` supports building vector indexes to reduce the vector search I/O cost. Each vector index is a structured file containing vector IDs and their locations in the data files. The index can be organized as an inverted list or a graph. In order to reduce the number of random page visits, `GaussDB-Vector` clusters closest vectors into physically contiguous data blocks. Moreover, the logical structures of the vector indexes are designed to reduce I/O and improve cache locality of hot data. `GaussDB-Vector` supports data sharding according to both the scalar column and the vector column to support high scalability on large-scale datasets. For graph-based indexes, `GaussDB-Vector` also optimizes the edge pruning algorithm to reduce index construction time. The details of ANN indexing are described in Section 3.

Hardware Acceleration. Vector search queries depend on many compute-bound operators, including clustering, quantization, similarity computations, etc. `GaussDB-Vector` optimizes these implementation for machines with Ascend/Kunpeng CPUs. On Kunpeng CPUs, parallel float instructions can be carried out using SIMD capability, where multiple float numbers are calculated in one instruction cycle. Kunpeng CPUs have more cores than other CPUs of the same grade, and thus thread parallelism is a vital approach to reducing latency of vector queries. `GaussDB-Vector` also supports parallel vector index construction and search, and selects the degree of parallelism and balances the load automatically. On Ascend NPUs, `GaussDB-Vector` supports batching vectors into matrices, synchronizing data in a unified memory space between the CPU and NPU to accelerate computations using NPU tensor cores.

2.2 SQL Interface

`GaussDB-Vector` extends SQL to support vector search. Below, we present some typical usage examples.

Data Definition Language (DDL). The DDL can be used to define vector table schemas and build indexes on vectors and scalars. As vector index construction can take several hours for trillions of data items, users can add the `CONCURRENTLY` keyword to avoid blocking other business applications, which builds the index using a data snapshot and then incrementally updates it based on the delta data.

Data Manipulation Language (DML). The DML can be used for querying and manipulating vector data. Users can orchestrate different operations using standard SQL. For simplicity, we use <cos>,<euc> and <ham> to denote operators of Cosine distance, Euclidean distance and Hamming distance respectively.

```
-- DDL: create a vector table
CREATE TABLE [schema_name.]vectortab
     (id int, vec floatvector(128), doc text)
WITH (ORIENTATION={ROW|COL})
[PARTITION BY {
     {RANGE (partition_key)} |
     {RANGE (partition_key)} |
     {LIST|HASH (partition_key)}];
-- create a vector index
CREATE INDEX [CONCURRENTLY][IF NOT EXISTS]
     [[schema_name.]vectoridx]
ON table_name [USING {IVF|VAMANA}]
     ({{column_name {L2|Cosine|IP|Hamming}}
[LOCAL]
[WITH ({storage_parameter=value} [, ...])]
[{VISIBLE|INVISIBLE}]
[WHERE predicate];
-- alter parameters of index
ALTER INDEX vectoridx
SET(storage_parameter=value);
-- rebuild the index
REINDEX [CONCURRENTLY] vectoridx;
```

```
-- DML: copy vector data from csv file
COPY t1 FROM 'floatvector.csv' CSV HEADER;
-- insert vector into table
INSERT INTO t1 VALUES
    (9711839, '[30,12,12,25]');
-- update vector data
UPDATE t1
SET repr = '[30,12,12,25]'
WHERE id = 1;
-- top-k search (Cosine)
SELECT id, repr <cos> '[1,1,3,2]' as s
FROM t1 ORDER BY s LIMIT 2;
-- range search (Cosine)
SELECT id, repr <cos> '[1,1,3,2]' as s
FROM t1 WHERE s < 0.8;
-- delete vector from table
DELETE FROM t1 WHERE id = 1;
```

3 VECTOR INDEXING

Vector indexing reorganizes vector data into index files in order to accelerate ANN search, and it is the core part of a vector database. Various ANN indexing algorithms have been proposed, and they can be classified as IVF-based methods [19], hashing-based methods [18], graph-based methods [10, 11, 13, 36] and quantization-based methods [20, 24, 25]. IVF-based methods and graph-based methods can shape the index according to the data distribution and have better accuracy on top-k search, thus `GaussDB-Vector` focuses on IVF and Vamana graph [32] as basic algorithms for ANN indexing. Based on these two algorithms, `GaussDB-Vector` adopts product quantization to further accelerate the search, and it also supports real-time updates.

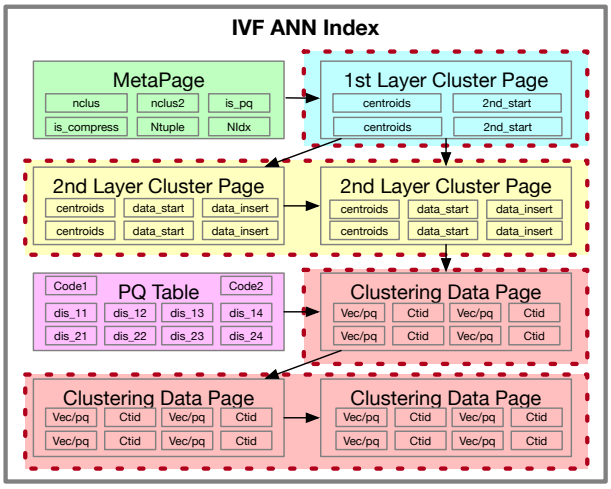


Figure 2: Structure of IVF Index

3.1 IVF-based Index

An IVF-based index splits the high-dimensional vector space by clustering, and then locates the data from clusters with nearest centroids. IVF-based indexes are popular for their small data expansion and sequential data access characteristics. IVF-based indexes are especially suitable for data sizes in the millions, and they are fast and easy to maintain. In this section, we describe the implementation details for the IVF-based index in `GaussDB-Vector`.

Index Structure. `GaussDB-Vector` uses two-layer clustering for the IVF index in order to balance the load of clusters. As Figure 2 shows, the index is composed of a sequence of linked pages, organized logically as a three-layer tree structure. The metapage is the first page in the index file, and it stores the configuration and statistics of the index, including the number of clusters in the upper two layers, number of vectors, whether the index is compressed, and whether the index uses product quantization. Following this page are the first-layer clustering pages, containing centroids and start positions of secondary-layer clusters. Each secondary-layer cluster page contains the centroid and start point of full data in this sub-cluster.

In order to accelerate distance computing and reduce the space overhead, ANN search engines can take advantage of product quantization (PQ) techniques. PQ splits each of the vectors into segments, and each segment is represented by the closest centroid. If the index uses PQ, then `GaussDB-Vector` will build a PQ table structure according to the distribution of data and store it before the data pages. The data pages then store PQ codes instead of full vectors in order to reduce storage and distance computing cost. When the query computes the distance of two vectors, it finds the centroids and computes sum of distances between centroids. `GaussDB-Vector` adopts PQ-ADC [24] to increase the precision of product quantization, PQ-ADC only quantizes the data in the index, and the query vector remains unchanged. On the other hand if PQ is not used, data pages directly follow the cluster pages, and all the indexed data is stored in the data pages. Each tuple in a data page contains vector data and the corresponding tuple ID in the heap table. The vector is used to select nearest neighbors by distance, and the tuple ID is used to locate the row tuple in the original table.

Index Construction. During index construction, `GaussDB-Vector` first collects a set of vectors from heap table into memory by using reservoir sampling and then conducts k-means clustering on the samples, constructing cluster pages according to the results. `GaussDB-Vector` then scans all the vectors in the table and labels each vector with the ID of its nearest cluster (centroid), then sorts all the vectors by label. Next `GaussDB-Vector` builds data pages for each cluster according to the sorted vectors while also updating the information in the cluster pages. Building data pages in sorted order results in clusters that are stored contiguously, allowing search to be optimized by batch processing and data prefetching.

Vector Top-k Search. When a top-k query comes, the query vector is first compared against the centroids in the first-layer cluster pages to find N nearest clusters, along with M nearest sub-clusters from each cluster. In our evaluation, we observed that setting $N \times M$ to cover 5–10% of the dataset proves effective in achieving high recall with minimal latency. Therefore, we set N and M as $\sqrt{10\%}$ of total number of first/second layer buckets respectively by default (at least 1). Then, `GaussDB-Vector` scans the candidate vector set, $\mathcal{V}_c$, consisting of vectors from the $\sum_{i=1}^{N} M_i$ sub-clusters, and computes distances to each candidate vector. Next, `GaussDB-Vector` conducts binary sort on these vectors according to distance to the query. The binary sort iteratively picks a vector with distance δ and removes vectors from $\mathcal{V}_c$ with distances greater than δ, stopping once $|\mathcal{V}_c|$ is less than $2k$ `GaussDB-Vector` then fetches the corresponding rows from the table in the order of distances for vectors in $\mathcal{V}_c$ and checks their visibility. If a row is not visible (based on transactions), `GaussDB-Vector` scans the next row. If there are not enough visible rows, `GaussDB-Vector` sorts and scans the next partition of candidate vectors (resulting from the binary sort). In this way, the sorting complexity should be $O(|\mathcal{V}_c| + \frac{1}{2}|\mathcal{V}_c| + \frac{1}{4}|\mathcal{V}_c| + \frac{1}{8}|\mathcal{V}_c| + \cdots + \frac{1}{2^n}|\mathcal{V}_c|)$, which is $O(2|\mathcal{V}_c|)$ when n is infinite, and the total top-k fetching complexity is $O(2|\mathcal{V}_c| + k + N_{invisible})$, where $N_{invisible}$ is the number of rows being lazy deleted. Since the term $|\mathcal{V}_c|$ often dominates over other two terms, we simplify it as $O(|\mathcal{V}_c|)$. The full sorting complexity is $O(|\mathcal{V}_c| log(|\mathcal{V}_c|))$ for any query. This mechanism allows `GaussDB-Vector` to handle invalid entries appearing in the candidate set due to situations such as asynchronous deletion and multi-attribute filtering.

Vector Insert. For an insert query, `GaussDB-Vector` compares distances between centroids of clusters to find the nearest cluster. It then obtains the insert block ID that points to the last data page corresponding to the nearest cluster from the cluster page and appends the context ID (`ctid`) of the query vector to this page. The `ctid` gives the location of the vector in the heap table. If the last page is full, `GaussDB-Vector` tries to insert the vector into the next page and then updates the insert block ID upon success. But if the last page is full and it is also the last valid page, `GaussDB-Vector` requests a sequence of recycled pages from the free space manager (FSM) or creates new pages from the file system.

Vector Delete. For vector delete, `GaussDB-Vector` first identifies the target row based on the query, then locates the cluster containing the corresponding vector by finding the cluster with centroid nearest to the vector. The vector is then marked for deletion by setting a deletion flag. In order to achieve low latency, physical removal is conducted during asynchronous batch cleanup.

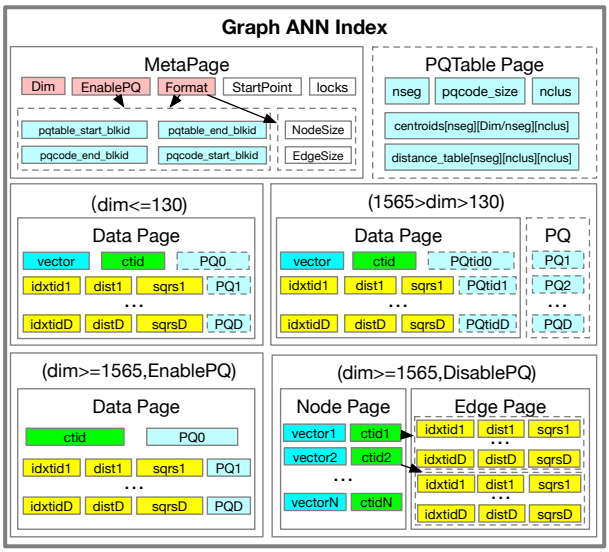


Figure 3: Structure of Graph-Based Index

Vector Vacuum. GaussDB-Vector relies on a vacuum operation to do space vacuum and batch cleanup of index files and tables. The vacuum operation is executed automatically when the number of tuples marked for deletion exceeds 10% of the table. For the IVF-based index, the vacuum operation scans all clusters one by one. For each cluster, GaussDB-Vector acquires new pages from FSM or file system, adds a shared lock on the data pages in this cluster, and then copies the undeleted vectors in this cluster to the new pages sequentially. Then, GaussDB-Vector adds an exclusive lock on the secondary cluster page and updates the start/insert block ID. Once all vectors have been copied, GaussDB-Vector releases all the locks and puts the replaced data pages for the cluster into FSM. This procedure is executed iteratively until all clusters have been processed. In this way, only update/insert queries on certain clusters are blocked by vacuum, and only for a short period of time.

3.2 Graph-based Index

Graph-based indexes have the advantage of fast and accurate approximation for different data distributions. GaussDB-Vector adopts Vamana graph [32] as the basic algorithm. Vamana graph is a directed monotonic relative neighborhood graph which guarantees a search complexity close to logarithmic time, and Vamana graph restricts the maximal out-degree for each node to ease management on the disk. Compared to NSG [21], Simhadri et. al. [32] show that adding a soft coefficient to the distance when pruning edges brings benefits to the performance. Aditi et. al. [22] propose methods for data modification and show that the soft coefficient can make the performance of Vamana graph stable, even after 40% of the data is deleted. Based on these previous works, GaussDB-Vector judiciously implements the structure of Vamana graph in terms of page design for reducing I/O cost, lock management for supporting high concurrency, and buffer management for reducing query latency. GaussDB-Vector proposes a two-phase neighbor pruning method to further reduce the overhead of vector insert and graph construction. Next we introduce the details of the graph-based index.

Index Structure. GaussDB-Vector builds and stores the graph structure in the blocks of index files. The first page in the index files is the metapage, and it stores metadata about the index including vector dimension, storage method, object size, start locations of difference page types, product quantization configurations, and locks for the current index. Following the metapage, the PQTable page stores the centroids of each PQ segment along with distances of all centroid pairs in each segment. The main components of the graph-based index are the nodes, edges, and PQ codes.

GaussDB-Vector designs three structures for adapting to different scenarios. (1) If the vector dimension is not very large, GaussDB-Vector stores all data associated with a single vector (i.e. its PQ code and neighbor codes, etc.) together in one page. When a query navigates the graph index, it can easily get the PQ code of the current vector as well as all the PQ codes of the neighbors by reading a single page, thereby reducing overhead of I/O. For example in Figure 3, the threshold value of 130 dimensions is calculated assuming 96 neighbors, 6-byte tuple IDs, 4-byte float vector values, 96-byte PQ codes (maximum), and 8*1024 bytes of space per page. (2) If the dimension of vectors is larger, the neighbor codes are too large to be put on one page. GaussDB-Vector moves the PQ codes out to a separate zone to avoid redundancy, and the data page only stores the tuple IDs of each neighbor. Even so, as the total size of PQ codes for all the vectors is small compared to the index, they can easily be cached in memory. Therefore, this structure is still very efficient even though GaussDB-Vector needs to read multiple PQ code pages for getting the neighbors. For example in Figure 3, the threshold value of 1565 dimensions is calculated using the same assumptions as before. (3) Users can also select a compression mode for the index structure if the data volume is very large (e.g. 1 billion vectors on one machine) or the dimension is very high (e.g. 2048). In this structure, GaussDB-Vector separates nodes and edges completely and each of them can be compactly arranged. If PQ is enabled for a query, the raw vectors can be dropped from the index to further reduce storage cost.

Index Construction. To quickly build the graph-based index, GaussDB-Vector clusters the vectors into $\mathcal{N}_g$ overlapped groups so that each vector belongs to its two nearest clusters. For each group, GaussDB-Vector constructs a small graph with only half the maximum out-degrees of the full graph. In this way, the index can be built in memory to avoid frequent data swaps, and global connectivity can also be maintained as each vector belongs to two subgraphs. The index construction procedure is shown in Algorithm 1.

Each subgraph is constructed by incrementally inserting vectors into a randomly initialized graph, as shown in Algorithm 2. In this way, early inserted nodes can retain some longer edges to reduce the radius of the graph. The insertion operation relies on vector search and neighbor pruning to keep the neighbors close and maximize the edge angle (this feature is proposed by MRNG [21]). For vector search, Algorithm 4 shows the greedy search procedure. Given a query vector, a root node is retrieved from the min-heap representing the closest vector so far to the query, and then the distances between its unvisited neighbors and the query vector are computed. Next, these neighbors are inserted into the min-heap. The min-heap has a limited size so that nodes far away from the query are gradually evicted. This process continues until there are no more unvisited nodes. Note that duplicate vectors are linked into one node, and only the main node participates in neighbor selection. This greatly improves the quality of the graph because

Algorithm 1: BuildGraphIndex

Input: A column of vector data in table $\mathcal{S}$, number of segments for PQ S_{pq}, number of clusters in each PQ segment C_{pq}, number of subgraphs N_g
Output: A graph-structured index $\mathcal{G}$.

1 $\mathcal{G} \leftarrow$ initialize an empty graph structure on the disk ;
2 $PQ \leftarrow$ calculate PQ table on $\mathcal{S}$;
3 $\mathcal{G} \leftarrow$ calculate PQ code for each node ;
4 $Sample \leftarrow$ sample from $\mathcal{S}$;
5 $Centroids \leftarrow$ cluster $Sample$ into N_g clusters ;
6 **for (** $i \leftarrow 0; i < N_g; i \leftarrow i + 1$ **)**
7 $\quad \mathcal{G}_{sub} \leftarrow \emptyset$;
8 $\quad$ **foreach (** $v \in \mathcal{S}$ **)**
9 $\quad\quad ids \leftarrow$ calculate the nearest two centroid ids ;
10 $\quad\quad$ **if** $i \in ids$ **then**
11 $\quad\quad\quad \mathcal{G}_{sub} \leftarrow \mathcal{G}_{sub} \cup \{v\}$;
12 $\quad \mathcal{G}_{sub} \leftarrow$ Call BuildSubgraph() to build the Vamana graph with 48 out-degree on vectors in $\mathcal{G}_{sub}$;
13 $\quad \mathcal{G} \leftarrow \mathcal{G} \cup \mathcal{G}_{sub}$ ▷ Merge the in-memory subgraph into on-disk index with 96 out-degree for each node ;
14 **return** $\mathcal{G}$;

Algorithm 2: BuildSubgraph

Input: A column of vector data in this subgraph $\mathcal{S}$.
Output: An in-memory Vamana graph $\mathcal{G}$.

1 $\mathcal{G} \leftarrow$ initialize an in-memory regular graph ;
2 **parallelly foreach(** $v \in \mathcal{S}$ **)**
3 $\quad \mathcal{G} \leftarrow$ insert v in $\mathcal{G}$;
4 **return** $\mathcal{G}$;

having too many duplicate neighbors can create cliques, leading to local minima traps that result in low recall during searches.

The total cost consists of subgraph construction, which for each subgraph is proportional to $|S|/|N_g|$ times the insertion complexity, along with the overhead of the partitioning phase (Algorithm 1 lines 2–5). For this phase in particular, the exact cost depends on the number of PQ segments and clusters in each segment, but in general is linear with respect to $|S|$.

Vector Top-k Search. ANN search on the graph-based index follows the procedure in Algorithm 4, returning results from the candidate set. However, as the index is stored on disk, the search query needs several random I/Os to get the candidates and their neighbors. As the size of the shared buffer is limited, only the data that will be frequently visited is worth caching. Based on this principal, `GaussDB-Vector` caches nodes and edges visited in the first two steps when searching, along with the PQ table and codes, while other data is fetched from disk. `GaussDB-Vector` also optimizes SSD read speed by fetching neighbors using multiple I/O threads. Specifically, the default value of M_{graph} is 128. If the total number of data rows is smaller than 128, the candidate set will decrease to the number of rows automatically. If we cannot get enough results (k-NN vectors) from candidate set due to invisible or filtered rows,

Algorithm 3: VectorInsert

Input: An in-memory graph $\mathcal{G}$, a query vector v.
Output: An in-memory graph with query vector $\mathcal{G}$.

1 $pool \leftarrow$ get candidates list by searching query v from $\mathcal{G}$;
2 $\alpha \leftarrow 1.2$;
3 $\mathcal{N}(v) \leftarrow \emptyset$;
4 $\mathcal{D} \leftarrow \{distance(k, v) | k \in pool\}$;
5 **for (** $\alpha' \leftarrow 1.0; \alpha' \leq \alpha \wedge |\mathcal{N}(v)| < degree; \alpha' \leftarrow \alpha' * 1.2$ **)**
6 $\quad$ **foreach (** $k \in SORT(pool, ascending)$ **)**
7 $\quad\quad d_{kk'} \leftarrow Min(distance(k, k') | k' \in \mathcal{N}(v))$;
8 $\quad\quad$ **if** $D_{kv} \leq d_{kk'} * \alpha'$ **then**
9 $\quad\quad\quad \mathcal{N}(v) \leftarrow \mathcal{N}(v) \cup \{k\}$ ▷ As k is closer to v than to any existing neighbor of v, v connects to k ;
10 $\mathcal{G} \leftarrow$ set neighbors $\mathcal{N}(v)$;
11 **foreach (** $p \in \mathcal{N}(v)$ **)**
12 $\quad$ **foreach (** $p' \in SORT(\mathcal{N}(p), descending)$ **)**
13 $\quad\quad$ **if** $D_{pp'} \leq D_{pv}$ **then**
14 $\quad\quad\quad$ **break**;
15 $\quad\quad$ **if** $distance(v, p') < D_{pp'} * \alpha$ **then**
16 $\quad\quad\quad \mathcal{N}(p) \leftarrow \mathcal{N}(p) - p' + v$ ▷ As p' is closer to v than to p, p connects to v instead ;
17 $\quad\quad\quad \mathcal{G} \leftarrow$ set neighbors $\mathcal{N}(p)$;
18 $\quad\quad\quad$ **break**;
19 **return** $\mathcal{G}$;

Algorithm 4: VectorSearch

Input: In-memory graph $\mathcal{G}$, query v, candidate size M_{graph}.
Output: A set of vectors C visited when searching.

1 $C \leftarrow \emptyset$;
2 $\mathcal{H} \leftarrow$ a min-heap with size M_{graph} with the fixed start point in the graph $\mathcal{G}$;
3 $p \leftarrow$ pop a node from $\mathcal{H}$;
4 $C \leftarrow C \cup \{p\}$;
5 **while** p *is valid* **do**
6 $\quad \mathcal{N}(p) \leftarrow$ unvisited neighbors of p ;
7 $\quad$ **foreach (** $p' \in \mathcal{N}(p)$ **)**
8 $\quad\quad \mathcal{H} \leftarrow$ insert p' and its distance to p ;
9 $\quad p \leftarrow$ pop a node from $\mathcal{H}$;
10 $\quad C \leftarrow C \cup \{p\}$;
11 **return** C ;

`GaussDB-Vector` will double the candidate size M_{graph} and search more candidates starting from the existing candidates.

Vector Insert. Given a vector to insert, the candidate set is obtained by searching the graph for the nearest neighbors, and then selecting some of the nodes in the candidate set to keep as the neighbors of the vector, as shown in Algorithm 3. The aim of neighbor selection is to find the nodes whose distance to the vector is smaller than the distances to the other kept neighbors of the vector. Vamana [32] introduces a relaxation coefficient to keep more connections, and this can enhance the overall connectivity of the graph. Additionally, in order to make the vector reachable by existing nodes, `GaussDB-Vector` tries to add the vector as a neighbor of each of its neighbors.

If the vector is selected as a valid neighbor of one of its neighbors, then the farthest neighbor of the neighbor is replaced by this vector. For each constructed subgraph, `GaussDB-Vector` writes the neighbors of each node to the full graph on disk.

Compared to vector search, vector insert is more costly as it needs to compute the distances not only between the inserted vector and its neighbors, but between the inserted vector and its neighbors' neighbors. For maximum out-degree m, the total complexity is approximately $O(m^2)$, which is smaller than that in DiskANN [32], where the worst-case complexity is $O(m^3)$. When adding the reverse edge (in-edge from neighbor p to the inserted node v), DiskANN finds edge-dominated nodes by checking each neighbor of p with every other neighbor of p, with complexity $O(m^2)$. Because this happens for every neighbor p of node v, the total complexity of inserting v is $O(m^3)$. On the other hand, the neighborhood of p is linearly scanned (Alg. 3 line 12), using the farthest neighbor of p as a heuristic for identifying dominated nodes. The total complexity of vector insert is reduced to $O(m^2)$.

Vector Delete. Like for the IVF-based index, lazy deletion is also used for the graph-based index. For a delete query, `GaussDB-Vector` needs to find the vector in the graph and mark it deleted. However, different from the IVF-based index, there is no guarantee that the vector can be located using ANN search. To solve this problem, `GaussDB-Vector` adds an inverted list that can be used to locate the tuple (row) ID (`tid`) of a node in the index by using the `ctid` of the vector in the heap file. In this way, deleted nodes will not be put into the candidate set, meanwhile avoiding the overhead of performing a table lookup for checking visibility. In this way, `GaussDB-Vector` supports in-place update (which is equivalent to a insertion after a deletion) in order to avoid garbage accumulation.

Vector Vacuum. `GaussDB-Vector` regularly recycles dead nodes when more than 10% data is deleted or updated. As Algorithm 5 shows, `GaussDB-Vector` first finds the locations of all dead nodes by scanning the index in parallel. But before pruning edges connected to dead nodes, `GaussDB-Vector` needs to ensure that existing insert queries have been completed, as they may add connections to dead nodes. To achieve this, `GaussDB-Vector` uses two index-level exclusive locks, one for preventing new inserts and the other for waiting on old inserts to complete. Nodes connected to dead nodes are first reinserted into the graph in order to obtain new neighbors. Then after reconstructing the nodes connected to dead nodes, all dead nodes are free to be physically deleted and the associated pages can be recycled. In order to avoid existing delete and search queries visiting recycled pages, physical delete waits for these queries to terminate. `GaussDB-Vector` also tries to fix disconnections in the graph. Any unreachable nodes in the graph due to deleted vectors are reinserted into the graph. This can avoid decreasing index quality when a large number of vectors undergo update.

The number of reinsertions depends on the node in-degree. In general, the in-degree is close to the out-degree but in some cases can be 4-5 times higher for certain hot spots [34]. Even so, `GaussDB-Vector` can perform reinsertions in parallel in order to reduce the total latency.

4 SCALAR-VECTOR HYBRID QUERIES

Vector databases can support scalar filtering by extending the size of the candidate set for both graph and IVF indexes until enough

Algorithm 5: Index_Vacuum

Input: A graph-based index file $\mathcal{G}$ with dead tuples.
Output: A vacuumed graph-based index file $\mathcal{G}$.

1 $\mathcal{D} \leftarrow \emptyset$;
2 **parallelly foreach(** $n \in \mathcal{G}$ **)**
3 **if** *n is invisible or marked dead* **then**
4 $\mathcal{D} \leftarrow \mathcal{D} \cup \{n\}$;
5 prevent new inserts and wait for existing inserts to end ;
6 **parallelly foreach(** $n \in \mathcal{G}$ **)**
7 **if** $\mathcal{N}(n) \cap \mathcal{D} \neq \emptyset$ **then**
8 $\mathcal{G} \leftarrow$ reinsert n into $\mathcal{G}$;
9 prevent new delete/search and wait for existing delete/search to end ;
10 free all the delete nodes ;
11 **return** $\mathcal{G}$;

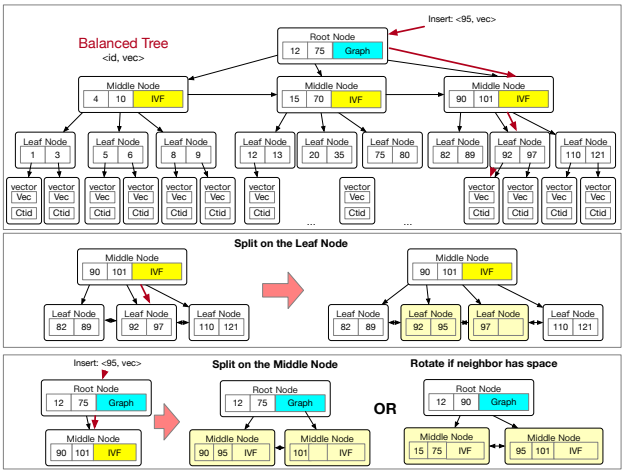


Figure 4: A scalar-vector index for hybrid search.

vectors are retrieved. However, this method is slow, especially when the selectivity of the condition is small, due to excessive distance computations and data accesses. `GaussDB-Vector` supports efficient hybrid search with a multi-column hybrid index, and the index integrates both IVF and graph structures.

As Figure 4 shows, `GaussDB-Vector` adopts a balanced tree as the basic structure for the hybrid index so that the scalar data can be quickly searched to locate the local vector index with higher selectivity. The resulting hybrid search can be even faster than a pure vector query, and the performance is also stable for different conditions. The hybrid index supports both equality and range conditions. Examples of hybrid search queries are shown below:

```
SELECT id FROM t WHERE t.id = 10 ORDER BY
    t.vec <euc> '[3,1,4,...,2]' LIMIT 3;
SELECT id FROM t WHERE t.id BETWEEN 10 AND 20
    ORDER BY t.vec <euc> '[3,1,4,...,2]'
    LIMIT 10;
SELECT id FROM t WHERE t.id < 20 ORDER BY
    t.vec <euc> '[3,1,4,...,2]' LIMIT 10;
```

4.1 Hybrid Index Structure

The scalar values are organized as a balanced tree structure that can be searched in $O(log(n))$ time. Just like a b-tree in a relational database, each node in the hybrid index occupies one page, and it also supports data updates by using `INSERT`, `SPLIT`, and `ROTATE` operations. In each root and middle node, a pointer linking the metapage of a vector subindex is arranged at the beginning, and a series of scalar division values and pointers linking to the next level are staggered and tightly arranged. In each leaf node, all the scalar values are tightly arranged, and each of the scalar values is linked to a vector and `ctid` in the heap table.

4.2 Hybrid Search

In the hybrid index, all the root and middle nodes have vector subindexes. When a query searches a node where the scalar predicate selects more than 50% of the data, the query can search for the answer from the vector subindex directly. Thus the query does not have to scan all the leaf data and individually rank the vectors. However, the leaf nodes themselves do not have vector subindexes. This is because a leaf node will not contain many tuples (less than 100 in general), and sequence scan is fast enough for small amount of data. Therefore, queries with small selectivities can quickly locate the scalar data and get the answer by sorting a small number of vectors, and queries with high selectivities can search the answer from a vector subindex with the minimal super-set of selected data.

4.3 Hybrid Data Update

An update query may involve insert, delete or both. Data delete is cheap because the deleted vector only needs to be marked as dead instead of removing it immediately. But data insert may trigger node splitting or rotating if the inserted page is full. Splitting happens when one node is full and the neighbors are also full. It creates a new node with half of the values, then inserts a pointer to the new node into the parent node. Rotating happens when the inserted node is full but neighbors still have space. For example in the bottom of Figure 4, the last middle node is full, but the left still has one free space, and thus 75 is moved down to the middle node and 90 moved up to the root node so that 95 can be inserted successfully. Rotating causes update operations in the last middle node and the root node along with an insert operation in the left middle node.

Node modification dominates the cost of data update in the hybrid index, especially for nodes with vector subindexes. This is also one of the reasons why vector subindexes are not built on the leaf nodes (the leaf node is highly likely to be split frequently). For the vector subindexes, the splitting and merging cost is higher for the graph-based index than for the IVF index. For the IVF index, the distribution of vectors does not change after node splitting/merging in most cases, and each bucket can be evenly split. But sometimes, different ranges of scalar values correspond to different vector distributions (even if the ranges are close to each other) so that the IVF index needs to be rebuilt over the new nodes. Likewise for the graph-based index, deleting half of the data in one subindex when splitting degrades the quality of the index, and `GaussDB-Vector` overcomes this by rebuilding on each of the new nodes. A graph-based index is only used in the root node or middle nodes containing more than 10 million vectors so that the construction cost can be compensated by the search performance gain.

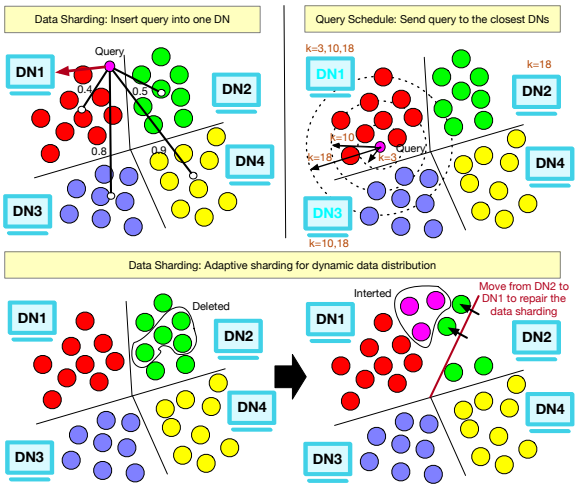


Figure 5: Data is inserted into the nearest node, and query will be sent to multiple near nodes.

In summary, the design of the hybrid index involves a tradeoff between search performance and update efficiency. Having more vector indexes on a greater number of scalar index nodes improves search performance but reduces update efficiency. `GaussDB-Vector` achieves a balance between search and updates by selectively building different types of vector indexes based on the node type and the volume of data associated with each node.

5 DISTRIBUTED VECTOR SEARCH

The two-tier architecture of `GaussDB-Vector` enables it to efficiently manage and scale large volumes of vectors. It supports billions of vectors on individual data nodes and can be expanded to handle trillions of vectors by adding more machines. In this section, we detail the methods used for distributing vectors and routing queries across the data nodes.

Data Sharding. In `GaussDB-Vector`, all the data is distributed by distance. Vectors are split into c clusters, with each cluster corresponding to one of the c data nodes. The centroid of each cluster is stored in a coordinate node. As Figure 5 shows, the distribution status has 4 types. In the `RANDOM` phase, the centroids information is invalid, and inserted vectors are randomly distributed to the data nodes. After the volume of data reaches a threshold (e.g. 50,000), `GaussDB-Vector` starts the `FULL REDISTRIBUTING` phase. In this phase, `GaussDB-Vector` clusters the data and moves the tuples to the corresponding data nodes. During this phase, clustering blocks data update queries for a short time, and search queries are still sent to all data nodes. After the centroids are decided, new vectors are inserted into the nearest data nodes, and old vectors are moved in the background. When the status becomes `CLUSTERED`, search queries are sent to close clusters, increasing the throughput. Data updates may cause the distribution to shift, degrading the quality of the clustering. `GaussDB-Vector` supports periodic incremental redistribution. In the background, `GaussDB-Vector` samples data from each data node and computes centroids for each data node. `GaussDB-Vector` then relabels each vector, and if the ratio of shifted vectors exceeds a threshold (e.g. 10%), `GaussDB-Vector` will update the centroids and start redistribution in the background.

Query Routing. One of the challenges in distributed vector search is determining how many data nodes each search query should be

sent to. If too few data nodes are searched, the recall of the ANN search will be very low. Conversely, if too many data nodes are involved, the performance will be suboptimal. We can estimate the cardinality of each data node for search queries [33], and when users do ANALYZE for distributed tables, GaussDB-Vector trains a cardinality model for each data node using a small data sample, and the models are stored in the coordinate node. For range queries, the query vector and distance threshold are used to directly estimate the cardinalities of each data node, allowing the query to be sent to only those data nodes with large cardinalities. For top-k queries, cardinality estimation of each data node is more complicated. GaussDB-Vector uses binary search on distances to find the threshold where the sum of all estimated cardinalities on data nodes exceeds k, and then queries are sent to those nodes with high contributions. For hybrid queries with a scalar filter condition, selectivity of the filter condition is considered during the cardinality estimation. However, cardinality estimation errors are unavoidable, even with sophisticated models. We find that error propagation across various query types showed that query vectors near a cluster centroid (i.e., assigned to a single machine in our distributed architecture) were least likely to suffer accuracy degradation. In contrast, queries with vectors near inter-cluster boundaries experienced the most significant performance decline. To address this issue, GaussDB-Vector additionally selects clusters whose centroids' distance to the query vector is equal to or less than the maximum distance among currently selected cluster centers.

6 HARDWARE ACCELERATION

GaussDB-Vector can improve vector search performance by utilizing specialized hardware like NPUs or GPUs, which are capable of calculating distances for batches of vector pairs. To utilize new hardware, the distance computation between two sets of vectors is transformed into basic matrix operations by using the formula $D(x, y) = x^2 - 2 \cdot (x \times y^T) + y^2$, where x^2 is a self sum-product of rows of x that can be precomputed when loading each vector into an index. Note that a fairly large matrix is needed to fully utilize the parallelism of NPUs.

Using NPUs/GPUs can significantly accelerate vector index construction and searching. For the IVF-based index, GaussDB-Vector uses NPUs to conduct data clustering. It computes distances of vectors to centroids in the NPU and labels the vectors according to their closest centroids, and then it sends vectors in the same cluster to the NPU to refine the centroid. After clustering is completed, the centroids are fixed, and vectors are labeled in batch according to their distances to centroids computed in the NPU. For the graph-based index, GaussDB-Vector builds a graph-based index by inserting each vector into a random graph in parallel, as mentioned in Section 3. For each inserted vector, we find the candidate set and select edges for it and its neighbors, and the performance bottleneck is distance computations during edge pruning. Therefore, we accumulate the inserted vectors, candidate sets closest to the vectors, and the neighbors of vectors in the candidate sets, and compute the distances in the NPU. Based on the distance table, we run the neighbor selection algorithm for each node.

GaussDB-Vector also supports using SIMD, available in mainstream CPUs, to accelerate distance computing for single vector pairs by folding the vectors into segments.

Table 1: Datasets

dataset	#rows	#dims	#cols	scalar ndv	distance
SIFT	10M	128	1	/	Euclidean
GIST	1M	960	1	/	Euclidean
HUAWEINet	15M	1024	1	/	Cosine
SIFT-H	10M	128	2	1M	Euclidean
GIST-H	1M	960	2	500K	Euclidean
HUAWEINet-H	10M	1024	2	1M	Cosine
SIFT-10B	10B	128	1	/	Euclidean

7 EXPERIMENTS

7.1 Experimental Settings

In this section, we conduct experiments for testing vector search performance (see Section 7.2), hybrid search performance (see Section 7.3), scalability on large scale datasets (see Section 7.4) and acceleration by using heterogeneous hardware (see Section 7.6).

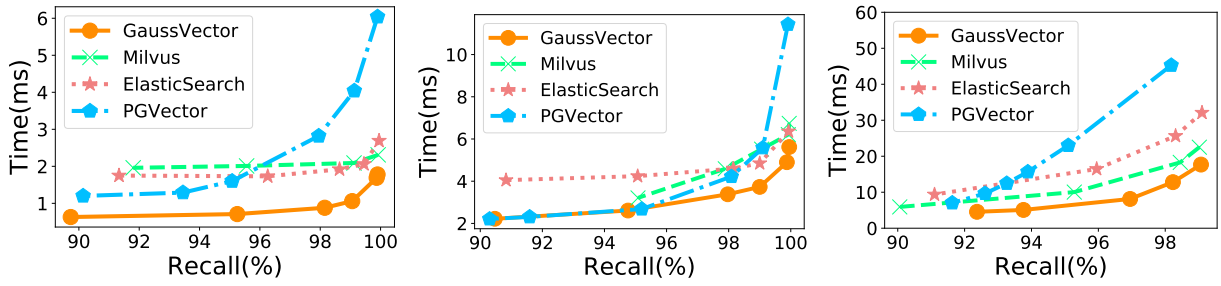
Datasets. We use three basic datasets with representative dimensions (128, 960, and 1,024). SIFT contains a collection of image embeddings generated using the SIFT algorithm [27]. GIST contains vectors describing the GIST features of images [29]. HUAWEINet is a collection of embedding vectors for documents in the telecommunication field provided by Huawei. We extend each dataset by adding an ID column that is a biased integer column with duplicates and a large range of distinct values, and we test hybrid search on these datasets. We extend SIFT to SIFT-10B to test the scalability of distributed GaussDB-Vector. Table 1 shows the details.

Baselines. We take three well-known and widely-used ANN systems, ElasticSearch, Milvus, and PGVector, as baselines. In order to ensure fairness of comparison, we only test the standalone versions of these systems. ElasticSearch is a NoSQL ANN system. It supports very comprehensive document and vector similarity retrieval features, stores semi-structured data (like JSON), and supports near real-time ANN query processing. Milvus is a storage-compute disaggregated system. It is also a vector-native ANN system, and all the data is organized as vector index structures. PGVector is an extension of PostgreSQL to support vector data storage, computing, and searching. Both GaussDB-Vector and PGVector are relational vector databases and support real-time ANN query processing. Additionally, we use HNSW indexes for all baselines, as HNSW performs the best on these datasets among all indexes supported by these systems. We build a graph-based index for single vector searching queries, and hybrid index for multi-column searching queries in GaussDB-Vector. And the default value for size of probing candidate set is 128. The index sizes of GaussDB-Vector are 9GB for 10M SIFT, 8.6 TB for 10B SIFT-10B, and 35GB for 10M HUAWEINet. These are comparable to the index sizes of the baseline systems, Milvus and ElasticSearch.

Hardware Environment. We conduct experiments on clusters of 40 machines. Each machine is equipped with 72 Intel 3.00GHz CPU cores, 64GB memory and 2TB of disk, connected by 10 Gigabit Ethernet. One machine also contains eight Ascend-920B type NPUs.

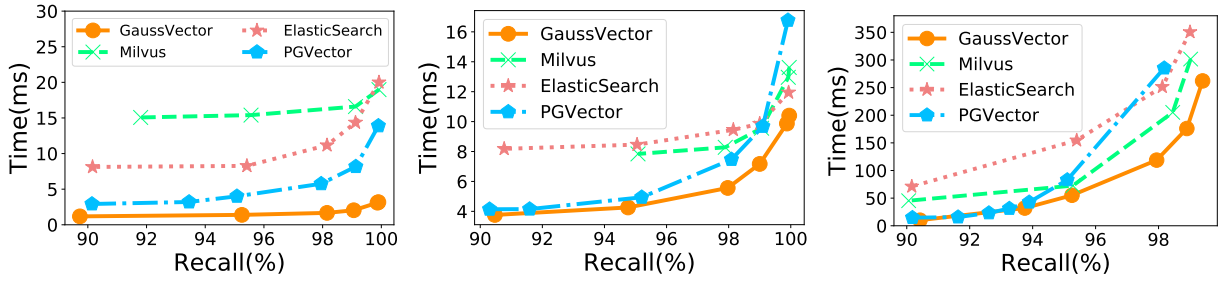
7.2 Vector Query Performance

Concurrent Execution. We compared query latency, recall, and throughput across the three datasets with an increased number of connections. We set the concurrency to 50 to ensure that the number of cores wouldn't become a bottleneck. Figure 7 and Figure 8



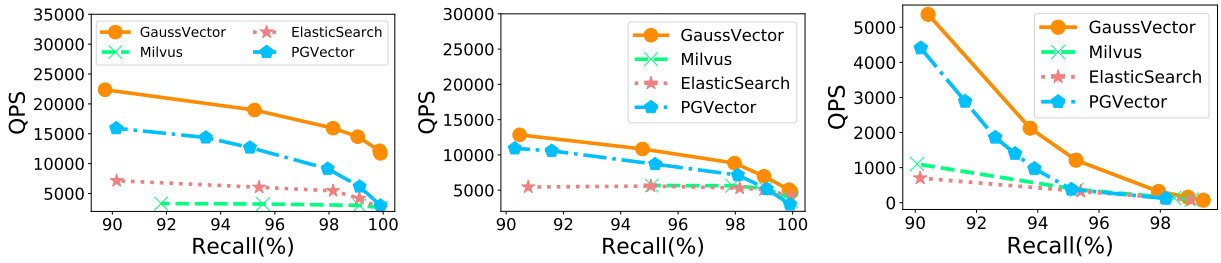
(a) SIFT dataset. (b) GIST dataset. (c) HUAWEINet dataset.

Figure 6: Average latency of ANN search with single connection.



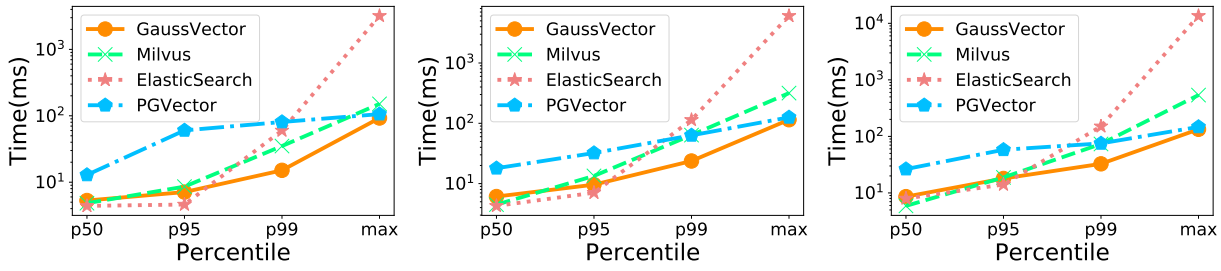
(a) SIFT dataset. (b) GIST dataset. (c) HUAWEINet dataset.

Figure 7: Average latency of ANN search with multiple connections.



(a) SIFT dataset. (b) GIST dataset. (c) HUAWEINet dataset.

Figure 8: QPS of ANN search with multiple connections.

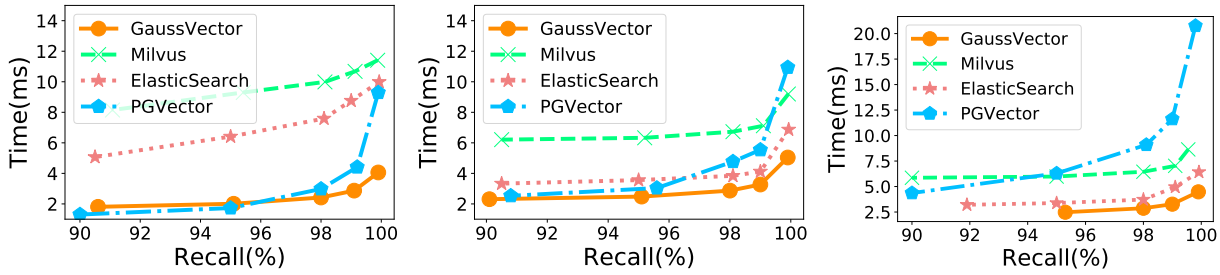


(a) SIFT dataset. (b) GIST dataset. (c) HUAWEINet dataset.

Figure 9: Insert latency.

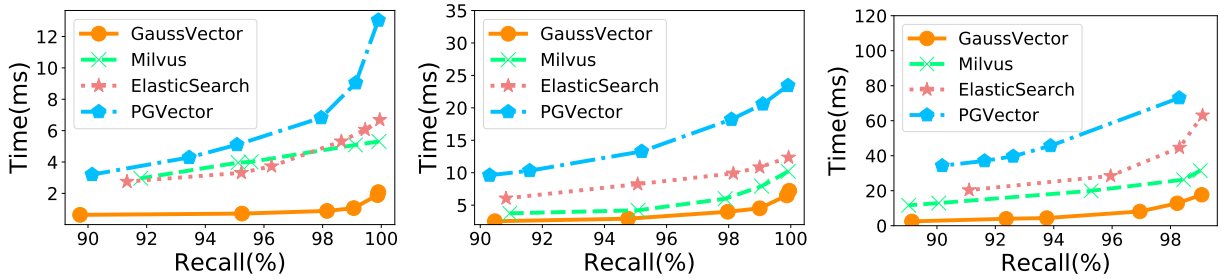
show the results. On all three datasets, GaussDB-Vector has the lowest query latency and the highest throughput. This is because GaussDB-Vector has a superior index structure compared to baseline methods, which significantly reduces resource contention by minimizing computational and I/O overhead during approximate nearest neighbor (ANN) searches. PGVector has better performance than ElasticSearch and Milvus because PGVector is designed for on-disk scenarios, and it has better buffer management, while ElasticSearch and Milvus have to load index segments for each query, incurring high I/O costs.

Insertion Performance. We also compared INSERT performance on the three datasets. Figure 9 shows the results. PGVector performs the worst compared to the other baselines. Only PGVector and GaussDB-Vector are real-time systems, which means that newly written tuples need to be flushed to disk index immediately. Milvus and ElasticSearch cache the newly added vectors in memory and build the index when the number of vectors exceeds a threshold.



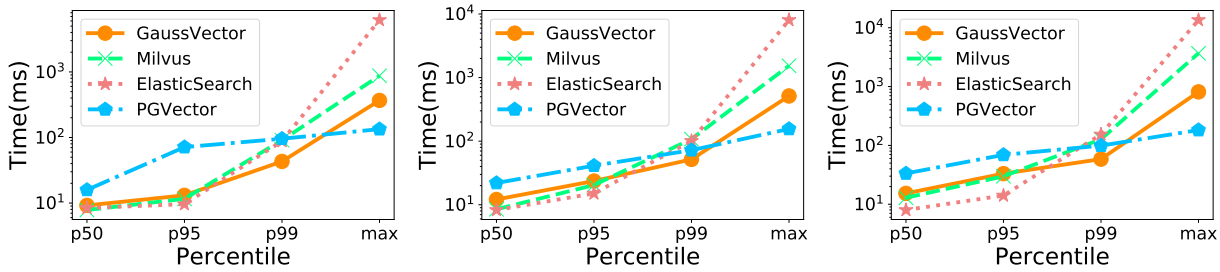
(a) SIFT dataset, 90% scalar selectivity. (b) SIFT dataset, 50% scalar selectivity. (c) SIFT dataset, 10% scalar selectivity.

Figure 10: Search latency for scalar-vector query.



(a) SIFT dataset. (b) GIST dataset. (c) HUAWEINet dataset.

Figure 11: Comparison of hybrid search latency.



(a) SIFT dataset. (b) GIST dataset. (c) HUAWEINet dataset.

Figure 12: Comparison of insertion with auto incremental id.

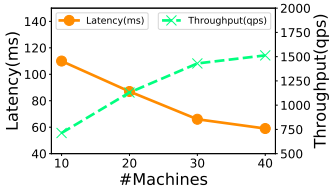


Figure 13: Scalability.

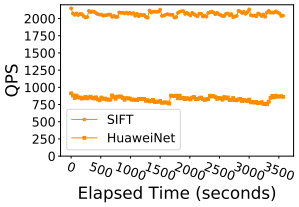


Figure 14: Update QPS.

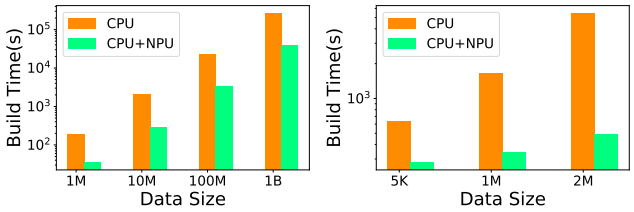
This can also explain why the maximal and 99th insert query latency of ElasticSearch and Milvus is much higher. Compared to PGVector, GaussDB-Vector controls the insertion cost much better because it has better search and edge pruning performance.

Serial Execution. We compared query latency and recall on the three datasets. Figure 6 shows the results. For SIFT, GaussDB-Vector performs the best, and has the least query time for all different recalls because the edge pruning algorithm used in GaussDB-Vector provides shorter paths for similar vectors, and the index structure design minimizes the random IO times during query execution. PGVector performs the worst when recall is high because it does not optimize the IO and computation for large candidate sets. For GIST and HUAWEINet, GaussDB-Vector and Milvus perform the best. This is because higher dimensions makes distance computing dominate the query time, and both GaussDB-Vector and Milvus have better SIMD optimization and quantization techniques to reduce distance computing cost.

7.3 Hybrid Search

Effect of Scalar Selectivity. In this experiment, we generate ANN queries with different scalar conditions and group them by selectivity on scalar values. Figure 10 shows the performance comparison on different selectivities. PGVector is most affected by selectivity. When the selectivity becomes low, the query will scan many redundant data in the index of PGVector, increasing the query time. Instead, ElasticSearch and Milvus organize data as segments, splitting data according to the scalar value. GaussDB-Vector performs the best on all selectivities, because the hybrid index can be adaptive to conditions with any selectivity, and it can select the best subindex to search for the query. ElasticSearch and Milvus have worse performance when the selectivity is high because they must merge results from different segments.

Overall Performance. In this experiment, we compare the overall performance on randomly generated scalar-vector hybrid queries.



(a) SIFT dataset, IVF Index. **(b) GIST dataset, Graph Index.**

Figure 15: Acceleration by using Ascend NPU.

Figure 11 shows the results. GaussDB-Vector performs the best because the hybrid index has stable performance on all queries.

Insertion Performance. The search performance and insert performance for hybrid scalar-vector data is a trade-off. Therefore, we tested the insertion latency by inserting tuples with auto-incremented ID as the indexed scalar field. Figure 12 shows the results of inserting one row per query. The overall insert performance of GaussDB-Vector is competitive compared to baselines due to the strategy that different layer of nodes use different types of vector subindex.

7.4 Scalability

We test distributed GaussDB-Vector on the large dataset SIFT-10B, and the evaluation results show that we can reach 99% recall, around 60 ms latency and 1,500 QPS throughput with 40 machines. As Figure 13 shows, GaussDB-Vector can have better performance with machine scale-out. When the number of machines is below 30, the performance has linear growth because GaussDB-Vector has good scalability due to vector data sharding and query scheduling techniques. When the number of machines increases further, the speed of performance growth decreases because the cost of data transformation and results merging becomes more significant.

7.5 Update Throughput

We also test update workloads on SIFT and HUAWEINet. As Figure 14 shows, throughput of updates slightly decreases periodically because the data deletions increase fragmentation. However, when the volume of updated data reaches 10% of total data size, vacuuming starts, restoring throughput. The maximal performance regression is around 5%-15%.

7.6 NPU Acceleration

We deploy GaussDB-Vector on a machine with NPU devices and run index construction and search queries on the system. Figure 15 shows that NPUs can bring 1-2 orders of magnitude improvement on index building tasks and IVF scan because NPUs can increase the parallelism of distance computing. However, the performance gain is not significant on graph index scan because graph search is hard to parallelize and the benefits of parallel computing cannot compensate for the cost of cross-device data transformation.

7.7 Real Use Case: Coding Copilot

GaussDB-Vector has been used to support software development, including code annotation, code completion, and test code generation. The copilot application schedules knowledge retrieval tasks from GaussDB-Vector and code/text generation tasks with an LLM, and then forms a chain-of-thought (CoT) to produce the code/text results. The knowledge data in GaussDB-Vector consists of code segments and documents and associated embeddings, and the total number of tuples are more than 1 billion. The application also relies on GaussDB-Vector to support frequent update for knowledge refreshing and correction. We observed that the peak throughput for search queries can reach 5,000 QPS and the throughput for update queries can reach over 3,000 QPS, and both the search and update queries can finish within 100 milliseconds on average.

8 RELATED WORKS

Vector similarity search has been studied for decades [14]. But the rise of high-quality embedding models has led to a new focus on large-scale high-dimensional vector search, as well as the development of new DBMSs designed to support this type of search [31].

Vector Similarity Search. State-of-art search algorithms tend to be based on IVF or graph indexes. IVF-based indexes like IVFADC [24] and SPANN [17] learn an optimal partitioning over the vectors to limit the scan to a small number of partitions most likely to contain near neighbors, and they have been shown to be more storage efficient compared to indexes like LSH [8, 23]. On the other hand, error guarantees are known for randomization-based indexes [9]. Graph-based indexes like NSG [21], Vamana [32], and HNSW [28] build a monotonic search network to support sub-linear time ANN search using greedy depth-first traversal.

Vector Database Management Systems. Many systems and search libraries such as PostgreSQL, SingleStore [15], ClickHouse[1], Redis[5], and ElasticSearch offer vector search capability. There are also "native" systems such as Milvus [35], Weaviate[7], and Pinecone[4] that are targeted specifically at vector datasets and workloads. For traditional systems, the challenge is to reduce the system overhead while adding vector search capabilities [39]. Many systems take advantage of distributed search and hardware acceleration using SIMD and GPUs. Zhang et al. propose to use relaxed monotonicity for early stopping when processing hybrid search [38], but the latency could be long when the selectivity of scalar condition is small. In GaussDB-Vector, we enhance performance at the index level, such as optimizing the storage structure and designing effective buffering strategies. We also implement system-level techniques to ensure high availability, scalability, and data freshness.

9 CONCLUSION

This paper presents GaussDB-Vector, a real-time persistent vector database aimed at high performance, high availability, and high scalability, while also offering high data freshness and hybrid scalar-vector filtered search capability. GaussDB-Vector demonstrates that vector search over disk-resident indexes can achieve low latency for search and update queries using storage structures and buffering strategies that are optimized for I/O in addition to compute. With appropriate techniques, these structures can also support efficient hybrid queries as well as distributed search that can scale to dataset sizes in the billions of vectors, and techniques utilizing new hardware can be used to further accelerate search.

ACKNOWLEDGMENTS

This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (62525202, 62232009), Shenzhen Project (CJGJZD20230724093403007), Zhongguancun Lab, Huawei, and Beijing National Research Center for Information Science and Technology (BNRist). Guoliang Li is the corresponding author.

REFERENCES

[1] http://clickhouse.com.
[2] http://elastic.co.
[3] http://faiss.ai.
[4] http://pinecone.io.
[5] http://redis.io.
[6] https://github.com/pgvector/pgvector.
[7] http://weaviate.io.
[8] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and optimal lsh for angular distance. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 1225–1233, Cambridge, MA, USA, 2015. MIT Press.
[9] A. Andoni, P. Indyk, and I. Razenshteyn. Approximate nearest neighbor search in high dimensions. In *ICM*, pages 3287–3318, 2018.
[10] M. Aumüller, E. Bernhardsson, and A. Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.*, 87(C), Jan. 2020.
[11] F. Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, Sept. 1991.
[12] F. Bang. GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings. In L. Tan, D. Milajevs, G. Chauhan, J. Gwinnup, and E. Rippeth, editors, *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, pages 212–218, Singapore, Dec. 2023. Association for Computational Linguistics.
[13] O. Beaumont, A.-M. Kermarrec, L. Marchal, and E. Riviere. Voronet: A scalable object network based on voronoi tessellations. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.
[14] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001.
[15] C. Chen, C. Jin, Y. Zhang, S. Podolsky, C. Wu, S.-P. Wang, E. Hanson, Z. Sun, R. Walzer, and J. Wang. SingleStore-V: An integrated vector database system in SingleStore. *Proc. VLDB Endow.*, 17(12):3772–3785, 2024.
[16] L. Chen, M. Zaharia, and J. Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.
[17] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, J. Wang, M. Yang, and J. Wang. SPANN: Highly-efficient billion-scale approximate nearest neighbor search. In *NeurIPS 2021*, 2021.
[18] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, page 253–262, New York, NY, USA, 2004. Association for Computing Machinery.
[19] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou. The faiss library. 2024.
[20] J. Fan, Z. Pan, L. Wang, and Y. Wang. Codebook-softened product quantization for high accuracy approximate nearest neighbor search. *Neurocomput.*, 507(C):107–116, Oct. 2022.
[21] C. Fu, C. Xiang, C. Wang, and D. Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5):461–474, 2019.
[22] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, S. Raz, Y. Lin, Y. Zhang, N. Mahapatro, P. Srinivasan, A. Singh, and H. V. Simhadri. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*, WWW '23, page 3406–3416, New York, NY, USA, 2023. Association for Computing Machinery.
[23] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 604–613, New York, NY, USA, 1998. Association for Computing Machinery.
[24] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.
[25] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2329–2336, 2014.
[26] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.
[27] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2), 2004.
[28] Y. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.
[29] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42(3), 2001.
[30] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inform.*, 33:351–385, 1996.
[31] J. J. Pan, J. Wang, and G. Li. Survey of vector database management systems. *VLDB Journal*, 2024.
[32] S. J. Subramanya, Devvrit, R. Kadekodi, R. Krishnaswamy, and H. V. Simhadri. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. In *NeurIPS 2019*, November 2019.
[33] J. Sun, G. Li, and N. Tang. Learned cardinality estimation for similarity queries. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1745–1757, New York, NY, USA, 2021. Association for Computing Machinery.
[34] H. Wang, W. Wu, C. Luo, A. Bian, C. Meng, Y. Wu, and J. Sun. Boosting Accuracy and Efficiency for Vector Retrieval with Local Scaling Graph . In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pages 336–348, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.
[35] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie. Milvus: A purpose-built vector data management system. In *SIGMOD*, pages 2614–2627, 2021.
[36] M. Wang, X. Xu, Q. Yue, and Y. Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.*, 14(11):1964–1978, July 2021.
[37] W. Yang, T. Li, G. Fang, and H. Wei. PASE: PostgreSQL ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2241–2253, New York, NY, USA, 2020. Association for Computing Machinery.
[38] Q. Zhang, S. Xu, Q. Chen, G. Sui, J. Xie, Z. Cai, Y. Chen, Y. He, Y. Yang, F. Yang, M. Yang, and L. Zhou. VBASE: Unifying online vector similarity search and relational queries via relaxed monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 377–395, Boston, MA, July 2023. USENIX Association.
[39] Y. Zhang, S. Liu, and J. Wang. Are there fundamental limitations in supporting vector data management in relational databases? A case study of PostgreSQL. In *ICDE*, 2024.