# LSM-VEC: A Large-Scale Disk-Based System for Dynamic Vector Search

Shurui Zhong shurui001@e.ntu.edu.sg Nanyang Technological University Singapore, Singapore Dingheng Mo dingheng001@e.ntu.edu.sg Nanyang Technological University Singapore, Singapore Siqiang Luo\* siqiang.luo@@ntu.edu.sg Nanyang Technological University Singapore, Singapore

### **Abstract**

Vector search underpins modern AI applications by supporting approximate nearest neighbor (ANN) queries over high-dimensional embeddings in tasks like retrieval-augmented generation (RAG), recommendation systems, and multimodal search. Traditional ANN search indices (e.g., HNSW) are limited by memory constraints at large data scale. Disk-based indices such as DiskANN [23] reduce memory overhead but rely on offline graph construction, resulting in costly and inefficient vector updates. The state-of-the-art clustering-based approach SPFresh [42] offers better scalability but suffers from reduced recall due to coarse partitioning. Moreover, SPFresh employs in-place updates to maintain its index structure, limiting its efficiency in handling high-throughput insertions and deletions under dynamic workloads.

This paper presents LSM-VEC, a disk-based dynamic vector index that integrates hierarchical graph indexing with LSM-tree storage. By distributing the proximity graph across multiple LSM-tree levels, LSM-VEC supports out-of-place vector updates. It enhances search efficiency via a sampling-based probabilistic search strategy with adaptive neighbor selection, and connectivity-aware graph reordering further reduces I/O without requiring global reconstruction. Experiments on billion-scale datasets demonstrate that LSM-VEC consistently outperforms existing disk-based ANN systems. It achieves higher recall, lower query and update latency, and reduces memory footprint by over 66.2%, making it well-suited for real-world large-scale vector search with dynamic updates.

#### **ACM Reference Format:**

# 1 Introduction

The proliferation of AI-driven applications such as retrievalaugmented generation (RAG) [6, 17, 22, 26], personalized recommendation [12, 32, 36], machine learning [7, 11] and multimodal search [34] has led to explosive growth in the deployment of vector databases—specialized systems that manage and query high-dimensional vector embeddings produced by large language models, vision encoders, and other machine learning models. These vector databases rely heavily on Approximate Nearest Neighbor (ANN) search to efficiently retrieve vectors that are close to a given query in high-dimensional space, balancing search accuracy and latency to support real-time or near-real-time applications [2, 3, 14, 23, 30, 45, 48].

ANN Search Indices. Numerous indexing methods have been proposed for efficient ANN search. The graph-based index [23, 30] has become the most widely used technique due to its superior recall-latency trade-offs in high-dimensional space. Meanwhile, tree-based approaches [3, 5, 37] suffer from the curse of dimensionality [5], and hash-based methods [15, 19] often require excessive memory to maintain hash tables [15]. In contrast, graph-based indexing exploits proximity relationships between vectors, enabling efficient neighbor exploration. However, classical graph-based methods such as HNSW typically assume a static vector set that resides entirely in memory. While suitable for moderate data scales, these methods become impractical at billionscale datasets, as the required memory capacity exceeds costeffective limits, especially in cloud or budget-constrained environments [4, 30].

As a result, disk-based ANN search systems have gained growing attention for large-scale deployments. DiskANN [23] extends graph-based search to disk-resident datasets by leveraging offline graph construction and aggressive pruning techniques to improve disk access locality and minimize random I/O during search. However, DiskANN is primarily designed for static datasets, where the entire dataset is available upfront, and the graph structure is carefully optimized during offline preprocessing.

Challenges within Dynamic Vector Search. The constant influx of new vector data in real-world applications drives the escalating demand for dynamic ANN search indexing. Unlike static datasets, modern systems such as recommendation engines, social networks, and generative AI models require vector databases that can efficiently handle real-time insertions, deletions, and updates. For instance, Amazon's recommendation system [1] continuously generates new product embeddings based on user interactions, requiring

 $<sup>^*{\</sup>rm Siqiang}$  Luo is the corresponding author.

immediate integration into the search index to maintain recommendation accuracy. Traditional ANN indexing methods are inadequate for such dynamic environments due to latency and computational overhead incurred by reindexing. For example, DiskANN either requires costly global rebuilding of the graph or suffers from degraded search performance due to poorly connected new nodes. Therefore, efficiently supporting continuous insertions, deletions, and evolving query patterns while maintaining high search accuracy and low disk I/O remains a crucial and open challenge for disk-based ANN systems.

Several recent studies [35, 38, 42] have explored methods to support dynamic updates in ANN indices. Among them, the state-of-the-art solution SPFresh [42] maintains incremental updates by applying clustering-based strategies rather than traditional graph indexing. SPFresh partitions the vector space into coarse-grained clusters and supports efficient in-place updates by assigning new vectors to the nearest cluster. This enables fast insertions and deletions without requiring global restructuring. However, SPFresh suffers from several key limitations. First, coarse partitioning introduces structural rigidity, that similar vectors may fall into different clusters, breaking neighborhood locality and leading to lower recall. For example, after the initial index is constructed, SPFresh achieves only around 0.75 Recall 10@10, which is significantly lower than that of graph-based methods. Second, the in-place update design restricts the flexibility of data layout optimization, making it difficult to improve disk locality over time.

Our Design: LSM-VEC. This paper presents LSM-VEC, a large-scale, disk-based vector database designed to achieve both efficient dynamic updates and high-recall in ANN search. LSM-VEC is the first system to integrate the LSM tree, a well-known indexing structure optimized for updates, to support efficient insertions and deletions in vector index. Specifically, we leverage AsterDB [33], a state-of-the-art graph-oriented LSM-tree, to maintain the HNSW proximity graph on disk, enabling efficient updates to the HNSW structure. LSM-VEC further incorporates two key techniques to reduce the query latency. (1) Selective neighbor exploration in HNSW. LSM-VEC avoids exhaustively evaluating all neighbors of each visited node. Instead, it adopts a probabilistic sampling strategy that selectively expands only a subset of neighbors. This technique is inspired by recent work on probabilistic graph traversal [28], originally proposed for inmemory ANN graphs. However, we extend this idea to the disk setting, where random I/O dominates the cost profile. Unlike the original formulation, where the primary overhead is computation, our adaptation must explicitly account for disk latency and data layout. To support this, LSM-VEC incorporates a new cost analysis that models the I/O savings from skipping neighbor evaluations, showing that even small reductions in sampling ratio can lead to substantial latency

gains without significant loss in recall. (2) LSM-VEC employs sampling-aware graph reordering to optimize vector placement on disk based on query-driven connectivity. Unlike traditional methods relying solely on static topology [41], LSM-VEC incorporates sampling-based edge weights reflecting actual traversal patterns. By co-locating vectors connected through frequently traversed edges, LSM-VEC enhances disk locality and significantly reduces random I/O operations during the traversal of the graph-based vector index.

Contributions. Overall, LSM-VEC integrates the writeoptimized characteristics of LSM-trees, the high recall of graph-based ANN search, the I/O efficiency of locality-aware reordering, and the update agility of sampling-based maintenance. This design yields a scalable and practical solution for billion-scale, dynamically evolving vector search. Experimental results on the SIFT1B dataset show that LSM-VEC consistently outperforms existing disk-based baselines. It achieves a higher Recall 10@10, lower update and query latency, and significantly lower memory usage. Compared to DiskANN, LSM-VEC reduces average update latency by up to 2.6× and memory usage by over 66.2%, while maintaining more stable and efficient query performance under dynamic workloads. These results demonstrate that LSM-VEC is a robust and efficient solution for real-world billion-scale ANN search.

In summary, this paper makes the following contributions:

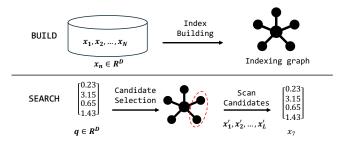
- We present a comprehensive analysis of the limitations in existing disk-based ANN systems and identify key challenges in supporting dynamic updates, efficient query execution, and scalable storage layout.
- We propose LSM-VEC, a disk-based vector search system that integrates hierarchical graph indexing with LSM-tree storage. LSM-VEC supports billionscale datasets with efficient insertions, deletions, and high-recall ANN queries.
- We implement LSM-VEC on top of AsterDB and evaluate it using billion-scale public datasets. Experimental results show that LSM-VEC achieves high accuracy and outperforms prior disk-based systems in both query and update efficiency.

## 2 Background

In this section, we introduce the fundamental task of ANN search, discuss existing solutions, and highlight the challenges in this domain.

### 2.1 ANN Search

Approximate nearest neighbor (ANN) search is a fundamental problem in large-scale vector retrieval, enabling fast similarity-based queries in applications such as retrieval-augmented generation (RAG) [26], recommendation systems [12], and multimodal search [34]. Given a query vector  $q \in \mathbb{R}^d$  and a database  $X = \{x_1, x_2, ..., x_n\}$ , the goal of ANN



**Figure 1.** An example of pipeline of approximate nearest neighbor (ANN) search, consisting of index construction, candidate selection, and distance computation.

search is to efficiently retrieve the most similar vectors to q based on a predefined distance metric.

The exact nearest neighbor (NN) search problem is formally defined as:

$$NN(q, X) = \arg\min_{x_i \in X} D(q, x_i), \tag{1}$$

where  $D(q, x_i)$  represents a similarity function, such as Euclidean distance:

$$D(q, x) = ||q - x||_2.$$
 (2)

Due to the high cost of exact nearest neighbor search in large-scale datasets, Approximate Nearest Neighbor (ANN) methods trade accuracy for efficiency by allowing approximate results instead of the exact nearest neighbors.

In practice, *Recall K@K* is commonly used to evaluate the effectiveness of ANN methods. Specifically, given a query, *Recall K@K* measures the fraction of the ground-truth *K* nearest neighbors that are successfully retrieved by the algorithm. Formally, it is defined as:

Recall K@K = 
$$\frac{|X \cap G|}{K}$$
, (3)

where X denotes the set of retrieved candidates and G is the ground-truth set of the K nearest neighbors.

Figure 1 illustrates the pipeline of a typical graph-based ANN search, which consists of two phases: index building and query processing. In the **build phase**, the system constructs a proximity-based index (e.g., graph) over a set of data vectors  $\{x_1, x_2, \ldots, x_N\} \subset \mathbb{R}^D$  based on their geometry property. Each vector is represented as a node, and edges are created between pairs of vectors that are considered close according to a chosen distance metric. In the **search phase**, given a query vector  $q \in \mathbb{R}^D$ , the system first performs candidate selection by traversing the index, followed by scanning and ranking the candidates based on distance. The final result is returned as the top-ranked candidates, representing the query vector's approximate nearest neighbors.

## 2.2 Indexing Techniques for ANN Search

Over the past decade, various ANN indexing techniques have been proposed, including tree-based [3, 5, 37], hashing-based [15, 19], and graph-based methods [21, 23, 29, 30]. Among these, graph-based approaches have emerged as the most effective for high-dimensional ANN search due to their superior recall-latency trade-off.

HNSW. Hierarchical navigable small world (HNSW) [30] is a widely adopted in-memory ANN indexing method that builds a hierarchical proximity graph. Each vector is assigned to a random maximum level based on an exponentially decaving distribution, and each layer maintains a navigable neighborhood structure. Higher layers include long-range links for coarse routing, while lower layers capture dense local neighborhoods for accurate refinement. HNSW achieves near-logarithmic search complexity and high recall. However, HNSW assumes that the entire graph resides in RAM, which makes it impractical for billion-scale datasets where memory costs become prohibitive. Furthermore, the incremental insertion procedure requires updating multiple graph layers, which leads to structural imbalance and degraded recall under high update rates. These limitations motivate the development of disk-based extensions that can retain the search quality of HNSW while supporting scalable, dynamic workloads.

DiskANN. DiskANN [23] adapts graph-based vector indices for disk-resident data by leveraging a pruned graph index [4] and combining it with disk-aware optimizations. It performs aggressive offline pruning and data reordering to improve disk locality. Neighbors with strong connectivity are placed close to each other on disk, reducing random I/O. At query time, DiskANN uses cache-aware traversal and prefetching strategies to efficiently access relevant parts of the graph. Although DiskANN significantly lowers memory consumption, it is fundamentally a static index. The graph is built entirely in memory and optimized before deployment. Insertions are appended at the end of the dataset without being properly integrated into the graph, which increases traversal cost and reduces recall. Deletions are not fully supported and may fragment the graph over time. While periodic full index reconstruction is possible, it incurs substantial computational overhead and is impractical for dynamic workloads. Consequently, DiskANN performs well in static environments but struggles to maintain high performance under continuous updates.

### 2.3 Dynamic Vector Index

While many ANN systems focus on optimizing static indexing performance, emerging workloads such as retrievalaugmented generation (RAG) and personalized search demand efficient dynamic support, where vectors are continuously inserted and deleted in real time.

3

**SPFresh.** SPFresh [42] proposes a fundamentally different design based on cluster-based indexing. Instead of maintaining a proximity graph, SPFresh organizes vectors into coarse-grained clusters via quantization. New vectors are assigned to their nearest clusters, enabling fast in-place updates and avoiding graph maintenance overhead. While this design enables efficient insertions and deletions, it suffers from structural rigidity. Similar vectors may fall into different clusters, harming recall unless many clusters are probed. This limitation is particularly severe under non-uniform data or evolving query distributions. Additionally, the system performs in-place updates, which simplifies maintenance but restricts opportunities for layout optimization. Vectors assigned near cluster boundaries may experience suboptimal placements, and SPFresh lacks mechanisms to adaptively refine these placements over time.

As a result, SPFresh trades accuracy for update speed. It achieves lower recall compared to graph-based systems, making it less suited for workloads where high precision and adaptive indexing are critical. In contrast, our approach combines the high-recall traversal of proximity graphs with update and disk-efficient mechanisms for scalable dynamic search.

#### 2.4 Our Motivation

Existing disk-based ANN systems face a fundamental tradeoff between achieving high recall, supporting efficient updates, and maintaining low search latency. Classical graphbased systems like DiskANN [23] achieve strong search accuracy by performing offline pruning and layout optimization, but they assume a static dataset and suffer from high maintenance costs when updates are required. To support dynamic workloads, recent systems take different design choices but introduce new limitations. SPFresh [42] adopts a clustering-based index with in-place updates, enabling efficient storage management but sacrificing accuracy. In contrast, FreshDiskANN [38] retains graph-based indexing to achieve better recall but lacks layout refinement during updates, resulting in sub-optimal search latency as the graph gradually deteriorates over time. Overall, none of the existing systems fully resolves the three-way trade-off among update efficiency, search performance, and accuracy in largescale, disk-based ANN search. Designing an index that simultaneously supports high-recall search, low-latency query processing, and efficient real-time updates remains a critical open challenge.

To address this gap, we propose LSM-VEC, a disk-based vector search system that integrates graph-based indexing, lightweight traversal, and storage-aware layout optimization. A key design decision in LSM-VEC is the use of a log-structured merge tree (LSM-tree) as the underlying storage architecture. Unlike traditional B<sup>+</sup>-tree or static file formats, LSM-trees are inherently write-optimized: they absorb random updates via a memory-resident buffer and organize data

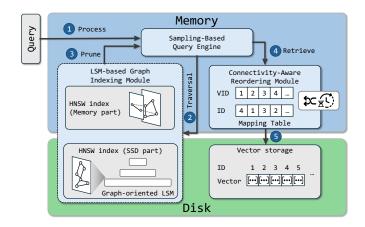


Figure 2. LSM-VEC architecture.

in sequentially written disk files through background compaction. This makes them particularly suitable for workloads with frequent insertions and deletions.

By combining a hierarchical graph-based vector index with LSM-tree-based storage and layout-aware maintenance, LSM-VEC achieves high recall and robust support for dynamic updates with minimal I/O overhead. Building on this foundation, we further introduce substantial query-time optimizations through a sampling-based probabilistic search strategy and connectivity-aware graph reordering. These techniques significantly reduce I/O during vector search, enabling the system to meet the performance and scalability requirements of real-world applications such as retrieval-augmented generation and personalized recommendation, where low-latency vector retrieval must coexist with massive data that continuously evolves.

## 3 The Design of LSM-VEC

## 3.1 Overview

Figure 2 presents the overview of LSM-VEC. LSM-VEC is constructed upon a graph-oriented log-structured merge tree [33] (LSM-tree) that enables efficient updates and queries for graph-based ANNS index. Building on this foundation, we have integrated three key modules further to enhance the performance of LSM-VEC, each tailored to address specific challenges associated with disk-based ANN searches and updates.

LSM-based Hierarchical Graph Indexing Module. This module employs a memory-disk hybrid hierarchical proximity graph inspired by the Hierarchical Navigable Small World (HNSW) model. It addresses scalability limitations of HNSW by partitioning the graph into memory-resident upper layers and a disk-resident bottom layer managed through an LSM-tree. The upper layers facilitate rapid long-range navigation, while the lower layers leverage efficient disk indexing and management. Vector storage and graph indexing

4

are decoupled to enhance storage efficiency and enable quick disk-based vector retrieval.

Sampling-Based Query Engine. Recognizing the computational overhead associated with naive neighbor exploration, this module implements a probabilistic neighbor selection mechanism. Utilizing a probabilistic filtering strategy based on projection-based similarity scores, the engine selectively evaluates neighbors, notably reducing disk I/O and computation.

Connectivity-Aware Reordering Module. To minimize random disk access, this module continuously optimizes the layout of data based on observed access patterns. Unlike traditional static reordering methods, it dynamically leverages runtime traversal statistics derived from the sampling-based query engine. Nodes frequently traversed together are incrementally co-located during regular LSM-tree compactions, enhancing data locality and reducing random disk I/O. This adaptive strategy is specifically designed for disk-resident graphs, efficiently handling updates without requiring extensive restructuring.

Collectively, these modules form an integrated solution tailored to the unique demands of large-scale, dynamic ANNS workloads. The LSM-based hierarchical indexing module ensures efficient index updating and querying at scale, the sampling-based query engine significantly reduces unnecessary IO overhead during search, and the connectivity-aware reordering module dynamically adapts storage layout to minimize disk latency. Detailed explanations and performance analyses of each module are provided in subsequent sections.

## 3.2 LSM-based Proximity Embedding: Efficient Indexing for Dynamic ANN Search

Disk-based approximate nearest neighbor search (ANNS) faces significant challenges in efficiently handling dynamic updates, as these updates often result in substantial random disk writes. LSM-VEC addresses this issue by extending hierarchical graph-based ANN search [30] for large-scale diskbased environments through an integration with an LSMtree-based storage engine. This design allows the system to retain the high recall and logarithmic query complexity of HNSW while addressing memory constraints and update inefficiencies encountered when scaling to billions of vectors. HNSW is known for its excellent balance between efficiency and accuracy due to its hierarchical structure, which facilitates efficient long-range navigation in higher layers and precise neighbor refinement in lower layers. However, the original HNSW design assumes that the entire graph structure resides in memory, making it unsuitable for large-scale disk-based scenarios.

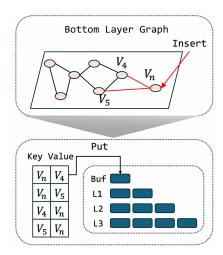
**Storage Layout in LSM-VEC.** To overcome this limitation, LSM-VEC decomposes the HNSW index into memory-resident upper layers and a disk-resident bottom layer. As shown in Figure 2, the upper layers of HNSW are retained

in RAM to support low-latency search entry and fast hierarchical navigation. According to the exponential decay distribution used in HNSW's level assignment [30], the upper layers are typically small. Empirically, less than 1% of all nodes reside above the bottom layer, which makes them suitable for in-memory storage even at billion-scale. Whereas the major layer of HNSW is stored on disk and maintained via an LSM-tree, facilitating efficient index updates. Since each vector insertion or deletion generates substantial new edges in the major layer, the adoption of an LSM-tree allows LSM-VEC to handle these updates efficiently without requiring a global restructuring of the entire index. In addition, LSM-VEC stores vector data separately from the graph index. All vectors are placed in a contiguous on-disk array, sorted by their corresponding ID. This layout allows constant-time retrieval via offset computation, avoiding redundant data storage while ensuring that vector access and neighbor traversal remain both efficient and write-friendly.

**Search in LSM-VEC.** Search in LSM-VEC follows a layered traversal strategy, optimized to minimize random disk I/O. The search process starts from the upper memory-resident layers, where long-range edges enable efficient navigation towards the target region. Once the search reaches the lower disk-resident layer, LSM-VEC employs the sampling-guided traversal technique introduced in Section 3.3 to selectively explore a small set of promising neighbors. This approach significantly reduces unnecessary disk accesses.

**Insertion in LSM-VEC.** Each newly inserted vector is indexed following a hierarchical HNSW-style process. The vector is assigned to a random level L sampled from an exponentially decaying distribution. The insertion then proceeds top-down through the hierarchy: at each level  $\ell$  (except the bottom layer), the system identifies approximate neighbors and connects the vector to the top-M closest nodes using in-memory search. At the bottom layer, neighbor search is conducted on the disk-resident graph stored in the LSM-tree. The vector is connected to the top-M nearest disk-resident nodes, and the resulting edges are written to the LSM-tree for durable storage.

Figure 3 presents a running example of the bottom-layer insertion procedure in LSM-VEC. In this example, a new vector  $v_n$  is inserted into the disk-resident graph. Through a disk-based nearest neighbor search, LSM-VEC identifies  $v_4$  and  $v_5$  as the top-M closest neighbors to  $v_n$ . The system then forms bidirectional links between  $v_n$  and these two nodes. As shown in the lower part of the figure, these edges are encoded as key-value pairs and inserted into the LSM-tree, where the key represents the source vector ID, and the value is its neighbor. All insertions are initially buffered in memory and eventually propagated to deeper LSM-tree levels via compaction. This example illustrates how LSM-VEC integrates new vectors into the disk-resident index with low overhead. The complete insertion procedure is detailed



**Figure 3.** An illustration of vector insertion in LSM-VEC. The new node  $v_n$  is connected to two bottom-layer neighbors  $v_4$  and  $v_5$ , and the resulting edges are stored in the LSM-tree.

in Algorithm 1, where  $NN(\cdot)$  denotes the nearest neighbor search performed over either the in-memory graph or the disk-resident index.

## Algorithm 1 Insertion in LSM-VEC.

```
Require: x \in \mathbb{R}^d: vector to insert; \mathcal{G}: in-memory graph; \mathcal{D}:
       disk-resident graph; L_{max}: current maximum level
   1: Sample level L \sim \Pr(L) \propto e^{-L}
  2: if L > L_{max} then
  3:
             L_{max} \leftarrow L
  4: end if
  5: E \leftarrow entry point from top layer of \mathcal{G}
  6: for \ell = L_{max}, ..., L + 1 do
             E \leftarrow \text{GreedySearch}(x, E, \mathcal{G}_{\ell})
  8: end for
      for \ell = L, ..., 2 do
            N_{\ell} \leftarrow \text{NN}(x, \mathcal{G}_{\ell})
 10:
            \mathcal{G}_{\ell} \leftarrow \mathcal{G}_{\ell} \cup \{(x, \text{TopM}(N_{\ell}))\}
 12: end for
 13: N_1' \leftarrow \text{NN}(x, \mathcal{D})
 14: \mathcal{D} \leftarrow \mathcal{D} \cup \{(x, \text{TopM}(N_1'))\}
 15: return (\mathcal{G}, \mathcal{D})
```

**Deletion in LSM-VEC.** To support efficient deletions in dynamic vector databases, LSM-VEC performs a local neighbor relinking strategy for both in-memory and disk-resident layers. When a vector is deleted, its immediate neighbors are reconnected using approximate neighbor search to preserve local graph connectivity. For the disk layer, LSM-VEC identifies affected nodes and inserts new edges into AsterDB, avoiding full reindexing.

In hierarchical HNSW indexing, the deleted node may exist in both the memory-resident upper layers and the disk-resident bottom layer. LSM-VEC ensures deletions are applied consistently across all levels. After relinking neighbors, the system removes all edges involving the deleted node from AsterDB and deletes the corresponding vector data. The full deletion procedure is described in Algorithm 2.

## **Algorithm 2** Deletion in LSM-VEC.

```
Require: x \in \mathbb{R}^d: vector to delete; \mathcal{G}: in-memory graph; \mathcal{D}:
      disk-resident graph
  1: for each layer \ell where x exists in \mathcal{G} do
           N_{\ell} \leftarrow \text{Neighbor}_{\ell}(x)
  2:
  3:
           for each p \in N_{\ell} do
                Remove edge (p, x) and (x, p) from G_{\ell}
  4:
  5:
           C \leftarrow \bigcup_{p \in N_{\ell}} \text{Neighbor}_{\ell}(p)
           for each p \in N_{\ell} do
  7:
                N_p' \leftarrow \text{NN}(p, C)
  8:
  9.
                Connect p to TopM(N'_p) in \mathcal{G}_\ell
10:
           Remove node x from G_{\ell}
11:
12: end for
 13: N_1 \leftarrow \text{Neighbor}_1(x) \text{ in } \mathcal{D}
14: for each p \in N_1 do
           Remove edge (p, x) and (x, p) from \mathcal{D}
16: end for
17: C \leftarrow \bigcup_{p \in N_1} \text{Neighbor}_1(p)
18: for each p \in N_1 do
           N_p' \leftarrow \text{NN}(p, C)
20:
           Connect p to TopM(N'_p) in \mathcal{D}
22: Remove vector x and all edges involving x from \mathcal{D}
23: return (\mathcal{G}, \mathcal{D})
```

# 3.3 Sampling-Guided Traversal: Fast and Robust Search over Disk-Based Graphs

Efficient ANN search on graph-based indices relies on exploring a minimal number of nodes and edges while ensuring high recall. Traditional graph-based ANNS methods, such as HNSW, typically employ greedy traversal strategies to navigate from an entry point to the target neighborhood. However, when applied to disk-based settings, naive greedy search often needs to exhaustively scan all neighbors of a node to make local routing decisions, incurring substantial random I/O overhead. To address this, LSM-VEC introduces a sampling-based filtering strategy inspired by probabilistic routing [8, 28], enabling efficient pruning of unlikely candidates with theoretical guarantees.

A key observation in graph-based ANN search is that not all neighbors need to be explored with equal probability. When expanding a node's neighbors during traversal, conventional greedy search evaluates all potential neighbors and selects the closest ones for further expansion. However, this approach results in redundant distance computations and excessive candidate evaluations, increasing query latency. This motivates LSM-VEC to adopt the probabilistic selection mechanism that dynamically adjusts the exploration probability of each neighbor based on its estimated proximity to the query. This sampling-based approach reduces unnecessary distance calculations while preserving high recall. For ease of understanding of our system, below we introduce the sampling techniques [8, 28] in detail.

At the initialization stage, the system samples m random projection vectors  $\{a_i\}_{i=1}^m \sim \mathcal{N}(0, I_d)$ , where d is the vector dimension. Each data vector  $x \in \mathbb{R}^d$  is encoded into a binary sign-hash code:

$$\operatorname{Hash}(x) = \left[\operatorname{sgn}(x^{\top} a_1), \dots, \operatorname{sgn}(x^{\top} a_m)\right] \in \{-1, 1\}^m, \quad (4)$$

where sgn(z) = 1 if  $z \ge 0$ , and -1 otherwise. These hash codes are stored in memory at insertion time.

At query time, given a query vector q, the system computes its hash code and compares it to each candidate u via:

$$#\operatorname{Col}(q, u) = \frac{1}{2} \left( m + \operatorname{Hash}(q)^{\top} \cdot \operatorname{Hash}(u) \right), \tag{5}$$

which counts the number of matching hash bits (collisions).

To ensure recall guarantees, a collision threshold is applied according to Hoeffding's inequality. Given a target error  $\epsilon$  and a maximum distance  $\delta$ , the threshold number of collisions is defined as  $T_{\epsilon}^{\rm SimHash}$ . Here,  $\delta$  typically corresponds to the distance between the query q and the farthest candidate in the current top-k candidate set, serving as a dynamic cutoff for evaluating new candidates.

Then, the filtering condition becomes:

$$\Pr\left[\|q - u\| \le \delta \mid \#\operatorname{Col}(q, u) \ge T_{\epsilon}^{\operatorname{SimHash}}\right] \ge 1 - \epsilon. \quad (6)$$

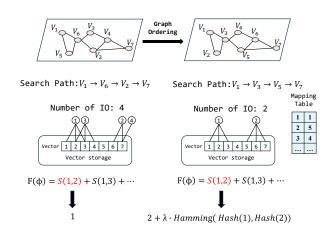
This allows the system to safely skip candidates with insufficient hash collisions, significantly reducing I/O and maintaining theoretical recall guarantees.

By integrating query-adaptive sampling and errorcontrolled hash filtering, LSM-VEC significantly accelerates the search on disk-based graphs while maintaining theoretical guarantees, making it highly suitable for billion-scale ANN applications.

**Theoretical Cost Analysis.** To quantify the effectiveness of sampling-guided traversal, we analyze and compare the expected search cost before and after applying sampling. Let T be the total number of visited nodes during the search, d be the average node degree,  $t_v$  be the time to fetch a single vector from disk, and  $t_n$  be the time to retrieve the neighbor list of a node from the LSM-tree.

In conventional graph traversal, all neighbors of each visited node are evaluated, resulting in a search cost of:

$$Cost_{full} = T \cdot (t_n + d \cdot t_v). \tag{7}$$



**Figure 4.** An example of graph ordering to improve I/O efficiency.

In contrast, LSM-VEC introduces a sampling ratio  $\rho \in (0,1]$ , which controls the fraction of neighbors to be accessed during traversal. A smaller  $\rho$  implies more aggressive pruning of neighbor evaluations. The corresponding search cost is reduced to:

$$Cost_{sampling} = T \cdot (t_n + \rho \cdot d \cdot t_v). \tag{8}$$

Thus, the expected I/O cost saving brought by sampling is:

$$\Delta = T \cdot (1 - \rho) \cdot d \cdot t_{v}. \tag{9}$$

This analysis highlights that sampling-based search effectively reduces vector I/O cost while preserving search quality, especially when the sampling ratio  $\rho$  is carefully tuned to balance recall and efficiency.

## 3.4 Locality-Aware Reordering: Adaptive Layout Optimization for Disk Traversal

To minimize random I/O overhead during disk-based ANN search, LSM-VEC adopts a graph reordering strategy to improve the physical locality of vectors stored on disk. This design is inspired by prior work on offline graph ordering [41], which aims to cluster closely related nodes together in memory to accelerate graph traversal.

Specifically, existing methods typically define a scoring function S(u, v) between two nodes u and v based on static graph topology. For example, the state-of-the-art approach [41] combines the number of shared in-neighbors  $S_s(u, v)$  and direct connections  $S_n(u, v)$  as:

$$S(u, v) = S_s(u, v) + S_n(u, v),$$
 (10)

where  $S_s(u, v) = |N_I(u) \cap N_I(v)|$  measures the number of common in-neighbors of u and v, and  $S_n(u, v)$  counts the

existence of direct edges between u and v. This static formulation captures structural proximity but ignores runtime query patterns.

In contrast, LSM-VEC introduces a fundamentally different score definition tailored for dynamic ANN search. Instead of relying solely on static graph structure, we derive S(u, v) from query-time traversal statistics. In particular, we define:

$$S(u,v) = S_s(u,v) + S_n(u,v) \cdot (1+\lambda)$$
· Hamming(Hash(q), Hash(u)). (11)

This sampling-driven score directly captures the runtime importance of each edge based on its frequency in sampled search paths, enabling the layout optimization to reflect actual query behavior.

Given this score definition, LSM-VEC aims to find a node permutation  $\phi(\cdot)$  that maximizes the total edge scores within a physical prefetch window of size w, following the formulation as:

$$F(\phi) = \sum_{0 < \phi(v) - \phi(u) \le w} S(u, v), \tag{12}$$

where  $v_i = \phi^{-1}(i)$  denotes the node placed at position i in the storage layout. Intuitively, this objective encourages frequently co-accessed nodes to be placed closely together, so that they can be fetched together within the same disk I/O block.

To achieve this goal, LSM-VEC periodically applies a global reordering pass over the disk-resident bottom-layer graph. The reordering is guided by the query-sampled edge heatmap, and the resulting layout naturally adapts to evolving query patterns without requiring prior knowledge of the full graph structure.

Running Example. In Figure 4, we demonstrate the effectiveness of locality-aware reordering with a running example. Left panel (original layout): The nodes of the proximity graph are stored in memory without any consideration for the query access pattern. As a result, frequently traversed nodes are dispersed across the vector storage. For instance, during the query traversal path  $V_1 \rightarrow V_6 \rightarrow$  $V_2 \rightarrow V_7$ , since these nodes are not physically contiguous, the system must perform four random I/O operations to retrieve the corresponding vectors during the search. Right panel (after reordering): After applying locality-aware reordering, the system rearranges the vector storage so that graph neighbors likely to be accessed in succession are placed adjacently in memory. In the reordered layout, the query that originally traversed  $V_1$ ,  $V_6$ ,  $V_2$ , and  $V_7$  is effectively transformed into a new, optimized traversal path  $V_1 \rightarrow V_3 \rightarrow V_5 \rightarrow V_7$ , where adjacent nodes in the graph now correspond to sequentially stored vectors. With this physical reordering, the number of random I/O operations required is reduced to only two. This example demonstrates how reordering can align the physical storage layout with runtime search paths, thereby improving I/O efficiency.

By integrating sampling-driven edge weights into reordering decisions and aligning them with the LSM-tree's compaction mechanism, LSM-VEC achieves high disk locality without sacrificing update efficiency. This approach ensures that the physical layout of the index remains closely aligned with the logical query paths, significantly reducing I/O cost in disk-based ANN search.

#### 4 Related Work

In-Memory ANN Indexing Graph-based ANN methods have emerged as the dominant paradigm for high-accuracy, low-latency vector search in RAM. Notably, HNSW [30] introduces a hierarchical small-world graph structure that enables logarithmic search time and strong recall guarantees. Variants and extensions of HNSW, including those in FAISS [13] and NGT [20], further improve indexing speed and graph quality. However, all these methods assume the index resides entirely in memory, limiting scalability in billion-scale scenarios. NSW [29] also use proximity-based neighborhood structures, but they tend to suffer from higher memory usage or lower recall under tight latency constraints.

Beyond graph-based approaches, several memory-efficient ANN systems adopt alternative indexing paradigms. SCANN [18] combines optimized quantization and learned pruning to reduce memory and latency, achieving state-of-the-art performance under inner-product similarity. BATL [27] proposes a learned tree-based index that achieves high recall and low latency using balanced partition trees trained with neural sequence prediction. PCNN [40] adopts error-correcting codes (polar codes) for efficient high-dimensional hashing, offering better trade-offs than classical LSH. While these systems achieve high performance under in-memory settings, their scalability and update efficiency degrade significantly in billion-scale, disk-based scenarios.

Disk-Based ANN Systems To overcome the memory limitations of in-memory ANN indices, several disk-based systems have been proposed to scale to billion-scale datasets. These methods reduce the memory footprint by optimizing disk access and leveraging SSD-friendly designs. DiskANN [23] builds a pruned graph offline and uses quantized vectors in memory to guide the search, loading only the best candidates from disk. SPANN [9] partitions the vector space via hierarchical clustering and builds local graphs, enabling efficient disk access but limiting the flexibility of update. ScaNN [16] combines quantization, reordering, and reranking in a multistage pipeline, balancing latency and accuracy but assuming a static corpus due to high retraining cost.

In summary, while these systems offer strong performance under static workloads, they lack native support for dynamic updates, limiting their applicability in evolving real-world deployments.

**Hardware-accelerated ANN systems.** A number of recent efforts have proposed ANN solutions based on GPUs

and FPGAs to accelerate large-scale vector search. For example, FusionANNS [39], BANG [25], RUMMY [47], iQAN [46] and ParlayANN [31] exploit GPU-friendly pipelines or CPU-GPU hybrid execution to enable low-latency approximate search. Others such as DF-GAS [44] design specialized FPGA-based infrastructures to support billion-scale ANN with high throughput and energy efficiency.

Despite their performance advantages, these systems are fundamentally designed for in-memory settings. They assume the index or its compressed form can fit into high-bandwidth accelerator memory and often lack support for real-time updates or truly out-of-core datasets.

In contrast, LSM-VEC addresses the orthogonal challenge of disk-based ANN search at billion scale. Rather than relying on dedicated hardware, it improves system-level efficiency through sampling-guided search, LSM-tree-based index maintenance, and graph reordering. Our design complements hardware accelerators and can be integrated with future hybrid pipelines that combine disk-resident storage with accelerator-based query processing.

Dynamic and Hybrid ANN Indexing SPFresh [42] represents a recent effort to support dynamic updates in largescale ANN search. Instead of graphs, it partitions the vector space into clusters and performs in-place updates via quantization-based assignments. While this strategy enables efficient insertions and deletions, it suffers from degraded recall due to coarse cluster boundaries and the inability to preserve fine-grained neighborhood structure. Moreover, SPFresh does not exploit graph connectivity or adaptive layout reordering. FreshDiskANN [38] improves upon the design of DiskANN by enabling update support without full reprocessing. It maintains a fixed disk layout and incrementally inserts new vectors by connecting them to diskresident neighbors. For deletions, it uses a localized neighborrelinking strategy, where neighbors of a deleted node are reconnected using a pruning rule. While FreshDiskANN supports efficient insertions and deletions, it does not perform any form of global reordering. As a result, the physical layout of vectors gradually deteriorates over time, which can negatively impact I/O locality and search performance. Recent systems like NV-tree [43] and PQ-based hybrid indexing [24] attempt to blend vector quantization with disk-aware indexing. Yet, these systems either fail to support dynamic updates or yield subpar search latency compared to graph-based methods.

#### 5 Evaluation

This section evaluates the performance of LSM-VEC against multiple baselines under various workloads.

## 5.1 Experimental Setting

**System Environment.** All experiments are performed on a dedicated server configured as follows:

- CPU: Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz (64 cores, 128 threads).
- Memory: 256 GB DDR4.
- Disk: 2 TB NVMe SSD.
- **Operating System**: Ubuntu 20.04.4 LTS (kernel 5.4.0-100-generic).

All indices are constructed on disk unless explicitly stated. For each configuration, we perform a warm-up phase to load frequently accessed pages into memory, followed by 10K randomly ordered queries for latency evaluation. We repeat each experiment three times and report the average result.

**Baselines.** We compare LSM-VEC with two representative disk-based ANN systems:

- **DiskANN** [23]: A graph-based disk-resident index that relies on offline pruning and reordering to improve search performance on static datasets.
- **SPFresh** [42]: A clustering-based dynamic ANN system that enables fast insertions and deletions via in-place updates but sacrifices recall due to coarse partitioning.

All systems are tuned to achieve comparable recall, and parameters such as the number of neighbors, search depth, and memory budget are carefully selected based on opensource implementations and prior work.

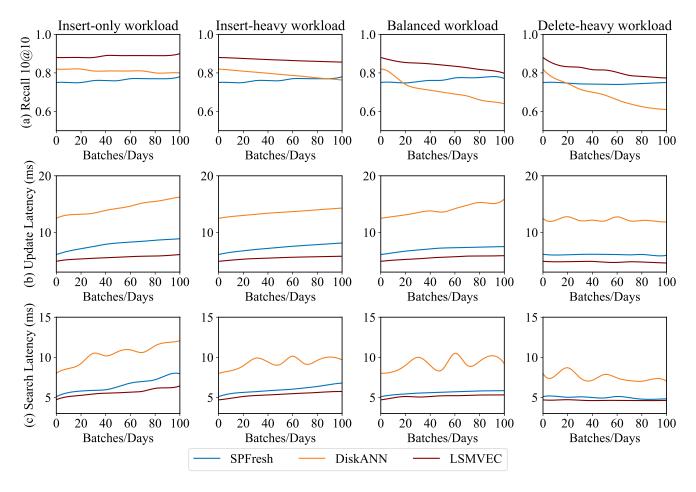
**Dataset.** We conduct all experiments on the widely-used **SIFT1B** [10] dataset, which contains one billion 128-dimensional SIFT descriptors extracted from image patches. Although SIFT1B is designed for billion-scale evaluation, we use a 100-million scale subset in our experiments due to hardware constraints. In particular, existing solutions such as DiskANN require several terabytes of memory to handle the full 1-billion dataset, which exceeds the memory capacity of our server. This experimental setting also reflects practical scenarios where billion-scale indices are typically partitioned or sharded in real-world systems.

**Evaluation Metrics.** We evaluate and report the following key metrics:

- Recall 10@10: Search accuracy, measuring the fraction of true nearest neighbors found within the top-10 returned results for each query.
- Query latency: Average search latency per query, computed over 100 randomly sampled queries.
- **Update latency**: Average latency of vector insertions under dynamic workloads, reflecting the efficiency of handling online updates.
- Memory usage: Peak memory consumption measured during search and update operations, including both index structures and graph buffers.

**Implementation Notes.** The on-disk index of LSM-VEC is implemented in C++ on top of AsterDB<sup>1</sup>, with concurrency support for graph construction and query processing. For

 $<sup>^1</sup> https://github.com/NTU-Siqiang-Group/Aster \\$ 



**Figure 5.** Evaluation of LSM-VEC under four update scenarios with different insert-delete ratios. We report recall, update latency, and search latency, simulating real-world dynamic workloads where the index continuously evolves. Each batch corresponds to 1% vector updates (1% insertion or 1% deletion).

DiskANN<sup>2</sup> and SPFresh<sup>3</sup>, we use the official open-source implementations and follow the recommended settings from their published papers. In all experiments, we avoid using SSD caching layers or memory-mapped file tricks to simulate a realistic deployment scenario for large-scale vector databases.

# 5.2 System Performance

LSM-VEC delivers robust and efficient performance across diverse update scenarios. To evaluate the robustness and efficiency of LSM-VEC under dynamic workloads, we simulate real-world update scenarios by designing a series of batch workloads with varying insert and delete ratios. In vector database applications like personalized search, recommendation, and RAG systems, vectors are frequently inserted, deleted, or updated to reflect evolving content or

user behavior. Notably, an update operation is commonly modeled as a delete followed by an insert.

We construct four representative workloads to reflect different application scenarios:

- **Insert-only workload**: 100% insert operations, simulating system initialization or rapid data growth.
- Insert-heavy workload: 70% insert and 30% delete operations, capturing scenarios with frequent new data but occasional clean-up.
- Balanced workload: 50% insert and 50% delete operations, representing mature systems with stable user bases.
- **Delete-heavy workload**: 30% insert and 70% delete operations, reflecting data refreshing or model retraining phases.

Figure 5 presents the comprehensive results of these experiments, reporting Recall 10@10, update latency, and search latency across different update workloads. Each batch corresponds to 1% vector updates (1% insertion or 1% deletion),

<sup>&</sup>lt;sup>2</sup>https://github.com/microsoft/DiskANN

 $<sup>^3</sup> https://github.com/SPFresh/SPFresh\\$ 

following the real-world dynamic update pattern as adopted in SPFresh [42].

For Recall 10@10 (Figure 5(a)), LSM-VEC consistently outperforms both SPFresh and DiskANN across all workloads. In the Balanced workload, LSM-VEC achieves 88.4% recall, significantly higher than SPFresh (75.5%) and DiskANN (82.0%). In the Delete-heavy workload, where extensive deletions severely impact graph quality, DiskANN's recall drops dramatically to 61.0%, while LSM-VEC still maintains 77.4% recall, demonstrating its robustness against dynamic data evolution.

For update latency (Figure 5(b)), LSM-VEC exhibits the lowest average update latency across all workloads. In the Insert-only workload, LSM-VEC achieves an average update latency of 4.90ms per vector, which is 1.2× faster than SPFresh (6.10ms) and 2.6× faster than DiskANN (12.5ms). As the workload becomes more delete-heavy, DiskANN's update latency increases to 11.86ms, while LSM-VEC remains stable at 4.60ms, benefiting from its write-optimized LSM-tree-based design.

As Figure 5(c) illustrates, for search latency, LSM-VEC consistently provides the lowest and most stable average search latency across all workloads. In the Insert-only workload, LSM-VEC achieves 4.70ms search latency, and this value remains nearly unchanged at 4.63ms in the Delete-heavy workload. In contrast, DiskANN suffers from degraded locality, increasing its search latency from 8.0ms to 12.0ms. SPFresh maintains relatively stable search latency (around 7ms), but this comes at the cost of lower recall due to its coarse-grained clustering.

Overall, these results demonstrate that LSM-VEC effectively addresses the challenges of dynamic ANN search. It achieves higher recall, lower update latency, and more stable search latency than both DiskANN and SPFresh, making it a strong candidate for real-world large-scale vector database deployments.

LSM-VEC achieves lower memory usage without sacrificing accuracy. Apart from update and query performance, memory usage is a critical concern for billion-scale vector search systems, especially in memory-constrained environments. Figure 6 reports the memory consumption of all systems over time, under four different update workloads. We include both the memory consumed by the in-memory index and the buffering layer for dynamic updates.

In the Insert-only workload, DiskANN exhibits rapid memory growth from 25GB to 76GB, as all inserted nodes and graph structures must be kept in memory. In contrast, both SPFresh and LSM-VEC demonstrate stable memory usage throughout the run. Specifically, SPFresh grows slightly from 20.1GB to 23.9GB, while LSM-VEC increases from 22.4GB to 26.5GB, benefiting from compact upper-layer storage and disk-resident bottom-layer graph. Notably, LSM-VEC maintains a flat memory curve even with 100% insert operations.

In the Insert-heavy workload, the memory gap between DiskANN and other systems further widens. DiskANN reaches 66GB due to increasing memory pressure from mixed inserts and deletes. Both SPFresh and LSM-VEC maintain low memory usage, stabilizing below 26GB. The final memory footprint of LSM-VEC is only 25.6GB, slightly higher than SPFresh's 25.3GB, demonstrating the effectiveness of LSM-tree-based storage in isolating on-disk graph maintenance from in-memory structures.

In the Balanced workload, DiskANN's memory usage spikes to 80GB due to frequent deletions and fragmented memory management. Meanwhile, SPFresh increases moderately to 27.0GB, and LSM-VEC remains highly compact at 27.0GB. Despite the high churn from 50% insert and 50% delete operations, both systems manage to cap memory usage effectively.

Finally, under the Delete-heavy workload, DiskANN continues to consume excessive memory (exceeding 69GB), while both SPFresh and LSM-VEC maintain excellent memory stability. SPFresh grows slowly to 25.7GB, and LSM-VEC stays within 22.4GB to 25.7GB across the entire run, showing strong adaptability to deletion-intensive environments.

Overall, this experiment demonstrates that both SPFresh and LSM-VEC provide low and stable memory usage across a wide range of dynamic workloads. Compared to DiskANN, which suffers from significant memory amplification in dynamic scenarios, LSM-VEC leverages its LSM-tree-based bottom-layer storage to efficiently bound memory usage while preserving high recall and low latency. This design makes LSM-VEC particularly suitable for billion-scale vector search deployments in resource-constrained environments.

LSM-VEC achieves superior search-update balance under dynamic workloads. Figure 7 presents the tradeoff between Recall 10@10 and latency in two critical dimensions: (a) query latency and (b) update latency. We vary the search parameters (e.g., efSearch and candidate pool size) to explore a wide recall range and report the corresponding latency of all baselines.

In terms of query latency (Figure 7(a)), LSM-VEC consistently achieves the best recall-latency tradeoff. Specifically, LSM-VEC reaches up to 94.0% recall with a query latency of only 6.2ms. In contrast, DiskANN requires 10.5ms query latency to achieve 92.0% recall, which is over 1.7× higher latency than LSM-VEC for lower accuracy. SPFresh exhibits the lowest recall range (75.0% to 82.0%) and incurs 6.2ms query latency at its best recall, significantly lagging behind both LSM-VEC in accuracy.

For update latency (Figure 7(b)), LSM-VEC again outperforms both baselines. LSM-VEC supports 88.4% recall with only 6.2ms update latency, thanks to its LSM-tree-based design that buffers and batches graph modifications. In comparison, DiskANN's update latency grows to 14.3ms under similar recall, which is 2.3× higher than LSM-VEC. SPFresh

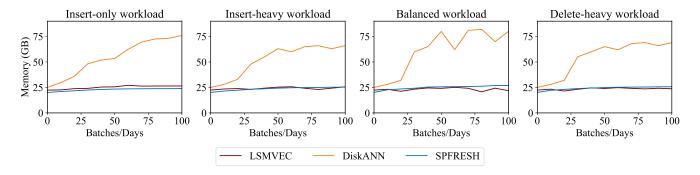
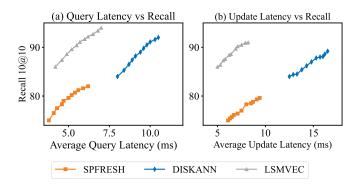


Figure 6. Memory usage over time under different batch workloads.



**Figure 7.** Tradeoff between recall and latency. (a) Search latency vs Recall. (b) Update latency vs Recall.

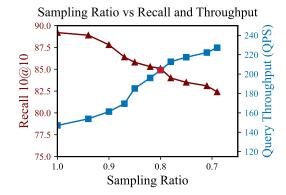
shows better update latency than DiskANN (6.1 ms- 9.4 ms), but suffers from limited recall improvements due to its coarse-grained cluster structure.

Overall, these results highlight that LSM-VEC provides the most efficient and scalable search-update tradeoff among all methods. It achieves higher recall with significantly lower query and update latency, making it well-suited for billionscale dynamic vector search.

Sampling enhances recall while preserving search efficiency. We further evaluate how sampling impacts the trade-off between recall and query latency in LSM-VEC. By reducing the sampling ratio, LSM-VEC selectively skips a portion of candidate neighbor evaluations to minimize computation and disk I/O.

Figure 8 presents the results as the sampling ratio varies from 1.0 (i.e., no sampling applied) to 0.7. As expected, query latency drops significantly as the sampling ratio decreases, from 6.81ms at ratio 1.0 to 4.72ms at ratio 0.7. This latency reduction comes at a modest cost in recall, which decreases from 89.2% to 82.4%.

Notably, LSM-VEC achieves a favorable balance at a sampling ratio of 0.8, highlighted by the red star. At this configuration, it attains 85.1% Recall 10@10 with only 4.90ms average query latency. Compared to full evaluation at ratio



**Figure 8.** Impact of the sampling ratio on recall, red star indicates our configuration.

1.0, this reduces latency by 30% while sacrificing only 4.1% recall.

These results demonstrate that sampling can substantially improve efficiency with minimal impact on accuracy. It is a key component of LSM-VEC, enabling scalable and latency-aware vector search under dynamic workloads.

#### 6 Conclusion

This paper presents LSM-VEC, a scalable disk-based ANN system designed for high-recall, low-latency vector search under dynamic workloads. By integrating hierarchical proximity graph indexing with an LSM-tree storage backend, LSM-VEC supports efficient insertions and deletions while maintaining search quality. The system further incorporates two key techniques: sampling-guided probabilistic traversal and connectivity-aware graph reordering. These techniques reduce disk I/O and enhance recall by aligning graph layout with real query patterns. Through extensive evaluation on billion-scale vector datasets, we demonstrate that LSM-VEC consistently outperforms state-of-the-art disk-based baselines such as DiskANN and SPFresh in terms of recall, latency, update efficiency, and memory usage. Our results show that

LSM-VEC is a practical and robust solution for large-scale, real-time vector retrieval in modern AI applications.

# Acknowledgments

## References

- [1] Amazon. 2025. Opensearch. https://aws.amazon.com/cn/opensearch-service/serverless-vector-database/.
- [2] Kazuo Aoyama, Kazumi Saito, Hiroshi Sawada, and Naonori Ueda. 2011. Fast approximate similarity search based on degree-reduced neighborhood graphs. In Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining. 1055–1063.
- [3] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. 2018. Hd-index: Pushing the scalability-accuracy boundary for approximate knn search in high-dimensional spaces. arXiv preprint arXiv:1804.06829 (2018).
- [4] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.
- [5] Erik Bernhardsson. 2017. Annoy: approximate nearest neighbors in C++/Python optimized for memory usage and loading/saving to disk. GitHub https://github. com/spotify/annoy (2017), 0-5.
- [6] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *International* conference on machine learning. PMLR, 2206–2240.
- [7] Yuan Cao, Heng Qi, Wenrui Zhou, Jien Kato, Keqiu Li, Xiulong Liu, and Jie Gui. 2017. Binary hashing for approximate nearest neighbor search on big data: A survey. IEEE Access 6 (2017), 2039–2054.
- [8] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In Proceedings of the thiry-fourth annual ACM symposium on Theory of computing. 380–388.
- [9] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highlyefficient billion-scale approximate nearest neighborhood search. Advances in Neural Information Processing Systems 34 (2021), 5199–5212.
- [10] SpaceV Contributors. 2021. SPACEV1B: A billion-Scale vector dataset for text descriptors. Webpage. Retrieved March 16 (2021), 2023.
- [11] Scott Cost and Steven Salzberg. 1993. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine learning* 10 (1993), 57–78.
- [12] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In Proceedings of the 10th ACM conference on recommender systems. 191–198.
- [13] Facebook. 2020. Faiss. https://github.com/facebookresearch/faiss. Accessed: 2025-03-28.
- [14] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast approximate nearest neighbor search with the navigating spreadingout graph. arXiv preprint arXiv:1707.00143 (2017).
- [15] Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. 2020. iDEC: indexable distance estimating codes for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 13, 9 (2020).
- [16] 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 *International Conference on Machine Learning*. PMLR, 3887–3896.
- [17] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In International conference on machine learning. PMLR, 3929–3938.
- [18] Shayan Hassantabar, Zeyu Wang, and Niraj K Jha. 2021. SCANN: Synthesis of compact and accurate neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 9 (2021), 3012–3025.

- [19] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 9, 1 (2015), 1–12.
- [20] Masajiro Iwasaki. 2015. Ngt: Neighborhood graph and tree for indexing.
- [21] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. arXiv preprint arXiv:1810.07355 (2018).
- [22] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2023. Atlas: Few-shot learning with retrieval augmented language models. *Journal of Machine Learning* Research 24, 251 (2023), 1–43.
- [23] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. Advances in neural information processing Systems 32 (2019).
- [24] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern* analysis and machine intelligence 33, 1 (2010), 117–128.
- [25] Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, Jyothi Vedurada, et al. 2024. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. arXiv preprint arXiv:2401.11324 (2024).
- [26] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in neural information processing systems 33 (2020), 9459–9474.
- [27] Wuchao Li, Chao Feng, Defu Lian, Yuxin Xie, Haifeng Liu, Yong Ge, and Enhong Chen. 2023. Learning balanced tree indexes for large-scale vector retrieval. In Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. 1353–1362.
- [28] Kejing Lu, Chuan Xiao, and Yoshiharu Ishikawa. 2024. Probabilistic routing for graph-based approximate nearest neighbor search. arXiv preprint arXiv:2402.11354 (2024).
- [29] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68.
- [30] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine* intelligence 42, 4 (2018), 824–836.
- [31] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. Parlayann: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms. In Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. 270–285.
- [32] Yitong Meng, Xinyan Dai, Xiao Yan, James Cheng, Weiwen Liu, Jun Guo, Benben Liao, and Guangyong Chen. 2020. Pmd: An optimal transportation-based user distance for recommender systems. In Advances in Information Retrieval: 42nd European Conference on IR Research, ECIR 2020, Lisbon, Portugal, April 14–17, 2020, Proceedings, Part II 42. Springer, 272–280.
- [33] Dingheng Mo, Junfeng Liu, Fan Wang, and Siqiang Luo. 2025. Aster: Enhancing LSM-structures for Scalable Graph Database. Proceedings of the ACM on Management of Data 3, 1 (2025), 1–26.
- [34] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PmLR, 8748–8763.
- [35] Jie Ren, Minjia Zhang, and Dong Li. 2020. Hm-ann: Efficient billionpoint nearest neighbor search on heterogeneous memory. Advances

- in Neural Information Processing Systems 33 (2020), 10672-10684.
- [36] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In Proceedings of the 10th international conference on World Wide Web. 285–295.
- [37] Chanop Silpa-Anan and Richard Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In 2008 IEEE conference on computer vision and pattern recognition. IEEE, 1–8.
- [38] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search. arXiv preprint arXiv:2105.09613 (2021).
- [39] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Hai Jin, Xuecang Zhang, Junhua Zhu, and Yu Zhang. 2025. Towards High-throughput and Low-latency Billion-scale Vector Search via CPU/GPU Collaborative Filtering and Re-ranking. In 23rd USENIX Conference on File and Storage Technologies (FAST 25). 171–185.
- [40] Noam Touitou and Nissim Halabi. 2023. Approximate Nearest Neighbor Search through Modern Error-Correcting Codes. In The Eleventh International Conference on Learning Representations.
- [41] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In Proceedings of the 2016 International Conference on Management of Data. 1813–1828.
- [42] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. 2023. Spfresh:

- Incremental in-place update for billion-scale vector search. In *Proceedings of the 29th Symposium on Operating Systems Principles.* 545–561.
- [43] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: reducing consistency cost for NVM-based single level systems. In 13th USENIX Conference on File and Storage Technologies (FAST 15). 167–181.
- [44] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, et al. 2023. DF-GAS: a distributed FPGA-as-a-service architecture towards billion-scale graph-based approximate nearest neighbor search. In Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture. 283–296.
- [45] Minjia Zhang and Yuxiong He. 2018. Zoom: Ssd-based vector search for optimizing accuracy, latency and memory. arXiv preprint arXiv:1809.04067 (2018).
- [46] Minjia Zhang, Jie Ren, Zhen Peng, Ruoming Jin, Dong Li, and Bin Ren. 2023. iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures. *IEEE Data Eng. Bull.* (2023).
- [47] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Vector Query Processing for Large Datasets Beyond GPU Memory with Reordered Pipelining. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). 23–40.
- [48] Wenhui Zhou, Chunfeng Yuan, Rong Gu, and Yihua Huang. 2013. Large scale nearest neighbors search based on neighborhood graph. In 2013 International Conference on Advanced Cloud and Big Data. IEEE, 181–186.