A Topology-Aware Localized Update Strategy for Graph-Based ANN Index

Song Yu
Northeastern Univ., China
yusong@stumail.neu.edu.cn

Shengyuan Lin
Northeastern Univ., China
linsy5847@gmail.com

Shufeng Gong
Northeastern Univ., China
gongsf@mail.neu.edu.cn

Yongqing Xie
Huawei Tec. Co., Ltd
xieyongqing1@huawei.com

Ruicheng Liu
Huawei Tec. Co., Ltd
liuruicheng1@huawei.com

Yijie Zhou
Northeastern Univ., China
zhouyijie@stumail.neu.edu.cn

Ji Sun
Huawei Tec. Co., Ltd
sunji11@huawei.com

Yanfeng Zhang
Northeastern Univ., China
zhangyf@mail.neu.edu.cn

Guoliang Li
Tsinghua Univ., China
liguoliang@tsinghua.edu.cn

Ge Yu
Northeastern Univ., China
yuge@mail.neu.edu.cn

ABSTRACT

Graph-based indices are widely used for approximate nearest neighbor search (ANNS). Under dynamic workloads, existing ANNS systems amortize update overhead with large batches, but large batches degrade index quality. We identify two key limitations in existing systems when handling small-batch updates. First, they still scan the entire index to repair the affected graph topology and rebuild the index, causing heavy I/O. Second, their naive repair introduces many edges, repeatedly triggering costly neighbor pruning with expensive distance computations. To address these issues, we propose a topology-aware localized update strategy that exploits the locality of small-batch updates to reduce unnecessary I/O and computation. Specifically, we introduce a lightweight graph topology that quickly identifies affected nodes without full index scans, and a localized update mechanism that restricts modifications to the pages containing these nodes. Moreover, we design a similarity-aware localized connection method that links each affected node to a small set of highly similar neighbors, avoiding redundant edges and costly pruning. Extensive experiments show that our update strategy achieves 2.39-5.96× higher update throughput than the state-of-the-art graph-based streaming ANNS system FreshDiskANN while maintaining high search efficiency and accuracy.

PVLDB Reference Format:
Song Yu, Shengyuan Lin, Shufeng Gong, Yongqing Xie, Ruicheng Liu, Yijie Zhou, Ji Sun, Yanfeng Zhang, Guoliang Li, and Ge Yu. A Topology-Aware Localized Update Strategy for Graph-Based ANN Index. PVLDB, 19(3): 495-508, 2025.
doi:10.14778/3778092.3778108

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/iDC-NEU/Greator.

Proceedings of the VLDB Endowment, Vol. 19, No. 3 ISSN 2150-8097.
doi:10.14778/3778092.3778108

1 INTRODUCTION

Approximate nearest neighbor search (ANNS) for high-dimensional vectors has become a critical component in modern data-driven applications, with widespread usage in information retrieval [6, 41, 49, 65], recommendation systems [14, 39, 42, 50], and large language models (LLMs) [9, 17, 26, 29, 30, 33, 47, 51, 72]. To accelerate ANNS while maintaining high search accuracy, some query indices have been developed. These indices can be categorized into two types: graph-based indices [16, 18, 19, 24, 36, 48, 53, 61] and partition-based (clustering-based) indices [10, 13, 20, 22, 25, 68], where graph-based indices have attracted widespread attention due to their superior performance in search efficiency and accuracy, particularly when dealing with high-dimensional vectors [8, 19, 34, 61, 62]. Consequently, many ANNS systems adopt graph-based indices to enable efficient retrieval [12, 23, 24, 37, 53, 61]. However, for a large number of high-dimensional vectors, graph-based indices tend to be substantially large, often exceeding the memory capacity of commodity PCs [61]. Therefore, existing systems commonly store indices on disk [23, 24, 53, 61], enabling cost-effective scalability.

However, in real-world applications, data is constantly evolving, such as product listings on e-commerce platforms like Alibaba's Taobao and content updates on social media platforms like Twitter and Instagram [35, 63, 68]. To keep up with these changes, ANNS systems must rapidly update their indices to maintain low search latency and high accuracy [68]. This capability is especially critical for AI applications processing streaming data, such as emails and chat histories, where timely insertion of new embeddings ensures that the most relevant and up-to-date content is retrieved for downstream tasks.

Motivation. Although existing graph-based ANNS systems have demonstrated excellent performance in search efficiency, accuracy and scalability, they experience significant performance bottlenecks when handling vector updates with frequent insertions and deletions. To amortize update overhead, these systems adopt batch processing and typically accumulate large batches to improve throughput [3, 53, 59, 63]. However, we find that batch update strategies face a trade-off between update efficiency and search accuracy. We evaluate FreshDiskANN [53], a state-of-the-art batch update ANNS system on disk [68], with different batch sizes on the GIST dataset

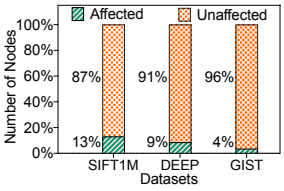


Figure 1: The ratio of affected nodes to unaffected nodes in the index.

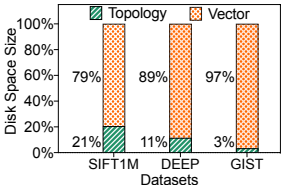


Figure 2: The ratio of disk space size occupied by vectors and graph topology in the index.

(detailed in Table 1). We find that as the batch size increases, the update throughput improves significantly, but the search accuracy (i.e., recall) decreases. Specifically, when the batch size increases from 0.1% to 8% of the dataset, the update throughput improves by 5.58×, while the search accuracy drops from 92.06% to 89.21%. This indicates that although increasing the batch size improves update throughput, it significantly sacrifices the search accuracy. *In light of this, is it possible to improve the update throughput of graph-based indices under small-batch updates without compromising accuracy?*

Our Goal. In this work, we aim to improve the update performance of graph-based indices under small-batch updates while maintaining high search efficiency and accuracy.

Drawbacks of Graph-Based Index Updates. Existing graph-based indices suffer from two main drawbacks under small-batch updates:

(1) Inefficient disk I/O. Existing graph-based indices suffer from substantial unnecessary I/O during updates, mainly due to the following two reasons. First, these index update methods scan the graph topology to identify and repair the affected topology, and then use the updated topology to rebuild a new index [68]. However, in small-batch update scenarios, the proportion of affected nodes is typically very small. For example, as shown in Figure 1, when 0.1% of the nodes are inserted and deleted in three real-world datasets, only 4%-13% of the nodes are actually affected. This implies that existing methods incur a large amount of unnecessary I/O (about 96% I/O in the GIST dataset).

Second, to optimize search performance, existing graph-based indices typically store each node's vector and its outgoing neighbors together, enabling a single I/O operation to retrieve the vector and outgoing neighbors [18, 23, 24, 53, 61]. However, this coupled storage design results in significant I/O waste when scanning the graph topology to locate nodes affected by deletions, as the vector data is not needed. This inefficiency is especially pronounced in high-dimensional vectors, where vector data is considerably larger than the graph topology. As shown in Figure 2, on three real-world datasets—SIFT1M (128 dimensions), DEEP (256 dimensions), and GIST (960 dimensions), the graph topology with 32 neighbors accounts for only 3%-21% of the total index size.

(2) Frequent heavy neighbor pruning. When a node's neighbors are deleted, existing repair methods connect it to all neighbors of deleted nodes to compensate for the lost connectivity and maintain the graph's navigability. However, in small-batch updates, few edges are typically deleted from a node, but the repair methods always add many new edges, greatly increasing the number of neighbors and degrading search performance. To mitigate this issue, existing methods prune neighbors to keep their number below a predefined threshold [53], e.g., 32. However, the neighbor pruning requires computing similarities (distances) between all pairs of neighbors, which imposes high computational overhead and severely degrades update performance, especially in high-dimensional spaces.

Insights and Solution. To overcome the drawbacks of index with small-batch updates, we design a topology-aware localized update strategy for graph-based ANNS indices on high-dimensional streaming vectors, which is designed based on two insights. One key insight in exploiting the localized nature of small-batch updates, only a small subset of nodes are actually affected, allowing us to perform targeted modifications to the index file, thereby significantly reducing unnecessary I/O. Another key insight is the locality of new connections, existing methods connect affected nodes to all outgoing neighbors of the deleted node, but most of these connections are eventually discarded after the neighbor pruning, e.g., about 84% new connections are discarded on the GIST dataset, which inspired us to connect affected nodes to only a few carefully selected outgoing neighbors of the deleted node.

Based on the above insights, we propose two key designs to update graph-based high-dimensional vector indices efficiently and locally. (1) **Topology-Aware Affected Node Identification and Access** (*localized I/O*). We first propose a topology-aware ANN index that includes a query index and a lightweight graph topology. The query index is used for efficient ANNS, while the lightweight graph topology enables fast identification of affected nodes without accessing vector data. After that, the affected vertices are read locally, instead of scanning the entire index file as existing methods [3, 53]. Then their neighbors are updated. (2) **Similarity-Aware Localized Connection** (*localized computation*). We propose a similarity-aware localized connection method that repairs only the affected region. For each affected node, it links the node to a small set of highly similar outgoing neighbors within the deleted node's local area. This preserves navigability toward the deleted region while minimizing redundant edges and avoiding the costly neighbor pruning typically required to enforce degree limits.

In summary, we make the following contributions.

- We propose a topology-aware index that efficiently identifies the affected nodes using a lightweight graph topology without accessing unnecessary vector data (Section 4.1).
- We provide a localized update method that restricts index modifications to only the pages containing affected nodes, thereby minimizing I/O overhead (Section 4.2).
- We propose a similarity-aware localized connection method that reduces redundant connections, avoiding frequent costly neighbor pruning (Section 5).
- We implement a graph-based streaming vector ANNS system, `Greator`, that incorporates a topology-aware localized update strategy (Section 6), and conduct a comprehensive evaluation on eight real-world datasets (Section 7). Results show that `Greator` achieves 2.39-5.96× higher update throughput than the state-of-the-art graph-based streaming ANNS system FreshDiskANN while maintaining high search efficiency and accuracy.

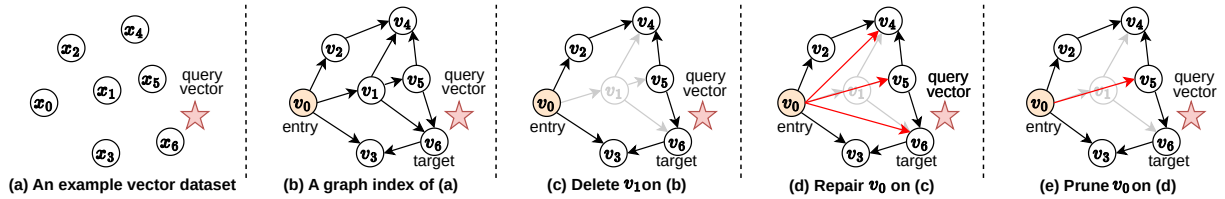


Figure 3: An example of a graph-based vector index and its update process. Each node has a maximum neighbor limit of $R = 3$. The red star represents the query vector, v_0 represents the entry of the index, and v_6 (x_6) represents the nearest target node (vector) for the query vector.

2 BACKGROUND

This section first introduces the background of graph-based approximate nearest neighbor search (ANNS), followed by a detailed explanation of the update methods for graph-based indices on disk.

2.1 Graph-Based ANN Index

Let $X = \{x_0, x_1, ..., x_{n-1}\}$ denote a vector dataset of n vectors, where each vector $x_i \in \mathbb{R}^d$ represents a d-dimensional vector. The distance between two vectors, $x_p \in \mathbb{R}^d$ and $x_q \in \mathbb{R}^d$, is represented as $dist(x_p, x_q)$. The distance function can be the Euclidean distance, cosine similarity, or other metrics.

Approximate Nearest Neighbor Search (ANNS). Given a vector dataset X and a query vector $x_q \in \mathbb{R}^d$, the goal of approximate nearest neighbor search (ANNS) is to retrieve a set R_{knn} of K vectors from X that are closest to x_q, note that the retrieved results are not guaranteed to be optimal. Typically, the accuracy of the search result R_{knn} is evaluated using the *recall*, defined as $RecallK@K = \frac{|R_{knn} \cap R_{exact}|}{K}$, where R_{exact} is the ground-truth set of the K closest vectors to x_q from X. The goal of ANNS is always to maximize the recall while retrieving results as quickly as possible, leading to a trade-off between recall and latency.

Graph-Based Vector Index. Given a vector dataset X and a graph-based index $G = (V, E)$ of X, where V is the set of nodes with $|V| = |X|$. Each node in V corresponds to a vector in X. Specifically, x_p denotes the vector data associated with node p, and $dist(p, q)$ represents the distance between two nodes p and q. E is the set of edges constructed by a specific index algorithm based on vector similarity. $N_{out}(p)$ and $N_{in}(p)$ denote the outgoing neighbor set and incoming neighbor set of p, defined as $N_{out}(p) = \{v | edge(p, v) \in E\}$ and $N_{in}(p) = \{v | edge(v, p) \in E\}$. Figure 3a illustrates an example vector dataset X, while Figure 3b presents its graph-based index.

Existing graph indices typically maintain only unidirectional graphs with outgoing neighbors, especially in disk-based implementations [68]. This is because, in graph-based indices [16, 23, 24, 36, 61], the number of outgoing neighbors for each node is relatively uniform and bounded by a predefined threshold R, allowing fixed space allocation for outgoing neighbors and independent updates. In contrast, the number of incoming neighbors varies significantly, and allocating fixed space for them leads to storage waste. Alternatively, compact storage requires adjusting the storage locations of subsequent nodes as the number of neighbors increases, incurring unacceptable overhead [31, 74, 79]. Moreover, maintaining consistency in bidirectional graphs during concurrent updates can also degrade index update performance.

ANNS with Graph-Based Index. Graph-based ANNS typically employs the beam search algorithm [24], a variant of best-first search. Specifically, the algorithm initializes a priority queue with a predefined entry, which is either randomly selected or determined heuristically. In each iteration, it greedily extracts the top-W closest nodes to the query vector from the queue, explores their neighbors, and inserts qualified candidates back into the queue. This process repeats until a stopping criterion is met, such as reaching a predefined search limit L. By adjusting L, beam search achieves a tunable trade-off between search accuracy and efficiency.

2.2 Disk-Based Graph Index Updates

As introduced in Section 1, modern applications increasingly require dynamic updates (insertion/deletion) and queries for vector data [40, 53, 59, 63, 68]. In these applications, the index needs to promptly respond to vector updates to ensure search accuracy and efficiency. For example, as shown in Figure 3c, when node v_1 (corresponding to vector x_1) is deleted, the existing index structure fails to return the target search result (i.e., node v_6) because there is no path from the entry (i.e., v_0) to reach it. Therefore, the index structure needs to be updated accordingly. Figure 3d illustrates a typical repair process used in existing methods [53], where the incoming neighbor v_0 of the deleted node v_1 is directly connected to v_1's outgoing neighbor set $\{v_4, v_5, v_6\}$. Subsequently, the neighbors of v_0 are pruned to satisfy a predefined limit R (set to $R = 3$ in the figure). The final updated index is shown in Figure 3e, where the modified index structure allows ANNS to efficiently and accurately retrieve the target node v_6.

However, efficiently updating indices while maintaining search accuracy remains a significant challenge. As shown in Figure 3d, when a node is deleted, its incoming neighbors must be repaired to maintain the navigability of the index. As discussed in Section 2.1, existing graph-based indices typically adopt a directed topology that stores only outgoing neighbors. Consequently, to identify and repair all affected nodes (i.e., incoming neighbors of deleted nodes), existing methods must scan the entire index. Furthermore, since graph-based indices in existing systems are often stored on disk to support large-scale vector data, the update overhead is further amplified by disk I/O.

Existing methods typically adopt a batch update mechanism [3, 53, 59, 63]. Next, we describe this procedure, which consists of the following three phases.

Deletion Phase. This phase repairs the neighbors of all nodes affected by each deleted node p (i.e., the incoming neighbors of p). However, since graph-based indices do not store the incoming neighbors of each node, they cannot directly access $N_{in}(p)$. As a result, nodes

Algorithm 1: Delete(L_D, R, α) [53]

Input: Graph $G(V, E)$ with $|V| = n$, set of nodes to be deleted L_D
Output: Graph on nodes V' where $V' = V \setminus L_D$

```
begin
    foreach p ∈ V \ L_D s.t. N_out(p) ∩ L_D ≠ ∅ do
        D ← N_out(p) ∩ L_D;
        C ← N_out(p) \ D ;      // initialize candidate set
        foreach v ∈ D do
            C ← C ∪ (N_out(v) \ L_D);
        N_out(p) ← RobustPrune(p, C, α, R);
```

and their neighbor lists must be loaded block by block from disk. For each block, Algorithm 1 (following the procedure in [24]) is executed on all resident nodes, and the modified blocks are written to a temporary intermediate index file on disk.

As shown in Algorithm 1, for each affected node p, a new candidate neighbor set C is constructed. All non-deleted nodes in $N_{\text{out}}(p)$ are first added to C. In addition, for each deleted node $v \in N_{out}(p)$, all non-deleted nodes in $N_{out}(v)$ are also added in C. Next, a costly neighbor pruning (i.e., `RobustPrune` algorithm [53]) is applied to C to enforce the neighbor size constraint. The pruned result is then assigned as the new neighbor set of p. It is worth noting that `RobustPrune` [53] involves numerous expensive pairwise distance computations between vectors, with a computational complexity of $O(|C|^2)$.

Insertion Phase. For each inserted node p, a greedy search (i.e., *GreedySearch* [53]) is performed on the graph, which requires random reads from the index file. Based on the search results, the neighbors $N_{out}(p)$ of p are constructed. Furthermore, for all outgoing neighbors of p (i.e., $N_{out}(p)$), a reverse edge to p needs to be added, i.e., $\{edge(p', p) | p' \in N_{out}(p)\}$. However, immediately inserting the vector, its neighbors, and corresponding reverse edges of p causes random writes. To avoid this, all updates are first stored in a temporary in-memory structure Δ, which merges updates targeting the same disk page, processing in the next phase.

Patch Phase. This phase applies the updates stored in Δ from the insertion phase to the temporary index file and generates a new index file. Specifically, it sequentially reads each node p from the temporary index file on the SSD in blocks, adds the edges of node p from Δ to $N_{out}(p)$, and checks whether the new degree $|N_{out}(p) \cup \Delta(p)|$ exceeds a given threshold (e.g., R). If the threshold is exceeded, neighbor pruning is triggered using the `RobustPrune` [53]. Finally, all updated blocks are written to the new index file on the SSD.

In summary, existing batch update methods scan the index file twice to identify and repair affected nodes [3, 53] during the deletion and patch phases, and generate new index files, leading to significant I/O overhead. Furthermore, during neighbor updates, existing repair algorithms (e.g., Algorithm 1) introduce excessive neighbors, frequently triggering neighbor pruning and further increasing computational overhead.

3 OVERVIEW

We present a topology-aware localized update strategy that enables fast updates of graph-based index structures in small-batch update

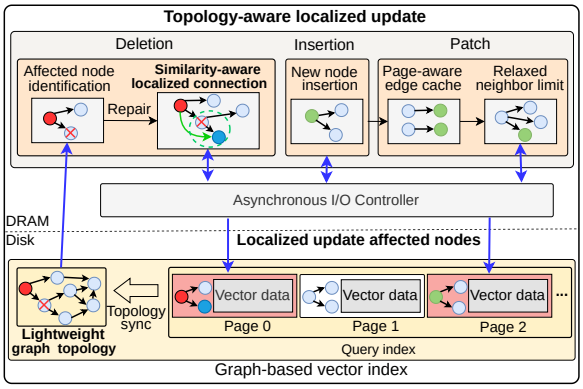


Figure 4: The update workflow of the topology-aware localized update strategy. The crossed-out circles represent deleted nodes, the red circles indicate nodes affected by the deletion, and the green circles denote newly inserted nodes. The red pages (e.g., page 0 and page 2) represent the affected pages that need to be updated.

scenarios. Our strategy primarily enhances update performance through the following core designs. To reduce I/O overhead, (1) we introduce a topology-aware ANN index, which includes a query index and a lightweight graph topology. The query index supports fast ANNS, while the **lightweight graph topology** efficiently identifies affected nodes, avoiding unnecessary I/O caused by scanning the vector data in the index; (2) we design a **localized update method**, updating the index locally based only on the affected nodes, thereby avoiding modifications to the entire index file. To reduce computational overhead, (3) we propose a **similarity-aware localized connection method**, which selectively connects affected nodes to a small number of the most similar outgoing neighbors within the local region of the deleted node. The method adaptively determines the number of connections based on the current neighbor count and ensures the neighbor limit is not exceeded, thereby reducing frequent and heavy neighbor pruning.

Update Workflow. After accumulating a small batch of updates, our strategy efficiently handles the batch and merges the updates to the graph-based vector index file on disk. As shown in Figure 4, this update workflow can be divided into the following three stages. 1) ***Deletion Phase.*** This phase begins by processing the deletion vectors. It first scans a lightweight graph topology significantly smaller than the query index file to quickly locate the incoming neighbors affected by each deleted node. Next, to reduce unnecessary I/O, only the affected pages (e.g., Page 0 and Page 2 in Figure 4) in the query index are loaded. It then repairs the neighbors of these nodes using the newly designed similarity-aware localized connection method, which significantly reduces the frequency of costly neighbor pruning. Finally, the updated pages are written back to the query index file. 2) ***Insertion Phase.*** This phase processes the inserted vectors. Each vector is treated as a new node and undergoes an approximate search on the query index to determine its outgoing neighbors. The newly inserted nodes, along with their neighbor set and vector data, are then written to the query index file. Meanwhile, the reverse edges corresponding to the outgoing neighbors of these nodes are cached in a *page-aware structure* ΔG

for deferred processing. 3) ***Patch Phase***. Based on the information in ΔG, this phase merges reverse edges targeting the same node. It then loads only the pages corresponding to the affected nodes, updates their neighbor set using a *relaxed neighbor limit*, and writes the updated pages back to the index file.

4 TOPOLOGY-AWARE ANN INDEX WITH LOCALIZED UPDATE

In this section, we propose a novel index structure that supports localized updates with minimized disk I/O.

4.1 A Topology-Aware ANN Index

As discussed in Section 2.2, existing methods face a fundamental dilemma when updating graph-based indices: they require two full index scans to locate and repair affected nodes because their designs prioritize search accuracy and efficiency while overlooking update performance. To optimize both, we propose a new index structure with two components: a query index for fast vector search and a lightweight graph topology for efficient updates, as detailed below.

Query Index. To meet the efficiency and accuracy requirements of vector search, we adopt the state-of-the-art disk-based graph index methods (i.e., DiskANN [24]) to construct a static query index. In the on-disk storage layout, each node's outgoing neighbors and vector data are stored together in a compact format, as shown in Figure 4. This co-location design enables a single I/O operation to retrieve both neighbor information and vector data during search, thereby reducing I/O overhead.

Although the query index satisfies the efficiency and accuracy requirements for vector search like DiskANN, it still faces the same update limitation as existing methods: it cannot efficiently identify the nodes affected by deletions, as incoming neighbors are not explicitly stored. As discussed in Section 1, existing methods identify affected nodes by scanning the entire index file. However, since the index file contains a large amount of vector data, such scans incur significant unnecessary I/O overhead. To address this issue, we introduce a lightweight graph topology, maintained separately from the query index, which serves as an auxiliary structure to efficiently identify affected nodes during updates.

Lightweight Graph Topology. We design a lightweight graph topology that stores only the outgoing neighbors of each node, as shown in Figure 4. Its content is consistent with the neighbor information in the query index. By eliminating vector data, the size of the lightweight topology is typically significantly smaller than that of the query index file. As shown in Section 1, on the GIST dataset, the graph topology accounts for only 3% of the total index size. Based on this design, the update process scans only the lightweight graph topology to efficiently identify affected nodes, avoiding the unnecessary I/O caused by scanning vector data in existing methods.

Notably, we adopt an independent lightweight graph topology alongside the query index, rather than fully decoupling and storing the topology and vector data separately. The main consideration is that fully decoupling and storing both on disk would degrade query performance, since queries require simultaneous access to topology and vectors, leading to additional I/O overhead. If the decoupled topology is stored in memory, update performance can be improved, but this approach is difficult to scale to large datasets.

Lazy Synchronization for Index Consistency. Since both the query index and the lightweight graph topology store the graph's structural information, it is necessary to maintain consistency between them. Note that the lightweight graph topology is used solely to identify affected nodes during updates and is not involved in the search process during query. To this end, we adopt a lazy synchronization strategy. Specifically, the strategy first repairs each affected node in the query index to quickly restore query availability. Then, when a node is updated in the query index, we avoid blocking to maintain consistency in the lightweight topology; instead, the corresponding structural update of the node is delegated to a background thread for asynchronous synchronization with the lightweight graph topology. The only requirement is that, before the next update begins, all modifications in the query index must have been synchronized to the lightweight topology to ensure correctness in identifying affected nodes. It is worth noting that updating the query index typically dominates the overall cost of ANN index maintenance, as it involves both large-scale vector data and expensive structural repairs with extensive computation. In contrast, the lightweight topology is much smaller and only requires simple data copying without additional computation, making its overhead negligible compared to the total update time.

4.2 Page-Level Localized Updates

As described in Section 2.2, existing systems complete index updates by sequentially scanning the entire index file and reconstructing the full index. This approach is efficient in large-batch update scenarios, as most of the sequentially loaded data requires modification, and the sequential scanning utilizes the high bandwidth of the disk. However, in small-batch update scenarios, the data loaded contains a large number of unaffected nodes. As shown in Figure 1, existing systems read a large number of unaffected nodes, leading to significant unnecessary I/O overhead.

Page-Level Localized Updates. To eliminate unnecessary I/O overhead in existing methods, we propose a fine-grained page-level localized update strategy, updating only the pages that contain affected nodes. This strategy follows a three-phase batch update workflow: deletion, insertion, and patch.

Deletion. During the deletion phase, two steps are performed for each deleted node. First, the node is removed from the index; second, the structures of the affected nodes (i.e., its incoming neighbors) are repaired in the existing index file, avoiding the overhead of regenerating the index file in existing systems. Specifically, for each deleted node p, our method removes p only from the mapping *Local_Map*, which records the association between node IDs and their locations in the index file. The freed location is then added to a recycling queue $Free_Q$. Next, the lightweight graph topology is used to efficiently identify the set of affected nodes (i.e., nodes whose neighbor set includes deleted nodes), denoted as $V_{affected}$. For each affected node $q \in V_{affected}$, our method first locates its corresponding page in the index file using *Local_Map*, and then loads the page via an asynchronous I/O interface. Once loaded, a lightweight neighbor repair algorithm (SALC, detailed in Section 5.1)

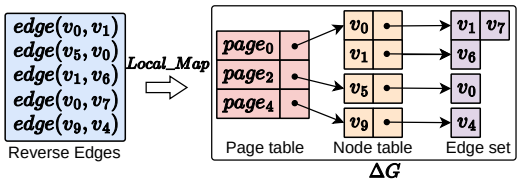


Figure 5: An example of storing reverse edges in ΔG.

updates the outgoing neighbor set $N_{out}(q)$. Finally, the modified page is written back to the index file.

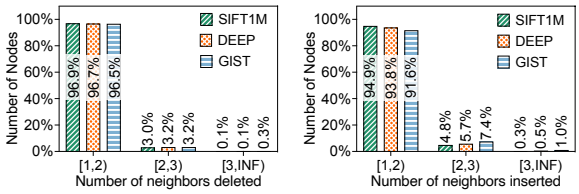
Insertion. For each inserted node p, our method constructs its outgoing neighbor set $N_{out}(p)$ using the same method as existing systems (detailed in Section 2.2). A free storage location is then retrieved from the recycling queue $Free_Q$; if $Free_Q$ is empty, a new location is allocated at the end of the file. As in the deletion phase, our method employs an asynchronous I/O interface to write the relevant data of p, including its vector x_p and neighbor set $N_{\text{out}}(p)$, to the designated position.

Page-Aware Cache Structure ΔG. Similar to existing methods, our method adds reverse edges $\{edge(p', p) \mid p' \in N_{out}(p)\}$ for each inserted node p to maintain good index quality. To avoid the excessive random I/O overhead incurred by writing reverse edges immediately upon insertion, we cache them in a novel page-aware cache structure, ΔG, as illustrated in Figure 5. By organizing reverse edges in pages, ΔG reduces redundant updates to the same page. For each reverse edge, the page ID of the source node in the index file is first resolved using *Local_Map* (e.g., resolving $edge(v_0, v_1)$ to $page_0$). Then, the corresponding node table is looked up in the page table of ΔG; if it does not exist, a new one is created. Within the node table, the edge set for the source node ID is retrieved (or created if absent), and the target node ID is appended to the set.

Patch. To merge the updates in ΔG into the query index file, our method first identifies the pages that need to be updated based on ΔG's page table (e.g., $page_0$ in Figure 5). After loading the page (e.g., $page_0$) from the query index file, it sequentially processes the node table associated with $page_0$ in ΔG. Taking node v_0 as an example, we retrieve its pending reverse edge set $\{v_1, v_7\}$ and merge them into the original neighbor set $N_{out}(v_0)$ in $page_0$, producing the updated set $N'_{out}(v_0)$. If $|N'_{out}(v_0)| > R$, a pruning algorithm is applied to maintain the neighbor count within the limit R (as detailed in Section 5). After processing v_0, the subsequent nodes in the node table (e.g., v_1) are traversed. Finally, once all node updates in $page_0$ are completed, the updated page is written back to the query index file via an asynchronous I/O interface.

4.3 I/O Overhead Analysis

We compare the I/O overhead during batch updates between our method and existing methods introduced in Section 2.2. For the deletion and patch phases, existing methods perform two full index scans, leading to a total I/O cost of $O(2(|X| + |G|))$, where $|X|$ is the size of the vector dataset. In contrast, our method identifies the affected nodes by scanning the separately stored lightweight graph topology file, thus reducing the cost to $O(|G|)$. Additionally, our method employs a page-level localized update mechanism, which only requires a small amount of random I/O operations on the affected node set $V_{affected}$ and the incremental graph ΔG, with a cost of $O(|V_{affected}| + |\Delta G|)$. Therefore, the total cost is



(a) Delete phase **(b) Patch phase**

Figure 6: Distribution of the number of deleted and inserted neighbors.

approximately $O(|G| + |V_{affected}| + |\Delta G|)$. In small-batch update scenarios where $|V_{affected}| \ll |X|$ and $|\Delta G| \ll |G|$, this design significantly reduces unnecessary I/O operations. For the insertion phase, both systems perform similar graph search-related random I/O operations. Overall, our method reduces a significant amount of I/O overhead in small-batch updates through decoupled lightweight graph topology and page-level localized update mechanisms. This I/O efficiency is validated by the experimental results in Figure 9 (Section 7.2).

5 LIGHTWEIGHT INCREMENTAL GRAPH REPAIR

To reduce the expensive vector distance computations caused by neighbor pruning during updates, we introduce a similarity-aware localized connection method that mitigates pruning triggered by both deletions and insertions.

5.1 Similarity-Aware Localized Connection

When a node's neighbors are deleted, existing methods connect the affected node to all outgoing neighbors of the deleted nodes to preserve navigability [53], enabling incremental maintenance without rebuilding. However, this naive repair inflates the degree and frequently triggers expensive pruning.

Example 1: Figure 7(a) illustrates a partial graph structure of a graph-based index, where edge lengths represent vector distances and each node is limited to at most $R = 3$ neighbors. Figure 7(b) demonstrates the naive neighbor repair process (e.g., Algorithm 1) applied to node v_0 after the deletion of its neighbor v_1. Specifically, the outgoing neighbors of v_1, $N_{out}(v_1) = \{v_4, v_5, v_6\}$, are added as candidates to the remaining neighbors of v_0. This expands v_0's candidate set to five nodes, exceeding the limit R, and thus triggers a costly pruning step. As a result, v_0 retains $\{v_2, v_3, v_5\}$ as its updated neighbors.

Observation. Based on the experiments described in Section 1 under small-batch update scenarios, we analyze the number of neighbors deleted per affected node on three real-world datasets. As shown in Figure 6a, 96% of affected nodes lose only a single neighbor. However, as illustrated in Figure 7b, even when only one neighbor is deleted, the naive neighbor repair approach (e.g., Algorithm 1) adds all outgoing neighbors of the deleted node to the candidate set. This often causes the candidate set size to exceed the neighbor limit, thereby triggering costly neighbor pruning.

Intuition. Based on the above observation, in small-batch update scenarios, an affected node p typically loses only a few neighbors due to deletion, so it can still maintain good connectivity in the graph through the remaining neighbors. To leverage this property,

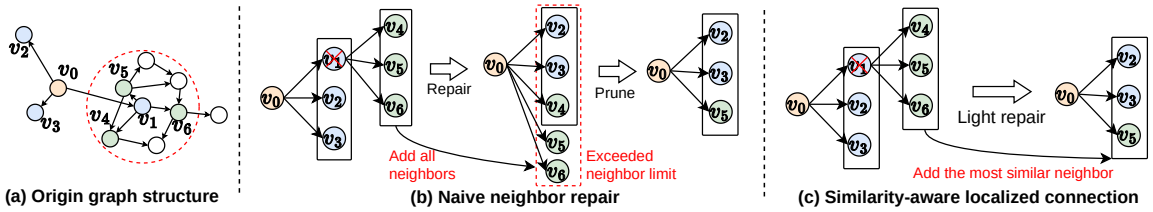


Figure 7: Illustrate the deleted neighbor repair process for naive neighbor repair and our method, respectively. (a) represents the partial graph structure of an index, where the red circle highlights the local region around v_1, while (b) and (c) depict the process of repairing the neighbors of v_0 after v_1 is deleted. The black rectangle indicates the neighbor limit $R = 3$.

Algorithm 2: SALC

Input: Graph $G(V, E)$ with $|V| = n$, set of nodes to be deleted L_D, affected node set $V_{affected}$ by the deleted nodes, out degree bound R, deleted-neighbor threshold T
Output: Graph on nodes V' where $V' = V \setminus L_D$

1 **foreach** $p \in V_{affected}$ **do**
2 $\quad \mathcal{D} \leftarrow N_{out}(p) \cap L_D$; // deleted neighbor set
3 $\quad C \leftarrow N_{out}(p) \setminus \mathcal{D}$; // candidate set
4 $\quad$ **if** $|\mathcal{D}| < T$ **then**
5 $\quad\quad slot \leftarrow R - |C|$; // available neighbor slots
6 $\quad\quad k_{slot} \leftarrow max(\left\lfloor \left(\frac{slot}{|N_{out}(p)|} \right) \right\rfloor, 1)$; // number of compensations
7 $\quad\quad$ **foreach** $v \in \mathcal{D}$ **do**
8 $\quad\quad\quad S \leftarrow N_{out}(v) \setminus L_D$;
9 $\quad\quad\quad$ Sort S by distance to v in ascending order;
10 $\quad\quad\quad C \leftarrow C \cup$ Top-k_{slot} nodes in S;
11 $\quad\quad N_{out}(p) \leftarrow C$;
12 $\quad$ **else**
13 $\quad\quad$ **foreach** $v \in \mathcal{D}$ **do**
14 $\quad\quad\quad C \leftarrow C \cup (N_{out}(v) \setminus L_D)$;
15 $\quad\quad$ **if** $|C| > R$ **then**
16 $\quad\quad\quad N_{out}(p) \leftarrow Prune(C, R)$;
17 $\quad\quad$ **else**
18 $\quad\quad\quad N_{out}(p) \leftarrow C$;

a lightweight incremental update algorithm should maximize the reuse of existing neighbor information and apply only localized compensation for the lost connections. Specifically, when a neighbor u of node p is deleted, p loses access only to the local region where u resides. Given that the index is typically constructed based on nearest-neighbor relationships, i.e., similar nodes tend to connect, u's neighbors usually include others from the same region where u resides. By exploiting this locality, we can propose a lightweight repair strategy that selects a small subset of u's most similar neighbors and reconnects p to them, thereby restoring p's access to the local region previously reached through u. Moreover, to avoid triggering costly pruning when the number of neighbors exceeds the threshold R, the number of added connections can be adaptively determined based on p's residual neighbor capacity.

Similarity-Aware Localized Connection. Based on the above intuition, we design a similarity-aware localized connection method, SALC, to perform a more lightweight repair of affected nodes by avoiding triggering costly neighbor pruning. The core idea of SALC is that when only a few of a node's neighbors are deleted, it compensates for this node by selectively adding a few nodes that are most similar to the deleted neighbors, instead of connecting to all outgoing neighbors of the deleted node. The detailed procedure of SALC is shown in Algorithm 2.

Specifically, for each affected node $p \in V_{affected}$, SALC first extracts its deleted outgoing neighbors $\mathcal{D} \leftarrow N_{out}(p) \cap L_D$ (line 2) and places the remaining neighbors into a candidate set $C \leftarrow N_{out}(p) \setminus \mathcal{D}$ (line 3). SALC then determines the repair strategy based on the number of deleted neighbors: if $|\mathcal{D}| < T$, where T is a predefined threshold on the number of deleted outgoing neighbors (default set to 2), a lightweight repair is performed (lines 4–11); otherwise, a heavy repair is executed, similar to the algorithm 1 in FreshDiskANN (lines 12–18). In the lightweight repair step, SALC first calculates the maximum number of compensatory neighbors allowed for node p as $slot \leftarrow R - |C|$, where R is the maximum number of outgoing neighbors for each node. Then, it calculates the number of compensating neighbors k_{slot} for each deleted neighbor (line 6). By treating each original neighbor of node p equally, the available number of compensable neighbors is evenly distributed among them, following the formula:

$$k_{slot} \leftarrow \max\left(\left\lfloor \frac{slot}{|N_{out}(p)|} \right\rfloor, 1\right).$$

Due to $k_{slot} \geq 1$, this ensures that each deleted neighbor receives at least one compensating edge. Moreover, when p has fewer original neighbors (i.e., $|N_{out}(p)|$) or more deleted neighbors, the available $slot$ becomes larger. This indicates that the deleted edges are more critical, thereby warranting stronger compensation. For each deleted neighbor $v \in \mathcal{D}$, SALC retrieves its surviving neighbors $S \leftarrow N_{out}(v) \setminus L_D$, sorts them by distance to v, and selects the top k_{slot} nodes from S to be added into C as compensations of deleted neighbors (lines 7-10). Since SALC selects the nearest neighbors, we can store neighbor distances in the topology to reduce the overhead of repeat distance computation (see Section 6 for details). Once all deleted neighbors in $\mathcal{D}$ have been compensated, the set C contains the candidate neighbors of p. Since the total number of neighbors to compensate for all deleted nodes is $|\mathcal{D}| \cdot k_{slot}$ and $|\mathcal{D}| \leq |N_{out}(p)|$, it follows that

$$|\mathcal{D}| \cdot k_{slot} \leq |N_{out}(p)| \cdot \frac{slot}{|N_{out}(p)|} \leq slot,$$

which ensures that the total number of newly added neighbors never exceeds the available slots, thereby guaranteeing that C does not surpass the maximum limit R. Finally, the neighbor set of p is updated as $N_{out}(p) \leftarrow C$ (line 11), completing the structural repair of node p in the index.

Example 2: Figure 7c illustrates the process of SALC repairing the neighbors of v_0 after v_1 is deleted. First, SALC collects the remaining neighbor set of v_0, denoted as $C = \{v_2, v_3\}$. It then examines $N_{out}(v_1)$ and computes $k_{slot} = 1$, selecting v_5 as the most similar non-deleted node to v_1. Finally, v_5 is added to C, resulting in the new neighbor set $N_{out}(v_0)$. Since $|N_{out}(v_0)| \leq R$, this approach avoids triggering costly neighbor pruning in contrast to Figure 7b.

As introduced in Section 4.2, the patch phase inserts cached reverse edges from ΔG into node neighbor sets. These additions often push degrees past R, triggering costly pruning. Similar to deletion analysis in Figure 6a, we count the number of reverse edges added when pruning is triggered. Figure 6b shows that 90% of events are caused by a single added edge. This is because existing batch update methods (e.g., FreshDiskANN) strictly enforce R: once a node reaches the limit, adding one edge violates the constraint and immediately triggers pruning.

Relaxed Neighbor Limit. To reduce expensive neighbor pruning costly pruning from a few reverse edges, we adopt a relaxed neighbor limit. Specifically, each node's neighbors are constrained by two parameters: a strict neighbor limit R and a relaxed neighbor limit R', where $R \leq R'$. When storing the neighbors of each node on disk, R' slots are allocated, with $R' - R$ reserved as additional space. During pruning, the strict limit is enforced to ensure that the number of neighbors does not exceed R. When adding reverse edges to a node, the relaxed limit is applied, allowing up to R' neighbors before pruning is triggered. A larger gap between R' and R lowers the pruning frequency but increases the number of neighbors, potentially raising search overhead. To balance this trade-off, we set R' based on Figure 6b, where 90% of pruning is triggered by adding a single edge. Thus, we default R' to $R + 1$, which significantly reduces pruning with negligible impact on search efficiency.

5.2 Computation Overhead Analysis

We briefly analyze the computational overhead of our method compared to the existing methods described in Section 2.2. During the deletion phase, existing methods use Algorithm 1 to repair a node when its neighbors include deleted nodes, triggering pruning with a complexity of $O(|C|^2)$, where $|C| \approx |\mathcal{D}| \cdot R + R$ and $\mathcal{D}$ is the set of deleted neighbors. In contrast, for a node with only a few deleted neighbors (as shown in Figure 6, accounting for 90% of cases), we use a similarity-aware localized connection method (i.e., Algorithm 2), ensuring that the number of neighbors after repair does not exceed R, thus avoiding pruning. The complexity of this method arises from finding similar neighbors, i.e., calculating the distance between each deleted node and its neighbors, which has a complexity of $O(|\mathcal{D}| \cdot R)$. By avoiding neighbor pruning, our method significantly reduces computational overhead. During the insertion phase, our method has the same overhead as existing methods. In the patch phase, although reverse edges are added, we allow the neighbor count to temporarily exceed R but remain within a relaxed bound R'. This relaxed constraint lowers pruning frequency compared to existing methods that strictly enforce R. Overall, our method reduces computational overhead by lowering the number of pruning operations in both the deletion and patch phases. The experimental results in Figure 10 of Section 7.2 further confirm that our method triggers pruning with significantly lower probability.

Table 1: Dataset description. T and D denote the dimensionality and data type of vectors.

Dataset	T	D	# Vector	# Query	Contents
SIFT1M	float	128	1,000,000	10,000	Image
Text2Img	float	200	1,000,000	1,000	Image & Text
DEEP	float	256	1,000,000	1,000	Image
Glove	float	300	1,000,000	1,000	Text
MSONG	float	420	994,185	1,000	Audio
GIST	float	960	1,000,000	1,000	Image
MSMARCO	float	1024	1,000,000	1,000	Text
MSMARCO10M	float	1024	10,000,000	1,000	Text
SIFT1B	uint8	128	1,000,000,000	1,000	Image

6 IMPLEMENTATION

We implement an ANNS system, Greator, based on our topology-aware index and built on top of FreshDiskANN [53]. Since optimization techniques of the strategy are independent of specific graph construction methods, they can be broadly applied to any other on-disk graph-based indices. In addition, Greator provides the following mechanisms to ensure thread safety during concurrent updates and further improve update performance.

Page-Level Concurrency Control. Greator supports concurrent vector search and vector updates, making it essential to ensure the correctness of these operations. In fine-grained update mode, each update touches only a few small pages and finishes quickly, so we adopt page-level concurrency control with read–write locks to protect page accesses. It is worth noting that, similar to existing systems (e.g., FreshDiskANN), Greator does not provide transaction support. If transactions were to be introduced, pages of all nodes involved in a batch update would need to be locked and updated simultaneously, which would significantly increase query latency. In such cases, adopting MVCC would be a more appropriate choice.

Storing Neighbor Vector Distances with the Topology. In the similarity-aware localized connection method, repairing a deleted node often requires selecting its closest neighbors, which typically involves computing distances between the deleted node and all of its neighbors. To reduce this overhead, we record these distances during index construction and store them in the topology, enabling direct reuse during updates and avoiding costly high-dimensional distance computations. This feature has been implemented in Greator as an optional optimization for users who wish to trade additional storage space for improved update performance.

7 EVALUATION

7.1 Evaluation Setup.

Evaluation Platform. All experiments are conducted on a server equipped with an Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz, 320GB DDR4 memory (@3200MT/s), 48 cores, and two 1.7TB SSDs with a sequential access bandwidth of up to 500MB/s. The operating system is Ubuntu 22.04 LTS, and the compiler used is GCC 11.4.0.

Datasets. We evaluate our system using eight public real-world datasets, with detailed statistics provided in Table 1, including SIFT1M [5], Text2Img [52], DEEP [70], Glove [70], MSONG [70], GIST [5], MSMARCO [70], MSMARCO10M [70] and SIFT1B [5]. They cover diverse dimensions, sizes, and content types, and have been widely adopted in prior studies [16, 24, 36, 53, 59, 68] for benchmark ANNS systems.

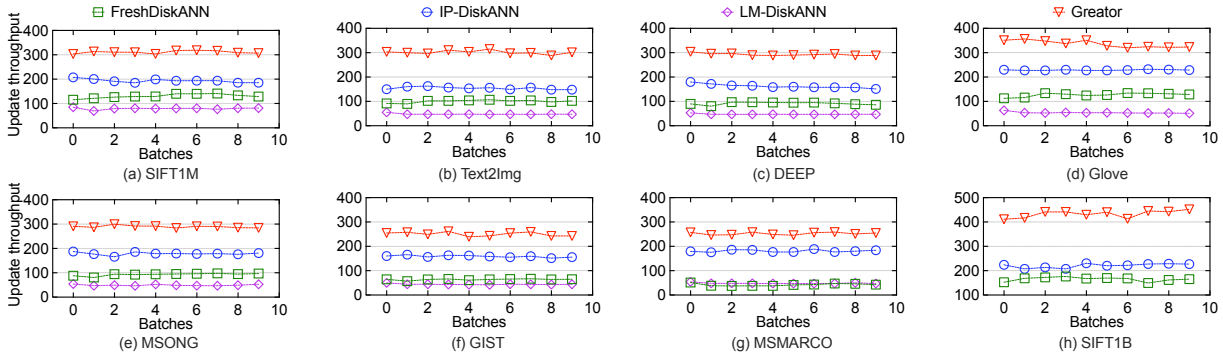


Figure 8: Update throughput comparison.

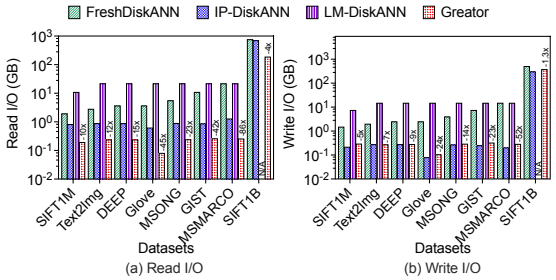


Figure 9: I/O amount comparison.

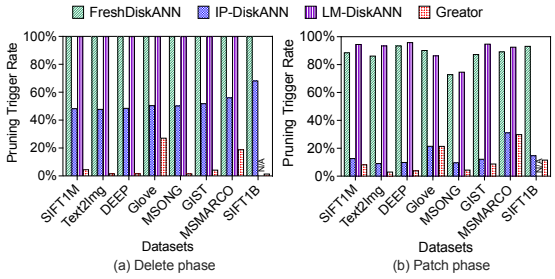


Figure 10: Pruning trigger rate comparison.

Compared Systems. In the experiments, we compare the following four systems. **FreshDiskANN** [53] is the state-of-the-art graph-based streaming ANNS system on disk [68], developed and open-sourced by Microsoft. **IP-DiskANN** [66] is a recently proposed algorithm for dynamically updating the index in DiskANN. We reproduce the method within the localized update framework of `Greator`, incorporating all of `Greator` 's I/O and disk update optimizations to ensure a fair comparison of repair algorithms. **LM-DiskANN** [45] is a disk-based ANNS system that supports index dynamic updates. It stores the compressed vectors of each node's neighbors on disk to reduce memory consumption. The graph indexes for all systems are built with $R = 32$, and we use 3 threads for search, 2 threads for insertion, 1 thread for deletion, and 10 threads for index batch update.

Metrics. `Greator` is designed to support fast index updates, and thus our evaluation primarily focuses on update performance. Update performance is measured by update throughput, which is defined as the number of vector insertions or deletions the system can process per second. Unless otherwise specified, the batch size in all update experiments is set to 1‰ of the dataset by default, ensuring a fair comparison of update performance under comparable index quality. To assess the broader impact of `Greator` on index quality and system overhead, we also evaluate several complementary metrics: Search accuracy is measured using recallK@K, with K set to 10; Search latency is reported using tail latency metrics, including P90, P95, P99, and P99.9; I/O overhead captures the amount of disk I/O generated by read and write operations; Space overhead reflects the disk space occupied by the index.

7.2 Update Performance

We first evaluate the update performance of different systems on eight real-world vector datasets (details are shown in Table 1). LM-DiskANN stores compressed neighbor vectors for each node in the index, which significantly increases the index size; consequently, the SIFT1B index exceeds our machine's storage capacity. The workload design follows the setup of FreshDiskANN. Specifically, we first use 99% of each dataset to statically construct the base index. Then, in each batch update, we delete 1‰ of the existing vectors and insert 1‰ of new vectors from the remaining 1% of the dataset. Due to randomness in base index construction, all experiments are repeated three times, and the average value is reported.

Update Throughput Comparison. To evaluate the update performance of different systems, we compare their update throughput over 10 consecutive batch updates, as shown in Figure 8. `Greator` consistently achieves higher throughput than FreshDiskANN and IP-DiskANN across all datasets. Specifically, `Greator` improves update throughput by 2.39×-5.96× compared to FreshDiskANN, 1.39×-1.96× compared to IP-DiskANN, 3.93×-6.28× compared to LM-DiskANN. Compared with baseline systems, the key advantage of `Greator` lies in its localized update design, which substantially reduces both I/O and computational overhead during updates. In the following, we provide a detailed analysis from these two perspectives.

I/O Amount Comparison. We measure the read and write I/O of each system during updates, as shown in Figure 9. `Greator` consistently achieves lower I/O than all baselines. Compared to FreshDiskANN, `Greator` reduces read I/O by 4.06×–85.58× and write I/O by 1.34×–52.08×, thanks to its topology-aware design: the lightweight topology avoids scanning unnecessary vectors, and the page-level localized update strategy minimizes redundant I/O. Compared to IP-DiskANN, `Greator` reduces read I/O by 3.31×–7.67×, while write I/O remains comparable because our reproduction of IP-DiskANN also benefits from `Greator` 's localized update strategy, further validating its effectiveness. Compared to LM-DiskANN, `Greator` reduces read I/O amount by 56.58×-271.30×, and write I/O amount by 25.51×-142.86×. This is mainly because LM-DiskANN

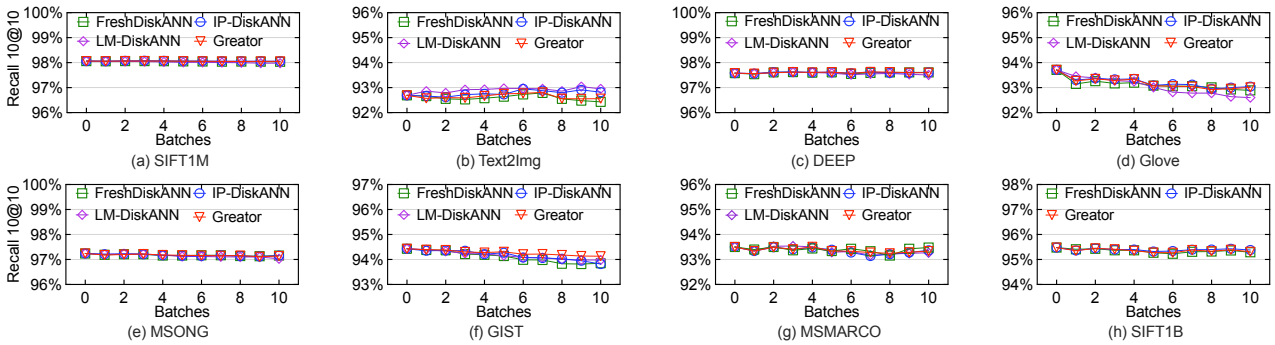


Figure 11: Recall comparison.

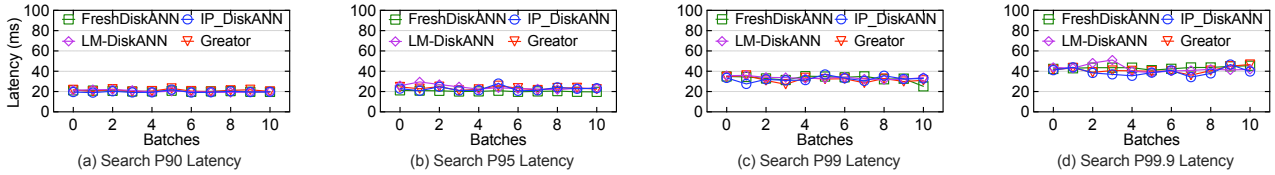


Figure 12: Search latency comparison on MSMARCO.

stores the compressed vectors of each node's neighbors in the index, which substantially enlarges the index size and consequently incurs higher I/O overhead during updates.

It is noteworthy that, as shown in Figure 9, compared to FreshDiskANN, `Greator` achieves the smallest I/O reduction on the SIFT1B dataset. This is because SIFT1B dataset has the lowest dimensionality and uses uint8 (see Table 1), resulting in the smallest proportion of vector data in the index file. Consequently, the lightweight topology of `Greator` saves the least redundant I/O introduced by vector data. As shown in Figure 9, under the same encoding length, higher-dimensional vectors further highlight the advantage of `Greator`, as it eliminates the overhead of reading high-dimensional vectors.

Pruning Trigger Rate Comparison. We further measure the neighbor pruning trigger rate during the delete and patch phases, and the results are shown in Figure 10. As observed, FreshDiskANN and LM-DiskANN exhibit similarly high pruning trigger rates, while both `Greator` and IP-DiskANN significantly reduce pruning across all datasets. This reduction in `Greator` is primarily attributed to its similarity-aware localized repair method, which prevents neighbor set overflow and thus minimizes unnecessary pruning. IP-DiskANN also reduces pruning compared to FreshDiskANN by reconstructing only a subset of neighbors during repair. In particular, during the patch phase, IP-DiskANN achieves a reduction comparable to `Greator`. This is because both `Greator` and IP-DiskANN benefit from relaxed neighbor constraints, which consistently lower pruning trigger rates relative to FreshDiskANN.

7.3 Index Quality

Although multiple designs in `Greator`, such as the localized update strategy and similarity-aware localized repair method, significantly improve update performance, it remains essential to evaluate whether `Greator` can maintain index quality. To this end, we assess search accuracy and search latency using the query vector sets provided by each dataset (see Table 1), based on the index state after each batch update evaluated in Section 7.2.

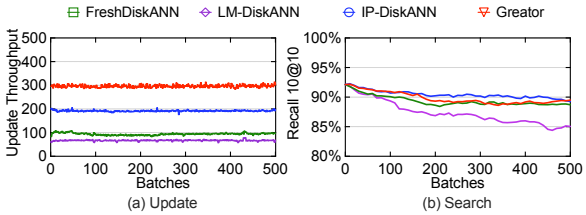


Figure 13: Update stability comparison.

Search Accuracy Comparison. As shown in Figure 11, we present the search recall of different systems after consecutive updates across all datasets. `Greator` achieves nearly identical recall to other systems on all datasets, demonstrating that our localized update strategy effectively preserves similar index quality while significantly improving update performance.

Search Latency Comparison. To further evaluate the impact of different update strategies on search performance, we measure the tail latency of queries (P90, P95, P99, and P99.9) on the MSMARCO dataset after consecutive updates. The results, shown in Figure 12, indicate that `Greator` achieves tail latency comparable other systems, and maintains consistently stable performance. This demonstrates that our similarity-aware localized repair strategy, combined with the relaxed neighbor limit, not only improves update performance but also preserves the high navigability of the index.

7.4 Update Stability

We also evaluate the stability of update performance under streaming data on GIST dataset. Specifically, we set the sliding window size to cover the first 50% of each dataset and use these nodes to build the initial index. In each batch of updates, the window slides forward by 1‰ of the datasets, i.e., we sequentially delete 1‰ of the data from the beginning of the window and insert 1‰ of the remaining data. This process is repeated 500 times, by which point the entire initial base index has been replaced and a completely new index has been constructed. As shown in Figure 13a, all systems maintain stable update performance, with Greator consistently

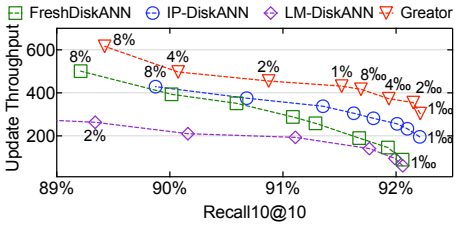


Figure 14: Varying batch size of updates.

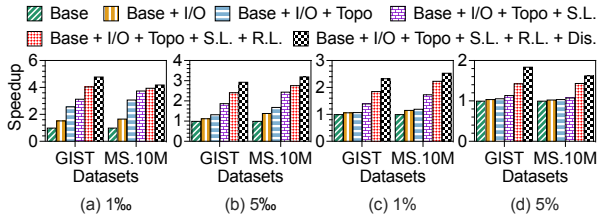


Figure 15: Performance gain.

demonstrating a stable performance advantage. In addition, we measure the query recall after each update, as shown in Figure 13b. While recall declines across all systems, our method maintains comparable performance to the baselines and achieves superior results over LM-DiskANN.

7.5 Sensitivity to Batch Size

We further evaluate the impact of batch size on update performance and index quality. Specifically, we measure the update throughput and recall of `Greator` and all baselines on the GIST dataset under varying batch sizes. As shown in Figure 14, as the batch size increases from 1‰ to 8% of the dataset, all systems show steadily improved update throughput. Furthermore, `Greator` consistently maintains a clear advantage over all three baselines: it achieves 1.23×–3.39× higher throughput than FreshDiskANN, 1.32×–1.55× higher throughput than IP-DiskANN, and 1.50×–4.97× higher throughput than LM-DiskANN. Notably, as batch sizes increase, the advantages of our optimizations diminish. This is because our localized update strategy is specifically designed for small batches, which affect only a small scope of the index. In contrast, larger batches introduce changes across the global index, thereby reducing the effectiveness of our optimizations.

Although larger batch sizes improve update throughput for these systems, their recall consistently drops (Figure 14). This shows that increasing batch size to boost throughput inevitably harms index quality. This observation motivates our design: prioritize small-batch updates to preserve quality while improving update performance.

7.6 Performance Gain

To evaluate the contribution of each core design in `Greator` to update performance, we begin with the baseline system (FreshDiskANN as base) and incrementally incorporate the proposed components of `Greator`. Each configuration is evaluated on two real-world datasets of different scales, GIST (1M vectors) and MSMARCO10M (10M vectors, abrv. MS.10M). For each dataset, we also vary the update batch size to examine the performance gains under different batch sizes.

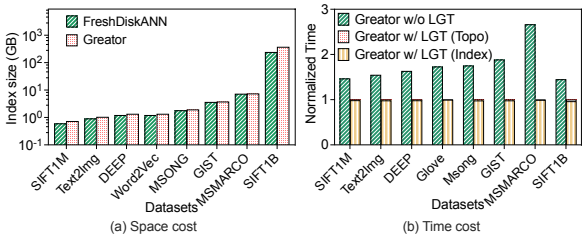


Figure 16: Cost of lightweight graph topology.

Across both datasets, when the batch size is set to 1‰ of the dataset size, all designs achieve substantial update acceleration. Specifically, Figure 15 presents the update speedup, measured as the normalized update throughput relative to the baseline. First, incorporating the localized update strategy (+I/O), which updates only the affected nodes, achieves a speedup of 1.54× and 1.67× on the two datasets, respectively. Next, introducing the lightweight graph topology (+Topo) improves performance to 2.58× and 3.09× by avoiding full index file scans. Subsequently, integrating the similarity-aware localized repair method (+S.L.) further enhances performance to 3.15× and 3.76× by reducing neighbor pruning. Next, applying the relaxed neighbor limit strategy (+R.L.) further decreases the computational overhead of pruning, ultimately achieving a performance boost to 4.06× and 3.96×. Finally, we also evaluate storing neighbor distances in the topology (+Dis.), which reduces the computation overhead of structure repair and improves update performance by up to 4.80× and 4.21×. These results demonstrate that all designs in `Greator` effectively contribute to improving update performance.

Furthermore, as shown in Figure 15, the results under different batch sizes indicate that the performance gains of most designs diminish as the batch size increases. This is because our designs are primarily tailored for small-batch updates: in I/O optimization, the benefit arises from reducing unnecessary block loading, but larger batches inevitably require loading more blocks, reducing the optimization space. Similarly, in computation optimization, larger batches cause more neighbors of each node to be updated, thereby increasing the likelihood of triggering costly pruning operations. Notably, compared to other designs, the strategies of relaxed neighbor limit and storing neighbor distances remain effective even under larger batch sizes. This is because, although larger batches require updating more nodes, these mechanisms still help reduce the computation cost for each updated node.

7.7 Cost of Lightweight Graph Topology

Although the lightweight graph topology in `Greator` significantly reduces I/O overhead and improves update performance, it also incurs additional storage and maintenance costs. We next conduct a detailed experimental analysis of these overheads.

Space Cost. We measure the storage overheads of `Greator` compared to FreshDiskANN. as illustrated in Figure 16a. On average, `Greator` incurs a 1.16× larger index file, primarily due to the additional lightweight graph topology. Notably, these files are stored on inexpensive and high-capacity disks, making the slight increase in space overhead generally acceptable. Furthermore, as the dataset dimensionality increases (when the data types are the same), the difference in index file size between `Greator` and FreshDiskANN

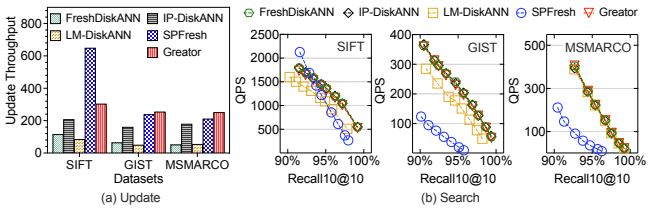


Figure 17: Comparison with SPFresh.

gradually decreases, resulting in a less pronounced space amplification effect. For example, on the MSMARCO dataset with 1024 dimensions, the overhead is only 1.02×.

Time Cost. In `Greator`, we measure both the time spent maintaining the lightweight graph topology (i.e., Greator w/ LGT (Topo)) and the time spent updating the query index (i.e., Greator w/ LGT (Index)). We also normalize update time by taking `Greator` as the baseline and report the update time without the lightweight graph topology (i.e., Greator w/o LGT) to show its benefits and overhead clearly. As shown in Figure 16b, the lightweight graph topology substantially reduces update time, while its maintenance accounts for only 0.59%-3.78% of the total, making its impact on overall performance negligible. These results confirm that the design introduces low overhead yet delivers significant performance gains.

7.8 Comparison with Partition-Based Graph ANNS System

We evaluate SPFresh [68], a state-of-the-art partition-based streaming ANNS system, on three real-world datasets. As shown in Figures 17, SPFresh achieves higher update performance since it only updates the partitions containing the affected vectors, whereas graph-based indices must update all neighbors. Its advantage is most evident on the low-dimensional SIFT dataset, where partitions are easier to separate. However, this advantage diminishes on high-dimensional datasets (e.g., GIST and MSMARCO), where overlapping partitions increase update costs. In contrast, for queries, graph-based indices (including `Greator`) consistently outperform SPFresh except on SIFT under low recall, since partition-based methods require probing many partitions to achieve high recall in high-dimensional settings. Overall, while SPFresh excel at updates, `Greator` and other graph-based approaches achieve a better balance by combining strong update efficiency with superior query performance.

8 RELATED WORK

Vector Indices. The widespread application of ANNS has led to significant research in vector indexing [7, 10, 15, 16, 19, 20, 22, 24, 25, 36, 45, 46, 48, 54, 60, 66, 73, 75], such as HNSW [36], IVFADC [25], and Vamana [24]. Most of these algorithms focus on the offline construction of high-quality indices for high-precision and low-latency vector queries. However, only a few algorithms address the online construction and updating of index structures (including insertions and deletions), such as FreshVamana [53]. FreshVamana [53] is the first graph-based index to support insertions and deletions. It incrementally adjusts and optimizes the index based on updates to avoid the expensive overhead of rebuilding the graph structure, while maintaining high index quality.

Vector Search Systems. To meet the vector search performance requirements in different real-world scenarios, numerous vector search systems have been developed [1–4, 11–13, 18, 21, 23, 24, 32, 37, 44, 53, 56, 58, 59, 61, 63, 64, 67, 68, 71, 76, 78]. These systems typically optimize vector search performance by combining specific index structures and search algorithm characteristics. Some work further explores combining disk and other external storage devices to support large-scale vector search under limited memory (e.g., [13, 24]). However, most of these works focus on query performance optimization in static vector data indexing. Only a few systems support the dynamic update of index structures in dynamic vector scenarios, such as SPFresh [68] and FreshDiskANN [53]. SPFresh [68] uses a cluster-based index and performs poorly in high-dimensional vector scenarios, while FreshDiskANN provides limited update performance.

Auxiliary Structure and Localizing I/O. Previous research in the database field has explored small auxiliary structures and I/O-localization techniques for update optimization, such as zone maps in column stores [27, 55], delta stores [28, 77], LSM-trees [38, 43, 57, 69]. While `Greator` draws inspiration from these approaches, they are not directly applicable to ANN indices: statistical summaries (e.g., zone maps for value pruning) accelerate queries but cannot efficiently locate nodes affected by deletions, and delta stores or LSM-trees absorb updates through additional layers, often incurring query slowdowns. In contrast, Greator leverages a lightweight auxiliary graph topology to rapidly identify affected nodes and performs localized in-place updates to avoid redundant vector I/O and costly global index repairs. Since this structure is used only for updates and not for queries, it significantly improves update efficiency without impacting query performance.

9 CONCLUSION

This paper proposes a topology-aware localized update strategy for graph-based ANN indices that enables rapid updates while preserving high search efficiency and accuracy. It incorporates three key designs to enhance update performance. First, a lightweight graph topology facilitates efficient identification of affected nodes without incurring the I/O cost of reading unneeded vector data. Second, a localized update mechanism restricts modifications to only the pages containing affected nodes, avoiding global traversals and reducing disk access. Third, a similarity-aware local connection method minimizes costly neighbor pruning by selecting only the most similar candidates for repair. We implement a graph-based streaming vector ANNS system, `Greator`, based on this strategy. Extensive experiments show that `Greator` achieves efficient updates while maintaining competitive index quality.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (62202088, 62461146205, U2241212), the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501), the Fundamental Research Funds for the Central Universities (N25ZLL045), 111 Project (B16009), and CCF-Huawei Populus Grove Fund. Yanfeng Zhang and Shufeng Gong are the corresponding authors.

REFERENCES

[1] 2025. Faiss. https://github.com/facebookresearch/faiss.

[2] 2025. Kgraph. https://github.com/aaalgo/kgraph.

[3] 2025. pgvector. https://github.com/pgvector/pgvector.

[4] 2025. Pinecone. https://www.pinecone.io.

[5] Laurent Amsaleg and Hervé Jegou. 2010. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/.

[6] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. 2023. Retrieval-based Language Models and Applications. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts, ACL 2023, Toronto, Canada, July 9-14, 2023*. Association for Computational Linguistics, 41–46.

[7] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. Elpis: Graph-based similarity search for scalable data science. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1548–1559.

[8] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2025. Graph-Based Vector Search: An Experimental Evaluation of the State-of-the-Art. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–31.

[9] Fu Bang. 2023. Gptcache: An open-source semantic cache for llm applications enabling faster answers and cost savings. In *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*. 212–218.

[10] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*. ACM Press, 322–331.

[11] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. 2024. Navigating Labels and Vectors: A Unified Approach to Filtered Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 6 (2024), 246:1–246:27.

[12] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. *SPTAG: A library for fast approximate nearest neighbor search*. https://github.com/Microsoft/SPTAG

[13] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 5199–5212.

[14] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*. ACM, 191–198.

[15] Sanjoy Dasgupta and Yoav Freund. 2008. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*. ACM, 537–546.

[16] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.

[17] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. 2023. Retrieval-Augmented Generation for Large Language Models: A Survey. *CoRR* abs/2312.10997 (2023).

[18] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *Proceedings of the ACM Web Conference 2023, WWW 2023, Austin, TX, USA, 30 April 2023 - 4 May 2023*. ACM, 3406–3416.

[19] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. 2025. SymphonyQG: Towards Symphonious Integration of Quantization and Graph for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[20] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*. ACM Press, 47–57.

[21] Han Hu, Jiye Qiu, Hongzhi Wang, Bin Liang, and Songling Zou. 2024. DIDS: Double Indices and Double Summarizations for Fast Similarity Search. *Proc. VLDB Endow.* 17, 9 (2024), 2198–2211.

[22] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 9, 1 (2015), 1–12.

[23] Shikhar Jaiswal, Ravishankar Krishnaswamy, Ankit Garg, Harsha Vardhan Simhadri, and Sheshansh Agrawal. 2022. OOD-DiskANN: Efficient and Scalable Graph ANNS for Out-of-Distribution Queries. *CoRR* abs/2211.12850 (2022).

[24] 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).

[25] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: Re-rank with source coding. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011, May 22-27, 2011, Prague Congress Center, Prague, Czech Republic*. IEEE, 861–864.

[26] Zhi Jing, Yongye Su, Yikun Han, Bo Yuan, Haiyun Xu, Chunjiang Liu, Kehai Chen, and Min Zhang. 2024. When large language models meet vector databases: A survey. *arXiv preprint arXiv:2402.01763* (2024).

[27] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (2012), 1790–1801.

[28] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 302–313.

[29] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[30] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. 2022. A Survey on Retrieval-Augmented Text Generation. *CoRR* abs/2202.01110 (2022).

[31] Hongfu Li, Qian Tao, Song Yu, Shufeng Gong, Yanfeng Zhang, Feng Yao, Wenyuan Yu, Ge Yu, and Jingren Zhou. 2024. GastCoCo: Graph Storage and Coroutine-Based Prefetch Co-Design for Dynamic Graph Processing. *Proc. VLDB Endow.* 17, 13 (2024), 4827–4839.

[32] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. 2018. The Design and Implementation of a Real Time Visual Search System on JD E-commerce Platform. In *Proceedings of the 19th International Middleware Conference, Middleware Industrial Track 2018, Rennes, France, December 10-14, 2018*. ACM, 9–16.

[33] Nan Li, Bo Kang, and Tijl De Bie. 2023. SkillGPT: a RESTful API service for skill extraction and standardization using a Large Language Model. *CoRR* abs/2304.11060 (2023).

[34] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *IEEE Trans. Knowl. Data Eng.* 32, 8 (2020), 1475–1488.

[35] Ruiyao Ma, Yifan Zhu, Baihua Zheng, Lu Chen, Congcong Ge, and Yunjun Gao. 2024. GTI: Graph-Based Tree Index with Logarithm Updates for Nearest Neighbor Search in High-Dimensional Spaces. *Proceedings of the VLDB Endowment* 18, 4 (2024), 986–999.

[36] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.

[37] Magdalen Dobson Manohar, Zheqi Shen, Guy E. 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, PPoPP 2024, Edinburgh, United Kingdom, March 2-6, 2024*. ACM, 270–285.

[38] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12 (2020), 3217–3230.

[39] 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 (Lecture Notes in Computer Science)*, Vol. 12036. Springer, 272–280.

[40] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Umar Farooq Minhas, Jeffrey Pound, Cédric Renggli, Nima Reyhani, Ihab F. Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. 2024. Incremental IVF Index Maintenance for Streaming Vector Search. *CoRR* abs/2411.00970 (2024).

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

[42] Shumpei Okura, Yukihiro Tagami, Shingo Ono, and Akira Tajima. 2017. Embedding-based News Recommendation for Millions of Users. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 1933–1942.

[43] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[44] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024,*

Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 597–604.

[45] Yu Pan, Jianxin Sun, and Hongfeng Yu. 2023. LM-DiskANN: Low Memory Footprint in Disk-Native Dynamic Graph-Based ANN Indexing. In *IEEE International Conference on Big Data, BigData 2023, Sorrento, Italy, December 15-18, 2023*. IEEE, 5987–5996.

[46] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proc. ACM Manag. Data* 2, 3 (2024), 120.

[47] Sajal Regmi and Chetan Phakami Pun. 2024. GPT Semantic Cache: Reducing LLM Costs and Latency via Semantic Embedding Caching. *CoRR* abs/2411.05276 (2024).

[48] Jie Ren, Minjia Zhang, and Dong Li. 2020. Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory. *Advances in Neural Information Processing Systems* 33 (2020), 10672–10684.

[49] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. 2022. ColBERTv2: Effective and Efficient Retrieval via Lightweight Late Interaction. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*. Association for Computational Linguistics, 3715–3734.

[50] Badrul Munir Sarwar, George Karypis, Joseph A. Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, May 1-5, 2001*. ACM, 285–295.

[51] Luis Gaspar Schroeder, Shu Liu, Alejandro Cuadron, Mark Zhao, Stephan Krusche, Alfons Kemper, Matei Zaharia, and Joseph E. Gonzalez. 2025. Adaptive Semantic Prompt Caching with VectorQ. *CoRR* abs/2502.03771 (2025).

[52] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. 2022. Results of the NeurIPS'21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track*. PMLR, 177–189.

[53] 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. *CoRR* abs/2105.09613 (2021).

[54] Yitong Song, Kai Wang, Bin Yao, Zhida Chen, Jiong Xie, and Feifei Li. 2024. Efficient Reverse k Approximate Nearest Neighbor Search Over High-Dimensional Vectors. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 4262–4274.

[55] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM, 553–564.

[56] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. 2024. Vexless: A Serverless Vector Data Management System Using Cloud Functions. *Proc. ACM Manag. Data* 2, 3 (2024), 187.

[57] Viraj Thakkar, Dongha Kim, Yingchun Lai, Hokeun Kim, and Zhichao Cao. 2025. SHIELD: Encrypting Persistent Data of LSM-KVS from Monolithic to Disaggregated Storage. *Proc. ACM Manag. Data* 3, 3 (2025), 217:1–217:28.

[58] Karthik V., Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. 2024. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. *CoRR* abs/2401.11324 (2024).

[59] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2614–2627.

[60] Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Yifan Zhu, and Wenchao Zhou. 2025. Accelerating Graph Indexing for ANNS on Modern CPUs. *arXiv preprint arXiv:2502.18113* (2025).

[61] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *CoRR* abs/2401.02116 (2024).

[62] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 14, 11 (2021), 1964–1978.

[63] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proc. VLDB Endow.* 13, 12 (2020), 3152–3165.

[64] Jiuqi Wei, Xiaodong Lee, Zhenyu Liao, Themis Palpanas, and Botao Peng. 2024. Subspace Collision: An Efficient and Accurate Framework for High-dimensional Approximate Nearest Neighbor Search. *CoRR* abs/2411.14754 (2024).

[65] Kyle Williams, Lichi Li, Madian Khabsa, Jian Wu, Patrick C. Shih, and C. Lee Giles. 2014. A Web Service for Scholarly Big Data Information Extraction. In *2014 IEEE International Conference on Web Services, ICWS, 2014, Anchorage, AK, USA, June 27 - July 2, 2014*. IEEE Computer Society, 105–112.

[66] Haike Xu, Magdalen Dobson Manohar, Philip A. Bernstein, Badrish Chandramouli, Richard Wen, and Harsha Vardhan Simhadri. 2025. In-Place Updates of a Graph Index for Streaming Approximate Nearest Neighbor Search. *CoRR* abs/2502.13826 (2025).

[67] Qian Xu, Juan Yang, Feng Zhang, Junda Pan, Kang Chen, Youren Shen, Amelie Chi Zhou, and Xiaoyong Du. 2025. Tribase: A Vector Data Query Engine for Reliable and Lossless Pruning Compression using Triangle Inequalities. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–28.

[68] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. ACM, 545–561.

[69] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Kenry Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory. *Proc. VLDB Endow.* 14, 10 (2021), 1872–1885.

[70] Mingyu Yang, Wentao Li, and Wei Wang. 2025. Fast High-dimensional Approximate Nearest Neighbor Search with Efficient Index Time and Space. arXiv:2411.06158 [cs.DB] https://arxiv.org/abs/2411.06158

[71] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2241–2253.

[72] Zhi Yao, Zhiqing Tang, Jiong Lou, Ping Shen, and Weijia Jia. 2024. VELO: A Vector Database-Assisted Cloud-Edge Collaborative LLM QoS Optimization Framework. In *IEEE International Conference on Web Services, ICWS 2024, Shenzhen, China, July 7-13, 2024*. IEEE, 865–876.

[73] Ziqi Yin, Jianyang Gao, Pasquale Balsebre, Gao Cong, and Cheng Long. 2025. DEG: Efficient Hybrid Vector Search Using the Dynamic Edge Navigation Graph. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–28.

[74] Song Yu, Shufeng Gong, Qian Tao, Sijie Shen, Yanfeng Zhang, Wenyuan Yu, Pengxi Liu, Zhixin Zhang, Hongfu Li, Xiaojian Luo, Ge Yu, and Jingren Zhou. 2024. LSMGraph: A High-Performance Dynamic Graph Storage System with Multi-Level CSR. *Proc. ACM Manag. Data* 2, 6 (2024), 243:1–243:28.

[75] 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.* 46, 3 (2023), 22–38.

[76] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 377–395.

[77] Wei Zhang, Cheng Chen, Qiange Wang, Wei Wang, Shijiao Yang, Bingyu Zhou, Huiming Zhu, Chao Chen, Yongjun Zhao, Yingqian Hu, Miaomiao Cheng, Meng Li, Hongfei Tan, Mengjin Liu, Hexiang Lin, Shuai Zhang, and Lei Zhang. 2024. BG3: A Cost Effective and I/O Efficient Graph Database in Bytedance. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago, Chile, June 9-15, 2024*. ACM, 360–372.

[78] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. 2023. Fast, Approximate Vector Queries on Very Large Unstructured Datasets. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*. USENIX Association, 995–1011.

[79] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (2020), 1020–1034.