Wolverine: Highly Efficient Monotonic Search Path Repair for Graph-based ANN Index Updates

Dawei Liu
Huazhong University of Science and Technology

Bolong Zheng*
Huazhong University of Science and Technology

Ziyang Yue
Huazhong University of Science and Technology

Fuhao Ruan
Huazhong University of Science and Technology

Xiaofang Zhou
Hong Kong University of Science and Technology

Christian S. Jensen
Aalborg University

ABSTRACT

Approximate nearest neighbor (ANN) search on high-dimensional vector data is core functionality in an increasing number of real-world applications. However, most existing methods only focus on accelerating search by means of indexing that assumes that the data is static. The few methods capable of contending with dynamic data often face challenges such as decreased query accuracy following updates and low update efficiency. In this study, we propose Wolverine, the first proposal that, to our knowledge, enables efficient monotonic search path repair, thereby solving the graph-based ANN index update problem. Wolverine repairs disrupted monotonic search paths by adding in-edges to the out-neighbors of a point to be deleted. To improve efficiency, Wolverine+ restricts the search space to be within the 2-hop neighbors of the point to be deleted. In addition, Wolverine++ employs a sophisticated candidate selection policy to find high-quality candidates in the reduced search space, simultaneously improving accuracy and efficiency. An experimental study on 9 real-world datasets demonstrates that Wolverine is capable of accelerating the deletion throughput by up to 11× and achieving more stable recall during updates compared to the state-of-the-art dynamic ANN search method.

PVLDB Reference Format:
Dawei Liu, Bolong Zheng, Ziyang Yue, Fuhao Ruan, Xiaofang Zhou, and Christian S. Jensen. Wolverine: Highly Efficient Monotonic Search Path Repair for Graph-based ANN Index Updates. PVLDB, 18(7): 2268 - 2280, 2025.
doi:10.14778/3734839.3734860

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/LDW2020/Wolverine.

1 INTRODUCTION

We are witnessing a great breakthrough of Large Language Model (LLM), which exhibits unprecedented capability in understanding, generating, and reasoning with human language. This capability is rooted in the vector representations of knowledge exploited from multi-modal data. After being embedded into the vector space, the correlation of cross-domain data can be measured by the distance between their vector representations. Therefore, approximate nearest neighbor (ANN) search can be used to search for knowledge relevant to the user-provided prompts and thus is a fundamental function in LLM.

ANN search is also a core function ability in applications involving data mining [6], recommendation [8], and information retrieval [1]. Most existing studies focus on building high-quality indexes for static datasets [4, 11, 12, 16, 19, 20, 26, 29, 37, 39, 40], with graph-based ANN search methods emerging as the most promising direction due to high accuracy and efficiency [34].

However, static ANN search methods cannot adapt to dynamic scenarios, where data is updated frequently. For example, consider an e-commerce scenario, such as Amazon and Taobao, where the system indexes the images and descriptions of products. The index requires continuous updates to accommodate changes in products, e.g., product launches or retirements. Assume that the system only employs static ANN indexes, which support insertions but do not allow deletions. To avoid search results containing products that have been retired, it is inevitable to search more results and filter these retired products from the result set. Furthermore, the accumulation of data in the index further significantly reduces query and insertion efficiency, forcing periodic rebuilding of these large-scale indexes, which can take days or even months. Therefore, dynamic ANN search, which can accommodate these rapid changes, is becoming increasingly important and warrants more attention.

A common approach to address the graph-based ANN index update problem is to extend static graph-based indexes with straightforward deletion strategies. In addition, FreshDiskAnn [30] modifies the graph structure to enhance the update performance and is the state-of-the-art dynamic ANN search method. However, these methods face two significant challenges:

- **Decreased accuracy after updates.** During updates, the deletion operation destroys so-called monotonic search paths that are necessary for accurate ANN search in a graph index. However, existing methods neglect this issue and try to repair the index without explicitly recovering destroyed monotonic search paths. As a result, these methods exhibit substantially decreased accuracies following updates.
- **Low update efficiency.** When updating the graph index after deletion operations, existing methods modify the edge lists of a large number of nodes relevant to a deleted node. Then, these

*Bolong Zheng is the corresponding author

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734860

nodes may go through a time-consuming edge trimming process, leading to low update efficiency.

We propose Wolverine, a novel graph-based dynamic ANN search method. We choose the name Wolverine because the superhero's healing power serves as a fitting analogy to the capabilities of the proposed algorithm, which is designed to repair and maintain the robustness of graph-based ANN indexes when subjected to updates. The idea of Wolverine is universal and can be integrated into other graph-based ANN indexes to enable accurate and efficient updates. To the best of our knowledge, Wolverine is the first proposal that enables monotonic search path repair, thereby solving the graph-based ANN index update problem. Specifically, Wolverine performs ANN search by taking each out-neighbor of a node to be deleted as a query, aiming to repair disrupted monotonic search paths by adding edges to those out-neighbors. To further enhance efficiency, we propose Wolverine+ that limits the search space to only the 2-hop neighbors of the node to be deleted. Finally, we propose Wolverine++, which employs a candidate selection policy that identifies high-quality candidates in a further reduced search space, thus improving both accuracy and efficiency. Extensive experiments show that Wolverine++ is capable of efficient deletion while preserving the accuracy of the graph index. Compared to the SOTA dynamic ANN search algorithm, FreshDiskAnn, Wolverine++ offers stability of the updated graph index while also improving accuracy and achieving an update efficiency that is up to 11× higher than that of FreshDiskAnn.

We summarize the key contributions as follows.

(1) We propose a novel performance quantification method called sliding window quantification, which simulates dynamic update scenarios and provides a realistic measure of the performance of dynamic ANNS algorithms.
(2) We develop the novel Wolverine algorithm that considers the impact of deletion operations from the perspective of repairing monotonic search paths in graph indexes.
(3) We further propose two optimized algorithms, Wolverine+ and Wolverine++, with delicately designed candidate set generation policy to obtain high-quality candidates within small search space, such that both accuracy and efficiency are improved.
(4) We report on extensive experiments on 9 datasets, showing that Wolverine++ is able to outperform baselines consistently, delivering stable index performance with improved accuracy and up to 11× higher update efficiency.

2 PRELIMINARIES

We proceed to provide the background knowledge of dynamic ANN search and analyze existing update methods.

2.1 Graph-based ANN Index

The ANN search problem is formally defined as follows, which is easy to extend to kANN search when the context is clear.

Definition 2.1 (ANN Search). Given a set $P \subseteq \mathbb{E}^d$ of n points in a d-dimensional Euclidean space, and let $q \in \mathbb{E}^d$ be a query point. The ANN search aims to construct an index structure that enables fast retrieval of an approximate nearest neighbor $p' \in P$ for q, such that:

$$d(p', q) \leq (1 + \epsilon) \cdot d(p^*, q)$$

where:

- p^* is the true nearest neighbor of q in the set P,
- $\epsilon \geq 0$ is the approximation factor, and
- $d(p, q)$ is the Euclidean distance between two points p and q.

Multiple index structures [11, 20, 26, 29, 37, 39, 40] have been proposed for ANN search on static datasets. Graph-based ANN indexes demonstrate potential in terms of query efficiency and accuracy, surpassing other indexes [11], primarily because they enable a greedy search strategy or a variant of it to process queries [11, 16, 26, 30].

Greedy Search Strategy. Let G be a graph-based ANN index built on dataset P such that each node represents a point, and edges between nodes represent proximity relationships between the points. Let s be the search starting node of G, where the greedy search algorithm starts from this node. $N_{out}(p)$ is the set of out-neighbors of node p, and p_{out} is one such out-neighbor. Similarly, $N_{in}(p)$ is the set of in-neighbors of node p, with p_{in} being one of its in-neighbors. The greedy search starts at node s and iteratively moves toward the query node q. At each step, the algorithm explores the out-neighbors $N_{out}(p)$ of the current node p and selects the neighbor closest to q. Then, it moves to this neighbor by updating the current node to it. The process repeats until no node in $N_{out}(p)$ is closer to q than p.

Monotonic Search Networks. The effectiveness of this greedy search strategy on proximity graph is due to the monotonic search property. Indexes with this property are called Monotonic Search Networks (MSNET) [9], and when performing greedy search on an MSNET, the query point can be found without backtracking. Therefore, in this study, we employ graph-based ANN index as the foundation for developing our update operations. To enable better understanding, we include the definition as follows.

Definition 2.2 (Monotonic Search Network (MSNET)). We use $P(v_1, v_2, \ldots, v_k)$ to represent a path of the graph G, i.e., $\forall i = 1, \ldots, k-1, \overrightarrow{v_i, v_{i+1}} \in G$. This path is called a monotonic search path to a node q if and only if $d(v_i, q) > d(v_{i+1}, q), \forall i = 1, \ldots, k-1$, which is denoted as $MSP(v_1, v_k)$ of q. In other words, the nodes in $MSP(v_1, v_k)$ gradually approach to q, so we normally have $v_k = q$.

The graph index G built on P is called a monotonic search network if and only if there exists at least one monotonic search path $MSP(p, q)$ of q between any two nodes $p, q \in P$.

Insertion Operation. The sparse neighborhood graph (SNG) [2] is one of the most prominent members of the MSNET family. It is employed widely in graph-based indexing algorithms, such as HNSW [26], NSG [11], FANNG [16], and DiskAnn [31]. Although these algorithms are designed for static data and do not support deletions, some do support insertions, enabling similar insertion strategies for incremental index construction. Specifically, when inserting a point p into an index, a kANN search is first conducted to locate p, and the search results form a candidate set C. A so-called EdgeTrim(p, C, θ_d) function is then employed to select $N_{out}(p)$ from the candidate set C, where θ_d is the degree threshold of nodes in the graph. Finally, bidirectional edges are added between p and $N_{out}(p)$. The EdgeTrim($\cdot$) function is the core of the insertion operation, and is described as follows.

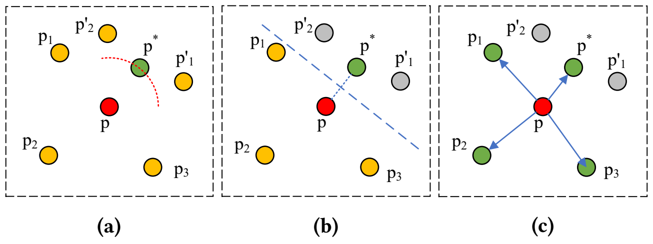


Figure 1: An Example of EdgeTrim(·)

EdgeTrim(·) Function. As shown in Fig. 1, the result set V of EdgeTrim(p, C, θ_d) is initialized to $V = \emptyset$. The function first adds the node p^* closest to p to V. It then removes p^* and all nodes $p' \in C$ with $d(p', p^*) < d(p', p)$ from C. In other words, it eliminates all nodes p' located on the same side as p^* of the space, partitioned by the perpendicular bisector between p and p^*. This process is repeated until the candidate set is empty or the size of V reaches θ_d. Fig. 1 illustrates this process. By trimming the out-edges through the removal of edges in similar directions, EdgeTrim(·) ensures a relative even distribution of out-edges in all directions. Additionally, since the function always selects the node closest to p from the remaining candidates, it prioritizes retaining shorter edges. This is referred to as the short edge priority principle.

Due to the high time complexity of constructing an exact SNG, existing proposals employ approximate SNGs as their underlying structure. While approximate SNGs accelerate index construction, they no longer guarantee the existence of a monotonic search path between every pair of nodes. Instead, they maintain monotonic search paths only from the search starting node to other nodes, which makes it more challenging to update the structure. Although many SNG-based indexes perform well in static settings, they do not support updates or only support insertions, limiting their applicability in dynamic settings.

2.2 Dynamic ANN Search

Existing static ANN search methods focus on accelerating query performance without accommodating updates to the data. However, in real-world applications, the entire dataset P is often unavailable when building an index, and we need to repeatedly add new data and remove outdated data. Therefore, the index must be updated accordingly. ANN search on a dynamic dataset thus causes additional challenges to indexing. In particular, existing indexes designed for static datasets thus struggle to adapt to dynamic scenarios. In order for an index to facilitate ANN search on a dynamic dataset, called dynamic ANN search, it must support three operations [30]:

(1) **Querying**: Finding the ANN for a given query point;
(2) **Insertion**: Inserting a new point;
(3) **Deletion**: Deleting an existing point.

Existing ANN indexes can support querying and the insertion of new points, but they often struggle to support deletion. Take HNSW [26] for example. When points must be deleted, HNSW employs a so-called pseudo-deletion, where points are marked as deleted so that they can be excluded from query results. However, as points are not removed physically from the index, "deleted" points continue to occupy storage and incur computational overhead. As the amount of pseudo-deleted points increases, queries increasingly traverse "deleted" points, slowing down query processing. Therefore, having accumulated a certain amount of pseudo-deleted points, the index must be rebuilt to ensure query efficiency. Unfortunately, for large datasets, the cost of rebuilding an index is high, often taking days to complete. This motivates us to explore new index update strategies.

Evaluation Metrics. We employ the following metrics to evaluate dynamic ANN search solutions:

(1) **Recall**: The query accuracy should remain stable or decrease at most slightly. For kNNS, we use k-recall@k [30], which is defined as $\frac{|R \cap T|}{k}$, where R is the query result set, T is the query ground truth, and $|R| = |T| = k$.
(2) **Query Throughput**: The query throughput, or average query latency, must not increase. We use OPS (operations per second) to measure the throughput. Specifically, we use Search_OPS to measure the query throughput.
(3) **Update Throughput**: The efficiency of update operations, i.e., insertions and deletions, must be sufficiently high, ideally exceeding that of periodic index rebuilding. Similar to Search_OPS, we use Delete_OPS and Insert_OPS to measure deletion throughput and insertion throughput, respectively.

2.3 Update Performance Quantification

Studies on static ANN search focus primarily on evaluating metrics, such as recall and query throughput, to assess performance. However, in dynamic scenarios, where a dataset is updated continuously, new quantification strategies that simulate evolving datasets are needed to evaluate performance. To enable this, we employ two quantification strategies to examine the performance of update operations: (1) Delete-and-Reinsert Quantification (DRQ), and (2) Sliding Window Quantification (SWQ).

Delete-and-Reinsert Quantification (DRQ). The DRQ strategy follows the method used in FreshDiskAnn [30]. Let UR be the update rate, which is the percentage of points to update. Initially, all points in the dataset are inserted into the index. During repeated rounds of DRQ, we first randomly delete $UR * n$ points (in experiments, we set UR to 1%, 5%, 10%, and 20% of the points), and we then reinsert them back into the index. Since the deleted points are reinserted, the size and distribution of the dataset remain unaltered. Therefore, a high-quality index should maintain stable recall and update throughput, and the update throughput should remain high.

Sliding Window Quantification (SWQ). Although DRQ can quantify the performance of update operations, it does not reflect real-world scenarios where deleted data is typically not reinserted into the index. Thus, while DRQ provides useful insight into the stability of recall and throughput during updates, its setup is limited in simulating realistic dynamic scenarios. To address this, we propose a novel sliding window quantification (SWQ) strategy that mimics practical update scenarios by replacing deleted data with new data. We begin by randomly shuffling the points in a dataset and setting a sliding window over it. The window is initially set to include the first w points, and the index is built using these points. In each round of SWQ, we slide the window backward by $UR * w$ points, updating the index to match the points within the window. Throughout the process, the number of points in the index remains constant, and since the data originates from the dataset, its distribution does not change substantially.

3 ANALYSIS OF EXISTING UPDATE METHODS

We proceed to present an analysis of existing methods for updating graph-based ANN indexes. Specifically, we report on pre-experiments using the SIFT1M dataset, to gain insights into (1) HNSW extended with three straightforward update operations and (2) the state-of-the-art dynamic ANN algorithm, FreshDiskAnn [30]. Based on the findings, we identify the reasons for the performance of these methods, and we highlight the key challenges in maintaining both efficiency and accuracy during updates, particularly on large-scale, dynamic datasets.

3.1 Baseline Update Operations

We consider HNSW and FreshDiskAnn because they are the most prevalent static and dynamic graph-based ANN algorithms, respectively. HNSW employs SNGs, while FreshDiskAnn approximates the α-Relative Neighborhood Graph (α-RNG), an SNG variant. Since HNSW is an in-memory index, we use the in-memory version of FreshDiskAnn to enable a fair comparison. We extend HNSW with the following three update operations:

Pseudo-deletion (Pseudo). Let p be a node to be deleted, Pseudo only marks it as deleted without actually removing it from the graph index. The deleted nodes can still be accessed during ANN search, but are excluded from the results.

Deletion Only (Do). Do directly removes p from the graph, along with all its in-edges and out-edges. After the deletion, no additional edges are added to repair the connectivity loss resulting from the removal of p.

Deletion with Fully Connections (DwFC). DwFC first performs the Do operation and then inserts edges between the in-neighbors and out-neighbors of p. Specifically, for each pair of an in-neighbor p_{in} and an out-neighbor p_{out}, an edge (p_{in}, p_{out}) is inserted. Finally, for p_{in} whose out-degree exceeds the degree threshold θ_d, its out-edges is trimmed again by calling EdgeTrim($\cdot$) to ensure that the out-degree of p_{in} does not exceed θ_d. The deletion operation in FreshDiskAnn is also based on this idea, with the main distinction being the strategy of EdgeTrim($\cdot$).

Since HNSW uses a hierarchical structure, where a node can exist in multiple graphs across different levels, we apply the update operations at all levels containing the nodes to be deleted. To assess the performance of each deletion operation, we perform delete-and-reinsert quantification on the SIFT1M dataset with update rate 5%. To make HNSW and FreshDiskAnn have nearly the same accuracy before deletions, we set the degree threshold θ_d to 64 and the size of the dynamic candidate list *ef* to 200 in HNSW, and we set θ_d to 32 and the insert candidate list size L to 200 in FreshDiskAnn. This is because if the accuracies differ, this may affect the relative difficulty in maintaining query performance after updates. Regarding search parameters, we set the search candidate list size L_s to 100 for FreshDiskAnn. HNSW has no search parameters.

Fig. 2 shows that Pseudo achieves the best accuracy at the beginning. It outperforms Do and DwFC, and only falls behind FreshDiskANN after 27 rounds. This is because it retains the deleted

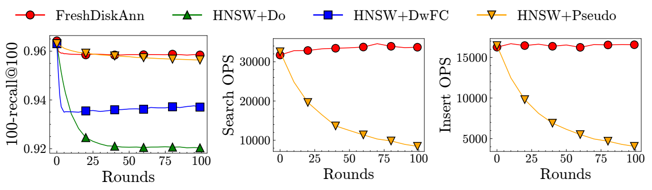


Figure 2: Performance of Update Operations

nodes and relevant edges in the index, so that it avoids damaging the monotonic search paths. However, the cost is that the index cannot shrink. During rounds of insertions and deletions, the index continuously expands, which increases the difficulty of finding the correct ANN search results. Therefore, Pseudo also suffers from the decline in accuracy. Due to the same reason, Search_OPS and Insert_OPS of Pseudo exhibit an obvious downward trend. After 100 rounds, both Search_OPS and Insert_OPS of Pseudo drop to about 25% of the values before updates. To handle the performance decline, periodic rebuilding is required.

For other update operations, both HNSW+Do and HNSW+DwFC exhibit notable decreases in accuracy after deletions. Specifically, the accuracy of HNSW+Do decreases sharply and eventually stabilizes at 0.92, which is lower than a stable accuracy of 0.937 achieved by HNSW+DwFC. Additionally, we observe a slight increase in recall of HNSW+DwFC after a sharp decline. Interestingly and counter-intuitively, while HNSW+DwFC attempts to compensate for connectivity losses by adding edges, it exhibits a faster accuracy decline than HNSW+Do in first few rounds. FreshDiskAnn, which adopts a similar strategy as HNSW+DwFC, also exhibits this behavior in the first 5 rounds. This suggests that indiscriminately adding edges to compensate for connectivity loss does not improve accuracy; instead, it can degrade performance in high-recall regions.

3.2 Insights into Performance Inefficiencies

We proceed to analyze the reasons why Do and DwFC fail to maintain consistent accuracy. Since FreshDiskAnn and DwFC employ similar approaches, we only have to consider DwFC.

We believe that the primary reason of this drop in accuracy is that the deletion operation in MSNET disrupts monotonic search paths. Since deleted nodes are often part of multiple monotonic search paths, their removal causes the destruction of many such paths, consequently lowering the quality of the index. Furthermore, existing graph indexes are constructed using approximate SNGs, which provide fewer monotonic search paths than exact SNGs do. Therefore, once a monotonic search path is destroyed, there are only few, if any, alternative monotonic search paths that lead to the same region. As a result, some nodes become unreachable during the greedy search, which renders it more difficult to implement update operations on the approximate SNG.

Since Do deletes all edges connected to node p, it destroys all monotonic search paths passing through p. While DwFC attempts to repair these paths by creating connections between all pairs of p_{in} and p_{out}. Such connections are not always effective, because there is no guarantee that p_{in} is close to p_{out}. In particular, p_{in} is usually close to p, and in the worst-case scenario, $d(p_{in}, p_{out}) = d(p_{in}, p) + d(p, p_{out})$. Following the short edge priority principle, the connection between p_{in} and p_{out} is likely to fail due to the use of EdgeTrim().

Moreover, after adding all edges (p_{in}, p_{out}), the degree of p_{in} may exceed θ_d and EdgeTrim(·) is called. Some edges that originally contribute to graph connectivity but violate the edge-trimming rule are removed, further diminishing the connectivity of the graph. This causes DwFC performing even worse than Do in the first few rounds. After removing edges, the number of edges in the graph decreases, making it less likely for adding edges (p_{in}, p_{out}) to trigger EdgeTrim(·). In this case, DwFC successfully adds the edges (p_{in}, p_{out}) and reconnects the monotonic search paths. Therefore, the accuracy of DwFC slightly increases after rapidly dropping to low accuracy. Although FreshDiskAnn mitigates this issue by relaxing the edge trimming criteria—allowing edges (p_{in}, p_{out}) to be retained more often—it still cannot fully restore all broken monotonic search paths. As a result, FreshDiskAnn experiences a drop in accuracy during the initial cycles.

The decrease in accuracy caused by the deletions makes it challenging to obtain an accurate candidate set during reinsertions. This implies although deleted nodes are reinserted, they cannot fully restore connectivity due to the inaccurate candidate set. Consequently, updates reduce the index accuracy. However, if a deletion operation can ensure that the graph's quality either remains unaffected or is reduced only slightly, a more accurate candidate set would result during subsequent insertions. This would help maintain or minimally affect the accuracy of the graph index after updates.

Therefore, designing a stable deletion algorithm becomes the crucial challenge for graph-based index update. A well-designed deletion algorithm must ensure that the index accuracy remains unaffected by updates. Consequently, Wolverine places emphasis on optimizing deletion operations, while insertion operations follow existing methods.

4 THE WOLVERINE ALGORITHMS

Wolverine algorithms can be integrated with existing graph-based ANN indexes to ensure accurate and efficient updates. The basic Wolverine algorithm repairs disrupted monotonic search paths by adding in-edges for the out-neighbors of the point to be deleted. To enhance efficiency, Wolverine+ restricts the search space to the 2-hop neighbors of a point to be deleted. Finally, Wolverine++ introduces a candidate selection policy to ensure that there are high-quality candidates within a reduced search space, thus improving both accuracy and efficiency.

4.1 Wolverine with ANN Search for all p_{out}

When a node is deleted, any monotonic search path that passes through this node is divided into two subpaths. Although the subpath following the deleted node remains monotonic, it becomes unreachable from the search starting node. In order to repair the disrupted monotonic search path, we re-construct monotonic search paths to the first node of this subpath. In addition, we make this node reachable from evenly distributed directions to repair as many disrupted monotonic search paths as possible.

Specifically, for a query node q, assume that there exists only one monotonic search path $MSP(s, \dots, p, \dots, q)$ monotonic to q that passes p that to be deleted. This path is split into two subpaths, $P(s, \dots, p_{in})$ and $P(p_{out}, \dots, q)$. The nodes on $P(s, \dots, p_{in})$ remain reachable from the starting node s, while the nodes on $P(p_{out}, \dots, q)$ become unreachable from s due to the disruption in connectivity at p_{out}. In such case, we call q is affected by the deletion. Therefore, it is intuitive to deduce the following lemma.

Lemma 4.1. *If $MSP(s, p_{out})$ of q is repaired, the monotonic search path $MSP(s, q)$ is repaired.*

Proof. Let $MSP(s, p_{out})$ be $MSP(v_1, \dots, v_k)$, where $v_1 = s$ and $v_k = p_{out}$. As $MSP(v_1, \dots, v_k)$ is monotonic to q, $d(v_1, q) > \dots > d(v_k, q)$. Let $P(p_{out}, q)$ be $P(u_1, \dots, u_m)$, where $u_1 = p_{out}$ and $u_m = q$. $P(u_1, \dots, u_m)$ is also monotonic to q, so $d(u_1, q) > \dots > d(u_b, q)$. Note that $v_k = u_1 = p_{out}$, therefore $d(v_1, q) > \dots > d(p_{out}, q) > \dots > d(u_m, q)$. According to the definition of the monotonic search path, the concatenation of $MSP(s, p_{out})$ and $P(p_{out}, q)$ form a monotonic search path $MSP(s, q)$ of q. □

Therefore, we believe that the key to repairing damaged monotonic search paths $MSP(s, q)$ lies in repairing the paths from the search starting node s to the nodes in $N_{out}(p)$ that are monotonic to q. Recall that in the insertion operation, the kANN search results are used as in-neighbors of the inserted node to establish monotonic search paths from the search starting node s to the inserted node. Drawing inspiration from this, we use the nodes in ANN search path of q after deleting p to establish $MSP(s, p_{out})$ of q. We denote the set of these nodes as $SP(q)$.

Lemma 4.2. *There exists a node $sp \in SP(q)$ such that $d(sp, q) > d(p_{out}, q)$. Connecting the edge (sp, p_{out}) repairs $MSP(s, q)$ of q.*

Proof. We first prove the existence of the node $sp \in SP(q)$ such that $d(sp, q) > d(p_{out}, q)$. After deleting p, the ANN search path includes the subpath $P(s, p_{in})$. For each node u on this subpath, we have $u \in SP(q)$ and $d(u, q) > d(p_{out}, q)$.

Then, we prove connecting (sp, p_{out}) repairs the monotonic search path. To facilitate the analysis, we denote the ANN search path after deleting p as $P(v_1, \dots, v_m)$. Assume one of its subpath is $p(v_1, \dots, sp)$, where $v_1 = s$ and sp is on $P(s, p_{in})$, we have $d(v_1, q) > \dots > d(sp, q)$. Since $d(sp, q) > d(p_{out}, q)$, it holds that $d(v_1, q) > \dots > d(sp, q) > d(p_{out}, q)$. Therefore, connecting (sp, p_{out}) repairs the monotonic search path $P(v_1, \dots, p_{out})$. According to lemma 4.1, $MSP(s, q)$ of q is hence repaired. □

According to Lemma 4.2, we can propose a method for repairing the damaged monotonic search paths. For each node q with $MSP(s, \dots, p, p_{out}, \dots, q)$ that passes through the edge (p, p_{out}) before deleting p, we take the set $SP(q)$ as the candidate set for the new in-neighbors of p_{out}. Then, for each $sp \in SP(q)$ that $d(sp, q) > d(p_{out}, q)$, we connect the edge (sp, p_{out}). During the process, for nodes whose out-degree exceeds the threshold θ_d, the EdgeTrim(·) function is applied to enforce the out-degree under θ_d.

This method guarantees that all monotonic search paths are repaired. However, it is often challenging to determine which nodes are affected. Identifying these nodes requires checking whether it is reachable by the ANN search starting from the search starting node s. Since deleting a node may affect multiple nodes, we need to search all nodes in the graph to identify which nodes are affected. This approach is undoubtedly highly inefficient. Therefore, based

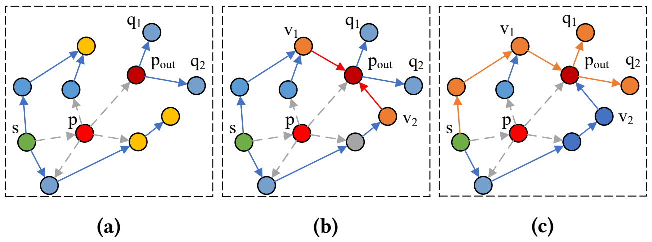


Figure 3: The process of Wolverine

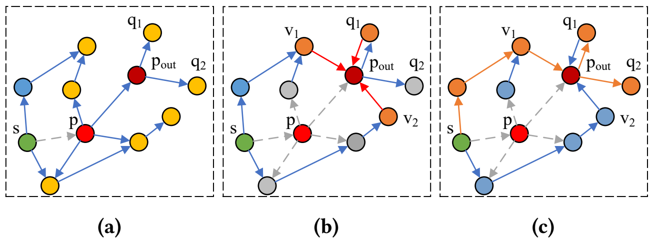


Figure 4: The process of Wolverine+

on this idea, we propose a practical method called Wolverine, which does not identify these nodes and uses the kANN search results of p_{out} to repair the monotonic search paths of p_{out}. Additionally, it tries to repair other $MSP(s, q)$ as much as possible by diversifying the directions of the added edges.

Specifically, Wolverine requires setting two parameters, δ_{in} and C_s, where δ_{in} represents the number of new in-neighbors to p_{out}, and C_s represents the size of the candidate set. Here, C_s is set to k, the size of the kANN search results. When deleting p, Wolverine first deletes the in-edges and out-edges of p. Then, for each p_{out}, we perform a kANN search for p_{out} from the search starting node s to obtain C_s nodes (yellow nodes in Fig. 3a). We take the these nodes as the candidates, denoted as C, and then employ the EdgeTrim($p_{out}, C \setminus \{p_{out}\}, \delta_{in}$) function, which returns the new in-neighbors set V (orange nodes in Fig. 3b). Then, we add an edge pointing to p_{out} for each node $v \in V$. For v whose degree exceeds the degree threshold, the EdgeTrim($v, N_{out}(v), \theta_d$) function is used again to limit their out-degree to the degree threshold.

We use kANN search results to repair the monotonic search paths of p_{out} for two reasons. First, to ensure that the degree of each node remains below the threshold θ_d, the EdgeTrim($\cdot$) function is applied to trim edges for nodes exceeding θ_d after adding edges. This function prioritizes shorter edges, making longer edges more likely to be removed. As a result, in the method derived from Lemma 4.2, the nodes in $SP(q)$ that successfully connect to p_{out} are those near p_{out}. Therefore, the ANN search results of p_{out} can cover these nodes and replace $SP(q)$ as the in-neighbors candidate set. Second, p_{out} can be directly retrieved from the edge table of p, eliminating the additional searches for identifying and significantly improving repair efficiency.

Subsequently, we also need to ensure that the added edges monotonic to the affected nodes as much as possible. Intuitively, we assume that the affected nodes are evenly distributed in all directions around each p_{out}. To maximize the chances of the repaired edge being monotonic to as many nodes as possible, we aim for the newly added in-edges of p_{out} point to all directions. Therefore, we need to perform kANN search for p_{out} to obtain a sufficiently large candidate set that contain nodes from diverse directions. Then, we select nodes in all directions to add edges pointing to p_{out}. Since the EdgeTrim($\cdot$) function can select nodes from all directions of p_{out} uniformly from the candidate set, we use the EdgeTrim($\cdot$) function to filter the candidate set to obtain nodes connected to p_{out}.

The study [28] proves that the ANN search time complexity is at least $O(n^{\frac{2}{\theta_d}} \ln n)$ in graph. So, the time complexity of Wolverine is $O(\theta_d + \theta_d n^{\frac{2}{\theta_d}} \ln n + \delta_{in}\theta_d)$, depending on the number of graph nodes, which may cause Wolverine inefficient on large datasets.

4.2 Wolverine+ with 2-Hop Neighbors

Wolverine employs ANN search for each out-neighbor of the node to be deleted to form the candidate set. However, ANN search across multiple out-neighbors (often dozens) is time-consuming, leading to inefficient updates. To address this, we introduce Wolverine+, which obtains the candidate set more efficiently.

Intuitively, we aim to quickly obtain an alternative candidate set that is similar to the kANN search results. In Wolverine, the kANN search results for each p_{out} are naturally close to p_{out}. Since p is near p_{out}, so are p's neighbors. Therefore, Wolverine+ considers directly using p's 2-hop neighbors, denoted as $Hop_2(p)$, to form the candidate set, allowing for rapid retrieval. Compared to Wolverine, this approach significantly reduces the number of nodes to be checked. We next explain the reasons behind this choice.

First, inspired from Wolverine, the idea of using $Hop_2(p_{out})$ as the candidate set naturally emerges. Note that, bidirectional edges are added between $N_{out}(p_{out})$ and p_{out} when p_{out} is inserted into the graph. Consequently, the nodes in $Hop_2(p_{out})$ are already likely to reach p_{out}. Therefore, adding edges from the nodes in $Hop_2(p_{out})$ to p_{out} is redundant and does not help to repair $MSP(s, p_{out})$. Instead, we focus on utilizing $Hop_2(p)$. Second, since $Hop_2(p)$ is close to all p_{out}, it can be considered as the candidate set of all p_{out}. In this way, we only need to obtain the candidate set once. In contrast, Wolverine obtains the candidate set multiple times. Therefore, Wolverine+ effectively replaces the kANN search results.

Wolverine+ and Wolverine use the same parameters δ_{in} and C_s, and a similar deletion process. However, it is worth noting that we delete the in-edges and retain the out-edges of p at first. This is because we need these out-edges to obtain $Hop_2(p)$. In addition, since C_s may be smaller than $|Hop_2(p)|$, we randomly select C_s nodes as candidates in this case. Then, same as Wolverine does, we call EdgeTrim($\cdot$) for each p_{out} and each new in-neighbor whose out-degree exceeds θ_d. At last, the out-edges of p and p itself are deleted. Fig. 4 shows an example of Wolverine+.

By limiting the candidate set to 2-hop neighbors, Wolverine+ reduces the number of nodes that need to check considerably. The time complexity of Wolverine+ is $O(\theta_d + \min({\theta_d}^2 + \theta_d, C_s) + \delta_{in}\theta_d)$. As the time complexity of Wolverine+ is independent of the dataset size n, it achieves robust performance even on large datasets.

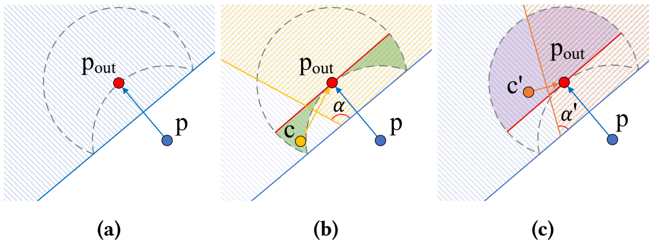


Figure 5: The region surrounded by the gray line is $\{x|d(x,p_{out}) < d(p,p_{out}) \wedge d(x,p) > d(p,p_{out})\}$. The hatched blue region in 5a is $MSR(p,p_{out})$. The hatched yellow region in 5b is the overlapping region of $MSR(p,p_{out})$ and $MSR(c,p_{out})$. The hatched orange region in 5c is the overlapping region of $MSR(p,p_{out})$ and $MSR(c',p_{out})$.

4.3 Analysis of Candidate Sets

While Wolverine+ reduces the number of nodes to check, it still examines too many. For a graph with $\theta_d = 64$, $Hop_2(p)$ could include as many as ${\theta_d}^2 = 4096$ nodes, which is excessive for a candidate set. Furthermore, since Wolverine+ randomly selects C_s nodes from this candidate set, it may overlook valuable candidates that could help repair the monotonic search path. To further refine the candidate set, the goal is to select high-quality candidates that effectively repair the monotonic search paths from $Hop_2(p)$. We define a high-quality candidate as one that meets three key conditions.

Ensure Successful Connection. We aim for the repaired edges to be effectively connected. According to the short edge priority principle, the EdgeTrim(·) function gives priority to retaining shorter edges. If an edge (c,p_{out}) is long, it is more likely to be trimmed. Therefore, we strive for $c \in C$ to be as close to p_{out} as possible. To achieve this, we constrain the candidate set to a sphere centered at p_{out} with a radius $d(p,p_{out})$. That is, $\forall c \in C$, $d(c,p_{out}) < d(p,p_{out})$.

Stay Untouched in Deletion. We aim for the nodes in the candidate set to be minimally affected by the deletion. If we select nodes that are significantly affected by the deletion of p as candidates, these nodes may not be reachable from s. Consequently, connecting these nodes to p_{out} may still leave p_{out} unreachable from s. Therefore, we focus on selecting nodes that are less affected by the deletion of p. We posit that nodes farther from p are less likely to be influenced by its deletion. After p is deleted, these distant nodes are more likely to maintain the monotonic search paths originating from s and are also more likely to be discovered by the greedy search algorithm. By connecting these nodes to p_{out}, we increase the probability of p_{out} being reached, thereby repairing the monotonic search path $MSP(s,p_{out})$. Thus, we require that for all $c \in C$, $d(c,p) > d(p,p_{out})$.

One Repair, Multiple Benefits. We aim for the repaired edges to restore multiple monotonic search paths. When node p is deleted, all monotonic search paths that pass through the edge (p,p_{out}) are disrupted due to the removal of this edge. To enable the newly connected edges to repair these disrupted paths, we seek to ensure that (c,p_{out}) is monotonic with respect to the endpoints of these paths. In previous approaches, we accomplished this by having (c,p_{out}) point to all directions. However, we demonstrate that this goal can be achieved with nodes located within a smaller region.

We aim for the added edge (c,p_{out}) to repair $MSP(s,q)$ for as many affected nodes q as possible. However, since q may be located at an arbitrary position with unknown paths leading to it, it is complicated to analyze the sufficient conditions, i.e., adding (c,p_{out}) ensures the repair of $MSP(s,q)$. Therefore, we employ the necessary conditions instead. In other words, we are interested in that, if an edge (c,p_{out}) can repair $MSP(s,q)$ ((c,p_{out}) belongs to $MSP(s,q)$), in what region p necessarily locates in. We call this region monotonic search region (MSR) w.r.t. (c,p_{out}), denoted as $MSR(c,p_{out})$. Assume that the nodes are evenly distributed, the area of $MSR(c,p_{out})$ that contains nodes reflects the posterior probability of that adding (c,p_{out}) repairs $MSP(s,q)$. Sequentially, we hope that $MSR(c,p_{out})$ and $MSR(p,p_{out})$ overlap as much as possible, so that a single (c,p_{out}) can compensate more for the deletion of (p,p_{out}). We formally define MSR as follows.

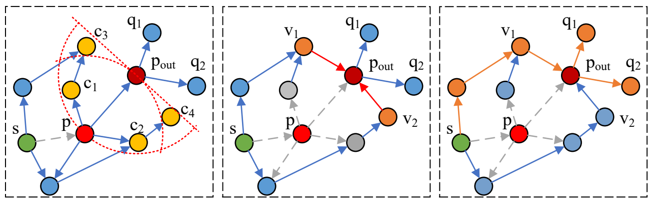
Definition 4.3 (monotonic search region). Given any directed edge (p,p_{out}), the monotonic search region $MSR(p,p_{out})$ of the directed edge is defined as $MSR(p,p_{out}) = \{x|d(x,p_{out}) < d(x,p)\}$.

When $MSP(s,q)$ contains (p,p_{out}), according to the nature of MSP, we have $d(q,p_{out}) < d(q,p)$. Therefore, $q \in MSR(p,p_{out})$ is an necessary condition. Figs. 5a illustrates an example of MSR in 2D space. The blue line is the perpendicular bisector of (p,p_{out}), and $MSR(p,p_{out})$ is the region above it (represented by hatched blue). For any node x in the region, it holds that $d(x,p_{out})) < d(x,p)$.

Consider Conditions Together. With the three proposed conditions, we proceed to introduce what region simultaneously satisfies these conditions. Take Fig. 5 as an example. The gray dashed outline belongs to two circles that are centered at p_{in} and p_{out} with radii of $d(p_{in},p_{out})$, respectively. Therefore, the crescent region represents $\{x|d(x,p_{out}) < d(p,p_{out}) \wedge d(x,p) > d(p,p_{out})\}$. All nodes in this crescent region satisfy the first two conditions, i.e., ensure successful connection and stay untouched in deletion.

Regarding the third condition, we hope that $MSR(c,p_{out})$ and $MSR(p,p_{out})$ overlap as much as possible. We divide the crescent region into two parts by the red line in Fig. 5b, which is parallel to the perpendicular bisector (blue line) of (p,p_{out}), and we consider the nodes in the region below the red line (the highlighted solid green region in Fig. 5b) to be suitable candidates. Fig. 5b illustrates the situation where c locates in the region. Similarly, we obtain $MSR(c,p_{out})$ and represent $MSR(c,p_{out}) \cap MSR(p,p_{out})$ by hatched yellow. In this case, since the angle α between the perpendicular bisectors of (c,p_{out}) and that of (p,p_{out}) is greater than $90°$, the overlapping region occupies at least 50% of $MSR(p,p_{out})$. In turn, (c,p_{out}) can compensate for above 50% nodes that are affected by the deletion of (p,p_{out}). In contrast, in Fig. 5c where c locates in the upper part (the highlighted solid purple region) and the angle α is less than $90°$, the overlapping ratio is lower than 50%. It means that (c,p_{out}) can only compensate for fewer affected nodes. Therefore, we consider nodes in the solid green region in Fig. 5b as high quality candidates that can satisfy the three conditions simultaneously.

A high quality candidate c can be distinguished totally based on distances. For the first condition, we have $d(c,p) > d(p,p_{out})$. For the second condition, we have $d(c,p_{out}) < d(p,p_{out})$. For the third condition, we leverage the law of cosines. When c locates in the



(a) (b) (c)

Figure 6: The process of Wolverine++

solid yellow area in 5b, the angle $cp_{out}p$ is an acute angle. Therefore, it holds that $d^2(c, p_{out}) + d^2(p, p_{out}) > d^2(c, p)$. In contrast, when c locates in the solid orange area in Fig. 5c, the angle $cp_{out}p$ is no more an acute angle. Although the square of the distance is used here, the existing algorithm uses the square of the Euclidean distance instead of the accurate Euclidean distance when implementing it. Therefore, in Euclidean space, this judgment method does not generate additional calculations.

Based on the above analysis, we define the candidate region w.r.t. (p, p_{out}) that contains high quality candidates as $CR(p, p_{out}) = \{x | d(x, p) > d(p, p_{out}) \wedge d(x, p_{out}) < d(p, p_{out}) \wedge d^2(x, p_{out}) + d^2(p, p_{out}) > d^2(x, p)\}$. The candidate set C should satisfy that $C \subset CR(p, p_{out})$.

4.4 Wolverine++

We propose Wolverine++ based on the above analysis of the candidate set. When searching for candidates connected to p_{out}, Wolverine++ quickly obtains a high-quality candidate set. Since Wolverine++ requires only a few additional edges to maintain accuracy, it efficiently restores the connectivity of the graph index after deletions, ensuring stable performance.

The deletion operations of Wolverine+ and Wolverine++ are almost the same except that Wolverine++ further filters out nodes in $Hop_2(p)$. For each p_{out} in $N_{out}(p)$, the candidate set C is initialized by the nodes from $N_{out}(p)$ whose distance to p_{out} is smaller than $d(p, p_{out})$. In this initialization step, we do not require the nodes added to C to be strictly located in $CR(p, p_{out})$, because there are only a few nodes in $N_{out}(p)$ that are located in the region. In order to obtain enough candidates, we relax the conditions for adding to the candidate set in this step. Therefore, at least p_{out} itself can be added to the candidate set. Next follows the expansion step, where we expand the nodes in the candidate set C. For each node c in C, we add the nodes in $N_{out}(c)$ that locate in $CR(p, p_{out})$ to C. The process terminates when the size of the candidate set reaches C_s. Fig. 6 shows the process of Wolverine++.

Specially, when $CR(p, p_{out})$ contains no nodes, Wolverine++ does not add in-edges for p_{out}. Because expanding the candidate region or searching more nodes to obtain candidates may introduce inappropriate edges and reduce efficiency. Since such p_{out} typically lies in sparse regions, even if the monotonic search paths are not repaired, Wolverine++ yields satisfactory performance.

The time complexity of Wolverine++ is $O(\theta_d + \theta_d \times \min({\theta_d}^2 + \theta_d, C_s) + \delta_{in}\theta_d)$. Although Wolverine++ needs to obtain a candidate set for each p_{out} and cannot obtain the candidate set by one pass, it selects candidate sets from smaller regions. Therefore, Wolverine++ still reduces the time required. Furthermore, compared to Wolverine+ that is prone to missing useful candidates due to early termination, Wolverine++ is less likely to encounter this issue due to its reduced candidate region. This makes it more effective at identifying and retaining useful candidates.

In scenarios that require high-frequent updates, Wolverine++ needs to perform deletions in batches. Directly employing multiple threads to perform deletions in parallel leads to competition, because the edge table of a node may be modified by multiple threads of different deletions. To achieve efficient multi-thread concurrency control, Wolverine++ performs batch deletion in three stages. In the first stage, we remove the in-edges of each node in a batch. Each node is processed by a single thread, which removes the edges connected to the deleted nodes. Although this approach involves checking each edge table, the time spent is minimal due to the elimination of thread contention. Furthermore, by ensuring that no multiple threads modify an edge table simultaneously, we avoid thread competition, making the deletion of in-edges more efficient. In the second stage, for each deleted node p, we use one thread to repair the monotonic search path $MSP(s, p_{out})$ of its out-neighbors with Wolverine++. In the third stage, we release the memory resources occupied by deleted nodes.

5 EVALUATION

We report on extensive experiments on real-world datasets that address the following questions:

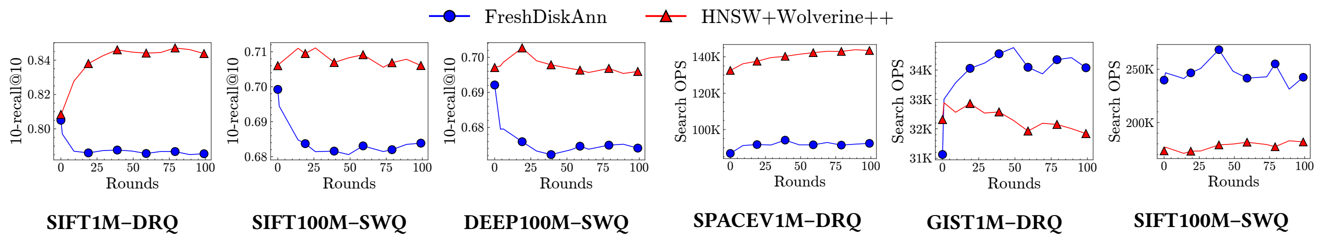
(Q1): How does Wolverine++ compare with the SOTA baselines?
(Q2): How does the update rate UR affect the performance?
(Q3): How does the value k of kANN affect the performance?
(Q4): Does Wolverine++ generalize to all SNG-based algorithm?
(Q5): How does Wolverine++ perform compared to the two basic algorithms, Wolverine and Wolverine+?
(Q6): How to properly configure Wolverine++?
(Q7): How many repaired edges are visited during query?

5.1 Experimental Setup

Datasets. We use 9 public real-world datasets to evaluate Wolverine algorithms, where SIFT1M, SIFT10M, SIFT100M, and GIST1M [21] are image vector dataset, SPACEV1M and SPACEV10M are sampled from SPACEV1B [27], released by Microsoft, and DEEP1M, DEEP10M, and DEEP100M are sampled from DEEP1B [3]. Statistics of the datasets are shown in Table 1.

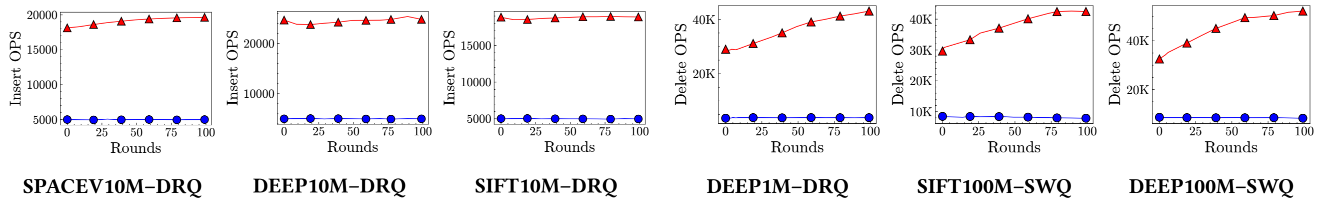
Table 1: Dataset Statistics

Dataset	Dim.	# of points	# of queries
SIFT1M	128	1,000,000	10,000
GIST1M	960	1,000,000	1,000
SPACEV1M	100	1,000,000	29,316
DEEP1M	96	1,000,000	10,000
SIFT10M	128	10,000,000	10,000
SPACEV10M	100	10,000,000	29,316
DEEP10M	96	10,000,000	10,000
SIFT100M	128	100,000,000	10,000
DEEP100M	96	100,000,000	10,000



(a) Query Accuracy (10-recall@10)

(b) Query Efficiency (Search_OPS)



(c) Insert Efficiency (Insert_OPS)

(d) Deletion Efficiency (Delete_OPS)

Figure 7: FreshDiskAnn vs. Wolverine++ under DRQ and SWQ

Machine Configuration. All experiments are conducted on a machine with an AMD EPYC 7K62 48-Core CPU and 256GB of memory. We use 64 threads to update the graph index in parallel. All experiments are performed in-memory.

Baseline. We compare Wolverine++ with FreshDiskAnn[30], the SOTA graph-based dynamic indexing algorithm. In experiments, we use the in-memory version of FreshDiskAnn for fair comparison. We configure FreshDiskAnn by setting the degree threshold θ_d to 32 and the insert candidate list size L to 200. For search, FreshDiskAnn sets the search candidate list size L_s to 10 for 10-recall@10.

Update Performance Quantification Strategies. We use the two quantification strategies presented in Sec. 2.3 to evaluate the update performance of Wolverine++. Compared to DRQ, we choose a larger dataset for SWQ to ensure sufficient new data for insertion in each round. However, we construct the graph index with the same number of vectors in DRQ and SWQ to avoid the influence caused by different data scales.

5.2 Performance Overview (Q1)

We compare Wolverine++ with FreshDiskAnn under DRQ and SWQ to examine the performance of update operations. For both DRQ and SWQ, we set $UR = 1\%$, and for SWQ, we set $w = 10M$. We observe that Wolverine++ consistently outperforms FreshDiskAnn across all datasets in terms of the metrics of 10-recall@10, Insert_OPS, and Delete_OPS. For Search_OPS, Wolverine++ outperforms FreshDiskAnn in most datasets.

5.2.1 *Wolverine Implementation.*

Wolverine++ adopts the insertion operation from HNSW. We refer to the implementation of Wolverine++ on HNSW as HNSW+Wolverine++. We adjust the graph construction parameters of HNSW according to different datasets. With these parameter settings, the recall gap between FreshDiskAnn and HNSW for the same query is less than 0.01 when no updates have been made. Since the graph is constructed with insertions, insertions share the same parameters as graph construction. For Wolverine++, we set the threshold number of new edges δ_{in} to 32, and always set candidate set size Cs to $2 \times \delta_{in}$.

5.2.2 *Query Accuracy and Efficiency.*

We investigate query accuracy and efficiency during updates using both DRQ and SWQ.

High Query Accuracy. Fig. 7a presents the query accuracy of the methods under DRQ and SWQ performance quantification strategies. We use 10-recall@10 as the evaluation metric. It is evident that HNSW+Wolverine++ consistently achieves higher query accuracy compared to FreshDiskAnn, due to its ability to repair monotonic search paths disrupted by deletions. In most cases, the recall of HNSW+Wolverine++ gradually increases as the connectivity of the graph index is less impacted by deletions, allowing newly inserted points to still be accurately located. Even in the worst-case scenarios, HNSW+Wolverine++ maintains relatively stable recall performance. The increase in recall is due to the increase in the average degree of the graph after Wolverine++ updates. During graph construction, to avoid frequent edge trimming, when a node's degree exceeds θ_d, HNSW calls the EdgeTrim$(\cdot)$ function to reduce a node's degree to less than $\theta_d/2$, which lowers the average degree of initial graph. In contrast, Wolverine++ only ensures that the degree remains below θ_d when calling the EdgeTrim$(\cdot)$ function. As a result, the average degree of the graph increases after updates with Wolverine++, enhancing graph connectivity. The increase in average degree improves recall but also increases search latency. However, this trade-off between recall and search efficiency can be controlled by adjusting the number δ_{in} of new in-neighbors in Wolverine++. The details of the trade-off analysis are presented in Sec. 5.7. Although FreshDiskAnn proposes a repair strategy by fully connecting the in-neighbors and out-neighbors of deleted points, this approach does not effectively repair the monotonic search paths.

Stable Query Efficiency. Fig. 7b illustrates the query efficiency of methods under DRQ and SWQ. We use Search_OPS as the evaluation metric. The Search_OPS for HNSW+Wolverine++ remains

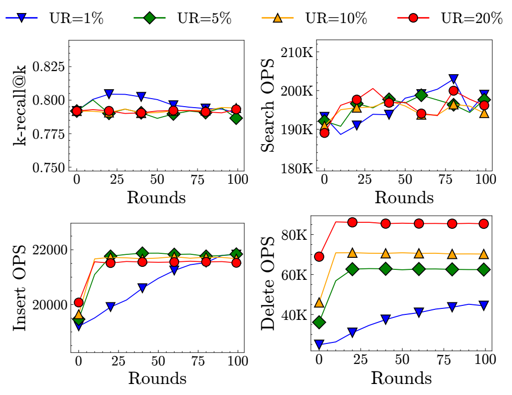


Figure 8: Comparison on Different Update Rates

stable in the experiment. Even under SWQ, which closely approximates real-world update scenarios, Search_OPS stays consistent at the level before updates. On small datasets, HNSW+Wolverine++ is 1.4 ~ 1.7× faster than FreshDiskAnn. In constrast, it is slightly slower on the ultra-high-dimensional dataset GIST1M. On large datasets, HNSW+Wolverine++ is also slower than FreshDiskAnn. However, since Wolverine++ does not optimize the query processing, Search_OPS primarily depends on the index itself, so we only have to focus on the stability of the query efficiency after updates, which is sufficiently verified by the experimental results.

5.2.3 Update Efficiency. We investigate insertion and deletion efficiency during updates using both DRQ and SWQ.

Insertion Efficiency. Fig. 7c shows the insertion efficiency of the methods under DRQ and SWQ. We use Insert_OPS as the evaluation metric. Although HNSW+Wolverine++ and FreshDiskAnn use similar insertion methods, Insert_OPS of HNSW+Wolverine++ is 3.6 ~ 4.8× greater than that of FreshDiskAnn. This is primarily due to FreshDiskAnn relaxing its edge trimming condition. As a result, during insertions, FreshDiskAnn needs to connect more edges between the inserted nodes and their neighbors. This requires modifying the edge tables of a large number of nodes. Moreover, modifying a greater number of edge tables introduces more thread contention, which further degrades FreshDiskAnn's performance in multi-threaded processing.

Deletion Efficiency. Fig. 7d shows the deletion efficiency of the methods under DRQ and SWQ. We use Delete_OPS as the evaluation metric. We can see that HNSW+Wolverine++ achieves a significantly higher and more stable deletion throughput, with Delete_OPS being 2.7 ~ 11.0× higher than FreshDiskAnn. Additionally, as updates progress, the Delete_OPS of HNSW+Wolverine++ gradually increases. In contrast, the Delete_OPS of FreshDiskAnn remains consistently lower. This is due to FreshDiskAnn's approach of checking all edges, identifying in-neighbors for deleted nodes, and immediately repairing out-neighbors by connecting in-neighbors to them. If the out-degree of the in-neighbors exceeds a threshold, trimming is triggered. Since FreshDiskAnn generally does not limit the number of in-neighbors, this can lead to excessive trimming, lowering deletion efficiency. On the other hand, while Wolverine++ also checks all edges, it only deletes and then selects δ_{in} points to repair the out-neighbors of the deleted points, requiring significantly fewer trimming operations.

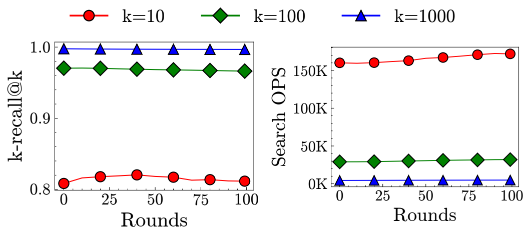


Figure 9: Impact of k

5.3 Impact of Update Rate (Q2)

In order to verify the effects of Wolverine++ for different update rates, we used two quantification methods, DRQ and SWQ, to evaluate the performance of HNSW+Wolverine++ on DEEP1M and DEEP100M. We set the update rates of DRQ and SWQ as 1%, 5%, 10%, and 20%, and we set $w = 1M$ for SWQ. We believe that such experimental settings can cover most application scenarios. We set δ_{in} to 24 for Wolverine++. Since both quantification methods exhibit the same trend, we only present the results of SWQ.

Fig. 8 shows the performance of Wolverine++ under different update rates in terms of recall, Search_OPS, Delete_OPS, and Insert_OPS. The recall of Wolverine++ remains stable across all experiments. Search_OPS also remains stable with a slight increase. This shows that Wolverine++ is able to avoid the deterioration of recall and Search_OPS with different update rates. For updates efficiency, Insert_OPS of Wolverine++ increases after updates and stabilizes at a higher level. For Delete_OPS, higher values are observed under larger update rates. This is due to the higher throughput of the first stage of deletion at higher update rates. A larger update rate means deleting more nodes in the batch. In the first stage, regardless of the number of deleted nodes, we scan all nodes in graph to remove the in-edges of the deleted nodes. The time overhead of this stage is only dependent on the number of nodes in the index. Therefore, for this stage, the more nodes are deleted in a batch, the same amount of time is consumed, resulting in higher throughput. Therefore, Wolverine++ performs better under higher update rates.

5.4 Impact of the Value of k (Q3)

To evaluate the performance of Wolverine++ on the kANN problem with different values of k, we use DRQ to test the performance of HNSW+Wolverine++ on SIFT1M. We set k to 10, 100, and 1000, and set δ_{in} to 20. We focus on recall and Search_OPS and do not observe Delete_OPS and Insert_OPS, as these metrics do not exhibit changes with variations in k.

Fig. 9 indicates that Wolverine++ avoids the decline in recall and Search_OPS for different kANN queries. For larger values of k, the accuracy remains stable after updates, as the index accuracy before update is already very high. Regarding Search_OPS, larger values of k typically require examining more nodes, resulting in lower Search_OPS for $k = 100$ and $k = 1000$ compared to $k = 10$.

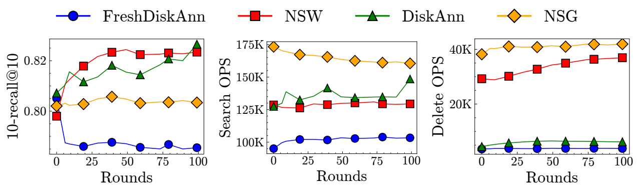


Figure 10: Other Methods vs. FreshDiskAnn

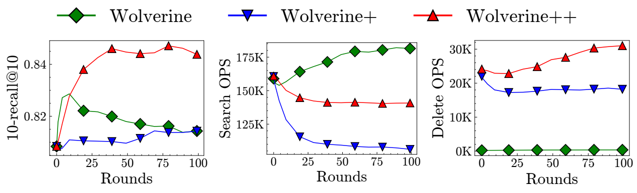


Figure 11: Comparison on Wolverine Algorithms

5.5 Algorithm Generalization (Q4)

Since Wolverine++ is not optimized for a specific graph index, it can be applied to any graph index that meets the requirements of SNG and supports insertion operations. To verify the generalization of Wolverine, we implement other graph indexs, NSW [25], NSG [11], and DiskAnn [31], in conjunction with Wolverine++. We use DRQ with update rate of 1% to evaluate the performance of the three methods and FreshDiskAnn on SIFT1M. Fig. 10 shows the performance of these methods in terms of query accuracy, query efficiency, and deletion efficiency with updates. The three methods with Wolverine++ outperform FreshDiskAnn in all metrics. The accuracy remains stable or improves after updates, and Search_OPS remains steady. We observe a slight decline in the Search_OPS of NSG with Wolverine++, dropping to 92% of its initial value by the 100th round. This is because iterative construction considers the dataset more comprehensively, resulting in shorter monotonic search paths constructed by iteration compared to those constructed by insertions. As more portions in NSG are constructed by insertion rather than iteration, the ANN search paths become longer.

We remark that Delete_OPS of DiskAnn+Wolverine++ is relatively low, only slightly higher than that of FreshDiskAnn. This is due to the relaxing edge trimming condition of DiskAnn, resulting in deleted nodes having more 2-hop neighbors, which leads to more candidate nodes within the candidate region. This increases the time overhead for filtering the candidate set and establishing edges.

Nevertheless, the three methods with Wolverine++ achieves performance comparable to that of HNSW+Wolverine++ and outperforms FreshDiskAnn across all metrics. This experiment demonstrates that Wolverine++ exhibits good generalization capabilities and can be effectively extended to other SNG-based graph indexes.

5.6 Comparison on Wolverine Algorithms (Q5)

In this experiment, we employ the DRQ performance quantification strategy with update rate 1% on the SIFT1M dataset to analyze the performance differences among the three algorithms we proposed: Wolverine, Wolverine+, and Wolverine++. All three algorithms are applied to HNSW using the same construction parameters and the same δ_{in} of 32. The experimental results indicate that, when considering all metrics comprehensively, Wolverine++ achieves a strong balance in performance.

Fig. 11 shows the changes in recall, Search_OPS, and Delete_OPS of the three proposed algorithms during updates. The accuracy of HNSW+Wolverine increases rapidly during the first few experiments, slightly declines after about 10 rounds, and stabilizes above the initial accuracy after about 20 rounds, indicating that the Wolverine algorithm effectively repairs the graph after deletion. However, Wolverine exhibits the lowest Delete_OPS among the three algorithms, aligning with our earlier analysis that its reliance on ANNS results for candidates is too time-consuming. In contrast, Wolverine+, which uses $Hop_2(p)$ as the candidate set, demonstrates a significantly higher Delete_OPS, highlighting that this approach accelerates candidate selection. However, its accuracy does not match that of Wolverine. This reflects our previous conclusion that Wolverine+ has an overly broad candidate range and may miss effective points in $Hop_2(p)$ for repairing monotonic search paths. Wolverine++, with its refined candidate selection, avoids the accuracy decline observed in Wolverine+. It shows notable improvements in accuracy, even outperforming Wolverine. Besides, Wolverine++ achieves higher Delete_OPS, which shows an upward trend. As for Search_OPS, Wolverine++ still performs well. In the first few rounds, the Search_OPS of HNSW+Wolverine++ is higher than that of HNSW+Wolverine+. This indicates that the monotonic search paths repaired by Wolverine++ are shorter than those of Wolverine+, further demonstrating the effectiveness of the candidate selection strategy of Wolverine++. The success of Wolverine++ in all metrics verifies the effectiveness of selected candidate regions, enabling fast graph updates while maintaining precision.

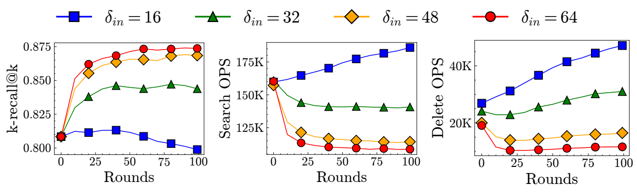


Figure 12: Parameter Study: δ_{in}

5.7 Parameter Study (Q6)

In this experiment, we investigate the proper parameter configuration for Wolverine++ to achieve the best performance. The results indicate that adjusting the parameter, δ_{in}, enables us to strike a favorable balance between deletion efficiency and search accuracy. The parameter δ_{in} represents the threshold number of new edges allowed, which corresponds to the neighbors that can be added when repairing the out-neighbors of the deleted points. In this experiment, we employ the DRQ performance quantification strategy with update rate 1% on SIFT1M, setting the construction parameters for HNSW to a degree threshold of $\theta_d = 64$ and a dynamic candidate list size of $ef = 200$. In Fig. 12, we examine the impact of varying δ_{in} on query accuracy and deletion efficiency, ranging from an upper limit of 16 neighbors to 64 neighbors. When δ_{in} is low (e.g., $\delta_{in} = 16$), recall diminishes as updates progress, as the

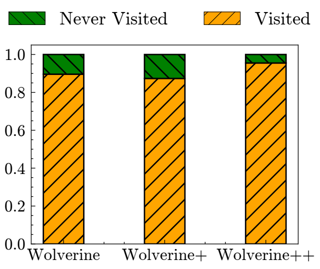


Figure 13: Proportion of Repaired Edges Visited

limited number of neighbors may insufficiently repair the disrupted monotonic search paths. Conversely, a higher δ_{in} allows for more effective repairs of monotonic search paths. While increasing δ_{in} leads to improved recall after updates, it also results in decreased deletion efficiency and query efficiency. This is because that, with higher δ_{in}, the more edges are added into the graph. This increases the average out-degree of the index. It expands the candidate set for Wolverine++ when repairing monotonic search paths while also increasing the search time complexity. Notably, at $\delta_{in} = 32$, recall consistently improves with updates, and deletion efficiency remains stable. Consequently, we select 32 as the default δ_{in} for Wolverine++.

5.8 Proportion of Repaired Edges Visited (Q7)

In this experiment, we test the proportion of repaired edges visited during queries after deletion for the three methods of the Wolverine family. The results indicate that the edges repaired by Wolverine are necessary for maintaining the accuracy of the index. We conduct the experiment on NSW built on SIFT1M. In this experiment, we first randomly delete 1% of the nodes using Wolverine, then perform the same queries as in the previous experiment, recording the repaired edges visited in each query. The experimental results are shown in the Fig. 13. Specifically, the proportions of repaired edges used in queries for Wolverine, Wolverine+, and Wolverine++ methods are 89.6%, 87.3%, and 95.5%, respectively. This indicates that the most of repaired edges are necessary, highlighting the effectiveness of Wolverine family. We observe that Wolverine+ has the lowest proportion, which is due to its broad candidate set obtaining strategy that connects too many unnecessary edges. In contrast, Wolverine++ obtains a significantly more accurate candidate set through candidate region filtering, demonstrating the success of Wolverine++ candidate obtaining strategy.

6 RELATED WORK

6.1 Static ANN Search Algorithms

Existing static ANNS algorithms can be mainly divided into 4 categories, namely tree-based [4, 5, 7, 29, 37], LSH-based [12, 18, 19, 23, 33, 35, 36, 40], quantization-based [13, 14, 17, 20, 22, 39], and graph-based [10, 11, 15, 16, 24–26, 32] methods. Graph-based methods are regarded widely as the most promising approach to solving the ANN search problem [34], with proposals including HNSW [26], NSG [11], and FANNG [16]. HNSW [26] approximates Delaunay Graph (DG) and Relative Neighbor Graph (RNG), employing a hierarchical structure with exponentially decreasing nodes from the bottom up. In each layer, a graph is constructed, with edges at higher levels functioning as shortcuts. NSG [11] proposes a monotonic RNG, relaxing the edge trimming strategy of RNG to create more monotonic paths. To reduce construction overhead, it utilizes an approximate kNN graph to limit the number of in-neighbor candidates. FANNG [16] approximates a sparse neighbor graph (SNG) by initializing a graph with all nodes but no edges, iteratively enhancing connectivity. During each iteration, two randomly selected nodes are connected with an edge if one cannot be reached via greedy search from the other.

6.2 Dynamic ANN Search Algorithms

Among the studies on the dynamic ANNS problem, FreshDiskAnn [30] and SPFresh [38] stand out as the most prominent approaches.

FreshDiskAnn [30] is a graph-based method that supports complete update operations, including both insertion and deletion. It introduces a novel proximity graph known as FreshVamana, which is constructed by approximating the α-RNG. This method compensates for connectivity loss in a fully connected manner. Due to its relaxed edge trimming policy, FreshDiskAnn minimizes the risk of newly added edges being trimmed, thereby maintaining the accuracy of the updated index. However, its accuracy tends to decrease slightly during the initial updates before stabilizing at a lower level compared to the original index. Additionally, the update efficiency of FreshDiskAnn is hindered by its fully connected strategy, as the in-degree of the deleted node can be unlimited and potentially comparable to the index size in the worst-case scenario.

SPFresh [38] combines proximity graphs with clustering to construct the index. It partitions vectors through clustering and builds a proximity graph for the centroids. During searching, it navigates to multiple nearest centroids via the proximity graph and evaluates the vectors in the corresponding clusters. For updates, SPFresh updates the partition as needed, enabling efficient updates. However, it struggles to achieve high accuracy due to its coarse-grained partitioning, which requires scanning numerous nearby clusters to ensure precision. Additionally, since SPFresh is not a graph-based algorithm, we do not include it in the comparison with Wolverine.

7 CONCLUSION

We propose Wolverine for the dynamic ANN search problem. First, Wolverine repairs disrupted monotonic search paths by adding in-edges for the out-neighbors of points to be deleted. Second, Wolverine+ improves efficiency by restricting the search space to be within the 2-hop neighbors of the point to be deleted. Third, Wolverine++ employs a sophisticated candidate selection policy to ensure high-quality candidates within the reduced search space, improving both accuracy and efficiency. Extensive experiments offer evidence that compared to FreshDiskAnn, Wolverine++ is capable of improving the deletion throughput by up to 11× and achieves more stable recall during updates.

ACKNOWLEDGMENTS

This work was supported in part by NSFC Grant No. 62372194.

REFERENCES

[1] 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. *PVLDB.* 11, 8 (2018), 906–919.

[2] Sunil Arya and David M. Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *SODA.* 271–280.

[3] Artem Babenko and Victor S. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *CVPR.* 2055–2063.

[4] 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 *SIGMOD.* 322–331.

[5] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[6] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn J. Keogh. 2010. iSAX 2.0: Indexing and Mining One Billion Time Series. In *ICDM.* 58–67.

[7] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. *PVLDB.*, 426–435.

[8] Kunal Dahiya, Deepak Saini, Anshul Mittal, Ankush Shaw, Kushal Dave, Akshay Soni, Himanshu Jain, Sumeet Agarwal, and Manik Varma. 2021. DeepXML: A Deep Extreme Multi-Label Learning Framework Applied to Short Text Documents. In *WSDM.* 31–39.

[9] DW Dearholt, N Gonzales, and G Kurup. 1988. Monotonic search networks for computer vision databases. In *Twenty-Second Asilomar Conference on Signals, Systems and Computers*, Vol. 2. 548–553.

[10] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.* 44, 8 (2022), 4139–4150.

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

[12] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD.* 541–552.

[13] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR.* 2946–2953.

[14] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 4 (2014), 744–755.

[15] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. 2011. Fast Approximate Nearest-Neighbor Search with k-Nearest Neighbor Graph. In *IJCAI.* 1312–1317.

[16] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *CVPR.* 5713–5722.

[17] Jae-Pil Heo, Zhe L. Lin, and Sung-Eui Yoon. 2019. Distance Encoded Product Quantization for Approximate K-Nearest Neighbor Search in High-Dimensional Space. *IEEE Trans. Pattern Anal. Mach. Intell.* 41, 9 (2019), 2084–2097.

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

[19] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC.* 604–613.

[20] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128.

[21] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/.

[22] Yannis Kalantidis and Yannis Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR.* 2329–2336.

[23] Yifan Lei, Qiang Huang, Mohan S. Kankanhalli, and Anthony K. H. Tung. 2020. Locality-Sensitive Hashing Scheme based on Longest Circular Co-Substring. In *SIGMOD.* 2589–2599.

[24] 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.

[25] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.* 45 (2014), 61–68.

[26] 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.

[27] Microsoft. 2021. SPACEV1B: A billion-Scale vector dataset for text descriptors. https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B.

[28] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *SIGMOD* 1, 1 (2023), 54:1–54:27.

[29] Chanop Silpa-Anan and Richard I. Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *CVPR.* 1–8.

[30] 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).

[31] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NeurIPS 2019.* 24–34.

[32] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NeurIPS.* 13748–13758.

[33] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: Solving c-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *PVLDB.* 8, 1 (2014), 1–12.

[34] Yao Tian, Ziyang Yue, Ruiyuan Zhang, Xi Zhao, Bolong Zheng, and Xiaofang Zhou. 2023. Approximate Nearest Neighbor Search in High Dimensional Vector Databases: Current Research and Future Directions. *IEEE Data Eng. Bull.* 46, 3 (2023), 39–54.

[35] Yao Tian, Xi Zhao, and Xiaofang Zhou. 2022. DB-LSH: Locality-Sensitive Hashing with Query-based Dynamic Bucketing. In *ICDE.* 2250–2262.

[36] Yao Tian, Xi Zhao, and Xiaofang Zhou. 2024. DB-LSH 2.0: Locality-Sensitive Hashing With Query-Based Dynamic Bucketing. *IEEE Trans. Knowl. Data Eng.* 36, 3 (2024), 1000–1015.

[37] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A Data-adaptive and Dynamic Segmentation Index for Whole Matching on Time Series. *PVLDB.* 6, 10 (2013), 793–804.

[38] 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 *SOSP.* 545–561.

[39] Bolong Zheng, Ziyang Yue, Qi Hu, Xiaomeng Yi, Xiaofan Luan, Charles Xie, Xiaofang Zhou, and Christian S. Jensen. 2023. Learned Probing Cardinality Estimation for High-Dimensional Approximate NN Search. In *ICDE.* 3209–3221.

[40] Bolong Zheng, Xi Zhao, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S. Jensen. 2020. PM-LSH: A Fast and Accurate LSH Framework for High-Dimensional Approximate NN Search. *PVLDB.* 13, 5 (2020), 643–655.