PRO-HNSW: Proactive Repair and Optimization for High-Performance Dynamic HNSW Indexes

Huijun Jin
Computer Science
Yonsei University
Seoul, Korea
jinhuijun@yonsei.ac.kr

Jieun Lee
Computer Science
Yonsei University
Seoul, Korea
jieun199624@yonsei.ac.kr

Shengmin Piao
Artificial Intelligence
Yonsei University
Seoul, Korea
shengminp@yonsei.ac.kr

Sangmin Seo
Computer Science
Yonsei University
Seoul, Korea
ssm6410@yonsei.ac.kr

Sanghyun Park†
Computer Science
Yonsei University
Seoul, Korea
sanghyun@yonsei.ac.kr

***Abstract*—Graph-based Approximate Nearest Neighbor Search (ANNS), particularly using Hierarchical Navigable Small World (HNSW) graphs, offers state-of-the-art query performance for large-scale, high-dimensional data. However, the efficiency and accuracy of HNSW indexes degrade seriously under dynamic conditions involving frequent data deletions and reinsertions, often necessitating costly full index rebuilds. This paper introduces PRO-HNSW (Proactive Repair and Optimized HNSW), a novel and lightweight maintenance framework designed to efficiently preserve and even enhance the quality of the HNSW graph in dynamic environments. PRO-HNSW systematically addresses key structural degradation factors by incorporating three targeted modules: (1) a `RemoveObsoleteEdges` module to prune edges pointing to deleted nodes while preserving minimal connectivity, (2) a `RepairDisconnectedNodes` module that re-establishes connectivity for isolated nodes via BFS-based neighbor discovery, and (3) a `ResolveUnidirectionalEdges` module to ensure more balanced and navigable graph structures. Extensive experiments on eight diverse benchmark datasets demonstrate that PRO-HNSW significantly outperforms not only the standard HNSW but also the state-of-the-art MN-RU method. Notably, on the high-dimensional GIST1M dataset under high data churn, PRO-HNSW achieves a peak recall of 84.23%, a significant 2.56 percentage points higher than even the ideal HNSW-Rebuild baseline, showcasing its superior ability to produce a more optimal graph structure. Our findings also reveal that applying specific PRO-HNSW maintenance modules to a freshly built static HNSW index can improve its initial recall, highlighting its utility as a general graph optimization technique. PRO-HNSW thus provides a practical and robust solution for maintaining high-performance ANNS in evolving real-world applications.**

***Index Terms*—Approximate Nearest Neighbor Search, Hierarchical Navigable Small World, Update Performance Optimization, Lightweight Graph Repair**

I. Introduction

Approximate Nearest Neighbor Search (ANNS) is a fundamental operation in many applications, including recommendation systems [1]–[3], image retrieval [4], [5], and natural language processing [6]–[8]. Among various ANNS algorithms, Hierarchical Navigable Small World (HNSW) graphs have gained significant popularity due to their excellent search performance, achieving a remarkable balance between speed and accuracy across diverse datasets [9]. HNSW constructs a multi-layer graph where nodes are progressively added, with long-range edges at upper layers facilitating fast traversal and short-range edges at lower layers enabling precise navigation.

† Corresponding Author

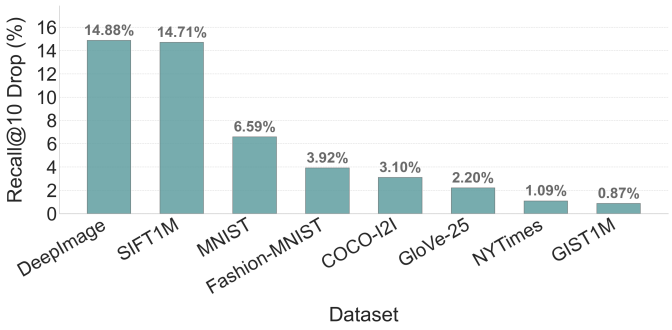

Fig. 1. Maximum recall drop during consecutive update scenarios.

Despite its efficacy for static datasets, the performance of HNSW indexes can seriously degrade in dynamic environments where nodes are frequently deleted and inserted. Real-world systems often deal with constantly evolving data, making efficient and effective update mechanisms crucial for maintaining search performance over time. For instance, in a large-scale e-commerce platform, product catalogs are continuously updated with new items being added and seasonal or out-of-stock items being removed [2], [4]. While HNSW supports deletions by marking nodes as deleted and reinserting new nodes into these vacated slots, this process often leads to a suboptimal graph structure over time. The cumulative effect of such updates can result in increased query latency, reduced recall, and an overall decline in search quality.

To quantify the impact of sustained updates on search accuracy, we evaluated the maximum recall drop for standard HNSW in a challenging consecutive update scenario across our eight benchmark datasets. In this experiment, for each dataset, we simulated a total data turnover of 50% by performing 100 consecutive update iterations, where each iteration involved randomly deleting and reinserting a distinct 0.5% of the base vectors. As illustrated in Fig. 1, the performance degradation under this sustained workload can be severe, with observed maximum recall drops reaching 14.88% for DeepImage and 14.71% for SIFT1M. This considerable degradation highlights the inherent vulnerability of the standard HNSW approach.

The conventional remedy, a full index reconstruction, is computationally prohibitive for many real-world scenarios characterized by high update throughput and stringent uptime requirements. This challenge underscores the pressing need for efficient and lightweight maintenance strategies for HNSW under dynamic workloads.

The primary sources of this performance degradation in updated HNSW graphs, as our investigation reveals, include: (1) the accumulation of *obsolete edges* pointing to marked-deleted nodes, which misguide search; (2) the emergence of *disconnected nodes*, resulting parts of the graph unreachable; and (3) the proliferation of *unidirectional edges*, which disrupt balanced graph traversal. While some existing approaches attempt to mitigate these issues through periodic localized rebuilding [10] or basic heuristic repairs [9], [11], they often fall short of comprehensively addressing the multifaceted nature of graph quality degradation or impose significant overhead themselves.

To overcome these limitations, this paper introduces **PRO-HNSW (Proactive Repair and Optimized HNSW)**, a novel and lightweight maintenance framework specifically designed to proactively and efficiently repair and even enhance the quality of HNSW graphs after a series of updates. Instead of resorting to costly rebuilds or overly simplistic heuristics, PRO-HNSW employs a suite of targeted, low-overhead algorithmic modules. These modules systematically address the identified root causes of degradation: a `RemoveObsoleteEdges` module tackles obsolete edges while preserving minimal connectivity; a `RepairDisconnectedNodes` module re-establishes crucial connectivity for isolated nodes; and a `ResolveUnidirectionalEdges` module ensures more balanced graph structures. Our contributions are:

- We propose PRO-HNSW, a novel and lightweight maintenance framework for HNSW indexes that efficiently recovers and sustains graph quality under various update scenarios, offering a significantly more cost-effective alternative to full index reconstructions.
- We systematically identify and analyze the root causes of HNSW graph quality degradation under dynamic updates—namely, obsolete edges, disconnected nodes, and unidirectional edges—and present targeted, low-overhead algorithmic solutions integral to PRO-HNSW, for each identified cause.
- We conduct extensive experiments demonstrating that PRO-HNSW significantly outperforms both standard HNSW and state-of-the-art MN-RU variants. We show that PRO-HNSW not only achieves the highest peak recall in all scenarios but also surpasses the theoretical optimum of the HNSW-Rebuild baseline.
- We validate the fundamental soundness of our repair modules by showing they can improve search recall even on static graphs, emphasizing their versatility as a general graph optimization approach.

II. Background and Related Work

A. Approximate Nearest Neighbor Search

The goal of a nearest neighbor search is to find the data points in a dataset X that are closest to a given query point q. While straightforward for low-dimensional data, finding the exact nearest neighbors becomes computationally prohibitive in high-dimensional spaces—a phenomenon widely known as the "curse of dimensionality" [12]–[14]. To overcome this challenge, Approximate Nearest Neighbor Search (ANNS) was proposed as a practical alternative. The core idea of ANNS is to trade a small, acceptable amount of accuracy for a massive gain in search speed. This trade-off is highly effective for many real-world applications where slight imperfections are tolerable, such as recommendation systems [1], [2], image retrieval [4], [5], and natural language processing [6], [7].

The trade-off between search accuracy (recall) and efficiency (queries per second, QPS) is a central theme in ANNS research. Various ANNS algorithms have been proposed, broadly categorized into tree-based [15]–[18], hashing-based [19]–[22], quantization-based [13], and graph-based methods [9], [23]–[25]. Among these, graph-based ANNS algorithms have recently demonstrated state-of-the-art performance on many benchmark datasets [26], [27]. Their success stems from their ability to effectively capture complex neighborhood structures in high-dimensional data, enabling efficient greedy search traversals that balance global exploration and local refinement. Within this category, HNSW has become one of the most popular and widely adopted methods due to its superior accuracy-speed trade-off.

B. Hierarchical Navigable Small World Graphs

HNSW [9] is a graph-based ANNS algorithm known for its excellent search performance. It constructs a multi-layer graph where each layer is a Navigable Small World (NSW) graph [28], enabling both fast traversal and precise navigation.

Hierarchical Structure. The HNSW graph is built by inserting nodes one by one. Each new node is assigned a maximum level, l_{max}, chosen randomly from an exponentially decaying probability distribution. This approach creates a hierarchical structure: the bottom layer (level 0) is a dense graph containing all nodes, connected by short-range edges for fine-grained search, while upper layers are progressively sparser, featuring long-range "highway" edges for efficient, coarse-grained traversal across the vector space.

Construction. During construction, the algorithm traverses from the top layer downwards. At each level, it finds approximate neighbors for the new node, guided by the $efConstruction$ (ef_c) parameter, which controls the trade-off between build time and graph quality. A crucial step is the heuristic-based neighbor selection, which selects up to M neighbors for upper layers ($l > 0$) and M_0 (typically $2M$) for the base layer, balancing proximity and diversity to ensure robust navigability.

Search. A search in HNSW also starts at the top layer, greedily traversing edges to find the node closest to the query

at each level before descending. This process is repeated until the search reaches level 0, where a final, more refined search is performed. The trade-off between search speed and accuracy is controlled at query time by the $efSearch$ (ef_s) parameter, which dictates the size of the dynamic candidate list.

Update. For updates, standard implementations employ a *mark-and-replace* mechanism: a node is flagged as deleted, and its slot is later reused for a new insertion. While simple, this approach is the source of the structural degradation issues that motivate our work.

C. Challenges in Maintaining High-Performance HNSW

In dynamic environments, the continuous churn of insertions and deletions systematically damages the structural quality of the HNSW graph. This degradation directly compromises search efficiency, making robust maintenance strategies essential for real-world applications. However, the standard HNSW approach presents several critical challenges that progressively degrade the index:

Obsolete Edges. When a node u is marked for deletion in HNSW, its entry in the graph is typically only flagged, while other alive nodes v that previously established an edge $v \rightarrow u$ often retain this edge. These lingering connections become *obsolete* or "dead" edges. As updates accumulate, the graph becomes increasingly littered with them, leading to two primary negative consequences. First, during search, traversing these obsolete edges results in wasted computations and can misguide the search trajectory, which directly increases query latency and harms recall. Second, their accumulation unnecessarily inflates the out-degree of alive nodes, which damages the effectiveness of neighbor selection heuristics during subsequent insertions and graph maintenance by cluttering neighbor lists with useless connections.

Disconnected Nodes. Node deletions and reinsertions can cause more severe structural damage than just obsolete edges: parts of the graph can become entirely *disconnected* from the main structure. A node might lose all of its incoming edges, especially at the critical upper "highway" layers of HNSW. This happens when the few neighbors pointing to it are deleted or have their own connections updated. Such a node, now lacking any incoming paths, effectively becomes an "unreachable island" in the graph. During a search, there is no path from the main entry points to this "island." As a result, the isolated node itself, along with any other nodes exclusively connected to it, becomes effectively invisible to the search algorithm [11].

This problem can escalate to *graph fragmentation*, where the entire graph shatters into one large "continent" and multiple smaller, isolated "islands". Consequently, if a query's true nearest neighbors happen to reside on one of these "islands", they become entirely inaccessible, leading to a drastic and unpredictable drop in recall [11]. This issue is particularly acute because the success of HNSW's search fundamentally relies on the ability to navigate the entire graph seamlessly.

Unidirectional Edges. The HNSW construction heuristic does not strictly guarantee that edges are bidirectional. An edge $u \rightarrow v$ might be established because v is selected as one of the best neighbors for u, but the reverse is not guaranteed, as u may not be among the best neighbors from v's perspective. While this level of asymmetry is inherent to HNSW, dynamic updates can significantly exacerbate the issue by breaking existing symmetric pairs or creating new one-way connections during reinsertions into a structurally altered neighborhood.

These *unidirectional edges* effectively create "one-way streets" in the graph's navigational structure, which can severely disrupt the greedy search process. If a search traverses an edge from node u to v but cannot return or explore adjacent areas due to a missing reciprocal edge, it can get trapped in a suboptimal region of the graph. Unable to backtrack to a more promising path, the search is often forced to converge on a less accurate result, ultimately failing to discover the true nearest neighbors and thereby lowering recall.

Addressing these challenges without resorting to frequent and costly full index rebuilds is the primary motivation for our work. Our proposed lightweight maintenance modules are designed to proactively and efficiently counteract these specific degradation factors, thereby preserving and even enhancing the search performance of HNSW indexes in dynamic settings.

D. Related Work on Updatable Graph-Based ANN Indexes

Graph-based methods for ANNS have shown considerable success in providing efficient and accurate search results for high-dimensional data [27]. Algorithms such as HNSW [9], NSG [24], Vamana (used in DiskANN [25]), and others [23], [28]–[35] construct graph structures where nodes represent data points and edges represent proximity, enabling efficient greedy-search traversals. While these methods excel in static settings, their performance can significantly degrade in dynamic environments where data is frequently inserted or deleted, a common scenario in real-world applications. The primary challenge lies in maintaining the graph's structural integrity and search efficiency without resorting to costly full index rebuilds.

Addressing updates in disk-based graph ANN indexes, Yu et al. [36] highlight significant challenges related to I/O overhead and inefficient neighbor pruning, particularly with small-batch updates. Their system, Greator, proposes a lightweight topology for faster affected-vertex identification and localized page-level updates to minimize I/O. It also introduces a "similar neighbor replacement" method to reduce costly pruning operations. While Greator's emphasis on localized repairs to mitigate update overheads shares a conceptual similarity with PRO-HNSW, its solutions are specifically tailored for disk I/O efficiency and persistence, whereas PRO-HNSW focuses on lightweight computational fixes for in-memory HNSW structures.

For in-memory HNSW, Xiao et al. [11] investigate the "unreachable points phenomenon" and performance degradation stemming from standard HNSW update mechanisms. Their MN-RU algorithm aims to mitigate these issues by redesigning deletion and update strategies, including specific neighbor connection repairs. A notable aspect of their approach is

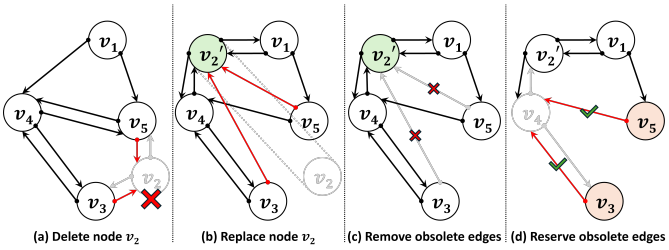

Fig. 2. Remove or reserve obsolete edges ($M = 2$).

the introduction of a dual-index architecture, comprising a main HNSW index and an auxiliary "HNSW Backup Index" designed to manage and retrieve otherwise unreachable points. Queries are then performed across both indexes ("dualSearch"). While this dual-index strategy can help in finding points that become isolated in the main index, it inherently incurs additional memory overhead for storing and maintaining the secondary index, and potentially adds complexity to the query process and overall system management.

In contrast, our PRO-HNSW framework offers a fundamentally different approach. Instead of introducing auxiliary data structures like the "Backup Index" in MN-RU, it operates with a suite of lightweight, in-place modification modules that directly enhance the structural integrity of the single, primary HNSW graph. This design not only addresses node unreachability but also systematically resolves distinct issues of obsolete edges and unidirectional edges through computationally efficient repairs. Our findings also underscore that PRO-HNSW serves as a foundational enhancement to the core HNSW framework for both dynamic and static scenarios. This corroborates a similar principle observed by Zardbani et al. [37] in the context of spatial indexing, where a method designed for adaptive updates also proved highly effective for building an efficient static index.

Furthermore, we believe the core principles of PRO-HNSW—proactively pruning obsolete edges (ROE), repairing graph connectivity for isolated nodes (RDN), and ensuring edge reciprocity (RUE)—address fundamental challenges inherent to any dynamic graph-based ANNS method, not just HNSW. For instance, popular algorithms such as NSG and Vamana, which also rely on greedy-search traversals, face similar structural degradation from frequent updates. While the direct application and evaluation of our modules on these alternative graph structures are beyond the scope of this paper, it presents a compelling avenue for future work. Our current research focuses exclusively on validating the effectiveness of these principles within the HNSW framework.

III. The PRO-HNSW Framework

The PRO-HNSW is designed to be lightweight and is consists of three core algorithmic modules, each targeting a specific type of structural degradation commonly observed after data deletions and reinsertions. These modules are: (1) `RemoveObsoleteEdges`, which prunes outdated connections to deleted nodes; (2) `RepairDisconnectedNodes`, which re-establishes connectivity for nodes that may have become isolated; and (3) `ResolveUnidirectionalEdge`, which ensures that critical navigational edges are bidirectional or otherwise balanced.

Algorithm 1: `RemoveObsoleteEdges`

Input: A set of deleted nodes $delSet$, alive nodes U, a threshold T

1 **for** alive node $u \in U$ **do**
2 **for** $l \leftarrow 0$ **to** L_u **do**
3 $N_{(u,l)} \leftarrow getNeighbors(u, l)$
4 $c_{an} \leftarrow countAliveNeighbors(N_{(u,l)}, delSet)$
5 **if** $c_{an} < T$ **then**
6 **continue**
7 $N'_{(u,l)} \leftarrow \emptyset$ `// initialize a new list`
8 **for** $n \in N_{(u,l)}$ **do**
9 **if** $n \notin delSet$ **then**
10 $N'_{(u,l)}.add(n)$
11 $updateNeighbors(u, l, N'_{(u,l)})$

A. Remove Obsolete Edges (ROE)

Conceptual Overview. The ROE module is designed to prune obsolete edges that point to deleted nodes. Fig. 2 conceptually illustrates the scenarios ROE handles and the core idea behind its heuristic. Initially, when a node (say v_2 in Fig. 2a) is marked for deletion, its outgoing edges are conceptually removed, but incoming edges from alive nodes (e.g., from v_3, v_5, highlighted in red) might persist. If the slot of v_2 is later reused by a new vector, these persistent incoming edges become obsolete, now pointing to the updated node v'_2 (Fig. 2b). Our ROE module targets such obsolete connections. As shown in Fig. 2c, obsolete edges (e.g., from v_3, v_5 to v'_2) are pruned if their source nodes possess other outgoing edges, thereby maintaining their connectivity. However, a crucial safeguard is incorporated: Fig. 2d depicts a situation where pruning an obsolete edge (e.g., from v_3 or v_5 to a deleted v_4) would leave the source node (v_3 or v_5) with no outgoing connections. In such cases, ROE preserves this last outgoing edge, even if obsolete, to prevent node isolation.

ROE Algorithm. The detailed procedure is outlined in Algorithm 1. The algorithm iterates through all alive nodes U and all their levels L, assuming that a set of deleted nodes denotes $delSet$. The neighbor list $N_{(u,l)}$ of the alive node u on the level l is first obtained (Line 3), and the number of its "alive" neighbors c_{an} (i.e., those not in the $delSet$) is counted using the $countAliveNeighbors$ function (Lines 4). A predefined threshold T, typically set to 1, denotes the minimum number of alive neighbors that a node should retain at any level. If c_{an} is less than T, the neighbor list $N_{(u,l)}$ is left unchanged for that node and level to preserve minimal connectivity (Lines 5 to 7). Otherwise, if c_{an} meets or exceeds T, a new list $N'_{(u,l)}$ is constructed containing only the alive

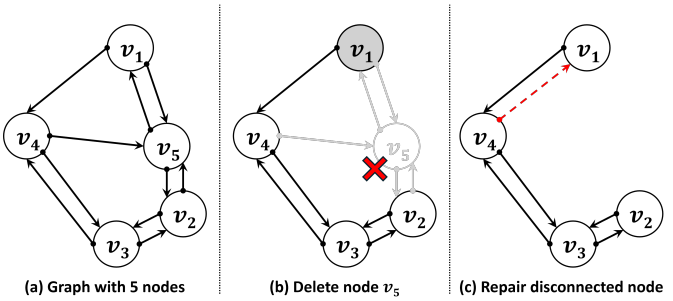

Fig. 3. Node deletion and repair disconnected node ($M = 2$).

neighbors (i.e., $n \in N_{(u,l)}$ such that n is NOT IN $delSet$) (Lines 8 to 10). Finally, the neighbor list of u at level l is updated to $N'_{(u,l)}$ via $updateNeighbors$ (Line 11). This selective pruning mechanism ensures the graph is cleansed of unhelpful obsolete edges while cautiously preventing node isolation, typically after a batch of nodes has been marked for deletion.

Time Complexity. The time complexity of Algorithm 1 is primarily determined by iterating through all alive nodes and their neighbors across all levels. Let N be the number of alive nodes in the graph, L_{avg} be the average maximum level of a node, and M_{avg} be the average number of neighbors per node per level. The algorithm accesses each neighbor list and checks each neighbor against the $delSet$. Thus, the total number of such checks, which constitutes the dominant part of the computation, can be approximated as $O(N \cdot L_{avg} \cdot M_{avg})$. The check against $delSet$ (an unordered set) takes, on average, $O(1)$ time per neighbor.

B. Repair Disconnected Nodes (RDN)

Conceptual Overview. The RDN module addresses the issue of graph fragmentation by identifying and re-establishing connectivity for isolated nodes. Fig. 3a depicts an example graph segment. If a node v_5 is subsequently deleted (Fig. 3b), v_1 loses its only remaining incoming connection, thereby becoming isolated. To rectify such situations, the RDN module is applied. Fig. 3c shows the outcome: the RDN procedure, by invoking the $limitedHopRepair$ subroutine (Algorithm 3), attempts to establish new incoming connections for v_1. In this illustrative example, a new incoming edge $v_4 \rightarrow v_1$ is successfully added, reintegrating v_1 into the navigable graph structure.

RDN Algorithm. The Algorithm 2 formalizes this repair process. It begins by invoking a $findAllComponents$ function (Line 1), which identifies all distinct connected components ($allComps$) within the graph. After determining the total number of alive nodes, c_{tn} (Line 2), a size threshold, T_c, is established as $c_{tn}/2$ (Line 3). This threshold is based on the observation that fragmented graphs typically consist of one main component and smaller outliers; thus, T_c targets these smaller "fragment" components for repair. For a node u within a small component ($comp$) that is NOT IN the $delSet$ (Lines 4 to 6), the algorithm iterates through all

Algorithm 2: `RepairDisconnectedNodes`

```
1  allComps ← findAllComponents()
2  c_tn ← countAliveNodes()
3  T_c ← c_tn / 2
4  for comp ∈ allComps do
5      if sizeOf(comp) < T_c then
6          for u ∈ comp and u ∉ delSet do
7              L_u ← getMaxLevel(u)
8              for l ← 0 to L_u do
9                  limitedHopRepair(u, l)
```

Algorithm 3: $limitedHopRepair(u_t, l_t)$

```
   Input: A target node u_t to be repaired, a level l_t
1  visited ← newSet(); visited.add(u_t)
2  frontier ← newQueue(); frontier.enqueue(u_t)
3  hop ← 0
4  while frontier ≠ ∅ do
5      hop ← hop + 1
6      isAddedInThisHop ← false
7      nextF ← ∅ // init a next frontier
8      for u ∈ frontier do
9          N_(u,l_t) ← getNeighbors(u, l_t)
10         for n ∈ N_(u,l_t) do
11             visited.add(n)
12             nextF.add(n)
13             if addIntoEmptySlot(n, u_t, l_t) then
14                 isAddedInThisHop ← true
15     if hop ≥ 3 and isAddedInThisHop then
16         break
17     frontier ← nextF
```

levels originally populated by u (up to L_u, Lines 7 to 8). Subsequently, it invokes the $limitedHopRepair$ procedure (Line 9) to establish up to this target number of connections for node u at that level.

The core repair mechanism is encapsulated within the $limitedHopRepair$ subroutine, as shown in Algorithm 3. This procedure aims to establish new incoming links to the target node u_t at the specified level l_t by performing a limited-hop BFS. Starting from u_t (Lines 1 to 2), the BFS explores nodes in its $frontier$ (Lines 8 to 17). For each valid, unvisited, and undeleted neighbor n discovered (by iterating $N_{(u,l_t)}$ obtained via $getNeighbors$), an attempt is made to add an incoming edge from n to u_t at l_t using the $addIntoEmptySlot$ function Lines 8 to 13). This function adds the edge only if n has an available slot at l_t. This strategy of targeting only available slots is designed to minimize computational overhead by avoiding costly heuristic-based neighbor selection when a simple addition suffices, while still effectively establishing new links to enhance the connectivity of the u_t node. The BFS

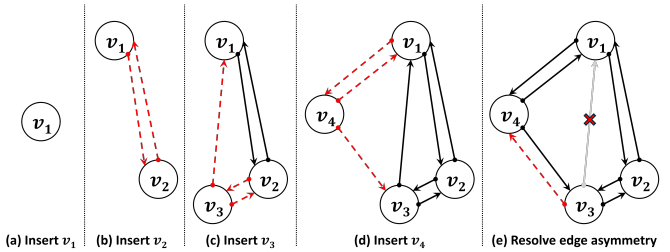

Fig. 4. Node insertion and resolve unidirectional edges ($M = 2$).

proceeds for a limited number of hops (e.g., up to 3 hops), terminating early if a new link is added in a hop after an initial exploration phase. This multi-level repair strategy is crucial for repairing not just local connectivity at level 0 but also the long-range navigational links at upper layers that are vital for HNSW's search efficiency.

Time Complexity. The RDN procedure involves two main computational stages. First, the $findAllComponents$ step, typically implemented using BFS across all graph layers, has a time complexity of approximately $O(N + E)$, where N is the total number of nodes and E is the total number of edges in the graph. Second, for each node u identified within a "small" component (let N_s be the total count of such nodes), the algorithm iterates up to its maximum original level L_u (with an average maximum level of L_{avg} across these nodes) and invokes the $limitedHopRepair$ subroutine. The $limitedHopRepair$ performs a BFS up to a small, constant number of hops k. In each hop, it explores up to M_{avg} neighbors, and for each of these, it again explores M_{avg} neighbors. The dominant operation within this BFS is often the neighbor retrieval and checks. If we consider the number of nodes visited in k hops to be roughly on the order of M_{avg}^k in sparse regions, and for each, M_{avg} neighbors are processed, the complexity of a single $limitedHopRepair$ call might be approximated as $O(M_{avg}^{k+1})$. Thus, the total complexity for the repair phase can be estimated as $O(N_s \cdot L_{avg} \cdot M_{avg}^{k+1})$. The overall complexity of RDN will be dominated by either the component finding or the cumulative repair phase, depending on the graph's state and the number of small components.

C. Resolve Unidirectional Edges (RUE)

Conceptual Overview. The RUE module is designed to resolve unidirectional edges by attempting to establish reciprocal connections where they are missing. Fig. 4 provides a step-by-step illustration of how unidirectional edges can arise during HNSW construction and how our RUE module. Initially, Fig. 4a depicts the insertion of the first node, v_1. Subsequently, as shown in Fig. 4b, node v_2 is inserted, and ideal bidirectional edges $v_1 \leftrightarrow v_2$ (newly formed edges are highlighted with red dashed lines) are established. However, unidirectional edges can emerge as more nodes are added. In Fig. 4c, node v_3 is inserted. While new edges $v_3 \rightarrow v_1$, $v_3 \rightarrow v_2$, and $v_2 \rightarrow v_3$ are formed, a reciprocal edge $v_1 \rightarrow v_3$ is notably absent. This type of unidirectional edges typically arises when node v_1's neighbor list is already full, or the neighbor selection heuristic does not consider v_3 a better candidate than its existing connections. This pattern continues in Fig. 4d with the insertion of v_4, where the edge $v_4 \rightarrow v_3$ lacks its reciprocal $v_3 \rightarrow v_4$. Fig. 4e illustrates the outcome after applying our RUE module. The procedure identifies such unidirectional edges (e.g., involving v_3 and v_4) and attempts to establish the missing reciprocal links, for instance, by adding the edge $v_3 \rightarrow v_4$ (shown in red). The outcome is a graph with resolved unidirectional edges, leading to a more balanced and robustly navigable structure.

Algorithm 4: `ResolveUnidirectionalEdges`

```
1  for alive node u in the graph do
2      N_(u,l0) ← getNeighbors(u, l0)
3      for n ∈ N_(u,l0) do
4          N_(n,l0) ← getNeighbors(n, l0)
5          isReciprocated ← false
6          for v ∈ N_(n,l0) do
7              if v = u then
8                  isReciprocated ← true
9                  break
10         if not isReciprocated then
11             attemptAddIncomingEdge(n, u, l0)
```

Algorithm 5: $attemptAddIncomingEdge(u_s, u_t, l)$

```
Input: A source node u_s, a target node u_t, a level l
1  N_(u_t,l) ← getNeighbors(u_t, l)
2  c_mn ← getMaxNeighbors(l)
3  if sizeOf(N_(u_t,l)) < c_mn then
4      N_(u_t,l).add(u_s)
5      return true
6  else
7      candidateSet ← N_(u_t,l) ∪ u_s
8      vec_t ← getDataVector(u_t)
9      N_s ← selectNeighbors(candidateSet, vec_t, c_mn)
10     Update N_(u_t,l) with N_s
```

RUE Algorithm. The Algorithm 4 formalizes this resolving process. Our approach deliberately focuses this procedure on level 0. This is because level 0 is the only layer guaranteed to contain all nodes and is the densest layer where the vast majority of unidirectional edges occur, making it the most impactful layer for resolving unidirectional edges while avoiding the significant computational overhead of processing all layers. The Algorithm 4 iterates through all alive nodes in the graph. For each outgoing neighbor n of the alive node u found in its level 0 neighbor list, $N_{(u,l_0)}$ (Line 2), it inspects n's level 0 neighbor list, $N_{(n,l_0)}$ (Line 4), to determine if u is present. The boolean variable $isReciprocated$ tracks this (Lines 5 to 9). If this reciprocal edge is missing, the procedure calls $attemptAddIncomingEdge$ (Line 13) to potentially add

TABLE I
DATASET STATISTICS, DATA TYPES, DISTANCE METRICS, AND TIER-SPECIFIC HNSW PARAMETERS.

Dataset	Size	Dim.	# Vec.	# Qry.	Data Type	Metric	Size Tier	M	ef_c	ef_{re}	ef_s
DeepImage [38]	3.6 GB	96	9,990,000	10,000	Image Features	Cosine	Large	16	100	75	75-125
GIST1M [13]	3.6 GB	960	1,000,000	1,000	Image Features	L2					
SIFT1M [13]	501 MB	128	1,000,000	10,000	Image Features	L2	Medium	12	75	50	50-100
NYTimes [39]	301 MB	256	290,000	10,000	Doc Embeddings	Cosine					
MNIST [40]	217 MB	784	60,000	10,000	Image Features	L2	Small	8	50	25	25-75
Fashion-MNIST [41]	217 MB	784	60,000	10,000	Image Features	L2					
COCO-I2I [42]	136 MB	512	113,287	10,000	Image Features	Cosine					
GloVe-25 [43]	122 MB	25	1,183,514	10,000	Word Embeddings	Cosine					

u to n's neighbor list at level 0.

The $attemptAddIncomingEdge$ subroutine, detailed in Algorithm 5, is responsible for the actual edge balancing. It first retrieves the current neighbor list $N_{(u_t,l)}$ for the node u_t at level l (Line 1) and determines the maximum allowed neighbors, c_{mn}, for the level (Line 2). If $N_{(u_t,l)}$ has fewer than c_{mn} entries (Line 3), u_s is directly added to u_t's neighbor list, and the subroutine returns true. However, if $N_{(u_t,l)}$ is full (Line 6), a heuristic selection is triggered. All existing neighbors in $N_{(u_t,l)}$ plus u_s form a candidate set $candidateSet$ (Line 7). The data vector vec_t is retrieved (Line 8), and a $selectNeighbors$ function (Line 9) is invoked. This heuristic, mirroring HNSW's neighbor selection, chooses the best c_{mn} neighbors. The neighbor list of u_t, $N_{(u_t,l)}$, is then updated (Line 10). This overall process aims to resolve unidirectional edges, thereby enhancing graph navigability.

Time Complexity. The RUE algorithm primarily operates on level 0 of the HNSW graph. It iterates through all alive nodes (let there be N such nodes) and, for each of their M_0 neighbors n, it again checks n's M_0 neighbors to find u. This check for a reciprocal edge takes $O(M_0)$ time. If an unidirectional edge is found, the $attemptAddIncomingEdge$ subroutine is called. In the best-case scenario where n has an available slot, adding the edge is typically $O(1)$ after checking existing neighbors ($O(M_0)$). However, if n's neighbor list is full, the heuristic neighbor selection process $selectNeighbors$ is invoked. This heuristic usually involves calculating distances from n to $M_0 + 1$ candidate neighbors and then selecting the best M_0. If distance calculations take $O(d)$ (where d is the vector dimensionality) and selection involves a sort or priority queue operations, this step might be around $O(M_0 \cdot d + M_0 \log M_0)$. Therefore, the overall time complexity of the RUE module can be approximated as $O(N \cdot M_0^2 \cdot (d + \log M_0))$ in the more complex case involving the heuristic. Given its operation primarily on the dense level 0, the number of edges considered can be substantial. To manage this computational load effectively, our implementation of the RUE module is designed to leverage parallel processing across multiple threads, thereby significantly reducing its effective execution time. While the theoretical worst-case complexity appears high, this scenario is rarely triggered in practice. The worst case occurs when adding a reciprocal edge to a node whose neighbor list is full, thereby invoking the costly distance-based $selectNeighbors$ heuristic. However, in the dynamic settings we address, frequent deletions naturally create open slots in neighbor lists, making the far more common operation a simple, low-cost edge insertion. This is empirically validated by our experimental results in Section IV-E (Table III), which demonstrate that the practical, normal-case overhead of the RUE module is minimal.

IV. EVALUATION

In this section, we present a comprehensive experimental evaluation of PRO-HNSW. We aim to demonstrate its effectiveness in maintaining high search performance under various update scenarios compared to standard HNSW and state-of-the-art methods.

A. Experimental Setup

Environment. All experiments were conducted on a server equipped with a 12th Gen Intel(R) Core(TM) i7-12700F CPU (2.50GHz base) and 94GB of RAM, running Ubuntu 20.04 LTS.

Metrics. The primary metrics for evaluating search performance are:

- **Recall@K**: We use Recall@10, defined as the proportion of queries for which at least one of the true top-10 nearest neighbors is found within the top-10 returned results.
- **Queries Per Second (QPS)**: Measures search throughput, averaged of 3 runs using a single query.

Datasets. We evaluated PRO-HNSW on eight publicly available benchmark datasets, which are standard in ANNS research. These datasets, detailed in Table I, cover a diverse range of dimensionalities, sizes, data types, and distance metrics. Specifically, we categorize these datasets into three tiers based on their approximate file size: Large, Medium, and Small. The ground truth nearest neighbors for all query sets were used as provided with the datasets.

Parameters. The HNSW parameters were selected using a tier-based configuration tailored to dataset characteristics. These include the core construction parameters: M, which defines the maximum number of outgoing edges each node can have, and ef_c ($efConstruction$), which controls the size of the dynamic candidate list during index building to determine graph quality. We also define two dynamic parameters: ef_{re}, which serves as the $efConstruction$ value during the element

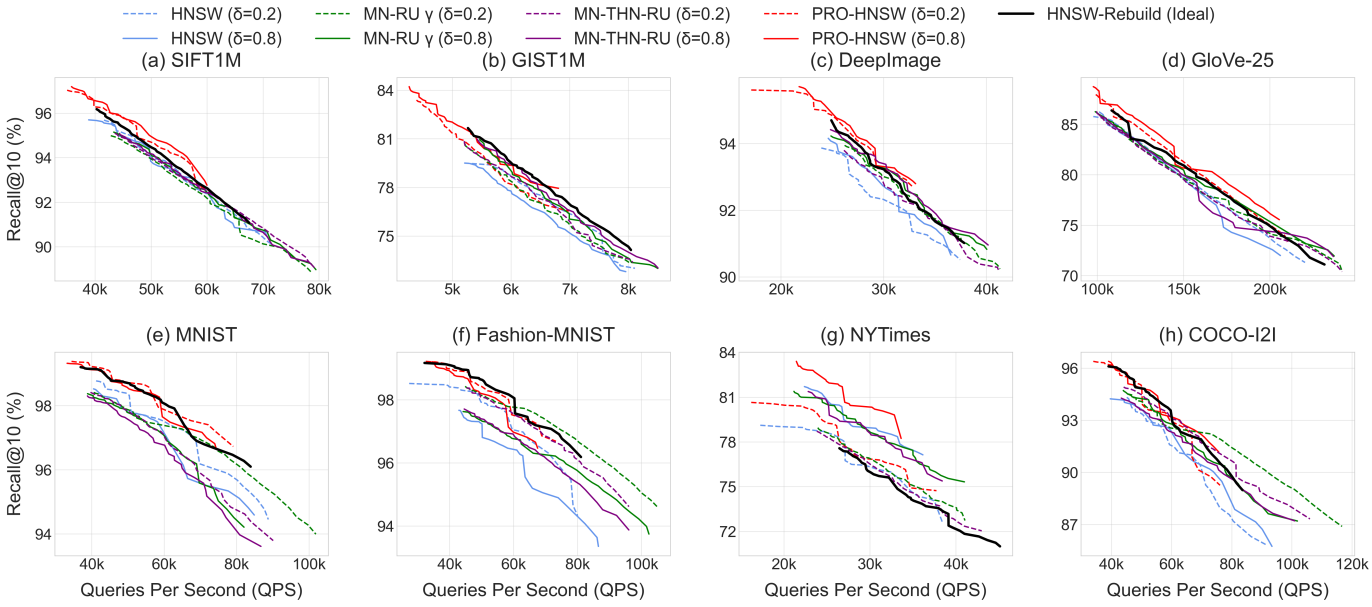

Fig. 5. QPS-Recall@10 comparison across datasets in bulk update scenario.

reinsertion phase, and ef_s, a query-time parameter that also controls the candidate list size to balance search speed and recall. Specific values for these parameters for each tier are detailed in Table I. For all experiments, unless otherwise noted, these tier-specific parameters were consistently applied to both HNSW and PRO-HNSW for fair comparison. The ef_s parameter was varied across the specified range for each tier in unit increments to generate Recall-QPS trade-off curves.

Baselines. To rigorously evaluate the performance of our proposed PRO-HNSW, we compare it against four distinct baselines that represent standard, state-of-the-art, and ideal performance scenarios:

- **HNSW**: Represents standard HNSW behavior where updates involve marking nodes as deleted and reinserting nodes into vacated slots using default heuristics, without further graph optimization.
- **HNSW-Rebuild(Ideal)**: This baseline represents the theoretical upper bound on search quality of standard HNSW. After the full set of updates is performed, a new HNSW index is built from scratch on the final dataset. While offering the best possible recall, this approach is computationally prohibitive in most real-world applications.
- **MN-RU** γ: A state-of-the-art method from Xiao et al. [11] that represents their core update algorithm which prioritizes update and query speed.
- **MN-THN-RU**: Another state-of-the-art variant from the same work. This method performs a more extensive connection repair, aiming for the highest possible recall and graph stability, often at the cost of speed.

B. Overall Performance Comparison

We conducted extensive experiments under two challenging dynamic update scenarios. The results are presented in Fig. 5 and Fig. 6. Within each subplot, we illustrate the QPS-Recall@10 performance using Pareto frontier lines for multiple conditions: four baselines with each at two distinct deletion ratios, $\delta = 0.2$ (small churn) and $\delta = 0.8$ (large churn).

Bulk Updates. We first evaluated all methods under a bulk update scenario, which simulates large, instantaneous data churn. For each dataset, a significant portion—$\delta = 0.2$ (20%) and $\delta = 0.8$ (80%)—of the base vectors was randomly selected, marked as deleted, and then reinserted in a single batch. Fig. 5 presents the QPS-Recall@10 performance of PRO-HNSW against all baselines.

The results reveal a clear and consistent performance hierarchy. Across all datasets and conditions, PRO-HNSW consistently demonstrates the most favorable QPS-Recall trade-off. Its Pareto frontiers are not only dominant over the HNSW variants and the MN-RU variants but, remarkably, also surpasses the performance of the ideal HNSW-Rebuild baseline in all scenarios except for the Fashion-MNIST dataset. This counter-intuitive result, outperforming a freshly rebuilt index, strongly suggests that PRO-HNSW's three maintenance modules work in synergy to overcome the inherent limitations of the standard HNSW construction heuristic, producing a graph structure that is not just repaired, but fundamentally more optimal and navigable. Furthermore, another noteworthy observation is that PRO-HNSW consistently achieves the highest maximum recall values in almost every scenario. For instance, on the high-dimensional GIST1M dataset with $\delta = 0.8$, PRO-HNSW reaches a peak recall of 84.23%, a significant 2.56 percentage points higher than the ideal HNSW-Rebuild baseline (81.67%) underscoring its exceptional ability to preserve the structural integrity of the graph under extreme stress.

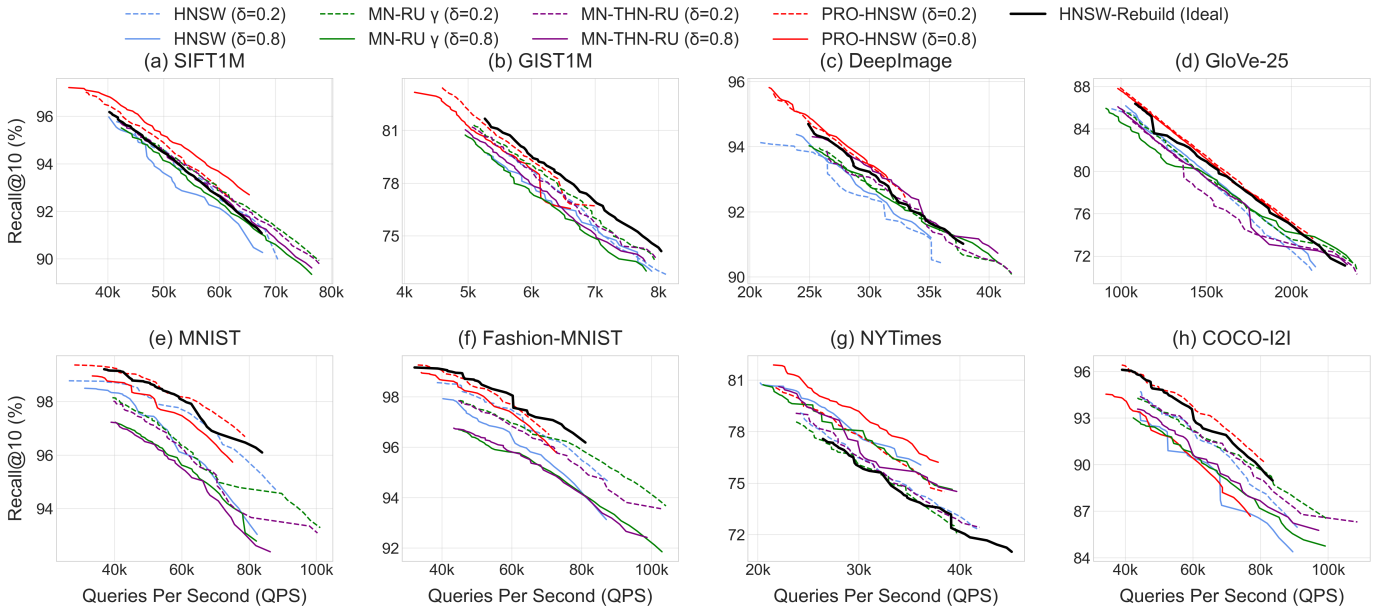

Fig. 6. QPS-Recall@10 comparison across datasets in consecutive update scenario.

An interesting exception is the NYTimes dataset (Fig. 5g), where HNSW-Rebuild does not form the optimal performance frontier. This suggests that for datasets with specific structural properties, such as the dense topical clusters found in document embeddings, a standard fresh build is not always superior.

In contrast, MN-RU, demonstrates clear limitations. While its variants show a slight advantage over the standard HNSW on some datasets like Fashion-MNIST (f) and COCO-I2I (h), their performance on most other datasets is often comparable or only marginally better than the degraded HNSW baseline. Critically, with the exception of the NYTimes anomaly, the MN-RU variants consistently fail to surpass the ideal HNSW-Rebuild frontier, indicating that their repair mechanisms are less effective at achieving a truly optimal graph structure compared to PRO-HNSW.

Consecutive Updates. To simulate more realistic, continuous data evolution, we evaluated all methods under a consecutive update scenario. In this setup, for a given total deletion ratio $\delta \in \{0.2, 0.8\}$, the corresponding fraction of distinct data vectors was randomly selected and then divided into 1000 sequential micro-batches for iterative updates. For PRO-HNSW, the lightweight ROE module was applied in every iteration, while the more intensive RDN and RUE modules were invoked only once after all iterations were completed, ensuring a balance between frequent cleaning and efficient deep repair. Fig. 6 presents the QPS-Recall@10 performance against all baselines under this sustained, iterative churn.

The results in the consecutive update scenario reinforce the conclusions from the bulk updates, with PRO-HNSW consistently exhibiting the most robust and balanced performance. This highlights the effectiveness of PRO-HNSW's periodic deep repair strategy in mitigating the cumulative damage from thousands of micro-updates. This superiority is particularly evident on the SIFT1M (Fig. 6a), DeepImage (Fig. 6c) and GloVe-25 (Fig. 6d) datasets, where PRO-HNSW's Pareto frontiers are clearly dominant, significantly surpassing all baselines.

Furthermore, PRO-HNSW consistently achieves the highest maximum recall across nearly all datasets and conditions. For instance, on the high-dimensional GIST1M dataset with $\delta = 0.8$ (Fig. 6b), PRO-HNSW achieves a peak recall of 83.44%, which is a notable 1.77 percentage points higher than even the ideal HNSW-Rebuild baseline (81.67%). This demonstrates its remarkable ability to not just preserve but actively enhance graph quality under sustained updates.

In contrast, the MN-RU variants again show their limitations, which are particularly stark on the MNIST dataset at $\delta = 0.2$ (Fig. 6e). In this scenario, both MN-RU variants not only fail to offer an improvement but their performance collapses, resulting in a Pareto frontier significantly worse than even the standard HNSW baseline. Conversely, PRO-HNSW's graph sits decisively above all other methods, including the ideal HNSW-Rebuild, showcasing that while the MN-RU repair heuristics can be detrimental on certain data distributions, PRO-HNSW's comprehensive repair strategy consistently produces a more robust and optimal graph structure in a realistic, continuous update environment.

C. Recall Resilience under Sustained Updates

To evaluate the robustness and long-term performance stability of PRO-HNSW under extensive data churn, we conducted a demanding sustained update experiment. This scenario is distinct from the previous consecutive update evaluations in its intensity and the application frequency of our optimization modules. Here, 100% of its original data points were effectively replaced over 1000 iterations. In every

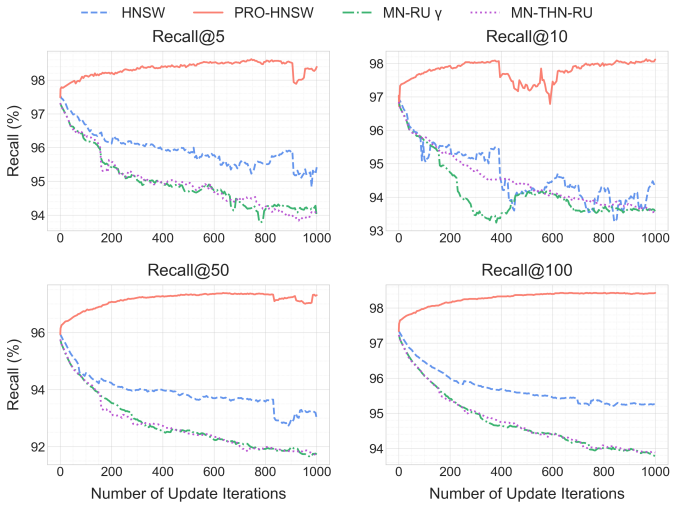

Fig. 7. Recall@K comparison through iterations.

iteration, 0.1% of the total distinct data points (pre-selected randomly from the entire dataset) were marked as deleted, and their corresponding original data vectors were immediately reinserted. For PRO-HNSW, after each update, all three of its maintenance modules were sequentially applied. This simulates a highly dynamic real-world system where the index is continuously maintained. Performance, in terms of Recall@10, was measured periodically at a fixed ef_s value of 30 for both standard HNSW and PRO-HNSW.

Fig. 7 plots the evolution of recall as a function of these 1000 update iterations on the Fashion-MNIST dataset. To provide a more comprehensive analysis as requested by our reviewers, we present the results for Recall@K across K=5, 10, 50, 100, and include a comparison against the MN-RU variants (γ and THN-RU).

The results illustrate a clear performance hierarchy under this rigorous update protocol. For all tested K values, the standard HNSW and both MN-RU variants exhibit a significant initial drop in recall, followed by a gradual, fluctuating degradation. Interestingly, while the MN-RU variants show competitive performance at Recall@10, their effectiveness diminishes at other K values (K=5, 50, 100), where their recall drops significantly, often performing worse than the standard HNSW baseline. This suggests their repair heuristics are specifically tuned for a certain search depth and may not preserve the broader neighborhood structure effectively.

In stark contrast, PRO-HNSW is the only method that demonstrates remarkable resilience and even performance enhancement. Instead of degrading, the recall for PRO-HNSW shows a rapid initial improvement across all K values, for instance rising from 97% to over 98% for Recall@10 within the first 200 iterations. It then maintains a stable, high-performance trajectory, ultimately stabilizing at a level surpassing its original state. Critically, as K increases, the performance curves for all methods become more stable, but PRO-HNSW consistently maintains a significant performance gap over all competitors.

TABLE II
RECALL@10 COMPARISON OF INITIAL VERSUS OPTIMIZED GRAPH

Dataset	ef_s	HNSW	PRO-HNSW	ΔRecall
DeepImage	75	91.07	**91.20**	+0.13
	100	93.27	**93.43**	+0.16
	125	94.71	**94.86**	+0.15
GIST1M	75	74.32	**74.78**	+0.46
	100	78.86	**79.08**	+0.22
	125	81.95	**82.13**	+0.18
SIFT1M	50	90.88	**91.09**	+0.21
	75	94.34	**94.50**	+0.16
	100	96.14	**96.27**	+0.13
NYTimes	50	70.41	**70.82**	+0.41
	75	74.65	**75.02**	+0.37
	100	77.53	**77.67**	+0.14
MNIST	25	95.96	**96.26**	+0.30
	50	98.62	**98.73**	+0.11
	75	99.23	**99.26**	+0.03
Fashion-MNIST	25	**96.09**	95.53	-0.56
	50	**98.61**	98.46	-0.15
	75	99.18	**99.23**	+0.05
COCO-I2I	25	88.74	**88.97**	+0.23
	50	94.35	**94.62**	+0.27
	75	96.09	**96.31**	+0.22
GloVe-25	25	71.26	**71.39**	+0.13
	50	81.56	**81.85**	+0.29
	75	86.42	**86.65**	+0.23

This significant and sustained high performance is a direct result of the iterative application of PRO-HNSW's three core maintenance modules. This continuous, fine-grained maintenance cycle not only counteracts structural decay but also progressively optimizes the graph structure, leading to a robust and superior recall performance across all search depths (K).

D. Static Graph Optimization (No Update Scenario)

Beyond addressing performance degradation in dynamic environments, we investigated whether our proposed maintenance modules could enhance the quality of an HNSW graph even in a static scenario, i.e., immediately after its initial construction and before any data updates occur. For this "no update" experiment, we first build a standard HNSW index for each dataset. Subsequently, we apply a subset of our maintenance framework—specifically, the RDN and RUE modules—to this freshly built index. The performance of this post-optimized graph is labeled as PRO-HNSW in Table II. The ROE module is not applied as no nodes are marked for deletion. Our primary goal here is to ascertain if these repair mechanisms can refine the initial graph structure formed by HNSW's construction heuristics, potentially leading to improved search recall.

Table II summarizes the Recall@10 performance. The results indicate that applying our RDN and RUE modules to a newly built HNSW index yields a modest but consistent improvement in recall across the vast majority of datasets and ef_s configurations. An exception was observed with Fashion-

TABLE III
CUMULATIVE OPERATIONAL METRICS AND TIME COST BREAKDOWN OF OPTIMIZATION MODULES.

Dataset	ROE		RDN		RUE		Other	Total Update Time (s)	
	Removed Edges	Time(%)	Repaired Nodes	Time(%)	Resolved Edges	Time(%)	Time(%)	HNSW	PRO-HNSW
DeepImage	31,021,679	0.17	53,783	0.91	61,838,755	2.69	96.22	**1606.73**	1618.31
GIST1M	1,352,758	0.03	150,864	0.82	6,365,267	3.02	96.12	378.52	**375.71**
SIFT1M	2,354,398	0.20	1,697	0.31	4,318,572	2.80	96.69	**67.05**	77.06
NYTimes	705,881	0.04	1,522	0.11	1,106,053	7.20	92.66	**71.64**	75.02
MNIST	87,277	0.16	316	0.35	158,081	1.70	97.79	**2.42**	3.36
Fashion-MNIST	66,416	0.21	3,651	0.54	153,863	2.06	97.19	**2.59**	2.71
COCO-I2I	134,995	0.17	4,015	0.47	293,702	2.57	96.78	**4.76**	5.70
GloVe-25	1,957,006	0.42	4,579	0.79	2,701,798	4.18	94.60	**34.08**	36.11

MNIST, where a slight decrease in recall occurred at lower ef_s values. This particular outcome might suggest that for some specific data distributions, the structural modifications aimed at improving global graph quality could marginally alter some optimal search paths effective only at lower search budgets. Nevertheless, the predominant positive trend across seven of eight datasets points towards a clear, beneficial refinement of the graph structure. These findings suggest that the HNSW construction process may not always yield a perfectly optimal graph, and our lightweight modules can act as a beneficial post-construction refinement step.

E. Efficiency of the Maintenance Modules

A critical aspect of our proposed framework, PRO-HNSW, is its ability to deliver significant search performance improvements with minimal computational overhead from its maintenance modules. To quantify this, we analyze both the operational metrics of each module and their respective time costs relative to the overall update process, comparing the total update time of PRO-HNSW with that of standard HNSW. The results discussed in this subsection, including those presented in Table III, correspond to a challenging bulk update scenario (Section IV-B) with a deletion ratio $\delta = 0.8$ and an ef_s setting of 75 used for evaluating the PRO-HNSW components.

Table III provides a detailed breakdown of the operational metrics (number of edges/nodes optimized) and the percentage of total PRO-HNSW update time consumed by each module for all eight benchmark datasets under these conditions. The data clearly demonstrates the lightweight nature of our individual optimization modules. For instance, in the DeepImage dataset, ROE, RDN, and RUE contribute only 0.17%, 0.91%, and 2.69% of the total PRO-HNSW update time, respectively. This trend is consistent across other datasets. The "Other" component, which encompasses the inherent HNSW reinsertion process, consistently accounts for the vast majority of the update time in PRO-HNSW (e.g., 96.22% for DeepImage and 96.69% for SIFT1M). This underscores that the primary cost within PRO-HNSW's update phase still stems from the fundamental mark-and-replace operations, with our optimization modules adding only a marginal overhead.

Despite these additional targeted repair steps, the total update time of PRO-HNSW remains highly competitive with that of standard HNSW, as shown in the final two columns of Table III. For GIST1M dataset, PRO-HNSW (375.71 seconds) was slightly faster than standard HNSW (378.52 seconds), even while performing extensive graph repairs. For most other datasets, such as DeepImage, PRO-HNSW's total update time (1618.31s) showed only a minimal increase (less than 1%) compared to standard HNSW (1606.73s). This highly comparable time cost yields definitive improvements in search accuracy, with PRO-HNSW achieving a higher Recall@10 across all datasets. This slight overhead is a modest trade-off for the substantial gains achieved in recall.

Furthermore, Table III quantifies the substantial amount of structural rectification performed by PRO-HNSW. For instance, on SIFT1M, the ROE module removed over 2.35 million obsolete edges, RDN repaired nearly 1,700 nodes, and RUE resolved over 4.31 million unidirectional edges. These results confirm that our modules are actively identifying and addressing a significant volume of graph inconsistencies.

PRO-HNSW thus achieves a compelling balance: the maintenance modules are demonstrably lightweight in terms of their time cost percentage, they perform a significant amount of corrective work on the graph structure, and the resulting total update time for PRO-HNSW is highly competitive. This efficiency, combined with the substantial search performance benefits, positions PRO-HNSW as a practical and effective solution for dynamic HNSW environments, offering a far more economical alternative to periodic full index rebuilds.

F. Analysis of Individual Modules

To understand the individual contribution of each proposed maintenance module, we conducted an analysis study where each optimization function was applied in isolation on top of the standard HNSW framework. The experiments were performed under the consecutive update scenario, focusing on the SIFT1M and GIST1M datasets. These two were chosen as they represent distinct and challenging characteristics within our benchmark suite: SIFT1M is a well-known dataset of local image features with moderate dimensionality, while GIST1M consists of very high-dimensional global image descriptors, allowing us to evaluate our modules under different data distributions and structural complexities. Fig. 8 illustrates the QPS-Recall@10 performance. Each pair of subplots (e.g., (a) SIFT1M and (b) GIST1M for ROE) shows the effect of a single optimization module. Within each subplot, four Pareto

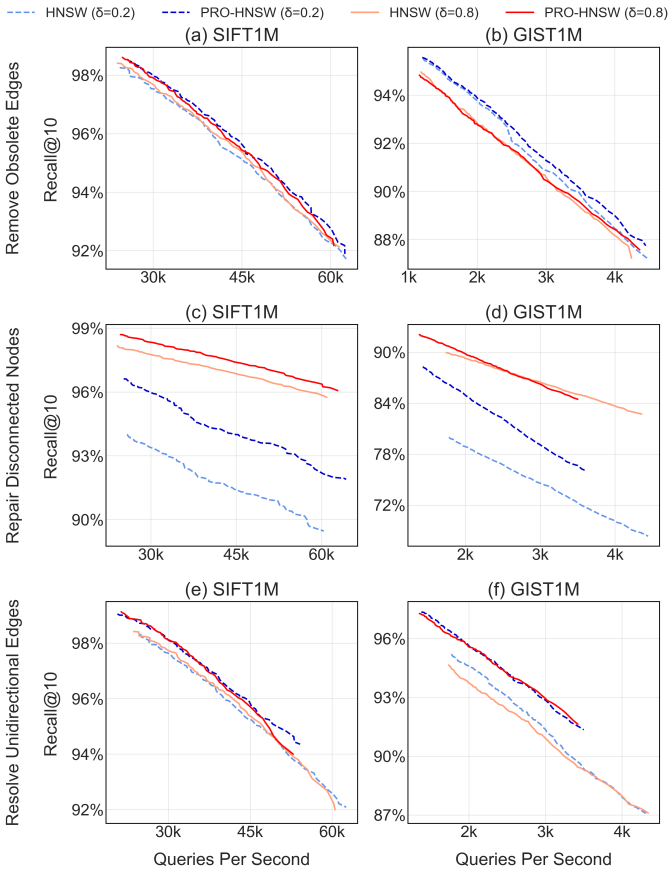

Fig. 8. QPS-Recall@10 comparison of individual PRO-HNSW modules.

frontiers are presented, comparing standard HNSW against HNSW augmented with the specific module, for both $\delta = 0.2$ and $\delta = 0.8$.

Impact of ROE. The effect of applying only the ROE module is shown in Fig. 8a and 8b For the SIFT1M dataset, the ROE module provides a discernible improvement in the QPS-Recall trade-off over standard HNSW, particularly at higher recall targets for both $\delta = 0.2$ and $\delta = 0.8$. This suggests that pruning obsolete edges, even as a standalone measure, enhances search efficiency. For the GIST1M dataset, however, particularly under the $\delta = 0.8$ update scenario (Fig. 8b), the Pareto frontier of HNSW+ROE closely aligns with that of standard HNSW, indicating a more limited impact of solely removing obsolete edges in this specific high-dimensional context after significant churn.

Impact of RDN. Fig. 8c and 8d illustrate the impact of the RDN module. For this particular experiment, to directly verify the effectiveness of RDN in reconnecting isolated nodes, the query set was uniquely constructed: after the update process, existing disconnected nodes within the graph were identified, and these specific nodes were then used as queries. On SIFT1M (Fig. 8c), applying the RDN module results in a very clear and substantial improvement in Recall@10 across the QPS spectrum for both $\delta = 0.2$ and $\delta = 0.8$ when compared to standard HNSW. For GIST1M (Fig. 8d) with $\delta = 0.8$, HNSW+RDN initially achieves a higher maximum recall than standard HNSW. However, a crossover is observed where, beyond approximately 2.5k QPS, the recall for HNSW+RDN slightly underperforms standard HNSW. This might suggest that while RDN effectively reconnects the queried disconnected nodes, the newly formed edges in the dense GIST1M graph might, under higher search efforts, occasionally lead to less globally optimal paths compared to the HNSW structure for this specific query set.

Impact of RUE. The contribution of the RUE module is depicted in Fig. 8e and 8f. The application of RUE consistently leads to improved Recall@10 performance over standard HNSW across both SIFT1M and GIST1M datasets for both $\delta = 0.2$ and $\delta = 0.8$. The Pareto frontiers for HNSW+RUE are visibly superior, indicating that resolving unidirectional edges enhances graph navigability and allows the search to discover true nearest neighbors more effectively. This consistent improvement highlights the general benefit of ensuring more balanced connectivity.

This analysis confirms that each module provides a distinct benefit: ROE offers foundational cleaning, RDN is critical for repairing reachability, and RUE consistently improves navigability. These individual contributions underscore their importance within the comprehensive PRO-HNSW framework, which achieves its superior performance by addressing these varied degradation factors synergistically.

V. Conclusion

In this paper, we introduced PRO-HNSW, a novel and lightweight maintenance framework designed to preserve and enhance the performance of HNSW graphs in dynamic environments. By systematically addressing obsolete edges, node disconnections, and unidirectional edges through three targeted algorithmic modules (ROE, RDN, and RUE), PRO-HNSW consistently outperformed standard HNSW update mechanisms across eight diverse datasets under various update scenarios, including bulk and consecutive updates. Our experiments demonstrated significant improvements in Recall@10 and QPS with only modest computational overhead. The study of analysis of individual modules confirmed the positive contribution of each module, and notably, we found that the same modules are so effective that they can improve the recall of even a freshly built static HNSW index. This highlights that PRO-HNSW serves not just as a reactive repair tool, but as a proactive graph optimization framework. PRO-HNSW thus offers a practical and robust solution for enhancing the resilience and search performance of HNSW indexes in evolving real-world applications, making graph-based ANNS more viable for dynamic real-world applications.

Acknowledgment

This research was supported by the National Research Foundation (NRF) funded by the Korean government (MSIT) (No. RS-2023-00229822).

REFERENCES

[1] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM conference on recommender systems*, 2016, pp. 191–198.

[2] R. Chen, B. Liu, H. Zhu, Y. Wang, Q. Li, B. Ma, Q. Hua, J. Jiang, Y. Xu, H. Deng *et al.*, "Approximate nearest neighbor search under neural similarity metric for large-scale recommendation," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022, pp. 3013–3022.

[3] S. Okura, Y. Tagami, S. Ono, and A. Tajima, "Embedding-based news recommendation for millions of users," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 1933–1942.

[4] J. Li, H. Liu, C. Gui, J. Chen, Z. Ni, N. Wang, and Y. Chen, "The design and implementation of a real time visual search system on jd e-commerce platform," in *Proceedings of the 19th International Middleware Conference Industry*, 2018, pp. 9–16.

[5] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

[6] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.

[7] H. Li, Y. Su, D. Cai, Y. Wang, and L. Liu, "A survey on retrieval-augmented text generation," *arXiv preprint arXiv:2202.01110*, 2022.

[8] K. Santhanam, O. Khattab, J. Saad-Falcon, C. Potts, and M. Zaharia, "Colbertv2: Effective and efficient retrieval via lightweight late interaction," *arXiv preprint arXiv:2112.01488*, 2021.

[9] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[10] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri, "Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search," *arXiv preprint arXiv:2105.09613*, 2021.

[11] W. Xiao, Y. Zhan, R. Xi, M. Hou, and J. Liao, "Enhancing hnsw index for real-time updates: Addressing unreachable points and performance degradation," *arXiv preprint arXiv:2407.07871*, 2024.

[12] R. Bellman, "Dynamic programming," *science*, vol. 153, no. 3731, pp. 34–37, 1966.

[13] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[14] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.

[15] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 1984, pp. 47–57.

[16] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[17] S. Dasgupta and Y. Freund, "Random projection trees and low dimensional manifolds," in *Proceedings of the fortieth annual ACM symposium on Theory of computing*, 2008, pp. 537–546.

[18] S. Dasgupta and K. Sinha, "Randomized partition trees for exact nearest neighbor search," in *Conference on learning theory*. PMLR, 2013, pp. 317–337.

[19] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604–613.

[20] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "Spann: Highly-efficient billion-scale approximate nearest neighborhood search," *Advances in Neural Information Processing Systems*, vol. 34, pp. 5199–5212, 2021.

[21] A. Andoni and I. Razenshteyn, "Optimal data-dependent hashing for approximate near neighbors," in *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, 2015, pp. 793–801.

[22] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, 2004, pp. 253–262.

[23] B. Harwood and T. Drummond, "Fanng: Fast approximate nearest neighbour graphs," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 5713–5722.

[24] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *arXiv preprint arXiv:1707.00143*, 2017.

[25] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," *Advances in neural information processing Systems*, vol. 32, 2019.

[26] M. Aumüller, E. Bernhardsson, and A. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Information Systems*, vol. 87, p. 101374, 2020.

[27] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *arXiv preprint arXiv:2101.12631*, 2021.

[28] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.

[29] M. Iwasaki, "Neighborhood graph and tree for indexing highdimensional data," *Yahoo Japan Corporation. Retrieved August*, vol. 22, p. 2020, 2015.

[30] Q. Chen, H. Wang, M. Li, G. Ren, S. Li, J. Zhu, J. Li, C. Liu, L. Zhang, and J. Wang, "Sptag: A library for fast approximate nearest neighbor search," 2018.

[31] Z. Jin, D. Zhang, Y. Hu, S. Lin, D. Cai, and X. He, "Fast and accurate hashing via iterative nearest neighbors expansion," *IEEE transactions on cybernetics*, vol. 44, no. 11, pp. 2167–2177, 2014.

[32] C. Fu and D. Cai, "Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph," *arXiv preprint arXiv:1609.07228*, 2016.

[33] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 8, pp. 1475–1488, 2019.

[34] J. V. Munoz, M. A. Gonçalves, Z. Dias, and R. d. S. Torres, "Hierarchical clustering-based graphs for large scale approximate nearest neighbor search," *Pattern Recognition*, vol. 96, p. 106970, 2019.

[35] C. Fu, C. Wang, and D. Cai, "High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 8, pp. 4139–4150, 2021.

[36] S. Yu, S. Lin, S. Gong, Y. Xie, R. Liu, Y. Zhou, J. Sun, Y. Zhang, G. Li, and G. Yu, "A topology-aware localized update strategy for graph-based ann index," *arXiv preprint arXiv:2503.00402*, 2025.

[37] F. Zardbani, N. Mamoulis, S. Idreos, and P. Karras, "Adaptive indexing of objects with spatial extent," *Proceedings of the VLDB Endowment*, vol. 16, no. 9, pp. 2248–2260, 2023.

[38] A. Babenko and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2055–2063.

[39] D. Newman, "Bag of Words," UCI Machine Learning Repository, 2008, DOI: https://doi.org/10.24432/C5ZG6P.

[40] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 2002.

[41] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[42] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer vision–ECCV 2014: 13th European conference, zurich, Switzerland, September 6-12, 2014, proceedings, part v 13*. Springer, 2014, pp. 740–755.

[43] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.