CleanANN: Efficient and Robust Full Dynamism in Graph-based Approximate Nearest Neighbor Search

Ziyu Zhang
Massachusetts Institute of Technology
Cambridge, USA
sylziyuz@mit.edu

Yuanhao Wei*
University of British Columbia
Vancouver, Canada
yuanhaow@cs.ubc.ca

Joshua Engels
Massachusetts Institute of Technology
Cambridge, USA
jengels@mit.edu

Julian Shun
Massachusetts Institute of Technology
Cambridge, USA
jshun@mit.edu

Abstract

The approximate nearest neighbor search (ANNS) problem has important applications, such as robotics, data mining, semantic search, and unstructured data retrieval. Graph-based ANNS indexes have superb empirical tradeoffs in indexing cost, query efficiency, and query approximation quality. Most existing graph-based indexes are designed for the static scenario, where there are no updates to the data after the index is constructed. However, full dynamism (insertions, deletions, and searches) is crucial to providing up-to-date responses in applications using vector databases. It is desirable that the index efficiently supports updates and search queries concurrently. Existing dynamic graph-based indexes have the following shortcomings: (1) graph repair operations incurred by delete queries involve many graph updates, hindering scalability; (2) data distribution shift and out-of-distribution queries are not addressed; and (3) in efficient systems implemented as directed graphs, clearing incoming edges after deletions is a global batch operation, delaying resource reuse and leading to periodic regressions in query quality.

To solve these problems, we propose the CleanANN system, which consists of three main components: (1) workload-aware linking of diverse search tree descendants to combat distribution shift; (2) query-adaptive on-the-fly neighborhood consolidation to efficiently handle deleted nodes; and (3) lock-free semi-lazy memory cleaning to accelerate the cleaning of stale information in the data structure and reduce the work spent by the first two components.

We evaluate CleanANN on three diverse datasets on fully-dynamic workloads and find that CleanANN exhibits higher overall throughput while delivering similar query quality compared to previous state-of-the-art solutions, particularly in more challenging datasets with data distribution shift and out-of-distribution queries. On 64 threads, compared to the recent state-of-the-art system Wolverine++, CleanANN delivers 12–18× overall throughput. Compared to the industry state-of-the-art IP-DiskANN system, CleanANN delivers 1.14–1.26× overall throughput.

CCS Concepts

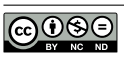
• **Information systems** → **Search engine indexing**; **Proximity search**; *Stream management*; • **Computing methodologies** → **Concurrent algorithms**; Knowledge representation and reasoning.

Keywords

Approximate nearest neighbor search, graph-based index

ACM Reference Format:
Ziyu Zhang, Yuanhao Wei, Joshua Engels, and Julian Shun. 2026. CleanANN: Efficient and Robust Full Dynamism in Graph-based Approximate Nearest Neighbor Search. In *38th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '26), July 6–10, 2026, London, United Kingdom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3816782.3819219

*This work was done while at Massachusetts Institute of Technology.



SPAA '26, London, United Kingdom

ACM ISBN 979-8-4007-2761-0/26/07
https://doi.org/10.1145/3816782.3819219

1 Introduction

k-nearest neighbor search is a well-studied problem with many applications where the relevance of data points can be captured by vector similarities, such as search engines [28], data clustering [3], and anomaly detection [43]. In these applications, datasets can be large and have several hundred dimensions, and need to be searched quickly and accurately.

Definition 1 (k-Nearest Neighbors (kNN)). *Given a dataset D with $|D| \gg k$ in a space M with distance function $d(\cdot,\cdot)$, for a query $q \in M$, its set of k nearest neighbors $kNN(q)$ is a subset of D of size k satisfying $\forall x \in D \setminus kNN(q), \forall y \in kNN(q), d(y,q) \leq d(x,q)$.*

Solving kNN exactly in high dimensions is believed to be computationally hard [42], and thus researchers focus on the approximate nearest neighbor search problem (ANNS) instead. This problem has become increasingly important due to the ubiquity of vector embedding-based semantic search for complex, multi-modal, unstructured data in AI applications [26]. Data often change continuously in many applications that need ANNS, such as recommendation systems [47], autonomous agents [37], and robotics [39]. Dynamism in ANNS is important for data systems that support efficient data updates and semantic search, as this enables fresh responses to real-time inputs in AI applications. This motivates the need for a *dynamic* ANNS index that can keep up with, for example, the 500+ hours of content uploaded to YouTube every minute, the one billion images updated on JD.com every day [27], or the constant ingestion of emails and code by an AI assistant. We are interested in the realistic *full dynamism* setting, where insertions, deletions, and searches can all happen concurrently.

Graph-based indexes such as Vamana [18] (the in-memory version of DiskANN), HNSW [34], NSG [9], and NGT [17] have been shown [1, 35] to achieve state-of-the-art performance on static

data (i.e., the index is not updated).[1] However, existing dynamic graph-based indexes [13, 23, 29, 46, 50, 54] suffer from at least one of the following problems: (1) graph repair operations incurred by delete queries involve many graph updates, hindering scalability; (2) data distribution shift and out-of-distribution queries are not addressed; and (3) in efficient systems implemented as directed graphs, clearing incoming edges after deletions is a global batch operation, delaying memory reuse and leading to periodic regressions. We present these problems in detail in Section 1.1.

We propose several techniques to address these issues. We propose *bridge building*, a data-aware method that improves the robustness of graph-based indexes under full dynamism. The key idea is to add additional edges between visited points that are moderately far apart during graph traversals. In addition, we propose a *dynamic neighborhood consolidation* method that identifies useful graph repair operations around deleted points and performs these repairs on the fly, without requiring global operations that degrade system throughput. We also propose *semi-lazy cleaning*, a novel memory management strategy that efficiently cleans deleted nodes in the graph while maintaining graph quality, eliminating reliance on global batch cleaning operations and their associated regressions in query quality. We present CleanANN, an ANNS index that achieves efficient full dynamism using these techniques. CleanANN supports any mix of concurrent insertions, deletions, and searches.

1.1 Limitations of Previous Work

Graph-based Indexes. In a graph-based ANNS index $G = (V, E)$ for dataset D, each node $v_x \in V$ represents a data point $x \in D$. Each graph-based index has its own algorithm for deciding how to connect edges among these nodes. Given a query point $q \in M$, the index generally conducts a *greedy search* for q on G from a set of starting nodes, i.e., the search repeatedly explores the node closest to q among unexplored neighbors in the search frontier. When there are no nodes among the unexplored neighbors closer than the best visited node, the best visited nodes are reported as the approximate k-nearest neighbors, or *aKNN(q)*. The graph is usually constructed under a *sparsity constraint*, typically expressed as an upper bound for the out-degree, to bound the complexity of the greedy searches. In practice, the de facto measure for the quality of *aKNN(q)* is the **recall**:[2]

Definition 2 (Recall k@k). *Given a dataset D and a query q, let $aKNN(q) \subseteq D$ with $|aKNN(q)| = k$ be the result set returned by an ANNS algorithm Alg. The recall k @ k of Alg for q is $\frac{1}{k}|kNN(q) \cap aKNN(q)|$.*

Supporting dynamic operations, especially deletions, is a notoriously difficult problem for graph-based ANNS indexes. Existing dynamic graph-based indexes [13, 23, 29, 46, 50, 54] suffer from at least one of the following problems:

[1]The best performing systems evaluated in [1] are graph indexes with fine-grained optimizations such as quantization. These optimizations can be applied independently to the graph structure techniques that we focus on in this paper.
[2]Usually "recall" means the proportion of correct data points returned, and "precision" means the proportion of correct data points among the ones returned. The measurement of interest here incorporates both and is colloquially referred to as recall.

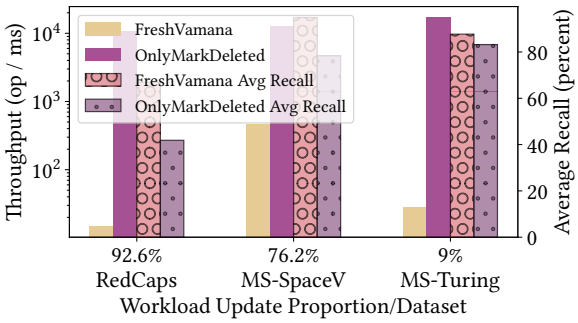


Figure 1: Search throughput and average recall with (FreshVamana) or without (OnlyMarkDeleted) concurrent global consolidation. The datasets are described in Section 5.

Challenge 1 (Deletion Efficiency Issue). *Deletion algorithms involve a number of graph updates not bounded by the sparsity constraints of the graph, hindering performance.*

Challenge 2 (Robustness Issue). *Deletion algorithms are local to the deleted points, and do not account for the insertion and search workloads or the overall graph structure. The insertion algorithms are not robust with respect to the order in which data points are inserted.*

Challenge 3 (Data Structure Staleness Issue). *In efficient graph systems implemented as directed graphs, handling remaining incoming edges to deleted points relies on batch operations, delaying resource reuse and leading to periodic regressions in query quality.*

Deletion Efficiency Issue. For each deleted point p, IP-DiskANN selects in-neighbors of p encountered during a search for p. Since there is no upper bound on the in-degree of graph nodes (as opposed to the out-degree), the number of graph updates needed to repair deletions in these methods can be high. While Wolverine++ [29] bounds the number of in-neighbors updated to repair the graph in the case of a Delete, it may inspect the full 2-hop neighborhood of neighbors of p, resulting in expensive Delete queries.

When data points are deleted, FreshVamana [46] (the in-memory version of FreshDiskANN) first marks the corresponding nodes as *tombstones* to filter them out in the responses for subsequent Search queries, and then periodically performs a global scan (known as *consolidation*) across the entire index to update edges and remove tombstones. The consolidation algorithm tries to connect all in-neighbors of each tombstone to all out-neighbors of the same tombstone, which incurs a significant cost. Our experiments show that the system throughput of FreshVamana can decrease by more than 10x during consolidation; however, consolidation is required to achieve good Search recall (Figure 1). LSH-APG [54] uses the same approach of connecting all in-neighbors to all out-neighbors to fix the graph after a node is deleted, implying similar efficiency problems.

Robustness Issue. Graph-based ANNS indexes are generally constructed by incrementally adding data points. A new data point $x \in D$ is added via the following insertion routine:

Routine 1 (New Data Point Insertion).
(1) Create a node $v_x \in V$ to represent x.
(2) Find a set of candidate neighbor nodes C by conducting a best-first search for x.

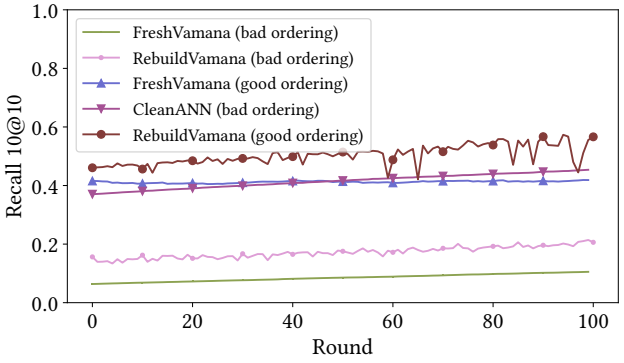


Figure 2: Synthetic dataset with a clustered distribution. A good ordering inserts whole clusters together and a bad ordering is uniformly random.

(3) Choose good neighbors $C^\star \subseteq C$ with some index-specific heuristics and connect v_x to $C^\star$.

For building the index, previous approaches combine Routine 1 with some additional optimizations. For example, HNSW [34] imposes a global hierarchical structure over the candidate set C in step (2) and the edge choices in step (3); and FANNG [12] runs global iterative optimizations after running Routine 1 on each data point. Naturally, Routine 1 is used as an Insert algorithm for indexes. However, since the global optimizations mentioned above do not directly apply to the dynamic setting, Routine 1 when used as an Insert algorithm faces robustness issues.

Suppose a node v_x has already been inserted. For a node v_y inserted later, let $C_y^\star$ be the nodes already in G that are good neighbors for v_y. An edge from v_x to v_y is created only if $v_x \in C_y^\star$. However, v_y being a good neighbor for v_x does not necessarily imply the converse, and an edge from v_x to v_y may never be created even if it is useful. Without global optimizations during the index build, Routine 1, when used as an Insert algorithm, is sensitive to the ordering in which points are inserted and therefore not *robust*.

In Figure 2, 1% of the dataset is inserted into the index on every round. We run a query batch after each insert batch, reporting the recall. FreshVamana, which directly uses Routine 1 to insert data points, shows a large recall difference with different orderings. Compared to RebuildVamana, which builds a Vamana index from scratch every round (a very inefficient algorithm in the dynamic setting), FreshVamana has about a 10% lower recall under the same bad ordering, since it does not benefit from the global two-pass optimization of RebuildVamana. These comparisons show that Routine 1 when used as an insertion algorithm is not robust.

Data Structure Staleness Issue. Since most performant graph indexes are implemented as adjacency lists that only store out-neighbors, given a deleted point p, one must scan the entire index to reliably identify all the in-neighbors of p. This is expensive, and thus existing methods (IP-DiskANN and FreshDiskANN) rely on periodic global batch operations to remove incoming edges to deleted points and free the corresponding resources. The amount of stale information (i.e., *bloat*) accumulates over time and leads to query quality degradation until the bloat is removed by periodic cleaning operations, as we will see in Section 5.

1.2 Our Contributions

In this paper, we present the CleanANN system with three main techniques that address the limitations discussed in Section 1.1.

(1) We propose *bridge building*, a data-aware algorithm that improves the robustness of dynamic graph-based indexes. The key idea is to add additional edges between visited points that are moderately far apart during an insertion. This helps address the insertion robustness issue and allows the algorithm to achieve good recall without needing a good ordering in the workload. Figure 2 shows that CleanANN with a bad ordering has the same average recall as FreshVamana with a good ordering.
(2) We propose a *dynamic neighborhood consolidation* method that makes graph repairs around deleted points on the fly, avoiding global operations that degrade system throughput. The graph repair cost needed to actually delete a point is amortized across future operations.
(3) We propose *semi-lazy cleaning*, a novel memory management strategy that efficiently cleans deleted nodes in the graph while maintaining graph quality. The key idea is to stop the consolidation and recycle the deleted nodes early to reduce work. Together with dynamic neighborhood consolidation, this addresses the deletion efficiency issue, enabling the system to achieve both high throughput and good recall.

Our techniques allow updates and queries to run concurrently. We evaluate CleanANN on three datasets. Our main experiments focus on the sliding window setting. We use a batch-based design to measure recall under dynamism and concurrency following previous work [18, 29, 50, 51]. For each sliding window, we first issue an update batch that concurrently deletes old data points and inserts new data points into the index. Then, a batch of search queries runs while the graph repairs incurred by the updates continue to run concurrently. We also present efficiency results from the same sliding windows but with inserts, deletes, and searches running concurrently. We compare against two recent state-of-the-art systems, IP-DiskANN [50] and Wolverine++ [29]. To ensure accurate comparison, we implemented CleanANN and Wolverine++ [29] on the same base index as IP-DiskANN [50].

Our experiments show that

(1) CleanANN efficiently supports both Inserts and Deletes without requiring expensive global operations or periodic rebuilds. In the full dynamism setting, CleanANN achieves higher overall throughput across different datasets and workloads while maintaining similar or higher recall than two recent state-of-the-art baseline systems Wolverine++ [29] and IP-DiskANN [50]. On 64 threads, compared to Wolverine++ [29], CleanANN delivers 12–18× overall throughput. Compared to the industry state-of-the-art system IP-DiskANN [50], CleanANN delivers 1.14–1.26× overall throughput.
(2) CleanANN maintains high recall and is robust against different insertion orderings and distribution shifts in the workload.
(3) CleanANN scales well with respect to both dataset size and thread count, achieving 30–100 concurrent operations/ms on 64 threads.

Table 1: Notation

Notation	Meaning
$d(p, q)$	Distance between points p and q
R	Graph out-degree bound
L	Backtracking budget for greedy search
L_I	Backtracking budget for greedy search during insertion
$N(v)$	Out-neighborhood of node v
$IN(v)$	In-neighborhood of node v
α	Sparsity heuristic parameter
$\pi(\cdot)$	Parent of node in a tree
C	Eagerness threshold

2 Preliminaries

In graph-based ANNS indexes, each node corresponds to a data point. We use the letters q, x, and y to denote data points and u, v, and w to denote graph nodes. A subscript on a node (e.g., v_x) means that v_x represents the data point x. Uppercase letters in the script typeface (e.g., $\mathcal{C}$ and $\mathcal{N}$) represent sets of graph nodes or edges. Subscripts on algorithm names (e.g., GreedyBeamSearch$_L$) denote parameters for the algorithms. Table 1 lists more notation that we use in our algorithm descriptions.

The vanilla GreedySearch algorithm common in graph-based ANNS indexes may converge to a local optimum. Therefore, ANNS indexes commonly *backtrack* to explore a few close-to-optimal unexplored nodes in the frontier to escape the local optimum. Our implementation is based on the Vamana graph. We refer readers to the FreshDiskANN [46] paper for details of algorithms in Vamana: the backtracking GreedySearch (Algorithm 1, [46]), Insert (Algorithm 2, [46]), and the RobustPrune (Algorithm 3, [46]) graph pruning algorithm in Vamana.

3 Guided Bridge Building

In the static setting, previous work generally combines Routine 1 with global optimizations to build the index. Existing methods [9, 12, 18, 32, 34, 46, 54] use the same routine for insertions after building the index, including FreshVamana, which we implement our algorithms on, and IP-DiskANN and Wolverine++ which we compare with. As discussed in Section 1.1, using Routine 1 as Insert is not robust and exhibits search quality degradation compared to a static build.

Our algorithm GuidedBridgeBuild addresses this issue by adaptively introducing new edges based on the nodes traversed in GreedySearch. We will see that guided bridge building is critical to maintaining the quality of the index in the fully-dynamic setting in Section 5, improving both efficiency and quality for Search queries.

3.1 Setup and Idea

We explain GuidedBridgeBuild in the context of FreshVamana, which uses GreedySearch (Algorithm 1, [46]) with RobustPrune (Algorithm 3, [46]) under the framework of Routine 1 to implement Insert. GuidedBridgeBuild selectively adds edges among the nodes visited by GreedySearch. Our method is orthogonal to the specifics of RobustPrune and backtracking in FreshVamana and may also be used to improve other graph-based indexes.

Let $\mathcal{V}$ be the set of nodes explored by the search phase of Insert. $\mathcal{V}$ can be viewed as a *search tree* $\mathcal{T}$, where v is the parent of w (denoted as $v = \pi(w)$) if w was added to the search *frontier* during the exploration of v, i.e., $w \in N(v)$ and w had not been visited when v was explored. To insert q, existing methods [9, 34, 46] connect q to a subset of $\mathcal{V}$. FreshVamana chooses this subset from $\mathcal{V}$ with RobustPrune, which connects q with nodes close to it in diverse directions to improve search quality while promoting shortcut edges to improve efficiency.

Our key observation is that creating bridge edges among *cousins* in moderately deep levels (younger generations) of the search tree during a fully-dynamic workload creates paths that help with the robustness issue in Insert, similar to what static ANNS indexes achieve with global operations during the index construction phase.

Deeper-level cousins in the same search tree are likely to be nodes relatively close but not connected to each other via a short path, which can be caused by the robustness problem described in Section 1.1. As these cousin nodes are spatially close to each other and to the query, subsequent queries will likely need to navigate among these vertices. Without direct connections between these nodes, search queries require a larger backtracking budget to navigate between them, possibly resulting in early pruning of the search, which leads to lower search quality.

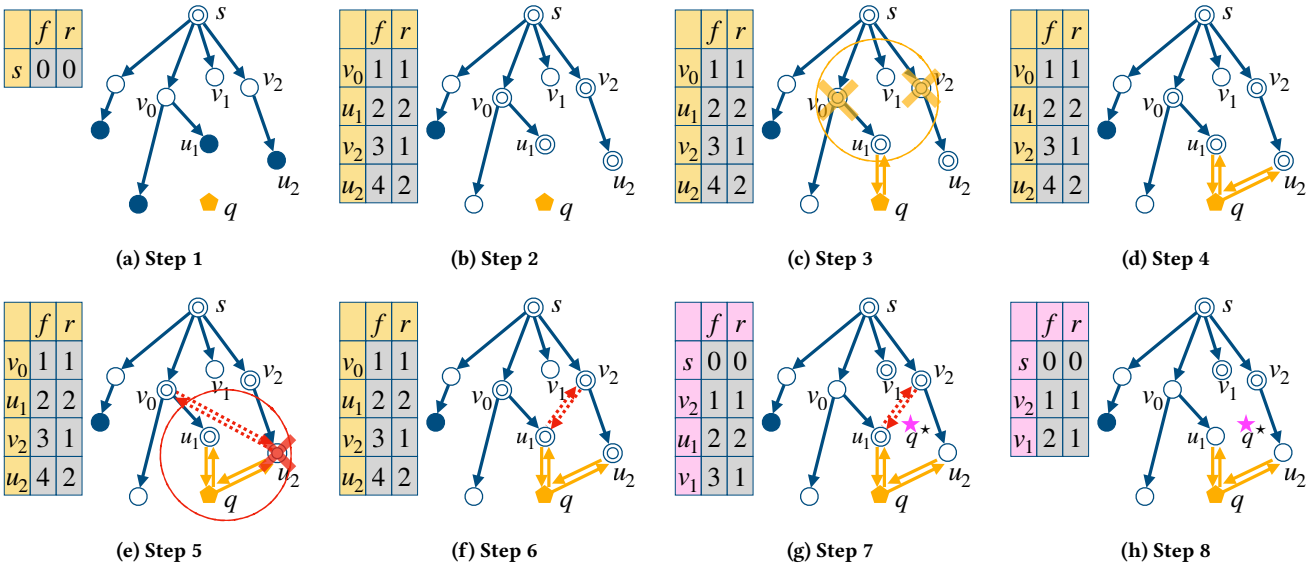
GuidedBridgeBuild introduces these missing connections. In addition to benefits in the dynamic case, we also find that GuidedBridgeBuild can improve the recall of static ANNS indexes by improving their robustness against bad orderings during index construction.

3.2 Algorithm Description

GuidedBridgeBuild identifies good bridge edges via an augmented GreedySearch: BridgeBuilderSearch (Algorithm 1). Lines 6–11 initialize the search frontier, the set of explored nodes, and the search tree $\mathcal{T}$. Each iteration of Lines 12–17 is a search step exploring the best unexplored node w. On Line 18, the algorithm marks unvisited out-neighbors of w added to the frontier in the current iteration as the children of w in $\mathcal{T}$. Line 19 invokes GuidedBridgeBuild to add the bridge edges based on $\mathcal{T}$. Nodes with search tree depths specified in $\mathcal{S}$ are tentatively bi-directionally connected (Lines 22–26), subject to additional heuristic constraints by a commutative predicate HeuristicPredicate (Line 25). Lastly, Line 27 finalizes the bridge edges and performs additional pruning if needed via Algorithm 2.

GuidedBridgeBuild applies to both Insert and Search queries.

3.2.1 Insert with GuidedBridgeBuild. Figure 3 illustrates RobustInsert, which uses GuidedBridgeBuild with insert queries. In the example, $L = 2$ and $R = 2$, and degree constraint R does not apply to the starting node s. $r(\cdot)$ is the depth in the search tree and $f(\cdot)$ is the order in which nodes are explored. q is the new data point being inserted. Steps 1–2 (Subfigures 3a–3b) explore $\mathcal{V} = \{s, v_0, u_1, v_2, u_2\}$. On Step 2, the search terminates with $L = 2$ and the local minima of $d(\cdot, q)$: u_1 and u_2. On Line 4 in Algorithm 3, RobustInsert uses RobustPrune to select a subset of neighbors for q from $\mathcal{V}$. In Step 3 (Subfigure 3c), RobustPrune selects u_1 as the closest neighbor candidate, pruning v_0 and v_2. u_2 is subsequently selected as the next neighbor, reaching the out-degree bound $R = 2$. q is therefore connected to u_1 and u_2 (Subfigure 3d).



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

(e) Step 5 (f) Step 6 (g) Step 7 (h) Step 8

Figure 3: Step-by-step illustration of RobustInsert (Algorithm 3). Solid blue edges are existing graph edges. Single hollow circles are the search frontier and double hollow circles are explored nodes. Solid yellow edges are graph edges added from inserting q, and dotted edges are bridge edges from inserting q. The × marks in Subfigures 3c and 3e are neighbor candidates pruned out by RobustPrune.

Algorithm 1 Guided Bridge Building

$\mathcal{S}$: Set of tree depths considered
HeuristicPredicate: Boolean function for imposing constraints on the bridge edges
$r(v) \equiv$ depth of v in $\mathcal{T}$
StartIds : Fixed or separately computed node set where the search starts
procedure BridgeBuilderSearch(q)
  frontier = *StartIds*
  $\mathcal{V} = \emptyset$ ▷ Explored nodes
  $\mathcal{T} = \emptyset$ ▷ Search tree
  for $s \in$ *StartIds* **do**
    Set $\pi(s) =$ *null* in $\mathcal{T}$
  $\mathcal{L} \equiv$ best L nodes ever seen in *frontier*
  while *frontier* $\cap \mathcal{L} \neq \emptyset$ **do**
    $w = \text{argmin}_{v_x \in \text{frontier} \cap \mathcal{L}} d(x, q)$
    $\mathcal{V} = \mathcal{V} \cup \{w\}$
    frontier = *frontier* $\setminus \{w\}$
    for $u \in N(w), u \notin \mathcal{V}$ **do**
      frontier = *frontier* $\cup \{u\}$
      Set $\pi(u) = w$ in $\mathcal{T}$
  GuidedBridgeBuild($\mathcal{T}$)
  return $\mathcal{L}, \mathcal{V}$
procedure GuidedBridgeBuild($\mathcal{T}$)
  for $v \in \mathcal{T}$ such that $r(v) \in \mathcal{S}$ **do**
    $C = \emptyset$
    for $w \in \mathcal{T}$ such that $r(w) \in \mathcal{S}$, and $w \neq v$ **do**
      if HeuristicPredicate(v, w) **then**
        $C = C \cup \{w\}$
    AddNeighbors(v, C) (Algorithm 2)

Next, GuidedBridgeBuild with $\mathcal{S} = \{1, 2\}$ and no HeuristicPredicate considers bridge edges between nodes satisfying $r(\cdot) \in \mathcal{S}$: edges $v_0 \leftrightarrow u_2$ and $u_1 \leftrightarrow v_2$. Subfigure 3e illustrates u_1 causing $v_0 \rightarrow u_2$ to be pruned. Similarly, v_2 causes $u_2 \rightarrow v_0$ to be pruned.

Algorithm 2 Add Neighbors

R: Global out-degree bound
procedure AddNeighbors(v, C)
  $\mathcal{N} = N(v) \cup C$
  if $|\mathcal{N}| \leq R$ **then** $N(v) = \mathcal{N}$
  else $N(v) =$ RobustPrune($v, \mathcal{N}$) (Algorithm 3, [46])

Algorithm 3 Robust Insert

L_I: Insert backtracking budget
procedure RobustInsert(v_x)
  $\mathcal{L}, \mathcal{V} =$ BridgeBuilderSearch$_{L=L_I}(x)$ (Algorithm 1)
  $N(v_x) =$ RobustPrune($v_x, \mathcal{V}$) (Algorithm 3, [46])
  for $w \in N(v_x)$ **do** AddNeighbors($w, \{v_x\}$) (Algorithm 2)

The edges $u_1 \leftrightarrow v_2$ are added as bridge edges (Subfigure 3f), concluding RobustInsert.

This example also shows the algorithmic benefits of GuidedBridgeBuild. Subfigure 3g illustrates a search for $q^\star$ with the bridge edges present in the graph. The nodes s, v_2, u_1, and v_1 are explored in this order, correctly finding u_1, the correct nearest neighbor for $q^\star$. Subfigure 3h illustrates the search for $q^\star$ without bridge edges. s, v_2, and v_1 are explored, but the search terminates at the incorrect local optima, v_1 and v_2. v_0, which navigates to u_1, is too far from the query compared to the other siblings of v_0. With the same backtracking budget as when the search uses bridge edges, v_0 and u_1 are missed in the search. Therefore, the bridge edges between u_1 and v_2 improve the navigability of the graph.

Algorithm 3 lists our insertion algorithm RobustInsert. Given a new node v_x, RobustInsert first runs BridgeBuilderSearch for x, and then selects $N(v_x)$ from $\mathcal{V}$ (nodes explored by BridgeBuilderSearch) and tries to add v_x to $N(w)$ for each $w \in N(v_x)$, using RobustPrune as needed.

3.2.2 Search with GuidedBridgeBuild. A Search query can also improve the part of the index that it traverses with BridgeBuilder-Search. Searches for other nearby points traverse similar parts of the graph and thus benefit from the improved navigability. We refer to Search queries that perform GuidedBridgeBuild as *training* queries hereafter. In Section 5, we find experimentally that training with even a small number of in-distribution queries helps the index adapt to distribution shifts in a query-aware fashion.

3.2.3 Hyperparameters $\mathcal{S}$ and HeuristicPredicate. We found experimentally that $\Theta(\log(|D|))$ with a $\Theta(1)$ width is a good range for $\mathcal{S}$. HeuristicPredicate further filters out candidate bridge edges to balance between the cost of potentially performing more RobustPrunes for bridge edges and the robustness benefits these additional connections provide. By default, HeuristicPredicate in our experiments requires two endpoints of a bridge to have the same depth in $\mathcal{T}$. We also compare this heuristic to several alternatives in Section 5.

3.2.4 Summary. Our bridge building algorithm has three key benefits: (1) indexes whose construction and insertions use our algorithm have increased search quality under the same backtracking budget compared to FreshVamana, achieving a better search quality-efficiency tradeoff; (2) in a sliding window update scenario, our algorithm consistently improves the recall of the index throughout updates; and (3) our method can effectively adapt to distribution shifts and out-of-distribution queries. One can adapt GuidedBridgeBuild to other graph-based indexes by plugging in their backtracking and neighbor pruning mechanisms.

4 Dynamic Concurrent Data Structure Cleaning

In this section, we discuss how to handle Deletes concurrently with Inserts and Searches. We guarantee that a client will not see a deleted point in the results of a Search query executed after the Delete has completed. To achieve this consistency guarantee, it suffices to mark a deleted point as a ***tombstone***, run subsequent GreedySearches normally, and filter out the tombstones when selecting the k best neighbors from $\mathcal{L}$. However, as the index becomes more bloated, the accumulated tombstones interfere with the graph traversal, leading to a degradation in recall, as observed by Xu et al. [51] and in our experiments. Therefore, to achieve high search quality, we need to remove the tombstones and repair the graph. Previous graph-based ANNS indexes use costly global operations to implement graph repairs [13, 46, 54]. If the graph repair or rebuild operations are performed in batches, more expensive searches with larger backtracking budgets (e.g., the pgvector HNSW iterative scan [22]) are required to maintain recall. To address these issues, we propose adaptive dynamic techniques that perform graph repairs and cleaning more efficiently.

4.1 Algorithm Description

We first outline the life cycle of any graph node and the corresponding states:

- **Live**: The node represents a valid data point.
- **Tombstone**: The data corresponding to the node is logically deleted and will not be returned by a Search query. The graph in- and out-neighborhoods and the data value of the node are still valid for graph traversal.
- **Replaceable**: The node slot is available in M for a new insertion. The corresponding data and the out-neighborhood are no longer valid but there may be remaining in-neighbors.

Algorithm 4 Consolidation (Subroutine of Algorithm 2 in [46])

```
1: R : Global out-degree bound
2: procedure Consolidate(v)
3:     C = ∅                                          ▷ Neighbor candidates
4:     for w ∈ N(v) do
5:         if IsLive(w) then C = C ∪ {w}
6:         else C = C ∪ {u ∈ N(w) | u ≠ v ∧ IsLive(u)}
7:     if |C| ≤ R then N(v) = C
8:     else N(v) = RobustPrune(v, C)
```

Throughout the discussion, let w_x be the tombstone node to be removed. Concretely, Delete had previously been called on x, the index has marked w_x as a tombstone, and the graph needs to be repaired to prepare for the actual deletion of w_x. We propose performing neighborhood consolidation on the fly when needed and an accompanying semi-lazy memory management strategy. The discussion and implementation are based on Vamana, but the techniques are applicable to any graph-based ANNS index.

Efficient Consolidation on the Fly. GreedySearch can detect tombstone nodes used for its navigation. If GreedySearch adds w_x to the search frontier when exploring v, i.e., $v = \pi(w_x)$ in $\mathcal{T}$, and v is live, then the edge $v \to w_x$ facilitated the $v \to w_x \to N(w_x)$ navigation. Therefore, it is useful to make v absorb $N(w_x)$ via neighborhood consolidation before we actually delete w_x. This can be carried out by either Insert or Search queries. Since all of v's unvisited tombstone neighbors are added to the frontier during the same search iteration, it is cost-effective for v to consolidate with all of its tombstone neighbors' out-neighborhoods (Consolidate in Algorithm 4). This makes tombstone nodes ready to be removed from the graph earlier.

Early Stopping of Consolidations. Immediately removing w_x after only one in-neighbor v detects and consolidates with it does not sufficiently repair the graph around w_x. However, waiting for all incoming neighbors of w_x to reach w_x through a GreedySearch and consolidate with w_x reduces to naively amortizing FreshVamana's expensive periodic consolidation of connecting all in-neighbors to all out-neighbors of a deleted node. Since our method chooses effective consolidation targets on the fly, we find that it is unnecessary to connect all nodes in $IN(w_x)$ to all nodes in $N(w_x)$, and that it suffices to visit a tombstone w_x just a few times with Consolidate(v) ($v = \pi(w_x)$ in some search tree $\mathcal{T}$) to preserve w_x's navigation functionality. Therefore, we maintain a count for each tombstone of the number of times it has been visited by Consolidate from a live parent and stop performing more Consolidates for w_x after the count hits a threshold C (which we call the eagerness threshold). We present an analysis of the sensitivity of C in Section 5.

Semi-Lazy Cleaning. The remaining issue is to remove a tombstone node w_x from the graph after it has been processed by live parent nodes a sufficient number of times with Consolidate. Existing methods (FreshVamana [46], IP-DiskANN [50], and pgvector HNSW [23]) remove dangling edges ($w_i \to w_x$) for any remaining $w_i \in IN(w_x)$ that has not consolidated with w_x with costly batch

Algorithm 5 Try Marking a Node as Replaceable

```
1: M: Linearizable bag data structure tracking available graph storage slots corresponding to replaceable nodes
2: C: Eagerness threshold
3: H: Per-node atomics for tracking node status ▷ For any node w, H(w) ≥ 0 iff w is a tombstone and w is not replaceable; H(w) = −2 iff w is replaceable; H(w) = −1 iff w is live
4: procedure TryMarkReplaceable(w)
5:     while true do
6:         current ← Load(H(w))
7:         if current < C then return current == −2
8:         if CAS(H(w), current, −2) then
9:             M ← M ∪ {w}
10:            return true
```

consolidations. We argue that cleaning the remaining dangling incoming edges is *not necessary*. The primary benefits of actually removing w_x and all of its *incoming* edges include (1) the storage for node w_x can be reused for future nodes that are inserted and (2) w_x's in-neighbors that did not call Consolidate on w_x's neighborhood do not unnecessarily explore w_x in future GreedySearches, avoiding potential incorrect local optima. However, if previous on-the-fly Consolidates for w_x already sufficiently repaired the graph around w_x, it is acceptable for a deleted point x to be *directly overwritten by a new data point* y, even when some in-neighbors still point to the old w_x node in the graph.

When a new data point y is inserted, if w_x has been Consolidated C times, then *RobustInsertData can turn* w_x *into* w_y, *representing* y *directly*. RobustInsert(w_y) then uses y in its distance computations and neighbor selection. The resulting $IN(w_y)$ contains both new in-neighbors from RobustInsert(w_y) and the remaining nodes from $IN(w_x)$. We call the remaining incoming edges from $IN(w_x)$ ***random edges***, with the randomness coming from the workload. We experimentally confirm that these random edges do not hurt the quality of the index, since GreedySearch naturally avoids exploring nodes that are far away, and RobustPrune naturally removes edges that are not useful. We call this technique ***semi-lazy cleaning***.

Algorithm. Algorithm 6 lists CleanDynamicSearch, the full algorithm with on-the-fly consolidation and semi-lazy memory cleaning, combined with the methods proposed in Section 3. Apart from the bookkeeping for the best-first search and GuidedBridgeBuild, we now also need the array H, which encodes the tombstone and consolidation count information (Line 5). For all live nodes w, we have $H(w) = -1$, and for all replaceable nodes, we have $H(w) = -2$. For tombstoned nodes w, $H(w)$ tracks the number of consolidations that have been performed for them, and is non-negative. The code to modify H uses *compare-and-swap (CAS)*, which takes three arguments—a memory location x, an expected old value *old*, and a desired new value *new*. It atomically does the following: reads the value of x, if it is equal to *old*, then writes *new* to x and returns *true*; otherwise, leaves x unchanged and returns *false*.

During the best-first search, after the current best node w is selected to be explored and marked as explored (Lines 12–14), for each neighbor u of w, if u is a tombstone and has been consolidated for at least C times, u is marked replaceable and is available to represent a new point (Line 15) via setting $H(u) = -2$. On Lines 15–21, w is explored, adding its unvisited out-neighbors to the frontier and recording them as its children in the search tree $\mathcal{T}$. If any of these children are tombstoned nodes, the algorithm performs CleanConsolidate(w) (Algorithm 7), consolidating the neighborhoods of all of w's deleted out-neighbors with w's neighborhood,

Algorithm 6 Clean Dynamic Search

```
1: L: Search backtracking budget
2: StartIds : Fixed or separately computed node set where the search starts
3: M: Linearizable bag data structure tracking available graph storage slots corresponding to replaceable nodes
4: C: Eagerness threshold
5: H: Per-node atomics for tracking node status
6: procedure CleanDynamicSearch(q, performance_sensitive)
7:     T: Data structure for maintaining the search tree
8:     frontier = StartIds
9:     L = StartIds ▷ Maintains the best L nodes visited
10:    V = ∅ ▷ Explored nodes
11:    while frontier ∩ L ≠ ∅ do
12:        w = argmin_{v_x ∈ frontier ∩ L} d(x, q)
13:        V = V ∪ {w}
14:        Remove w from frontier
15:        N = {u ∈ N(w) | u ∉ V, ¬TryMarkReplaceable(u)} ▷ Calls TryMarkReplaceable(u) on each u in N(w) to evaluate whether u is replaceable, and keeps nodes with H(u) > −2
16:        for u ∈ N do
17:            frontier = frontier ∪ {u}
18:            L = L ∪ {u}
19:            Delete argmax_{v_x ∈ L} d(q, x) from L if |L| > L
20:        for u ∈ N do
21:            Set w = π(u) in T
22:        if ∃u ∈ N, IsTombStoned(u) ∧ ¬IsTombStoned(w) then
23:            CleanConsolidate(w) (Algorithm 7)
24:    if ¬performance_sensitive then
25:        GuidedBridgeBuild(T) (Algorithm 1)
26:    return L, V
```

Algorithm 7 Clean Consolidation

```
1: H: Per-node atomics for tracking node status
2: w: Tombstone node whose consolidation counter is updated
3: r: Number of new consolidations
4: procedure RecordConsolidation(w, r)
5:     while true do
6:         current ← Load(H(w))
7:         if current < 0 then return
8:         if CAS(H(w), current, current + r) then return
9: procedure CleanConsolidate(v)
10:    N = {w ∈ N(v) | IsTombStoned(w)}
11:    Consolidate(v)
12:    for w ∈ N do RecordConsolidation(w, 1)
```

Algorithm 8 CleanANN Insert Data

```
1: H: Per-node atomics for tracking node status
2: M: Linearizable bag data structure tracking available graph storage slots corresponding to replaceable nodes
3: procedure RobustInsertData(x)
4:     w ← Pop any element in M
5:     current ← Load(H(w))
6:     assert(current == −2)
7:     H(w) ← −1
8:     RobustInsert(w_x) (Algorithm 3)
```

Algorithm 9 CleanANN Search

1: H: Per-node atomics for tracking node status
2: *performance_sensitive*: Boolean variable indicating whether the search is performance-sensitive.
3: **procedure** Search(k, q, *performance_sensitive*)
4: $\mathcal{L}$, _ = CleanDynamicSearch(q, *performance_sensitive*)
5: $\mathcal{L}' = \{v \in \mathcal{L} \mid H(v) = -1\}$
6: **return** k best points corresponding to nodes in $\mathcal{L}'$ using brute force comparison with q

Algorithm 10 CleanANN Delete

1: H: Per-node atomics for tracking node status
2: **procedure** Delete(w)
3: **while true do**
4: *current* ← Load($H(w)$)
5: **if** *current* ≠ −1 **then return false**
6: **if** CAS($H(w)$, *current*, 0) **then return true**

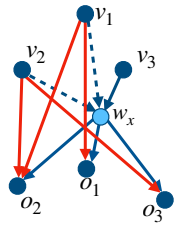


Figure 4: x is the data point being deleted. Blue edges existed before Delete(x). Red edges are added by CleanConsolidate (Algorithm 7). Solid edges exist and dashed edges are deleted. w_x is consolidated $C = 2$ times and marked replaceable.

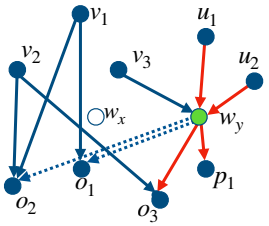


Figure 5: y replaces x to be the data point that w represents. Blue edges existed before the replacement. Red edges are determined using y. Solid edges exist after y is inserted and dashed edges are deleted as w_y's out-neighborhood is pruned using y.

incrementing H for the out-neighbors involved to record the consolidation, and pruning as appropriate (Lines 22–23).

For queries that are performance-sensitive (i.e., not training queries), GuidedBridgeBuild is not called. Performance-sensitive queries still perform on-the-fly consolidation.

Figures 4–5 illustrate an example of CleanDynamicSearch with $C = 2$. In Figure 4, initially, $N(w_x) = \{o_1, o_2, o_3\}$ and $IN(w_x) = \{v_1, v_2, v_3\}$. Delete($x$) marks w_x as a tombstone. As queries explore v_1 and v_2 via CleanDynamicSearch (Algorithm 6), v_1 and v_2 consolidate with w_x, absorbing $\{o_1, o_2, o_3\}$ into $N(v_1)$ and $N(v_2)$. The edges $(v_1 \to w_x)$ and $(v_2 \to w_x)$ are deleted. Since $C = 2$, v_3 does not perform CleanConsolidate anymore and the edge $v_3 \to w_x$ remains. w becomes available for representing other data points (it is replaceable).

In Figure 5, after y is inserted, w_x becomes w_y. $N(w_y)$ is formed with RobustInsert using y for distance computations. During RobustPrune for w_y, both the old out-neighborhood $N(w_x)$ and the visited node set $\mathcal{V}$ generated by CleanDynamicSearch(y) are candidates for $N(w_y)$. The random edge $v_3 \to w_y$ stays in the graph.

4.2 Implementing Operations using CleanDynamicSearch

Algorithm 9 lists the Search implementation in CleanANN. Similar to other graph-based indexes, it suffices to return the k best points corresponding to live nodes in $\mathcal{L}$ generated by CleanDynamicSearch. For Delete, it suffices to mark the node deleted by changing its H value from −1 to 0 (Algorithm 10).

4.3 Concurrency in CleanANN

Under on-the-fly consolidation, our concurrency protocol can support the concurrent execution of Search, Insert, and Delete operations.

The data structures that all queries synchronize over are the adjacency lists of the graph, the node status tracking array H, and a set that keeps track of all empty slots available for new graph nodes (we refer to this as the set of replaceable nodes). We use lock-free techniques on H consisting of an atomic variable for each node, and lock-based synchronization for graph adjacency lists.

Delete. Suppose node v previously represented point x. Since Delete only involves changing H (Algorithm 10), Delete(v) only needs to successfully perform one CAS on v. Any concurrent Search query that processes v during the final neighbor selection (Line 5 in Algorithm 9) after Delete(v_x) will not include x in the result.

Search. Following the concurrency techniques in FreshVamana, when the graph is traversed, adjacency lists are protected by read-write locks (shared reader and exclusive writer).

After C consolidations have been performed for a deleted node v_x, a Search calls TryMarkReplaceable on the node (Line 15 in Algorithm 6) when it encounters the node, which inserts v into the set M of replaceable nodes, making v available for representing a new data point. Only one thread will successfully perform the CAS, guaranteeing that v is released into the replaceable nodes set exactly once due to one previous Delete. If another thread concurrently calls TryMarkReplaceable on v and succeeds before the current thread, there are three possible scenarios: (1) the current thread sees that v is already replaceable and does not modify $H(v)$; (2) a new Insert and a sequence of consolidations are performed for v (now representing a new point y), the current thread sees $H(v) < C$ and does not modify $H(v)$, leaving v not replaceable as expected; and (3) a new insert and C consolidations are performed for v (now representing y), the current thread sees $H(v) = C$ and marks v as replaceable, stealing work from the last thread that consolidates v_y. In any case, the status of v is well-defined.

Insert. The concurrency control in Insert queries is mostly the same as for Search, except for two key differences. First, in RobustInsertData (Algorithm 8) where a graph node is chosen to represent the data point, the node must have replaceable status ($H(w) = -2$). Second, when the adjacency lists of the new node and its neighbors are updated (Lines 4–5 of Algorithm 3), the corresponding exclusive locks of the adjacency lists are held.

5 Experiments

We study the performance of CleanANN, which uses all of the methods from Sections 3 and 4, and compare with existing solutions.

5.1 Experiment Setup

Sliding Window Batched Update. Each experiment consists of multiple rounds. In each round, we insert a batch of new data points and delete an equal number of the oldest data points. The index is first constructed by inserting the first half of the entire dataset. The numbers of inserted and deleted points in each subsequent batch are each 1% of the current index size. Real-world datasets are streamed in the order given to reflect realistic distribution shifts where applicable. At the end of each round, we issue all search queries in the dataset, and calculate the recall (Definition 2). Unless otherwise specified, all Insert queries and a randomly selected 5% of Search queries perform GuidedBridgeBuild (denoted $\neg$*performance_sensitive* in Algorithm 6). Since CleanANN amortizes graph repairs over the workload, when CleanANN detects bloat above a threshold, we issue Search queries for 10% of deleted points so that more on-the-fly consolidations are performed. These Search queries are not counted as separate queries and their latencies are accounted for in the throughput calculation of delete batches.

Sliding Window Mixed Update. This setting runs all types of queries concurrently. Recall is not measured in this setting since the ground truth is not well-defined.

Implementations. The implementation[3] of CleanANN is based on the open-source in-memory version of DiskANN (Vamana) [45]. We use `Tokio` for parallelism following the DiskANN Rust codebase. CleanANN- is our system without GuidedBridgeBuild.

We compare with the following baselines:

- IP-DiskANN [50]. IP-DiskANN is implemented by the original authors and uses the same DiskANN codebase that we use.
- Wolverine++ [29]. Since Wolverine++ [29] was originally implemented on `HNSW` [33], we re-implemented the algorithm on the same DiskANN codebase.

On Delete(v_x), both baselines mark v_x as a tombstone before performing immediate graph repairs. Both baselines use global batch operations to clean up residual incoming edges for tombstone nodes. We refer readers to [50] for comparisons with FreshVamana.

Hyperparameter Choices. We now revisit the hyperparameters required by CleanANN and other baselines and discuss how they are set in the experiments. L configures the scope of Search backtracking. L_I configures the scope of Insert backtracking. R and α configure the sparsity of the graph via RobustPrune. In all experiments, we set $R = 64$, $\alpha = 1.2$, and $L_I = 128$. We report results from varying values of L to reflect the capabilities of different systems at different target recall ranges.

$\mathcal{S}$ and HeuristicPredicate configure GuidedBridgeBuild in CleanANN. We set $\mathcal{S} = \{\log_2 |D| - 1, \log_2 |D|, \log_2 |D| + 1\}$, and HeuristicPredicate$(v_1, v_2)$ = `True` if and only if $r(v_1) = r(v_2)$ in the search tree $\mathcal{T}$ of BridgeBuilderSearch (Algorithm 1). C controls how eagerly the semi-lazy cleaning method completely cleans a tombstone. We set $C = 7$ except in the microbenchmark studying the sensitivity of C in Section 5.3.

Datasets. We evaluate on 10-million and 100-million point segments of three datasets representing real-world large-scale embedding-based information retrieval using two different distance functions (ℓ_2 and cosine similarity) and different data distributions, with dimensionalities ranging from 100 to 512. By default, the results presented in the detailed analysis are on the 10-million point segments. The datasets are listed in Table 2. MS-Turing [15] and MS-SpaceV [4] are based on Web-scale general search queries, while RedCaps [8, 41] is based on site-level, more specialized search queries from Reddit. MS-Turing and RedCaps have out-of-distribution search queries. MS-SpaceV and RedCaps have real-world distribution shifts.

[3]Available at `https://github.com/SylviaZiyuZhang/cleanann-rust`

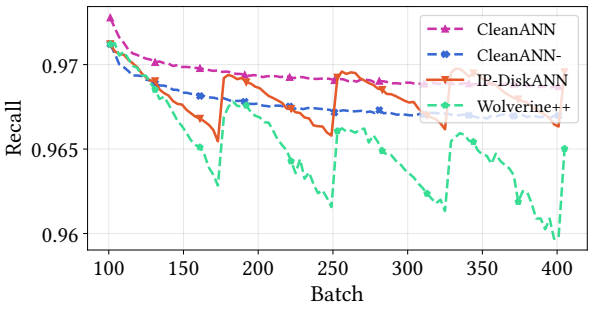


Figure 6: Recall 50@50 over time for MS-SpaceV with $L = 200$.

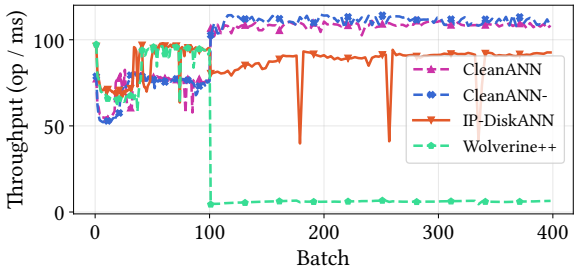


Figure 7: Overall throughput for MS-SpaceV in the Sliding Window Mixed Update setting.

Experiment Platform and Concurrency. All experiments are run on a machine with four Intel Xeon Platinum 8176 CPUs, with a total of 112 cores with 2-way hyper-threading. The machine has a total of 1.5TB DRAM. Unless otherwise specified, all systems use 64 hyper-threads throughout the benchmarks.

5.2 Main Benchmark Results

We report the recall for $k = 10$ and $k = 50$ for all the main benchmark experiments. We summarize the recalls (in the Sliding Window Batched Update setting) and comparisons of the overall throughput results (in the Sliding Window Mixed Update setting) of CleanANN, IP-DiskANN [50], and Wolverine++ [29] in Table 3, running both settings for 100 rounds after the index is constructed via insert-only batches (the first 100 batches). We also include results for CleanANN-, CleanANN without GuidedBridgeBuild.

CleanANN achieves recall competitive with IP-DiskANN [50] and Wolverine++ [29] while delivering higher overall throughput than both. For datasets with a real-world distribution shift and/or out-of-distribution queries, CleanANN does not exhibit recall degradation.

Search Quality. Figure 6 presents the recalls of our system and baselines on MS-SpaceV in the Sliding Window Batched Update setting, showing that *CleanANN maintains high index quality when undergoing updates.* Compared to CleanANN, IP-DiskANN [50] and Wolverine++ [29] both show recall degradation between global

Table 2: Dataset Information.

Name	#Queries	Dim.	Similarity	Distribution Shift	Domain	Out-of-Distribution Queries
MS-Turing [15]	10,000	128	ℓ_2	No	Web-scale Document Search	Yes
MS-SpaceV [4]	29,316	100	ℓ_2	Yes	Web-scale Document Search	No
RedCaps [41]	800	512	cosine	Yes	Text-to-Multimodal Search	Yes

Table 3: Main Benchmark Result Summary. IP is IP-DiskANN [50] and W++ is Wolverine++ [29]. "× throughput" is the multiplicative speedup of throughput (measured in the Sliding Window Mixed Update setting) of CleanANN compared to IP-DiskANN [50] and Wolverine++ [29]. In this table, all systems use $L = 200$ and $L_I = 128$. Wolverine++ [29] only supports Euclidean distance, which does not apply to RedCaps.

Name	Size	#Queries	CleanANN | IP | W++ recall10@10	CleanANN | IP | W++ recall50@50	IP | W++ × throughput
MS-SpaceV [4]	10M	29,316	98.11% | 98.11% | 97.68%	96.94% | 96.82% | 96.51%	1.24× | 18.14×
MS-SpaceV [4]	100M	29,316	97.49% | 97.59% | 97.14%	96.57% | 96.61% | 96.21%	1.25× | 16.9×
MS-Turing [15]	10M	10,000	96.1% | 96.32% | 95.17%	92.19% | 92.23% | 91.13%	1.21× | 14.76×
MS-Turing [15]	100M	10,000	95.62% | 95.84% | 94.67%	92.48% | 92.67% | 91.52%	1.14× | 11.75×
RedCaps [41]	10M	800	95.25% | 95.09% | N/A	94.79% | 92.20% | N/A	1.26× | N/A

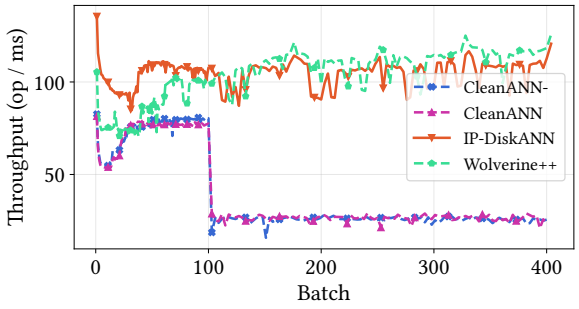


Figure 8: Insert throughput in the Sliding Window Batched Update setting on MS-SpaceV.

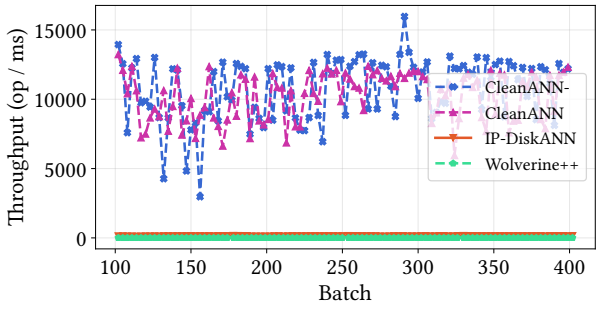


Figure 9: Delete throughput in the Sliding Window Batched Update setting on MS-SpaceV.

consolidation phases, with the degradation in Wolverine++ [29] being worse. This problem worsens as the search k gets closer to L. CleanANN- achieves 0.3% lower recall due to missing the graph structure improvements from GuidedBridgeBuild.

Efficiency. Figure 7 shows the overall throughput of different systems MS-SpaceV with 64 hyper-threads in the Sliding Window Mixed Update setting. The first 100 batches are streaming Inserts and the subsequent batches are fully concurrent mixed updates. Compared to IP-DiskANN, which maintains a similar recall, CleanANN has 1.14–1.26× higher overall throughput and does not experience significant throughput drops due to the batch cleaning. CleanANN has 12–18× higher overall throughput compared to Wolverine++ [29]. After the first 100 insert-only batches, the overall throughput of Wolverine++ drops significantly due to the cost of graph repair candidate evaluation for Deletes. CleanANN- has slightly higher overall throughput than CleanANN due to the absence of the cost of GuidedBridgeBuild.

5.3 Tradeoffs, Scalability, and Ablation Studies

Tradeoffs between Search/Update Throughput and Search Quality. Figures 8–10 show the tradeoffs between Search, Insert, and Delete throughputs among different systems on MS-SpaceV measured in the Sliding Window Batched Update setting. CleanANN has lower batched Insert and Search throughputs because graph repair operations are amortized onto the Insert and

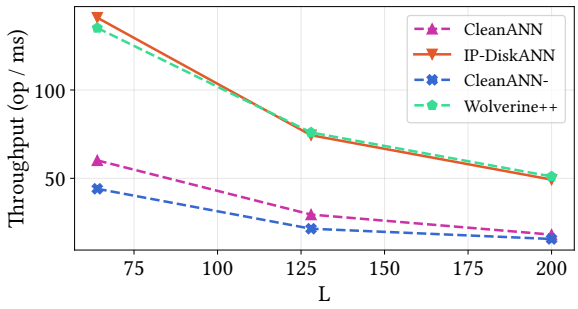


Figure 10: Search Throughput vs. different values of L in the Sliding Window Batched Update setting on MS-SpaceV.

Search batches. Correspondingly, the Delete batch throughput is much higher since Algorithm 10 does not perform graph updates (Figure 9). We present recalls at different values of L in Figure 11.

Parallel Scalability. We study the scalability of CleanANN in the Sliding Window Batched Update setting for 100 rounds on MS-SpaceV with an initial index size of 5 million using 1–224 hyper-threads. These results are shown in Figure 12. Compared to the single-threaded performance, the Search/Insert throughputs are 1.95×/1.95× at 2 threads, 6.32×/7.22× at 8 threads, 10.58×/14.03× at 16 threads, 14.64×/36.76× at 56 threads, 16.27×/53.30× at 112

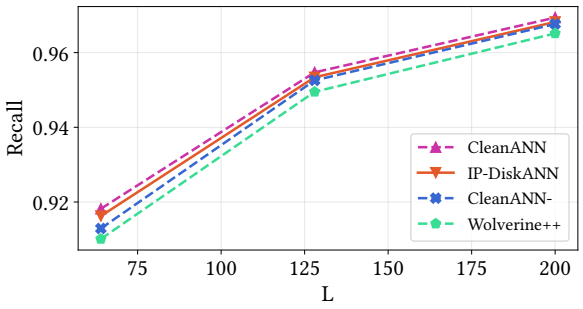


Figure 11: Recall 50@50 vs. different values of L in the Sliding Window Batched Update setting on MS-SpaceV.

threads, and 20.36×/62.98× at 224 threads. These findings are consistent with previous results [35] on Vamana. CleanANN maintains at least 90% recall throughout this experiment.

Ablation Study for GuidedBridgeBuild. By comparing CleanANN and CleanANN- in Figure 6 and Figure 7, we can see that GuidedBridgeBuild delivers a consistent recall improvement (0.3%) at the cost of 3% lower overall throughput. We note that in previous methods, achieving a small recall improvement in the high (95%+) recall region often requires significantly increasing the work required [1]. Furthermore, GuidedBridgeBuild *improves the convergence of greedy searches and makes Search queries more efficient.* With the same backtracking budget L, CleanANN converges faster (Figure 10) and to a higher recall (Figure 11), achieving a superior throughput-recall tradeoff.

Lastly, we compare the edges that GuidedBridgeBuild adds to the graph index with running index construction and Insert under a higher out-degree bound R to demonstrate that GuidedBridgeBuild improves the navigability of the graph even while controlling for the number of edges. We additionally run the experiment presented in Table 3 for MS-SpaceV on CleanANN with $R = 64$ and CleanANN- with $R = 74$. These two configurations achieve the same recall, with the former resulting in an average degree of 39 and the latter 44, suggesting that the benefits of GuidedBridgeBuild do not merely come from making the graph denser. These results suggest that GuidedBridgeBuild fundamentally improves the graph quality beyond RobustPrune.

Choice of GuidedBridgeBuild Heuristics. The HeuristicPredicate in the main benchmark constrains the depths of nodes to be the same. We considered several other plausible heuristics:

(1) Random Sampling: Given $\mathcal{S}$, for $v, w \in \{u \in \mathcal{T} \mid r(u) \in \mathcal{S}\}$, HeuristicPredicate$(v, w)$ = **true** with probability p.
(2) Distance of Lowest Common Ancestor (LCA): Given $\mathcal{S}$ and a threshold $l \in \mathbb{N}^+$, for $v, w \in \{u \in \mathcal{T} \mid r(u) \in \mathcal{S}\}$, let $u^\star$ be the LCA of v and w. HeuristicPredicate(v, w) = **true** if and only if $r(v) - r(u^\star) > l$ and $r(w) - r(u^\star) > l$.
(3) Connecting ancestors-descendants: Given $\mathcal{S}$ and a threshold $l \in \mathbb{N}^+$, for $v, w \in \{u \in \mathcal{T} \mid r(u) \in \mathcal{S}\}$, HeuristicPredicate$(v, w)$ = **true** if and only if $|r(v) - r(w)| > l$ and v is an ancestor or descendant of w.
(4) Bounded depth-difference constraint: Given $\mathcal{S}$, for $v, w \in \{u \in \mathcal{T} \mid r(u) \in \mathcal{S}\}$, HeuristicPredicate$(v, w)$ = **true** if and only if $|r(v) - r(w)| < \delta$.

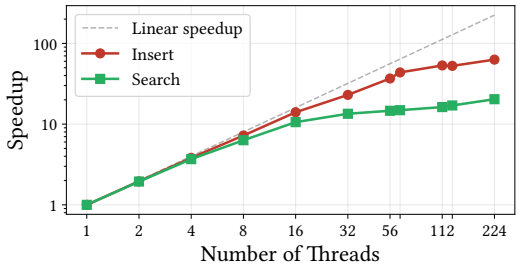


Figure 12: Parallel scaling of CleanANN on MS-SpaceV in the Sliding Window Batched Update setting.

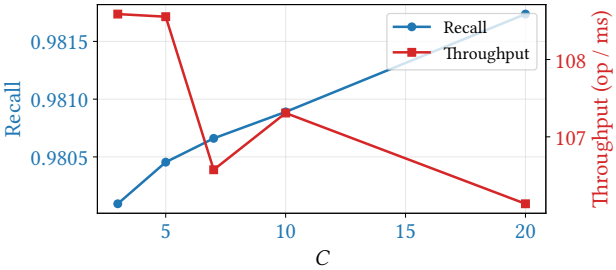


Figure 13: Average throughput and recall of CleanANN on MS-SpaceV vs. C.

We present results for these heuristics in Table 4. We find that the exact choice of heuristic does not generally produce a qualitative difference. This is likely because as long as candidate edges close the same gap among the same well-connected regions, the exact edge to add is not crucial for performance and accuracy. We suggest using the equal depth constraint because it has the most consistent performance and the smallest overhead across datasets. Furthermore, we found that the choice of $\mathcal{S}$ is much more important than the choice of the heuristic.

Semi-lazy Memory Cleaning Hyperparameter Sensitivity. We present a microbenchmark that studies the sensitivity of the cleaning threshold hyperparameter C on MS-SpaceV (Figure 13). The recall increases slightly and exhibits diminishing returns as C increases. A higher value of C means that more incoming edges of a tombstone need to be consolidated, causing lower throughput.

Effect of Random Edges. We study whether the technique of overwriting graph nodes with dangling incoming edges significantly affects the graph quality. We compare the recall and throughput under the Sliding Window Batched Update setting between the case where newly inserted data points reuse old graph nodes and the case where they do not, holding all other hyperparameters equal. The differences in these metrics are negligible. Additionally, we implemented a simple spatially aware strategy. When a point x is inserted, the index arbitrarily picks two replaceable nodes v_{y_1} and v_{y_2}, and chooses $\text{argmin}_{v_{y_j}} d(x, y_j)$ to represent x. We find that the recall and efficiency yielded by this strategy are not significantly different from using an arbitrary slot. Overall, our experiments indicate that random edges do not significantly impact the index performance.

Table 4: Recall improvement and Insert/Search efficiency compared to CleanANN- for different GuidedBridgeBuild heuristics. $\mathcal{S}$ is varied and the hyperparameter of each heuristic is fixed.

Heuristic	Hyperparameter	+ Recall at $L = 128$	× Insert Throughput	× Search Throughput
Random Sampling	$p = 0.05$	0.13–0.7%	0.73–0.88×	0.84–1.14×
Distance of LCA	$l = 3$	0.09–0.37%	0.95–1.09×	0.87–1.13×
Connecting ancestors-descendants	$l = 3$	0.03–0.04%	0.91–1.17×	0.99–1.06×
Bounded depth-difference constraint	$\delta = 1$	0.15–0.82%	0.75–0.9×	1.05–1.5×
Equal depth constraint (proposed)	—	0.09–0.56%	0.88–1.02×	0.99–1.06×

6 Related Work

Static ANNS indexes have been extensively studied in the literature. Previous indexing methods can be roughly categorized into three main categories of techniques: space-partition trees, locality-sensitive hashing, and graph-based indexes. Some work [2, 54] also combines multiple techniques for improved query performance. Space-partition trees and locality-sensitive hashing are partition-based approaches and inherently struggle with the *boundary issue*: if a query point is close to a boundary of different partitions, indexed points in every partition need to be compared with the query. This is especially an issue in high dimensions since the number of possible spatial partition neighbors grows with the dimensionality.

Space-partition trees. Space-partition indexes can be naturally extended to support ANNS. Given a query q, the index identifies the partition $\mathcal{P}(q)$ that q belongs to and potentially the partitions close to $\mathcal{P}(q)$. The indexed points in these identified partitions are further compared with q to find the nearest neighbors. Space-partition indexes that have been applied to ANNS include kd-trees [14, 20, 40], R-trees [25, 38], random projection (RP)-trees [6], spill (SP)-trees [11, 30, 48], Voronoi diagrams [24, 31], clustering [2], and product quantization [21]. These methods are also often combined with ideas from inverted indexes for improved performance.

Locality-sensitive hashing (LSH). Locality-sensitive hash functions hash points close to each other to hash values that are the same or close to each other with high probability. Indexed points in the same or similar hash buckets as the query point are compared with the query point. Multiple LSH functions are often used together to boost the success probability. LSH was first proposed for ANNS in [10, 16] and extensively studied subsequently [5, 7, 36]. LSH-based indexes have provable guarantees, but do not attain the performance of graph-based indexes in practice [1].

Graph-based indexes. In addition to the previous graph-based indexes that we discussed already [9, 12, 18, 32, 34], the recent work eXNG [52] makes the observation that connecting nodes with their intermediate near neighbors improves the graph. This observation is similar to the intuition for GuidedBridgeBuild. However, eXNG still focuses on the static setting. We refer readers to [49] for a more comprehensive review of graph-based indexes.

Dynamic indexes. Fully-dynamic ANNS indexes that maintain high recall under updates are less studied. Using different techniques for static ANNS mentioned above in the fully-dynamic setting introduces different challenges from the ones addressed in this paper for graph-based ANNS indexes. Naively applying static space-partition and LSH-based solutions in the dynamic setting by inserting points into or deleting points from the partition or hash bucket creates partition/bucket imbalance problems. This can degrade the efficiency of the index and may exacerbate the boundary issue discussed above, which worsens the recall. SPFresh [51] uses dynamic rebalancing (based on space partitioning) to solve this problem to obtain a dynamic version of the disk-oriented clustering-based index SPANN [2]. Dynamic rebalancing techniques are orthogonal to our methods, since we focus on graph-based techniques. A system such as SPFresh, which can combine a graph-based index with other algorithms, could benefit from our methods. The existing dynamic graph-based indexes FreshDiskANN [46], DEG [13], and LSH-APG [54], as well as graph-based entries in the NeurIPS 2023 streaming ANN competition [44], either use expensive global updates or suffer from query quality degradation.

Recent systems DIGRA [19] and RangePQ [53] propose algorithms for dynamic ANNS supporting range queries, leveraging tree structures. Their update algorithms focus on maintaining the tree structures themselves. Our work focuses instead on an efficient and robust solution for updating the graphs in graph-based ANNS.

The concurrent and independent work IP-DiskANN [50] focuses on efficiently supporting Delete queries in place. When a data point p is deleted, IP-DiskANN searches for p and heuristically selects in-neighbors of v_p visited during the search to absorb some other nodes close to p into their neighborhoods. On the other hand, CleanANN performs consolidation in subsequent operations and is informed by the query workload, while also providing methods to improve robustness.

7 Conclusion

We presented adaptive, robust, and efficient methods to improve the performance of graph-based ANNS indexes under full dynamism. We designed the CleanANN system, which implements our methods on top of Vamana. Our system avoids expensive global updates and adaptively improves the index. Future research opportunities include combining our methods with other graph-based ANNS approaches, adapting machine learning techniques from learned indexes to improve performance, and theoretically analyzing the performance and robustness of dynamic ANNS.

Acknowledgments

This work is funded by NSF awards #CCF-1845763, #CCF-2316235, and #CCF-2403237, a Google Faculty Research Award, a Google Research Scholar Award, MIT Jacobs Presidential Fellowship, NSF GRFP #DGE-2141064, and the MIT-Google Program for Computing Innovation.

References

[1] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2018. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. arXiv:1807.05614 [cs.IR]

[2] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search. *CoRR* abs/2111.08566 (2021). arXiv:2111.08566 https://arxiv.org/abs/2111.08566

[3] Yewang Chen, Shengyu Tang, Nizar Bouguila, Cheng Wang, Jixiang Du, and HaiLin Li. 2018. A fast clustering algorithm based on pruning unnecessary distance computations in DBSCAN for high-dimensional data. *Pattern Recognition* 83 (2018), 375–387.

[4] SpaceV Contributors. 2023. SPTAG/datasets/SPACEV1B at main · microsoft/SPTAG — github.com. https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B. [Accessed 30-07-2024].

[5] Anirban Dasgupta, Ravi Kumar, and Tamas Sarlos. 2011. Fast locality-sensitive hashing. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1073–1081. doi:10.1145/2020408.2020578

[6] Sanjoy Dasgupta and Kaushik Sinha. 2015. Randomized Partition Trees for Nearest Neighbor Search. *Algorithmica* 72, 1 (01 May 2015), 237–263. doi:10.1007/s00453-014-9885-5

[7] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*. 253–262. doi:10.1145/997817.997857

[8] Karan Desai, Gaurav Kaul, Zubin Aysola, and Justin Johnson. 2021. RedCaps: web-curated image-text data created by the people, for the people. arXiv:2111.11431 [cs.CV] https://arxiv.org/abs/2111.11431

[9] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.* 12, 5 (jan 2019), 461–474. doi:10.14778/3303753.3303754

[10] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. 518–529.

[11] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *International Conference on Machine Learning*. https://arxiv.org/abs/1908.10396

[12] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 5713–5722. doi:10.1109/CVPR.2016.616

[13] Nico Hezel, Kai Uwe Barthel, Bruno Schilling, Konstantin Schall, and Klaus Jung. 2025. Dynamic Exploration Graph: A Novel Approach for Efficient Nearest Neighbor Search in Evolving Multimedia Datasets. In *31st International Conference on Multimedia Modeling*. 333–347. doi:10.1007/978-981-96-2054-8_25

[14] Linjia Hu, Saeid Nooshabadi, and Majid Ahmadi. 2015. Massively parallel KD-tree construction and nearest neighbor search algorithms. In *IEEE International Symposium on Circuits and Systems (ISCAS)*. 2752–2755. doi:10.1109/ISCAS.2015.7169256

[15] Microsoft Inc. 2021. Billion-scale approximate nearest neighbor Search Challenge: NeurIPS'21 competition track. https://big-ann-benchmarks.com/neurips21.html#:~:text=Microsoft%20Turing%2DANNS%2D1B%20is,19%20paper%20and%20a%20blogpost.

[16] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. 604–613. doi:10.1145/276698.276876

[17] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. arXiv:1810.07355 [cs.DB]

[18] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems*, Vol. 32.

[19] Mengxu Jiang, Zhi Yang, Fangyuan Zhang, Guanhao Hou, Jieming Shi, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter. *Proc. ACM Manag. Data* 3, 3, Article 148 (June 2025), 26 pages. doi:10.1145/3725399

[20] Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete. 2017. A progressive k-d tree for approximate k-nearest neighbors. In *IEEE Workshop on Data Systems for Interactive Analysis (DSIA)*. 1–5. doi:10.1109/DSIA.2017.8339084

[21] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128. doi:10.1109/TPAMI.2010.57

[22] Andrew Kane. [n. d.]. Iterative index scans · Issue 678 · pgvector/pgvector — github.com. https://github.com/pgvector/pgvector/issues/678. [Accessed 29-09-2025].

[23] Andrew Kane, Heikki Linnakangas, Jonathan S. Katz, Jon Daniel, Fabian Fischer, Florents Tselai, Japin Li, Julien Rouhaud, Luca Giacchino, Narek Galstyan, Nathan Bossart, Pavel Borisov, Rui Chen, Samuel Marks, Will Laurance, jeffdavis, mulander, oneturkmen, and yihong. 2025. pgvector/pgvector. https://github.com/pgvector/pgvector

[24] Mohammad Kolahdouzan and Cyrus Shahabi. 2004. Voronoi-based K nearest neighbor search for spatial network databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB Endowment, 840–851.

[25] J. Kuan and P. Lewis. 1997. Fast k nearest neighbour search for R-tree family. In *International Conference on Information, Communications and Signal Processing*, Vol. 2. 924–928 vol.2. doi:10.1109/ICICS.1997.652114

[26] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*. Article 793, 16 pages.

[27] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. 2018. The design and implementation of a real time visual search system on JD E-commerce platform. In *Proceedings of the 19th International Middleware Conference Industry*. 9–16.

[28] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. 2021. Embedding-based product retrieval in taobao search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 3181–3189.

[29] Dawei Liu, Bolong Zheng, Ziyang Yue, Fuhao Ruan, Xiaofang Zhou, and Christian S. Jensen. 2025. Wolverine: Highly Efficient Monotonic Search Path Repair for Graph-Based ANN Index Updates. *Proc. VLDB Endow.* 18, 7 (March 2025), 2268–2280. doi:10.14778/3734839.3734860

[30] Ting Liu, Andrew Moore, Ke Yang, and Alexander Gray. 2004. An Investigation of Practical Approximate Nearest Neighbor Algorithms. In *Advances in Neural Information Processing Systems*, Vol. 17.

[31] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2021. HVS: hierarchical graph structure based on voronoi diagrams for solving approximate nearest neighbor search. *Proc. VLDB Endow.* 15, 2 (oct 2021), 246–258. doi:10.14778/3489496.3489506

[32] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate Nearest Neighbor Algorithm based on Navigable Small World Graphs. *Information Systems* 45 (2014), 61–68. doi:10.1016/j.is.2013.10.006

[33] Yury Malkov, Dmitry Yashunin, Dmitriy Bespalov, Marek Hanuš, Kishore Nallan, orrorcol, James Melville, Greg Roodt, Martin Dimitrov, Paul Brossier, Louis Abraham, alonre24, Taras Tsugrii, TakaakiFuruse, Aaron Lun, Atsushi Tatsuma, Alexander Vieth, Bo Li, Peter Sobot, He Yidong, Étienne Mollier, aurora, Cheng Yang, drons, dorosy-yeong, Vimal Mathew, Jisang Yoon, Dmitry, stephematician, and moritz-h. 2025. nmslib/hnswlib. https://github.com/nmslib/hnswlib

[34] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. doi:10.1109/TPAMI.2018.2889473

[35] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 270–285. doi:10.1145/3627535.3638475

[36] Rajeev Motwani, Assaf Naor, and Rina Panigrahi. 2006. Lower bounds on locality sensitive hashing. In *Proceedings of the Twenty-Second Annual Symposium on Computational Geometry*. 154–157. doi:10.1145/1137856.1137881

[37] Felix Ocker, Daniel Tanneberg, Julian Eggert, and Michael Gienger. 2024. Tulip Agent–Enabling LLM-Based Agents to Solve Tasks Using Large Tool Libraries. *arXiv preprint arXiv:2407.21778* (2024).

[38] Apostolos Papadopoulos and Yannis Manolopoulos. 1997. Performance of nearest neighbor queries in R-trees. In *International Conference on Database Theory*. 394–408.

[39] Erion Plaku and Lydia E Kavraki. 2008. Quantitative analysis of nearest-neighbors search in high-dimensional sampling-based motion planning. In *Algorithmic Foundation of Robotics VII: Selected Contributions of the Seventh International Workshop on the Algorithmic Foundations of Robotics*. 3–18.

[40] Parikshit Ram and Kaushik Sinha. 2019. Revisiting kd-tree for Nearest Neighbor Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1378–1388. doi:10.1145/3292500.3330875

[41] Reddit, Karan Desai, Gaurav Kaul, Zubin Aysola, Justin Johnson, Joshua Engels, Ziyu Zhang, and OpenAI (United States). 2024. CLIP-Embedded RedCaps Text-Image Dataset. https://doi.org/10.5281/zenodo.13137120

[42] Aviad Rubinstein. 2018. Hardness of approximate nearest neighbor search. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. 1260–1268. doi:10.1145/3188745.3188916

[43] Maying Shen, Xinghao Jiang, and Tanfeng Sun. 2018. Anomaly detection based on nearest neighbor search with locality-sensitive B-tree. *Neurocomputing* 289 (2018), 55–67.

[44] Harsha Vardhan Simhadri, Martin Aumüller, Amir Ingber, Matthijs Douze, George Williams, Magdalen Dobson Manohar, Dmitry Baranchuk, Edo Liberty, Frank Liu, Ben Landrum, Mazin Karjikar, Laxman Dhulipala, Meng Chen, Yue Chen, Rui Ma,

Kai Zhang, Yuzheng Cai, Jiayang Shi, Yizhuo Chen, Weiguo Zheng, Zihao Wan, Jie Yin, and Ben Huang. 2024. Results of the Big ANN: NeurIPS'23 competition. arXiv:2409.17424 [cs.IR] https://arxiv.org/abs/2409.17424

[45] Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, Andrija Antonijevic, Dax Pryce, David Kaczynski, Shane Williams, Siddarth Gollapudi, Varun Sivashankar, Neel Karia, Aditi Singh, Shikhar Jaiswal, Neelam Mahapatro, Philip Adams, Bryan Tower, and Yash Patel. 2023. DiskANN: Graph-structured Indices for Scalable, Fast, Fresh and Filtered Approximate Nearest Neighbor Search. https://github.com/Microsoft/DiskANN

[46] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. arXiv:2105.09613 [cs.IR]

[47] Ján Suchal and Pavol Návrat. 2010. Full text search engine as scalable k-nearest neighbor recommendation system. In *International Conference on Artificial Intelligence in Theory and Practice*. 165–173.

[48] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. 2023. SOAR: Improved Indexing for Approximate Nearest Neighbor Search. In *Neural Information Processing Systems*.

[49] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.* 14, 11 (jul 2021), 1964–1978. doi:10.14778/3476249.3476255

[50] Haike Xu, Magdalen Dobson Manohar, Philip A. Bernstein, Badrish Chandramouli, Richard Wen, and Harsha Vardhan Simhadri. 2025. In-Place Updates of a Graph Index for Streaming Approximate Nearest Neighbor Search. arXiv:2502.13826 [cs.IR]

[51] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 545–561. doi:10.1145/3600006.3613166

[52] Cheng Zhang, Jianzhi Wang, Wan-Lei Zhao, and Shihai Xiao. 2025. Highly Efficient Disk-based Nearest Neighbor Search on Extended Neighborhood Graph. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2513–2523. doi:10.1145/3726302.3729996

[53] Fangyuan Zhang, Mengxu Jiang, Guanhao Hou, Jieming Shi, Hua Fan, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 3, Article 152 (June 2025), 26 pages. doi:10.1145/3725401

[54] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards Efficient Index Construction and Approximate Nearest Neighbor Search in High-Dimensional Spaces. *Proc. VLDB Endow.* 16, 8 (apr 2023), 1979–1991. doi:10.14778/3594512.3594527