HNSW-Slim: A Light-Weight Vector Index for Approximate Nearest Neighbor Search

[†]Haobo Sun, [†]Jialu Gao, [†]Lu Chen, [†]Xiangyu Ke, [‡]Tianyi Li, [§]Jingwen Zhao
[†]Zhejiang University [‡]Aalborg University [§] Poisson Lab. of Huawei
{haobosun, jialugao, luchen, xiangyu.ke}@zju.edu.cn tianyi@cs.aau.dk zhaojingwen5@huawei.com

ABSTRACT

Hierarchical Navigable Small World (HNSW) is a leading index for approximate nearest neighbor search, prized for its high efficiency and accuracy in real-life applications. However, HNSW's multi-layer graph index incurs a substantial memory overhead—often matching or exceeding the size of the raw vector data—making it impractical for resource-limited deployments on terminals or mobile devices. Prior work on memory reduction mainly focused on vector data compression (e.g., quantization), which is inherently lossy and degrades search accuracy, leaving substantial index-structure overhead unaddressed. Moreover, existing graph-pruning techniques typically assume flat, static topologies, rendering them poorly suited for HNSW's hierarchical, dynamically evolving structure.

To close this gap, we propose HNSW-Slim, a light-weight, high-performance, and dynamically updatable HNSW variant. HNSW-Slim employs a parallelizable two-stage pruning pipeline: (1) Hierarchical Pruning removes redundant cross-layer links to collapse superfluous navigation shortcuts between levels; (2) Small-World Pruning selectively sparsifies intra-layer connections to preserve navigability while minimizing edge count. These steps are supported by a compact hierarchical adjacency list (CHAL) storage format that achieves high space utilization while enabling node-level updatability. Finally, we design a lightweight incremental-update protocol – optimized for client-server architectures – that transmits only a compact delta of affected nodes to minimize network overhead. Extensive experiments demonstrate that HNSW-Slim reduces HNSW's memory usage by up to 4.7×, sustains 2.6×-77.8× improvements of in search efficiency with similar accuracy compared with remote HNSW, and processes second-level updates. It is worth noting that HNSW-Slim and vector quantization are orthogonal and complementary, with vector quantization, the total memory footprint can be reduced by up to 6.3× to HNSW while maintaining 98% recall.

PVLDB Reference Format:
[†]Haobo Sun, [†]Jialu Gao, [†]Lu Chen, [†]Xiangyu Ke, [‡]Tianyi Li, [§]Jingwen Zhao. HNSW-Slim: A Light-Weight Vector Index for Approximate Nearest Neighbor Search. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

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

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Approximate Nearest Neighbor Search (ANNS) seeks, for a given query vector in a high-dimensional embedding space, to rapidly retrieve the most similar data points with high accuracy. It has become a core primitive in a wide range of systems: recommendation engines use ANNS to surface items that match user preferences [3, 5], document-retrieval pipelines employ ANNS over text embeddings to find relevant passages [27, 29], modern vector databases rely on ANNS for low-latency indexing and querying at scale [32, 41], and retrieval-augmented generation systems fetch support contexts via ANNS to ground large language models [28, 36]. As models and datasets continue to grow – often involving hundreds of millions or even billions of high-dimensional vectors – the ability to achieve millisecond-level latency without sacrificing recall has become critical for both academic research and industrial deployments.

Existing ANNS methods can be categorized into four types [42]: tree-based, hash-based, quantization-based, and graph-based. The first three categories succumb to the "curse of dimensionality", which drastically reduces search recall and throughput as vector dimensionality grows. In contrast, graph-based approaches construct a sparse proximity graph – treating data points as nodes and their similarities as edges – thereby reducing ANNS to a fast graph traversal problem. Among these, the Hierarchical Navigable Small World (HNSW) algorithm [31] has emerged as the de facto standard. HNSW organizes nodes into multiple layers for coarse-to-fine search and leverages the small-world property to ensure any neighbor can be reached in a few hops, delivering state-of-the-art recall–latency trade-offs. Its performance has driven adoption in major vector databases [15, 33, 35, 37, 38, 40, 45].

Despite its high search performance, HNSW suffers from **substantial memory consumption** [9]. We identify three primary contributors of HNSW: First, to support rapid index construction and dynamic updates, HNSW allocates a *constant-capacity array* for each node's neighbor list in every layer. Because actual node degrees vary widely, many buffer slots remain unused, wasting memory, especially for low-degree nodes. Second, the small-world graph relies on a handful of hub nodes with very high degree [44]. To store these hubs' large neighbor sets, HNSW uses a large buffer size for *all* nodes in the same layer, further inflating the index. Third, maintaining neighbor lists across multiple levels enables efficient "highway" routing but *duplicates storage* for the same nodes on different layers, compounding the index footprint [9].

The large memory footprint of HNSW *confines its use to server-side deployments*. As illustrated in the top half of Figure 1, edge

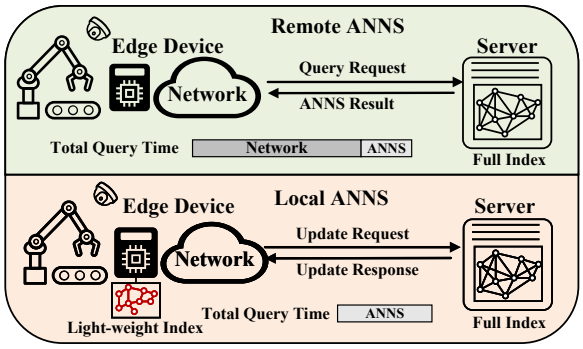


Figure 1: Two types of deployment in an industrial scenario.

devices must forward queries over the network to a server hosting a full set of indices, then await the results. On two real-world datasets [6, 19], network round-trip time accounts for 90.7% and 72.8% of end-to-end query latency, respectively. Such offloading introduces **unacceptable delays for latency-sensitive applications**. For example, real-time defect detection on industrial production lines, where cameras at the edge must quickly identify faulty parts to enable real-time action and cannot rely on remote search.

To support low-latency, on-device search, one must deploy a lightweight index locally while retaining a full-feature HNSW index on the server (Figure 1, bottom). Prior approaches for reducing ANNS memory fall into two broad categories: (1) *Vector Quantization* [1, 11, 14, 23, 42] compresses the embeddings themselves. Although it is effective for data reduction, quantization is inherently lossy and degrades search accuracy. Moreover, it leaves the HNSW graph structure intact – often yielding an index size comparable to or larger than the raw data. (2) *Pruning-based Graph Compression* [8, 9, 44] reduce index size through a static two-stage pipeline: constructing a *flat* proximity graph such as the k-nearest neighbor graph (kNNG), followed by pruning. However, this method still suffers from two limitations:

Reduced search efficiency. Pruning-based graph indexes are typically built upon simple proximity graphs (e.g., kNNG). Without HNSW's hierarchical layout and multi-layer routing, pruned flat graphs rely on greedy, local hops that increase average path length and query time. However, simple pruning in the hierarchical structure will disrupt the high-level connections, leading to the degradation of search performance, which requires a profound understanding of cross-level connectivity.

Poor dynamic adaptability. Their static build-and-prune pipelines incur high rebuild costs on updates, preventing efficient incremental maintenance on resource-constrained devices—an unacceptable drawback for time-sensitive edge applications such as security monitoring [16, 46] or live industrial inspection [26, 47]. The index must be rebuilt on a server and downloaded to the device, introducing significant latency and delayed updates. Although some pruning-based indexes [44] adopt compact formats such as Compressed Sparse Row (CSR) [25] to reduce memory footprint, these formats are optimized for static access patterns and lack support for efficient incremental updates.

These limitations highlight the urgent need for ANNS indexes that reconcile these competing demands: **achieving a lightweight memory usage without sacrificing the search efficiency and dynamic update capability**. We propose HNSW-Slim—a ***HNSW Space*** reduction and ***L***ightweight ***I***ndexing and ***M***aintenance approach. HNSW-Slim is a novel pruning-based ANNS index that achieves high search accuracy while supporting linear-time updates in dynamic scenarios. The core innovation is a synergistic two-stage pruning strategy **(1)** *Hierarchical Pruning* (§3): By analyzing cross-layer node reachability, we eliminate redundant connections while preserving searchability. This layer-aware strategy operates independently per node, enabling *full parallelism*. **(2)** *Small-World Pruning* (§4): Leveraging the small-world property [31], we compress intra-layer graphs using 1-hop neighbors as candidates – avoiding costly graph traversals. We further incorporate *compact degree statistics*, *lightweight connectivity enhancement*, and *parallel optimization*, enabling pruning on million-scale datasets in seconds – orders of magnitude faster than existing methods.

While pruning marginally affects search accuracy, the resulting index is compact enough for local deployment[1] and on-device query processing, hence eliminating network latency rooted in accessing the remote full HNSW index. Furthermore, HNSW-Slim is designed for dynamic environments, featuring an update-friendly storage format and an incremental update protocol (§5), which enables near real-time index maintenance (update batch of 1,000 vectors in 2 seconds, see Table 4). This is achieved with minimal computational overhead and network cost, as only a small delta is transmitted from the server to the client for each update.

We theoretically establish three key properties (§6): **(1)** HNSW-Slim reduces space complexity versus HNSW while retaining *logarithmic* search time; **(2)** We derive a theoretical minimum space cost of HNSW-Slim, under given user-controlled parameters, enabling principled hyperparameter tuning. **(3)** Both pruning stages run in linear time relative to dataset size, ensuring scalability.

Overall, we make the following main contributions:

- We propose HNSW-Slim, a two-stage pruning-based ANNS index that combines hierarchical and small-world pruning to remove redundant edges. It significantly reduces index size while maintaining high search performance.
- We design a compact and update-friendly storage format with an incremental update protocol, enabling efficient index maintenance and minimal network overhead in dynamic scenarios
- We provide a comprehensive theoretical analysis. Specifically, we prove that HNSW-Slim significantly reduces memory consumption compared to HNSW, achieves logarithmic search complexity and linear-time pruning, with a user-controlled trade-off between memory usage and search performance.
- We conduct extensive experiments in emulated terminal and server environments using four large real-world datasets. The results demonstrate that, compared to state-of-the-art methods, HNSW-Slim substantially reduces memory usage by up to 3.0×, enabling real-time index updates, and improves search efficiency by up to 6.7× with similar accuracy. Additionaly, when integrated with vector quantization, the total memory footprint can be reduced by up to 6.3× compared to the original HNSW, while maintaining 98% recall.

[1]Edge devices often operate under strict memory limits [47]. As shown in our experiments (Table 2), HNSW-Slim produces indexes well within this range (e.g., 43.7MB for 1M dataset), making it suitable for on-device deployment.

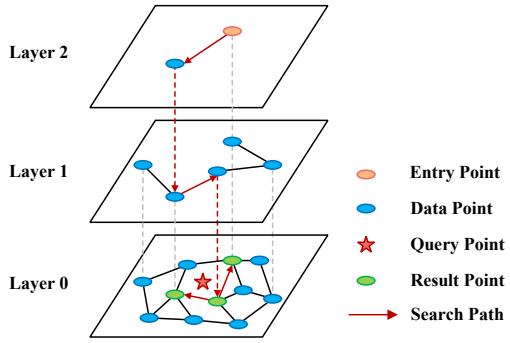


Figure 2: An illustration of the HNSW.

2 PRELIMINARIES

2.1 Approximate Nearest Neighbor Search

Nearest Neighbor Search (NNS) is a fundamental task, aiming to find the most similar data to a given query. Formally, given a query vector q and a dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_n\}$, it retrieves the k vectors in $\mathcal{D}$ that are closest to q under a certain distance metric, such as Euclidean distance (L_2) or cosine similarity.

Exact Nearest Neighbor Search (ENNS) requires returning the exact nearest neighbors without approximation. Formally, for a query q, ENNS solves:

$$\mathcal{R}_k^*(q) = \{x \in \mathcal{D} \mid \text{rank}(\text{dist}(q, x)) \leq k\} \tag{1}$$

where $\text{dist}(\cdot, \cdot)$ is the distance metric and $\text{rank}(\cdot)$ returns the position when sorted by distance. However, ENNS becomes computationally prohibitive as dataset size or data dimensionality grows, with brute-force methods requiring $O(dn)$ time per query (d denotes the data dimension, n denotes the dataset size).

To balance accuracy and efficiency, **Approximate Nearest Neighbor Search (ANNS)** has been introduced. ANNS allows returning neighbors that are close to the exact nearest neighbors within a bounded error, significantly improving search speed. Formally, given a query q and an approximation factor $c > 1$, ANNS returns:

$$\mathcal{R}_k(q) = \{x \in \mathcal{D} \mid \text{dist}(q, x)) \leq c \cdot \text{dist}(q, x^*))\} \tag{2}$$

where x^* is the exact nearest neighbor of q and $|\mathcal{R}_k(q)| = k$.

2.2 Hierarchical Navigable Small World Index

Hierarchical Navigable Small World (HNSW) [31] extends the Navigable Small World (NSW) [30] by introducing a multi-layer hierarchical structure, inspired by skip lists, as illustrated in Figure 2. This design enables exceptionally fast and accurate searches. HNSW incrementally constructs a multi-layered graph, where each layer is a proximity graph. The upper layers are sparse and act as a long-range highway for routing, while the lower, denser layers are used for fine-grained search. This allows HNSW to achieve logarithmic complexity for search operations on average.

Graph Structure. The HNSW graph consists of multiple layers, from layer 0 to L_{max}. Layer 0 is the base layer and contains all the data vectors. For any layer $l > 0$, the set of nodes in layer l is a subset of the nodes in layer $l - 1$. Each node is randomly assigned a maximum layer l_{max} up to which it appears, following an exponentially decaying probability distribution similar to a skip list, which ensures that the upper layers are sparse. Within each layer, the graph is constructed as an NSW graph, where nodes are connected to their neighbors to facilitate efficient greedy traversal.

Search. The search process mirrors the insertion logic. It starts from the entry point in the top layer and greedily traverses the graph to find the node closest to the query vector. This node then becomes the entry point for the layer below. This hierarchical routing continues until layer 0 is reached. In the base layer, a more exhaustive search (i.e., a beam search controlled by the *efSearch* parameter) is performed, maintaining a dynamic list of the best candidates found so far. This allows HNSW to explore the neighborhood around the query vector thoroughly, leading to high recall.

The remarkable performance of HNSW stems from its ability to balance long-range and short-range navigation. The hierarchical structure prevents the search from getting stuck in local minima, while the dense connections in the base layer ensure high accuracy. However, it needs high memory usage due to the storage of multi-layer connections and the pre-allocation of memory for neighbor lists, while we aim to develop a lightweight index structure.

2.3 Pruning-based Graph Index

To mitigate the high memory usage associated with dense proximity graphs, various pruning strategies have been developed for graph-based indexes. These methods typically begin with an over-connected graph, such as a k-Nearest Neighbor Graph (kNNG), and then strategically remove redundant edges to create a sparser index.

Notable examples include NSG [9], which prunes redundant edges by checking if a direct connection can be more efficiently traversed through a shared neighbor. NSSG [8] considers the angular separation of neighbors, pruning edges to promote a well-distributed, satellite-like neighborhood around each node. LEANN [44] allows a small number of critical hub nodes to retain many connections while heavily pruning the edges of less important nodes. However, the above methods need a high computational cost to identify the candidate neighbors of each node to prune. For each node, they often need to perform a computationally expensive graph traversal to gather a large set of candidate neighbors before applying pruning strategies. To alleviate this, NSSG restricts the candidate to the 2-hop neighborhood of a node, which significantly reduces the search space.

While effective at reducing the index size, the above methods adopt a two-stage build-and-prune strategy. The initial graph construction, often using iterative algorithms like NN-Descent to build a kNNG, which is computationally intensive and not designed for incremental updates. The subsequent pruning stage is also performed offline on the entire graph. This entire process is designed for static datasets, and an update typically requires a full reconstruction, making them ill-suited for dynamic environments where real-time updates are critical.

3 HIERARCHICAL PRUNING

In this section, we first analyze redundant edges across layers and propose a hierarchical pruning strategy to eliminate the redundancy with the corresponding modified search algorithm for the pruned graph. We also present a trade-off mechanism to balance memory savings and search performance.

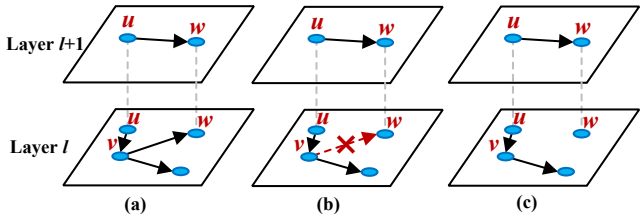


Figure 3: Original HNSW (arrows denote directed edges); (b) hierarchical redundancy (red arrows refer to pruned directed edges); (c) HNSW after eliminating hierarchical redundancy.

Algorithm 1: Hierarchical Pruning

Input: HNSW graph G
Output: Hierarchically pruned graph G_h

1 **foreach** $u \in G$ **do**
2 $u' \leftarrow u$ // Create a copy of the node
3 **for** $l = 0$ **to** $u.l_{max}$ **do**
4 $U \leftarrow \emptyset$
5 **foreach** $v \in u.neighbors[l]$ **do**
6 **if** $v.l_{max} = l$ **then**
7 $U \leftarrow U \cup \{v\}$
8 $u'.neighbors[l] \leftarrow U$
9 $G_h \leftarrow G_h \cup \{u'\}$

3.1 Hierarchical Redundancy in HNSW

The hierarchical design of HNSW, drawing inspiration from skip lists, guarantees that the node set in layer $l+1$ is a subset of that in layer l. While upper layers accelerate long-range routing, the multi-layer structure creates predictable duplicate connectivity. We provide a reachability-based criterion to identify duplicates and then present the hierarchical redundancy of HNSW.

Proposition 3.1 (Hierarchical Redundancy). *Let edge $(u \rightarrow w) \in E_{l+1}$ and edge $(v \rightarrow w) \in E_l$, then edge $(v \rightarrow w)$ in layer l is redundant.*

The detailed proof can be found in Appendix A. Figure 3 provides a running example of hierarchical redundancy.

3.2 Hierarchical Pruning Strategy

Based on the previous analysis of hierarchical redundancy, we propose a simple yet effective hierarchical pruning strategy. The core idea is to discard neighbors in a lower layer if they are already represented in a higher layer. Since a node existing in layer *l+1* is guaranteed to be accessible from its neighbors in that level, we can safely remove the direct connections to it in layer l without losing graph connectivity throughout the whole hierarchy.

The pseudo-code of the hierarchical pruning algorithm is depicted in Algorithm 1. We visit each node u (line 1) in each layer (line 3) of the HNSW graph G. For each neighbor v of u, we check its maximum layer $v.l_{max}$ that appears (line 5). If $v.l_{max}$ equals to the current layer l, it is retained (lines 6–7); otherwise, the edge u, v in layer l is redundant and removed from u's neighbor list in that layer. Finally, a new pruned HNSW graph is returned (line 12).

This pruning process results in a significantly sparser hierarchical graph. A key advantage of this strategy is its high efficiency and parallelizability. The pruning decision for each node is independent of all other nodes, allowing the entire process to be executed in parallel with no need for synchronization.

Algorithm 2: Search on Hierarchically Pruned HNSW

Input: Hierarchically pruned graph G_h, query q, number of query result k, beam width ef
Output: k nearest neighbors to q

1 $ep \leftarrow G_h$.entry_point, $L_{max} \leftarrow G_h$.max_layer
2 $V \leftarrow \{ep\}$ // Set of visited nodes
3 $W \leftarrow \{ep\}$ // Set of top candidates
4 **for** $l = L_{max}$ **to** 0 **do**
5 $W \leftarrow$ SearchLayer(G_h, q, l, W, ef, V)
6 **return** *Top k elements of W*

3.3 Search on Pruned HNSW

The removal of redundant edges necessitates a modification to the standard HNSW search algorithm. For the original HNSW, the greedy search in upper layers is sufficient to find a good entry point for the lower layer, and the beam search in the base layer is also sufficient to find the top-k nearest neighbors. However, for the hierarchically pruned HNSW, a node in layer l may have lost the reachability from the current layer. To compensate for this reachability loss, the search results from the upper layer must be carried down to the lower layer. The underlying principle is based on the following proposition:

Proposition 3.2. *Let V_l be the node set in layer l, and W_l be the set of true k-NN results to a query q in layer l. If a node $v \in V_{l+1}$ is in W_l, then v must also be in W_{l+1}.*

Based on Proposition 3.2, in order to find the true k-NN in layer l, we need to consider all nodes reachable from V_{l+1}. Specifically, the set of top candidates found in layer $l+1$ must be passed down and used to initialize the search in layer l. This leads to a fundamental change in the search process: the search in each layer must be a full k-nearest neighbor search (with a beam width of at least k, typically set as ef), not just a greedy search for a single best entry point.

To achieve this, we modify the `SearchLayer` of standard HNSW to perform a k-NN search on a single layer. The pseudo-code is presented in Algorithm 2. It starts from the top layer with the global entry point and iteratively calls `SearchLayer` for each subsequent layer (lines 4–5). The search reuses the candidates W (line 3) from the previous layer as the candidates for the current layer, and it maintains a shared visited set V (line 2) to ensure that each node is processed only once across all layers. Before searching, it initializes a min-heap C for search candidates by copying the candidates W and then converts it into a min-heap structure. The search beam width ef is used throughout all layers to ensure thorough exploration, and finally, the top k results in W are returned (line 6).

3.4 Tunable Trade-off

While hierarchical pruning yields substantial memory savings, it necessitates a more computationally expensive search strategy, which introduces performance overhead compared to the standard HNSW algorithm. This overhead stems primarily from two sources: the increased complexity of per-layer search and the cost of transferring

state between layers. Thus, it imposes a significant performance penalty. Considering the hierarchical structure of HNSW, the base layer contains the vast majority of nodes and edges, making it the primary source of both memory consumption and redundancy, while higher layers are substantially sparser, and the contribution of redundancy is much lower. This observation suggests that a trade-off to strike an optimal balance between memory footprint and search latency. We introduce a tunable trade-off mechanism controlled by a single parameter L_t. This parameter specifies the layer that remains fully connected, while all other layers are pruned. For the unpruned layer L_t, the original neighbors are retained, allowing the search to proceed in the same way as the standard HNSW. Specifically, the search in layers higher than L_t adopts a greedy search, while for layers lower than L_t, a full beam search is used to ensure the reachability of nodes. The trade-off parameter L_t can be set to any layer from 0 to L_{max}, allowing for flexible control over the trade-off between memory savings and search latency. The detailed algorithms and typical settings are in Appendix B.

4 SMALL WORLD PRUNING

While the hierarchical pruning removes the redundancy across layers of the HNSW graph, optimizing the graph structure *within* each layer is an equally critical task. It systematically reconstructs the neighborhood of each node by selecting a subset of edges that are most effective for navigation. The goal is to create a sparser, more efficient graph topology that reduces index size while preserving high search accuracy.

4.1 Systematic Review of Pruning Strategies

Existing graph pruning algorithms, while diverse in their specific implementations, generally follow the three-stage framework consists of initialization, iterative per-node pruning, and final connectivity enhancement.

Initialization prepares the necessary components for pruning. For instance, NSG [9] and NSSG [8] select the navigation points. LEANN [44] identifies a set of top α% high-degree nodes by analyzing the node degree distribution.

Per-Node Pruning iteratively visits each node in the graph to refine its neighborhood, which typically includes two steps: candidate selection and edge pruning. During the candidate selection, a set of potential neighbors is gathered for the current node. NSG and LEANN perform an ANN search to find candidate neighbors for the current node, and NSSG limits the candidate set to two-hop neighbors. During the edge pruning, the candidate neighbors are pruned based on specific heuristics. For example, NSG and LEANN primarily use distance-based heuristics, while NSSG uses angular-based heuristics. Note that LEANN further retains M (a larger number) neighbors for high-degree nodes and m (a smaller number) neighbors for all other nodes, in order to achieve high search accuracy by maintaining different numbers of neighbors for high-degree or low-degree nodes.

Connectivity Enhancement ensures the graph maintains strong connectivity, which is vital for search performance. NSG and NSSG traverse the graph and construct a DFS tree to link disparate components. In contrast, LEANN adopts a more direct yet efficient approach by adding reverse edges (i.e., if node u has v as a neighbor, an edge from v to u is added).

Algorithm 3: Small World Pruning

Input : HNSW Graph G, Parameters $\alpha_0, M_{h_0}, M_{l_0}, \alpha, M_h, M_l$
Output : Pruned Graph G'

```
/* Stage 1: Compute Degree Thresholds          */
// Initialize degree histogram for each layer
1  Hist ← InitializeDegreeHistogram(G)
// Compute degree thresholds for each layer
2  D ← ComputeDegreeThresholds(Hist, α_0, α)
/* Stage 2: Per-Node Pruning                   */
3  foreach node u ∈ V do
4      foreach level l from 0 to u.level do
5          C ← GetNeighbors(u, l)
           // Decide neighbor limit
6          M* ←
             GetNeighborLimit(d, D[l], l, M_{h_0}, M_{l_0}, M_h, M_l)
           // Prune neighbors using HNSW heuristic
7          G'.neighbors[l] ←
             GetNeighborsByHeuristic(C, M*)
/* Stage 3: Connectivity Enhancement           */
8  AddReverseEdge(G')
9  RefineEdge(G')
10 return G'
```

4.2 Small-World Pruning Strategy

Different from kNNG, HNSW has the *small-world* property, i.e., the distribution of nodes' degree is highly skewed. As LEANN identifies high-degree nodes and allows them to retain more neighbors during the pruning, this strategy is essentially in line with the small-world property. However, the pipelines of LEANN is still inefficient, involving parallel unfriendly initialization and expensive candidate selection. Based on the small-world property of HNSW, and inspired by existing pruning strategies [8, 9, 44], we propose an efficient small-world pruning strategy, as the auxiliary of the hierarchical pruning strategy, to further reduce the memory footprint within the layers of the HNSW graph. This method is designed to be lightweight and highly parallelizable, making it suitable for large-scale datasets and dynamic environments.

Algorithm 3 presents the small-world pruning strategy.

Initialization. We use a histogram-based approach to compute the degree distribution of each layer in the HNSW graph (line 1). Instead of storing the degree of each node, we maintain a histogram that counts the number of nodes at each degree. Considering the number of different degrees in each layer is small, we can use a fixed-size array to store the histogram, which requires only $O(d)$ space per layer, where d is the maximum degree of the layer. To identify high-degree nodes, we compute the degree threshold for each layer based on the histogram. For layer 0, we retain the top α_0% nodes, and for upper layers, we retain the top α% nodes. This avoids expensive sorting operations and allows us to efficiently identify high-degree nodes. This step can be parallelized across layers, as each layer's degree distribution is independent of others, furthermore, the maintenance of the histogram can also be done in parallel across nodes within each layer using atomic operations.

Per-Node Pruning. We perform the per-node pruning in parallel across all layers. In HNSW, the number of neighbors in the base layer is usually much greater than that in other layers. Therefore, we set different parameters for layer 0 and other layers: for layer 0, we retain M_{h_0} neighbors for high-degree nodes and M_{l_0} neighbors for others; otherwise, we retain M_h neighbors for high-degree nodes and M_l neighbors for others ($M_h \leq M_{h_0}$, $M_l \leq M_{l_0}$). The maximum number of remained neighbors (i.e., the neighbor limit M^*) depends on i) whether the node is high or low-degree, and ii) the layer of this node (line 6). Though NSSG limits the candidate set to two-hop neighbors, the size of the candidate set is still large, leading to expensive distance calculations. We find that one-hop neighbors are sufficient for pruning because the one-hop neighbors can already result in a well-connected graph. Thus, for each node, we treat its one-hop neighbors as the candidate set. The pruning is performed using the HNSW's heuristic [31], which selects M^* neighbors via distances (line 7). Note that the distance calculations for each neighbor are independent, which allows parallel computations. The pruned neighbors are then stored in the pruned graph G', which is structured similarly to the original HNSW graph.

Connectivity Enhancement. We further enhance the connectivity of the pruned graph by adding reverse edges. This operation, parallelizable across all layers, ensures that if node u has v as a neighbor, an edge from v to u is also guaranteed (line 8), which is crucial for maintaining strong graph connectivity. However, this process may introduce duplicate edges or cause a node's degree to exceed its original neighbor limit; thus, a final refinement step is necessary (line 9). We first deduplicate each node's neighbors, and then check if the neighbor count exceeds the original neighbor limit and reapply the HNSW's original distance-based heuristic to prune if needed, which is fully parallelizable across all nodes and layers, as each node's neighbors are independent of others.

By leveraging the small-world property of HNSW and parallelize the pruning as much as possible, our method is a lightweight, near-node-level operation rather than a slow, graph-level operation. The ANNS over the pruned graph is similar to the original HNSW graph, as it only searches a much smaller set of neighbors, and may lead to a slightly lower recall, but the impact is controllable through the parameters (e.g., $M_{h_0}, M_{l_0}, M_h, M_l$).

5 STORAGE FORMAT

We proceed to introduce the storage format, termed the Compact Hierarchical Adjacency List (CHAL), which is space-efficient and fully updatable. Subsequently, we present a lightweight, incremental update strategy tailored for client-server architectures.

5.1 Compact and Updatable Storage Format

A graph index's in-memory layout determines a crucial balance between update performance and storage size. Existing methods are generally polarized at either extreme of this spectrum. The native HNSW format, for instance, prioritizes fast updates by pre-allocating fixed-size memory blocks for each node's neighbors in every layer. However, it leads to substantial memory waste, as most nodes have far fewer neighbors than the allocated maximum number. Conversely, formats like Compressed Sparse Row (CSR) [25], employed by methods such as LEANN [44], achieve high space utilization by concatenating all neighbors into a single array. This, however, renders the graph extremely static, as any modification requires costly reallocation and copying of all data, making it unsuitable for dynamic environments.

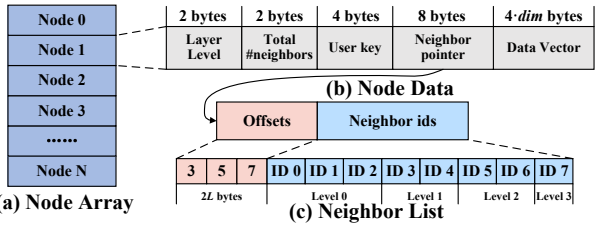


Figure 4: Storage format of CHAL

Motivated by these, we propose the Compact Hierarchical Adjacency List (CHAL), a new HNSW storage format designed to achieve both high space utilization and node-level update performance. As illustrated in Figure 4, CHAL is founded on the principle of decoupling node metadata from the variable-sized neighbor lists. The primary data structure is a node array (as depicted in Figure 4(a)), where each entry (as depicted in Figure 4(b)) stores fixed-size metadata for a single node: its maximum layer, the total number of neighbors, a pointer to its dedicated neighbor list, and its vector data. The neighbor list (as depicted in Figure 4(c)) is a contiguous memory block containing the node's variable-length adjacency information. It is composed of two parts: an *offsets* array, which stores the starting index of each layer's neighbors, and a *neighbor_ids* array, which stores the concatenated neighbor IDs, ordered by layer. Since the base layer's offset is always zero, it can be omitted to save space. The size of the *offsets* array is thus equal to the node's maximum layer, and the *neighbor_ids* array's size equals the node's total neighbor count.

This design ensures that each node's neighbor list is an independent, self-contained entity. When a node's adjacency list is modified, only its own neighbor list needs to be reallocated and its pointer updated in the node data. This localized operation does not affect any other node's data, thereby providing node-level granularity for updates. Consequently, CHAL maintains a compact storage footprint approaching that of LEANN's CSR format while offering the update flexibility required for dynamic applications.

We also discuss node deletion and long-term fragmentation under CHAL in Appendix F.

5.2 Incremental Update Protocol

Pruned graph indexes take much less memory storage and can be deployed locally on the terminal devices. When the data updates, the local indexes become outdated, but they lack the capability for efficient in-place updates. This limitation necessitates a remote update mechanism, whereby the server performs the index update or reconstruction, prunes the index, and then transmits the entire updated index to the client. However, this approach is highly inefficient, as it requires transferring the entire index even for minor updates, leading to significant network overhead/latency.

To address this limitation, we propose a lightweight incremental update protocol that minimizes both network traffic and client-side computation, as illustrated in Figure 5. Step ①, the client issues an update request to the server, transmitting new or modified data vectors with their internal IDs. Step ②, the server then performs the update on its HNSW index, applies the pruning strategy, and writes the updated neighbor lists back to the CHAL format of the

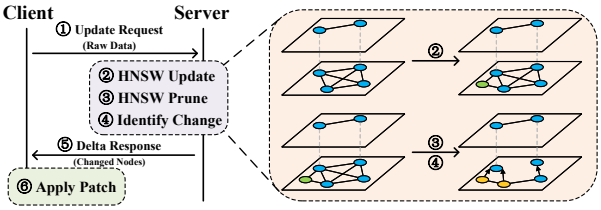


Figure 5: Workflow of the incremental update strategy

pruned index (Step ③). During this process, the server identifies all nodes whose neighbor lists have been changed (Step ④). Next, the server serializes the metadata and neighbor lists (in CHAL format) of only the affected nodes into a compact delta package, which is then transmitted to the client (Step ⑤). This selective transmission significantly reduces the amount of data transferred compared to sending the entire index. Finally, upon receiving the delta package, the client applies the patch to its local index by updating only the metadata and neighbor lists of the modified nodes (Step ⑥). This operation is highly efficient, as it involves modifying only a small subset of nodes rather than the entire index.

This incremental update protocol ensures that local indexes remain synchronized with the server while incurring minimal network and computational overhead. Thus, our approach enables practical deployment in real-world dynamic environments. Note that, HNSW-Slim follows a two-step build-then-prune pipeline rather than a single-pass build, with details in Appendix E.

6 THEORETICAL ANALYSIS

6.1 Index Size Estimation

Under the CHAL layout, the index footprint (excluding vector payloads and runtime buffers) can be decomposed into three parts:

(1) Per-node metadata. Each node stores: max layer (2 bytes), total neighbor count (2 bytes), user key (4 bytes), and a neighbor pointer (8 bytes). Thus

$$S_M = 16n \text{ bytes.} \tag{3}$$

(2) Offsets array. For each node, the offsets array records the starting index of each layer's neighbors, whose length equals the node's maximum layer, which follows a geometric distribution with decay p, and the expectation is $\frac{p}{1-p}$. Each offset uses 2 bytes, hence

$$S_O = \frac{2np}{1-p} \text{ bytes.} \tag{4}$$

(3) Neighbor IDs. It depends on the degree distribution and pruning parameters. Since every node ID occupies 4 bytes and the expected space complexity of HNSW-Slim is

$$\mathbb{E}[S_{\text{Slim}}] = \begin{cases} O\big(n(\alpha M_h + (1-\alpha)M_l)(2 - p + p^{L_t+1})\big), & L_t > 0 \\ O\big(n(\alpha M_h + (1-\alpha)M_l)(2+p)\big), & L_t = 0 \end{cases}$$

for the detailed derivation, see Appendix C.2, the total space is

$$S_N = \begin{cases} 4n(\alpha M_h + (1-\alpha)M_l)(2 - p + p^{L_t+1}), & L_t > 0 \\ 4n(\alpha M_h + (1-\alpha)M_l)(2+p), & L_t = 0 \end{cases} \text{ bytes.} \tag{5}$$

where α is the fraction of high-degree nodes, M_h and M_l are the high/low degree thresholds.

By summing up, the estimated index size is

$$S = S_M + S_O + S_N \text{ bytes.} \tag{6}$$

Note that the estimation is an approximation that ignores the connectivity enhancement stage at the end of small-world pruning, including the reserve edge addition and edge refinement. The reserve edge addition adds additional storage for one-direction edges, and the edge refinement prunes it for space saving; these two operations have opposite effects on the index size. For better estimation and practical parameter setup workflows, please see Appendix D.

6.2 Time Complexity Analysis

We analyze the time complexity of HNSW-Slim from two critical perspectives: the search time and the offline pruning time.

6.2.1 Search Time Complexity. The search process is divided into two phases based on the trade-off layer L_t:

Greedy Search Phase (for layers $l > L_t$): In layers that are higher than the trade-off layer, the search mimics the standard HNSW greedy traversal ($ef = 1$), analogous to the search in the skip list [34], is $O\left(\frac{\bar{M}}{p}\left(\log_{1/p}(n) - 1\right)\right)$. Since only layers $l > L_t$ are traversed, and the expected number of nodes above layer L_t is $n \cdot p^{L_t+1}$, therefore the greedy phase cost is as:

$$\mathbb{T}_{\text{Greedy}} = O\left(\frac{\bar{M}}{p}\left(\log_{1/p}(n\,p^{L_t+1}) - 1\right)\right) \tag{7}$$

Beam Search Phase (for layers $l \le L_t$): In layers $l \le L_t$, a beam search is applied. It first takes $O(ef)$ to convert the top candidate (a max-heap of size ef) into a min-heap to initialize the beam search. Then, a full beam search with beam width ef is performed to compensate for the pruned edges and ensure high recall. The total beam search cost across all layers $l \le L_t$ can be calculated using the greedy search cost derived above, as a beam search can be regard as ef greedy search.

$$\mathbb{T}_{\text{Beam}} = O(ef \cdot L_t + ef \cdot \mathbb{T}_{\text{Greedy},0\to l_t}) = O\left(ef \cdot L_t\left(1 + \frac{\bar{M}}{p}\right)\right) \tag{8}$$

where $ef * \frac{\bar{M}}{p}(\log_{1/p} n - 1)$ denotes the beam search cost over all the layers, while $ef * \frac{\bar{M}}{p}\left(\log_{1/p}(n\,p^{L_t}) - 1\right)$ denotes the beam search over layers $l > L_t$. The total search time complexity $\mathbb{T}_{\text{Slim}}$ is the sum of two above costs:

$$\begin{aligned} \mathbb{T}_{\text{Slim}} &= \mathbb{T}_{\text{Greedy}} + \mathbb{T}_{\text{Beam}} \\ &= \frac{\bar{M}}{p}\left(\log_{1/p}(n\,p^{L_t+1}) - 1\right) + efL_t\left(1 + \frac{\bar{M}}{p}\right) \\ &= \frac{\bar{M}}{p}\log_{\frac{1}{p}} n - \frac{2\bar{M}}{p} + ef + \left(ef - \frac{\bar{M}}{p}\right)L_t + \frac{\bar{M}}{p}efL_t \end{aligned} \tag{9}$$

The complexity can be further simplified to:

$$O(\mathbb{T}_{\text{Slim}}) = O\left(\frac{\bar{M}}{p}\left(\log_{1/p} n + ef(1 + L_t)\right)\right) \tag{10}$$

This demonstrates that the search time complexity of HNSW-Slim still keeps $O(\log n)$, similar to the original HNSW, with the trade-off controlled by L_t. A lower L_t (e.g., $L_t = 0$) reduces the number of layers requiring the expensive beam search, thus lowering latency at the cost of reduced memory savings. Conversely, a higher L_t maximizes memory savings but requires beam search in more layers, increasing the search time. This provides a clear theoretical basis for the optimization of the trade-off between search performance and memory footprint.

6.2.2 Pruning Time Complexity. The time complexity of pruning includes two parts: the cost of small-world pruning and the cost of hierarchical pruning. Note that, although the pruning process is designed to be highly parallelizable, the degree of parallelism is simply achieved by dividing a parallelism factor.

Small-World Pruning. This stage consists of three steps.

- **Initialization:** Computing degree histograms requires visiting all the edges in the layers to be pruned. The complexity is $O(\sum_{l=0}^{L_{max}} \mathbb{E}[N_l]M_l)$, where M_l is the average degree in layer l. While the commutations of threshold degrees in every layer take $O(\sum_{l=0}^{L_{max}} |D_l|)$ cost, where $|D_l|$ is the number of different degrees in layer l.
- **Per-Node Pruning:** This step iteratively visits nodes across all layers to prune edges based on the degree histograms. For each node, we use its one-hop neighbors as candidates. The pruning heuristic has a complexity of roughly $O(d \log d)$ for a node with d neighbors. The total complexity is approximately $O(\sum_{l=0}^{L_{max}} \mathbb{E}[N_l]M_l \log M_l)$, where $\mathbb{E}[N_l]$ is the average number of nodes in layer l.
- **Connectivity Enhancement:** Adding reverse edges requires visiting all the edges in the pruned graph, thus, the cost is proportional to the number of edges remaining after pruning, i.e., $O(\sum_{l=0}^{L_{max}} \mathbb{E}[N_l]\bar{M})$. For the edge refinement, which performs edge deduplication and final edge pruning, takes $O(\sum_{l=0}^{L_{max}} \mathbb{E}[N_l]\bar{M} \log \bar{M})$ cost, where $\bar{M}$ is the average degree after pruning.

The overall complexity for this stage $\mathbb{T}_{\text{SW-Pruning}}$ is dominated by the per-node pruning step: $O(\sum_{l=0}^{L_{max}} \mathbb{E}[N_l]M_l \log M_l)$, but it can be fully parallelized across the neighbors of all nodes in all layers, making it efficient for large graphs.

Hierarchical Pruning. This stage visits every neighbor of nodes in every layer except the trade-off layer L_t to check its maximum layer. The complexity $\mathbb{T}_{\text{H-Pruning}}$ is proportional to the number of neighbors being checked, which is:

$$O\left(\sum_{l=0}^{L_{\max}} \mathbb{E}[N'_l]M'_l - \mathbb{E}[N'_{L_t}]M'_{L_t}\right) \tag{11}$$

where $\mathbb{E}[N'_l]$ denotes the average number of nodes in layer l after pruning, while M'_l denotes the average node degree in layer l after pruning, which are smaller than $\mathbb{E}[N_l]$ and M_l respectively.

To sum up, the total time complexity of the pruning is as:

$$\begin{aligned} O(\mathbb{T}_{\text{Pruning}}) &= O\left(\mathbb{T}_{\text{SW-Pruning}}\right) + O\left(\mathbb{T}_{\text{H-Pruning}}\right) \\ &= O\left(\sum_{l=0}^{L_{\max}} \mathbb{E}[N_l]M_l \log M_l\right) \\ &= O\left(\sum_{l=0}^{L_{\max}} np^l M_l \log M_l\right) \\ &= O\left(\frac{nM \log M}{1-p}\right) \end{aligned} \tag{12}$$

where M denotes the node average degree of HNSW-Slim and n is the number of data vectors. Equation 12 demonstrates that the pruning complexity is linear in the dataset size, and thus, is efficient for large-scale datasets.

Table 1: Statistics of datasets.

Dataset	Dimension	Base Size	Query Size
COHERE [6]	768	1,000,000	10,000
GIST [18]	960	1,000,000	1,000
SIFT [19]	128	6,000,000	10,000
DEEP [2]	96	8,000,000	10,000

7 EXPERIMENTS

In this section, we conduct comprehensive experiments to empirically validate the effectiveness and efficiency of HNSW-Slim. Our evaluation is designed to answer the following questions:

Q1: How significant are the memory savings achieved by HNSW-Slim compared to state-of-the-art baselines (cf. Section 7.2)?

Q2: Does HNSW-Slim maintain competitive search performance (throughput and recall) despite its compressed size (cf. Section 7.3)?

Q3: How efficiently can HNSW-Slim handle dynamic updates in a client-server environment (cf. Section 7.4)?

Q4: How the parameter impacts the index and search performance of HNSW-Slim (cf. Section 7.5)?

7.1 Experimental Settings

7.1.1 Datasets. We evaluate all methods on four real-world datasets. Table 1 summarizes the key statistics of each dataset, including the vector dimension, the number of data vectors, and the number of queries. These datasets cover a range of modalities and application scenarios, ensuring the robustness and generality of our experimental evaluations.

7.1.2 Baselines. We compare HNSW-Slim against several state-of-the-art in-memory ANNS index algorithms, covering classic and recent graph-based methods, as well as quantization-graph indexes:

- **HNSW** [31]: Hierarchical Navigable Small World, which is a widely adopted multi-layer proximity graph structure that achieves high recall and efficiency, and serves as the foundation for our method. In our evaluation, HNSW is deployed on the server side, and the client issues search queries over the network to the server, simulating a typical cloud-based vector search scenario. Additionally, we evaluate **HNSW-PQ** on the client side, which applies Product Quantization (PQ) to compress vectors while maintaining the same index configuration as HNSW.
- **LEANN** [44]: A recent low-storage vector index designed for resource-constrained environments, which employs pruning based on degree distribution on HNSW to minimize memory usage while maintaining search quality.
- **NSG** [9]: Navigating Spreading-out Graph, a monotonic proximity graph that optimizes both search efficiency and index compactness through a distance-based pruning.
- **NSSG** [8]: Satellite System Graph, an extension of NSG that introduces efficient neighbor candidate selection and angle-based pruning to improve pruning efficiency.
- **Vamana** [22]: The in-memory graph index used in DiskANN, which builds a proximity graph with robust connectivity and supports efficient both disk-based and in-memory search.
- **Glass** [48]: An open-source ANNS library developed by Zilliz, which supports various graph indexes and quantization methods.

• **NGT-QG** [21]: A quantization-based graph index that utilizes FastScan [1] to accelerate quantized distance calculations.
• **SymphonyQG** [14]: A recent quantization-based graph index, which also adopts FastScan and integrates the RaBitQ quantization method into the graph index to further optimize memory usage and search performance.

For all baselines, we use their official implementations and parameters reported in the respective papers or repositories if available. Both NGT-QG and SymphonyQG utilize FastScan, which was originally designed for the IVF index to accelerate distance calculations for a batch of vectors within a bucket using SIMD instructions and improving cache hit rates. They adapt this technology to graph indices. However, this adaptation requires storing the quantized code of each neighbor for every node, leading to index size inflation. Additionally, to achieve high recall, they typically necessitate storing raw vectors for re-ranking, which significantly increases memory consumption. As reported [14], their memory footprint on GIST amounts to 22.9GB and 12.5GB, respectively. To enable a fair comparison in resource-constrained environments, we made the modifications in Appendix G.

7.1.3 Parameter Settings. For HNSW-Slim, we configure the default parameters as follows. In the index construction stage, we choose $M = 30$ and efConstruction = 128 to construct the initial HNSW index, with the decay factor $p = 1/32$. In the hierarchical pruning stage, we set the trade-off layer $L_t = 0$, meaning all layers except the base layer are subject to hierarchical pruning. In the small-world pruning stage, we set the degree distribution thresholds $\alpha = \alpha_0 = 2$, i.e., identifying the top 2% highest-degree nodes as hubs. For the base layer, high-degree nodes retain 32 neighbors (i.e., $M_{h_0} = 32$) while other nodes retain 8 neighbors (i.e., $M_{l_0} = 8$); for other layers, high-degree nodes retain 16 neighbors (i.e., $M_h = 16$) while other nodes retain 4 neighbors (i.e., $M_l = 4$).
7.1.4 Evaluation Metrics. To ensure a comprehensive and fair comparison, we adopt several widely used metrics. Index construction performance is measured by the index size and the construction time, where the latter includes both the initial index construction time and the index pruning time. Search performance is assessed by reporting recall-QPS curves, where recall@k (with $k = 3$ default) quantifies the accuracy and QPS (queries per second) quantifies the efficiency. To evaluate the index update performance, we report the average update time per batch and the bandwidth consumed during index updates in the client-server scenario.

Note we report only the size of the index structure, excluding auxiliary data structures such as locks, visited arrays, and other runtime buffers, except for quantization-graph indexes. This measurement ensures a consistent and fair comparison across all methods.
7.1.5 Environment Configuration. Experiments are conducted in a client-server environment to reflect practical deployment scenarios. The server is equipped with an AMD EPYC 9254 CPU (24 cores, 48 threads) and 256GB DDR5 memory, while the client is simulated on an Intel Core i7-12700 processor, restricted to 4 cores, 4GB memory using Docker and cgroups to emulate a resource-constrained edge device. Both server and client run Ubuntu 24.04 LTS and are connected via a 1000 Mbps network. The implementation is in C++ and compiled with g++ 13.3.0 using the *-Ofast*, *-flto*, and *-march=native* options. We utilize TCMalloc [13] for memory management, OpenMP [4] for parallelism, Folly [7] for concurrent data structures, cpp-httplib [20] for network communication, and Protocol Buffers [12] for data serialization.

For small datasets, index construction and pruning are performed on the server, while the resulting pruned index (except HNSW) is transferred to the client for search. HNSW is deployed on the server, and the client issues search queries over the network, simulating a typical server-based vector search scenario. All client-side indexes are evaluated in the resource-constrained environment. For index updates, the client initiates the request, and then the server performs the update and pruning; finally, the updated index (or delta) is sent back to the client. For large datasets, all the methods are deployed on the server.

7.2 Indexing Performance

Table 2 presents the index construction performance, including the index size and the construction time for each method across four real-life datasets. All the indexes are built and pruned on the server side. For pruning-based graph indexes, the indexing time is broken down into the initial construction time and the pruning time. As observed, HNSW-Slim consistently achieves the smallest index size across all datasets. Compared to HNSW, it reduces memory consumption by 4.5×-5.7×. It also outperforms other memory-optimized methods by 27%-299%, demonstrating its effectiveness in reducing memory footprint due to fully reducing HNSW's redundancies, and the acceleration of pruning is not achieved at the cost of space reduction.

Regarding pruning efficiency, HNSW-Slim demonstrates remarkably fast pruning (1.6-11.7s), achieving 238.3×-424.5× speedup over NSG and 16.5×-59.8× over LEANN. NSSG optimizes its pruning strategy, but HNSW-Slim still achieves 5.5×-16.3× speedup. Since Vamana does not follow the build-and-prune pipeline, HNSW-Slim's total indexing time remains competitive with Vamana. The pruning overhead is minimal (1.9%-7.4% of the initial construction time), showing its excellent lightweight and scalability.

Table 3 compares the memory footprint and index construction time with quantization-graph methods. Note that the memory footprint here denotes the actual memory usage on the client deployment to ensure a fair comparison. To adapt to the memory-constrained environments, all methods entail memory footprints close to 4GB. HNSW-Slim can safely run on the client with full-precision vectors because its graph structure is extremely compact. In contrast, quantization graph methods require storing additional codebooks, lookup tables, and quantized codes of every neighbor (for FastScan), which offsets the memory savings from vector quantization. NGT-QG and SymphonyQG must reduce the number of neighbors compared to the default settings, and Glass exceeds the memory limit on DEEP8M under the default settings. In terms of index construction time, HNSW-Slim is significantly faster, demonstrating the efficiency of our pruning-based construction compared to the complex optimization or quantization processes required by other methods. The search performance will be reported below.

7.3 Search Performance

As illustrated in Figure 6, the search performance of all methods is demonstrated through the plotting of their recall-QPS trade-off curves. The first observation is that the HNSW server-deployed

Table 2: Index construction performance (Time includes the initial construction time and the pruning time).

Dataset	COHERE		GIST		SIFT		DEEP	
Algorithm	Size (MB)	Time (s)	Size (MB)	Time (s)	Size (MB)	Time (s)	Size (MB)	Time (s)
HNSW-SLIM	**43.7**	82.8 + **1.6**	**48.0**	131.1 + **3.7**	**341.3**	141.6 + **9.2**	**466.6**	158.9 + **11.7**
LEANN	56.5	82.8 + 97.2	60.9	131.1 + 150.2	450.9	141.6 + 176.6	615.1	158.9 + 204.5
NSG	79.5	90.0 + 680.4	85.8	**79.9** + 887.1	1191.7	197.3 + 2490.9	968.9	**142.2** + 3423.2
NSSG	109.1	138.7 + 27.7	113.1	130.2 + 28.8	1362.6	277.3 + 79.5	1086.7	192.1 + 76.2
Vamana	102.8	**77.4**	110.1	104.8	1107.3	277.6	841.0	258.7
HNSW	248.0	82.8	248.0	131.1	1488.0	**141.6**	1983.9	158.9

Table 3: Index performance of quantization-graph indexes (Memory denotes memory footprint during runtime).

Dataset	COHERE		GIST		SIFT		DEEP	
Algorithm	Memory (MB)	Time (s)	Memory (MB)	Time (s)	Memory (MB)	Time (s)	Memory (MB)	Time (s)
HNSW-SLIM	**3112.0**	**84.4**	3783.4	**134.8**	3726.7	**150.8**	**3750.8**	**170.6**
Glass	3382.8	1029.4	3665.7	1296.7	-	-	3943.9	1397.9
NGT-QG	3546.5	640.8	3829.8	832.3	3919.6	729.2	3866.4	1028.2
SymphonyQG	3430.8	211.4	**3518.0**	222.7	**3126.8**	211.6	3772.0	279.7

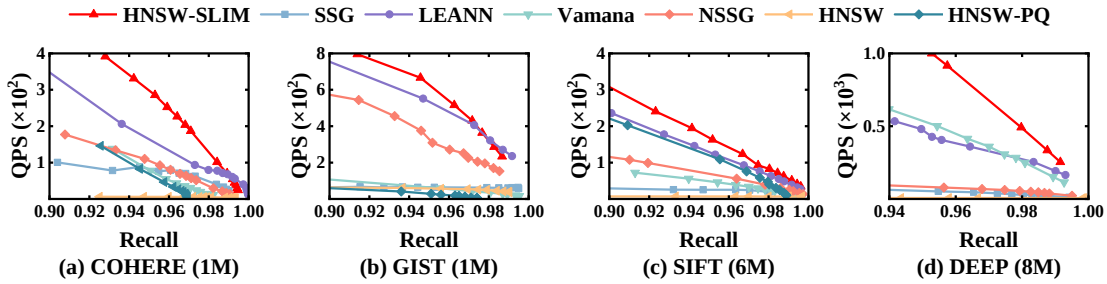


Figure 6: Search performance (recall-QPS trade-off curves).

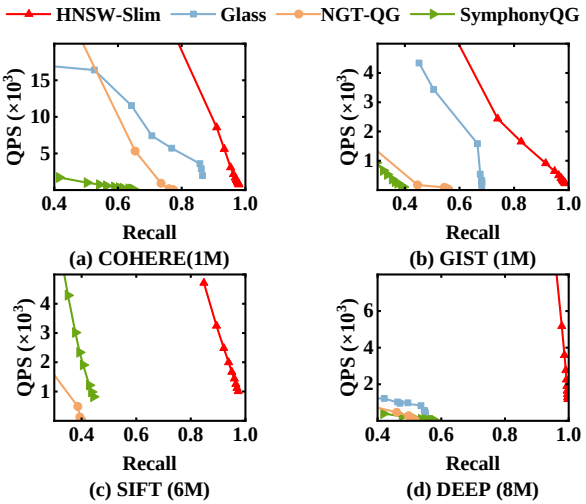


Figure 7: Search performance of quantization-graph indexes.

approach performs the worst in most cases due to the high network latency, while the others are locally deployed indexes. Second, HNSW-SLIM demonstrates superior search performance. Specifically, it outperforms all baselines on the large-scale SIFT and DEEP datasets; on COHERE and GIST datasets, it maintains a strong lead in most cases, with a slight lag behind LEANN at very high recalls, which is an expected trade-off for its significant memory reduction. Note that LEANN cannot support updates due to its underlying CSR storage format. Vamana achieves competitive performance with LEANN on DEEP dataset, but performs worse on other datasets. NSG and NSSG show inferior performance, while NSSG is worse than NSSG due to its trade-off with the pruning strategy. These results demonstrate that HNSW-SLIM successfully retains efficient search performance while achieving substantial memory savings, rendering it highly suitable for utilization in edge devices.

We also deploy HNSW-PQ in the client for comparison. For COHERE, GIST, and SIFT, we use PQ8. While it fits in memory, its performance varies. On SIFT and COHERE, HNSW-PQ is competitive with some graph baselines (e.g., Vamana, LEANN) but fails to reach high recall due to quantization loss. On GIST, it performs significantly worse than other methods. For DEEP, HNSW-PQ with PQ8 exceeds the memory limit. Although PQ4 fits, its maximum recall is only 0.875, falling below the plotted range (0.94-1.0). These results confirm that while quantization reduces memory, it trades recall compared to HNSW-SLIM with full-precision vectors.

Figure 7 presents the search performance of HNSW-SLIM compared with quantization-graph indexes. HNSW-SLIM consistently achieves the best recall-QPS trade-off. Glass employs PQ8 quantization, provides reasonable performance, but lags behind HNSW-SLIM in both QPS and maximum recall. Though integrating FastScan to accelerate the distance computation, NGT-QG and SymphonyQG perform poorly in this resource-constrained client environment. It is important to note that modern quantization-graph methods often utilize quantization not for compression, but to enable high-speed distance computation (e.g., via FastScan) and rely on storing raw vectors for re-ranking to ensure high recall. This design philosophy inherently demands abundant memory and advanced instruction sets (e.g., AVX512), making them ill-suited for resource-constrained edge devices. When forced to run in such environments—by removing re-ranking, reducing neighbors, and using lower-precision

Table 4: Update performance (T: update time, T/v, per-vector update time, BW: bandwidth).

Dataset	COHERE			GIST			SIFT			DEEP		
Algorithm	T (s)	T/v (ms)	BW (MB)	T (s)	T/v (ms)	BW (MB)	T (s)	T/v (ms)	BW (MB)	T (s)	T/v (ms)	BW (MB)
HNSW-Slim	**1.4**	**1.4**	**1.2**	**1.8**	**1.8**	**1.2**	**4.6**	**4.6**	**1.0**	**7.9**	**7.9**	**1.7**
LEANN	552.1	552.1	62.7	786.9	786.9	67.6	948.0	948.0	642.2	1066.9	1066.9	815.0
NSG	734.7	734.7	68.5	914.2	914.2	74.7	*Timeout*	*Timeout*	*Timeout*	*Timeout*	*Timeout*	*Timeout*
NSSG	156.7	156.7	95.0	150.4	150.4	99.1	299.0	299.0	1118.9	403.0	403.0	1290.6
Vamana	48.3	48.3	82.8	65.4	65.4	88.9	178.5	178.5	874.8	193.1	193.1	828.0

quantization (PQ4/3-bit RaBitQ) to fit the memory budget—their performance degrades significantly. In contrast, HNSW-Slim, by pruning the graph structure, offers a more robust and efficient solution for such scenarios.

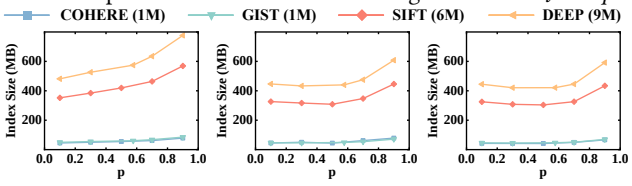
7.4 Update Performance

We evaluate the update performance in a client-server setting, simulating by deploying indexes on the client, initially built on 90% of the dataset, and the remaining 10% of the data is then incrementally inserted in batches of 1,000. We set the timeout to 30 minutes and measure the average update time, per-vector update time, and the network bandwidth required per batch. The results are presented in Table 4, demonstrating that HNSW-Slim achieves near real-time update capabilities, far surpassing all other baselines. Our method completes batch updates in seconds (i.e., effective per-vector update latency in the order of milliseconds), underscoring its suitability for dynamic environments. This high efficiency stems directly from the design that the CHAL storage format enables node-level update, while the incremental update protocol minimizes the network overhead by transmitting only a compact delta package containing the changed nodes' metadata and neighbors.

In contrast, other methods are ill-suited for dynamic updates, and NSG even times out in large datasets. In the absence of incremental protocols, it is necessary to transmit the entire index again, even for minor changes. Particularly, methods like LEANN and NSG, which are characterized by the requirement of computationally intensive pruning, result in a significant delay in the update. Furthermore, the update time of HNSW-Slim scales linearly with the dataset size, because of its linear time complexity. This confirms that our approach is scalable and capable for large-scale dynamic datasets. These findings validate that HNSW-Slim is a good solution for resource-constrained edge devices that require dynamic updates.

7.5 Parameter Study

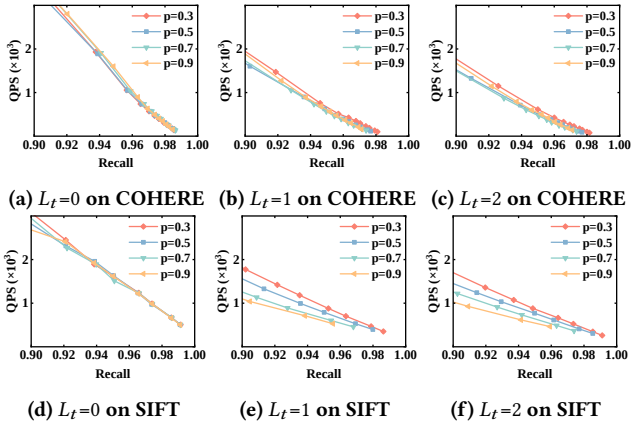
7.5.1 Hierarchical Pruning. To demonstrate the effectiveness of hierarchical pruning, we vary the trade-off layer L_t and the decay factor p used in the hierarchical pruning. We report the index size and search performance for different configurations of L_t and p.



(a) L_t=0 (b) L_t=1 (c) L_t=2

Figure 8: Effect of L_t and p on index size.

Figure 8 shows the index size for different L_t and p values on four datasets. As shown in Figure 8(a), when $L_t = 0$, the index size grows as p increases, which is consistent with the expected space complexity $O(n\bar{M}(2+p))$ analyzed in Sec. C. As shown in Figures 8(b)-8(c), when L_t is greater than 0, the index size first decreases and then increases with p growing, indicating there exists an optimal minimum index size. As discussed in our theoretical analysis, $p^* = \left(\frac{1}{L_t+1}\right)^{\frac{1}{L_t}}$ denotes the value to achieve the minimum index size, i.e., 0.5 when $L_t = 1$, and 0.57 when $L_t = 2$, which is consistent with the observation. Second, the index size decreases significantly when L_t increases from 0 to 1, while it slightly decreases when L_t increases from 1 to 2. This is because the base layer has the most redundancy, and when $L_t = 0$, no hierarchical pruning is performed on the base layer, especially for the large-scale datasets (i.e., SIFT and DEEP).



(a) L_t=0 on COHERE (b) L_t=1 on COHERE (c) L_t=2 on COHERE

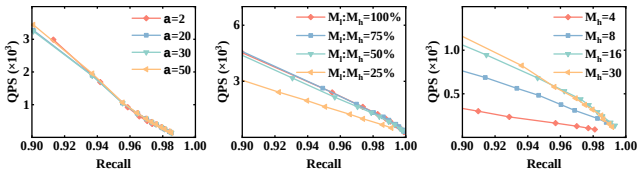
(d) L_t=0 on SIFT (e) L_t=1 on SIFT (f) L_t=2 on SIFT

Figure 9: Effect of L_t and p on search performance.

7.5.2 Small-World Pruning. Figure 9 shows the search performance under different L_t and p values. Here, we only report the results of two datasets due to space limitations and similar observations. As observed, the search performance decreases as L_t increases, as L_t determines the number of layers to perform beam search, which is consistent with the time complexity provided in Eq. 10. Second, search performance increases as p decreases, because p determines the number of neighbors to retain in higher layers. A smaller p means that fewer neighbors are retained, leading to fewer nodes being traversed in the search for higher layers, which is also consistent with the time complexity provided in Eq. 10.

Small-world pruning depends on multiple parameters (i.e., α, α_0, M_h, M_l, M_{h_0}, M_{l_0}). To study their impact on HNSW-Slim, we vary the proportions of parameter pairs. Due to space limits and similar trends, we report the most representative pairs.

In Figure 10(a), when varying $\alpha = \alpha_0$ from 2 to 50, both search performance and the index size are very stable. It reflects the characteristic of the small-world that a small portion of nodes have high degrees, which strongly affects the search performance. Thus, we



α	Size
2	43.7 MB
20	44.0 MB
30	44.2 MB
50	44.2 MB

(a) α on COHERE

M_l:M_h	Size
100%	735.7 MB
75%	661.2 MB
50%	537.2 MB
25%	341.4 MB

(b) M_l : M_h on SIFT

M_h	Size
4	25.3 MB
8	35.9 MB
16	48.0 MB
30	56.7 MB

(c) M_h on GIST

Figure 10: Effect of small-world pruning parameters.

only need to maintain more neighbors for these nodes to ensure high search performance while effectively reducing the index size.

In Figure 10(b), we fix M_h:M_{h_0} and M_h, and vary the ratio of M_l:M_h from 100% to 25%. As observed, search performance degrades while the index size decreases with the decrement of M_l:M_h. This verifies that M_l:M_h can achieve the trade-off between search performance and index size. However, the search performance is stable when M_l:$M_h \geq 50\%$, also reflecting the small-world property.

In Figure 10(c), we fix M_h:M_{h_0} and M_l:M_h, and vary M_h from 4 to 30. As observed, the search performance as well as the index size decrease as M_h decreases, demonstrating the effectiveness of pruning, and $M_h = 16$ is the sweet spot.

7.6 Additional Evaluations

We also conduct additional evaluations to further demonstrate the effectiveness of HNSW-Slim. First, we evaluate the search performance of different indexes under the same memory budget. As shown in Figure 12 in Appendix D, HNSW-Slim consistently outperforms other baselines when constrained to the same index size, confirming that pruning a larger graph yields better performance than constructing a smaller graph from scratch. Second, we evaluate the complementarity between HNSW-Slim and vector quantization. The results in Appendix H show that HNSW-Slim can be effectively integrated with quantization techniques to achieve further memory savings while maintaining competitive search performance. Third, we evaluate HNSW-Slim on large-scale datasets in Appendix I, showing the scalability of HNSW-Slim.

8 RELATED WORK

8.1 Resource-Constrained Vector Search

Deploying ANNS on devices with limited memory and computational power presents a unique set of challenges. To make on-device search feasible, research has explored several complementary directions as stated below.

The primary approach has been compressing the raw vectors and associated indexes. Techniques like Product Quantization (PQ) [23] and dimensionality reduction with PCA [24] reduce vector storage. Although effective, these methods are lossy and can compromise search accuracy. In contrast, graph pruning methods like NSG [9] and NSSG [8] shrink the graph index, without sacrificing the accuracy. However, existing static pruning techniques are often ill-suited for dynamic environments and not fully optimized for HNSW's unique properties. To address this, HNSW-Slim introduces a novel pruning method with an updatable storage format and lightweight update protocol, allowing for efficient memory usage while maintaining the dynamic capabilities of the index. It is worth emphasizing that index pruning and vector quantization are orthogonal and complementary; HNSW-Slim focuses on the former to minimize index overhead, but can be combined with quantization for further data compression.

Another line of works [22, 39, 43] focuses on disk-based indexes for datasets too large to fit in RAM. They employ sophisticated I/O-aware algorithms to minimize disk access, trading higher query latency for the ability to search massive datasets. While powerful, their performance is fundamentally limited by I/O bandwidth. As HNSW-Slim can be used to reduce the memory footprint of the in-memory index, it can be combined to further enhance the search capabilities of resource-constrained devices.

8.2 Pruning-based Graph Indexes

Pruning-based graph indexes start from a dense proximity graph and then remove edges to achieve compactness while preserving navigability. Representatives are NSG [9], NSSG [8], LEANN [44]. NSG and NSSG sparsify flat graphs (e.g., kNNG) via global traversal and distance/angle heuristics, which is effective but costly at scale, and thus, not designed for hierarchical indexes or incremental maintenance. LEANN prunes HNSW by degree distributions, yet still incurs expensive initialization and candidate selection, and adopts CSR storage format that impedes efficient updates.

In contrast, HNSW-Slim introduces two-stage pruning: the small-world pruning eliminates intra-level redundancy, which is inspired by the existing methods but use degree histogram, one-hop neighbors, and reverse edge addition with node-level parallelism for efficient pruning, resulting in 200× less pruning time compared with NSG; the hierarchical pruning removes edges by reachability-based criterion, which differs from the existing flat-graph sparsification, and demands a profound understanding of its cross-layer connectivity. Coupled with a compact, updatable storage format (CHAL) and an incremental update protocol, HNSW-Slim supports node-granular updates with minimal data transfer.

9 CONCLUSIONS

In this paper, we address the critical challenge of HNSW's substantial memory consumption by introducing HNSW-Slim, a novel pruning-based variant that significantly reduces index size while preserving high search performance and enabling near real-time dynamic updates. At its core, HNSW-Slim employs a highly parallelizable synergistic two-stage pruning method, by combining hierarchical and small-world pruning, it eliminates the inter-layer and intra-layer redundancies in the HNSW. This design is supported by a compact, update-friendly storage format and an incremental update protocol. Theoretical analysis provides rigorous guarantees on its space complexity, search time complexity, and pruning time complexity. Extensive evaluations demonstrate that HNSW-Slim reduces memory usage by efficiently pruning, outperforming state-of-the-art baselines while maintaining competitive search performance. By transforming HNSW into a lightweight and updatable index, HNSW-Slim offers a practical solution for deploying high-performance ANNS in resource-constrained environments.

REFERENCES

[1] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2016. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. 9, 4 (2016), 12.

[2] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.

[3] Payam Bahrani, Behrouz Minaei-Bidgoli, Hamid Parvin, Mitra Mirzarezaee, and Ahmad Keshavarz. 2024. A new improved KNN-based recommender system. *The Journal of Supercomputing* 80, 1 (2024), 800–834.

[4] OpenMP Architecture Review Board. 2021. OpenMP Application Program Interface Version 5.2. https://www.openmp.org/specifications. Accessed: 2025-07-10.

[5] Rihan Chen, Bin Liu, Han Zhu, Yaoxuan Wang, Qi Li, Buting Ma, Qingbo Hua, Jun Jiang, Yunlong Xu, Hongbo Deng, et al. 2022. Approximate nearest neighbor search under neural similarity metric for large-scale recommendation. In *CIKM*. 3013–3022.

[6] Cohere. 2022. Wikipedia 2022-12-es Embeddings. https://huggingface.co/datasets/Cohere/wikipedia-22-12-es-embeddings.

[7] Inc. Facebook. 2024. Folly: Facebook Open-source Library. https://github.com/facebook/folly. Accessed: 2025-07-10.

[8] Cong Fu, Changxu Wang, and Deng Cai. 2021. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 8 (2021), 4139–4150.

[9] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *PVLDB* 12, 5 (2019), 461–474.

[10] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2025. Practical and Asymptotically Optimal Quantization of High-Dimensional Vectors in Euclidean Space for Approximate Nearest Neighbor Search. 3, 3 (2025), 202:1–202:26.

[11] Jianyang Gao and Cheng Long. 2024. RaBitQ: quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *SIGMOD* 2, 3 (2024), 1–27.

[12] Google. 2024. Protocol Buffers. https://github.com/protocolbuffers/protobuf. Accessed: 2025-07-10.

[13] Google. 2024. TCMalloc: Thread-Caching Malloc. https://github.com/google/tcmalloc. Accessed: 2025-07-10.

[14] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. 2025. SymphonyQG: Towards Symphonious Integration of Quantization and Graph for Approximate Nearest Neighbor Search. *SIGMOD* 3, 1 (2025), 1–26.

[15] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. 2022. Manu: a cloud native vector database management system. *PVLDB* 15, 12 (2022), 3548–3561.

[16] Ali H Hamad. 2021. Smart campus monitoring based video surveillance using haar like features and k-nearest neighbour. *International Journal of Computing and Digital Systems* 10 (2021).

[17] harsha simhadri. 2003. https://github.com/harsha-simhadri/big-ann-benchmarks/blob/main/neurips23/README.md. Accessed: 2025-10-20.

[18] Jégou Hervé, Douze Matthijs, and Schmid Cordelia. 2008. The GIST Dataset. http://corpus-texmex.irisa.fr/. Accessed: July 9, 2025.

[19] Jégou Hervé, Douze Matthijs, and Schmid Cordelia. 2008. The SIFT Dataset. http://corpus-texmex.irisa.fr/. Accessed: 2025-07-09.

[20] Yuji Hirose. 2024. cpp-httplib: A C++11 single-file header-only HTTP/HTTPS server and client library. https://github.com/yhirose/cpp-httplib. Accessed: 2025-07-10.

[21] Yahoo Japan. 2018. Neighborhood Graph and Tree for Indexing High-dimensional Data. https://github.com/yahoojapan/NGT. Accessed: 2025-11-29.

[22] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing Systems* 32 (2019).

[23] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2010), 117–128.

[24] Ian T Jolliffe and Jorge Cadima. 2016. Principal component analysis: a review and recent developments. *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences* 374, 2065 (2016), 20150202.

[25] Humayun Kabir, Joshua Dennis Booth, and Padma Raghavan. 2014. A multilevel compressed sparse row format for efficient sparse computations on multicore processors. In *HiPC*. 1–10.

[26] Minseok Kim, Seunghwan Jung, Eunkyeong Kim, Baekcheon Kim, Jinyong Kim, and Sungshin Kim. 2023. A fault detection and isolation method via shared nearest neighbor for circulating fluidized bed boiler. *Processes* 11, 12 (2023), 3433.

[27] Hrishikesh Kulkarni, Nazli Goharian, Ophir Frieder, and Sean MacAvaney. 2024. LexBoost: Improving Lexical Document Retrieval with Nearest Neighbors. In *Proceedings of the ACM Symposium on Document Engineering*. 1–10.

[28] Shige Liu, Zhifang Zeng, Li Chen, Adil Ainihaer, Arun Ramasami, Songting Chen, Yu Xu, Mingxi Wu, and Jianguo Wang. 2025. TigerVector: Supporting vector search in graph databases for advanced RAGs. In *SIGMOD*. 553–565.

[29] Dario Lucarella. 1988. A document retrieval system based on nearest neighbour searching. *Journal of Information Science* 14, 1 (1988), 25–33.

[30] 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.

[31] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2018), 824–836.

[32] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector database management techniques and systems. In *Companion of the 2024 International Conference on Management of Data*. 597–604.

[33] Pinecone. 2023. HNSW: Hierarchical Navigable Small World Graphs. https://www.pinecone.io/learn/series/faiss/hnsw/. Accessed: 2025-07-09.

[34] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[35] Qdrant. 2021. Qdrant: Vector Database. https://qdrant.tech/. Accessed: 2025-07-09.

[36] Korakit Seemakhupt, Sihang Liu, and Samira Khan. 2024. EdgeRAG: Online-Indexed RAG for Edge Devices. *arXiv preprint arXiv:2412.21023* (2024).

[37] Tomer Sharon et al. 2024. Why Vector Search in Elasticsearch? The Rationale Behind the Design. https://www.elastic.co/search-labs/blog/vector-search-elasticsearch-rationale. Accessed: 2025-07-09.

[38] The Vald Team. 2019. Vald: A Highly Scalable Distributed Vector Search Engine. https://github.com/vdaas/vald. Accessed: 2025-07-09.

[39] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Xuecang Zhang, Junhua Zhu, and Yu Zhang. 2024. FusionANNS: An Efficient CPU/GPU Cooperative Processing Architecture for Billion-scale Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2409.16576* (2024).

[40] Vespa.ai. 2017. Vespa: The open source big data serving engine. https://vespa.ai. Accessed: 2025-07-09.

[41] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *SIGMOD*. 2614–2627.

[42] Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Yifan Zhu, and Wenchao Zhou. 2025. Accelerating Graph Indexing for ANNS on Modern CPUs. *SIGMOD* 3, 3 (2025), 1–29.

[43] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. *SIGMOD* 2, 1 (2024), 1–27.

[44] Yichuan Wang, Shu Liu, Zhifei Li, Yongji Wu, Ziming Mao, Yilong Zhao, Xiao Yan, Zhiying Xu, Yang Zhou, Ion Stoica, et al. 2025. LEANN: A Low-Storage Vector Index. *arXiv preprint arXiv:2506.08276* (2025).

[45] Weaviate. 2019. Weaviate: The Open Source Vector Database. https://github.com/weaviate/weaviate. Accessed: 2025-07-09.

[46] N Kadek Ayu Wirdiani, Praba Hridayami, N Putu Ayu Widiari, K Diva Rismawan, Putu Bagus Candradinata, and I Putu Deva Jayantha. 2019. Face identification based on K-nearest neighbor. *Scientific Journal of Informatics* 6, 2 (2019), 150–159.

[47] Zhe Zhou, Chenglin Wen, and Chunjie Yang. 2016. Fault isolation based on k-nearest neighbor rule for industrial processes. *IEEE Transactions on Industrial Electronics* 63, 4 (2016), 2578–2586.

[48] Zilliz. 2023. Pyglass - Graph Library for Approximate Similarity Search. https://github.com/zilliztech/pyglass. Accessed: 2025-11-29.

A PROOF OF PROPOSITIONS

Proposition A.1 (Hierarchical Redundancy). *Let edge* $(u \to w) \in E_{l+1}$ *and edge* $(v \to w) \in E_l$*, then edge* $(v \to w)$ *in layer* l *is redundant.*

Proof. As node w is the neighbor of u in the higher layer, once u is reached in layer $l+1$, the search is able to access w directly in layer l (due to the top-down traversal strategy). While v is the neighbor of w in layer l, the edge $(v \to w)$ in layer l does not improve reachability and is therefore redundant. □

B TRADE-OFF HIERARCHICAL PRUNING

Algorithm 4: Trade-off Hierarchical Pruning

Input: HNSW graph G, trade-off layer L_t
Output: Partially pruned graph G'_h

1 **foreach** $u \in G$ **do**
2 $u' \leftarrow u$ `// Create a copy of the node`
3 **for** $l = 0$ **to** $u.l_{max}$ **do**
4 **if** $l == L_t$ **then**
  `// Keep original neighbors for layer` L_t
5 $u'.neighbors[l] \leftarrow U$
6 **else**
7 $U \leftarrow \emptyset$
8 **foreach** $v \in u.neighbors[l]$ **do**
9 **if** $v.l_{max} == l$ **then**
10 $U \leftarrow U \cup \{v\}$
11 $u'.neighbors[l] \leftarrow U$
12 $G'_h \leftarrow G'_h \cup \{u'\}$
13 **return** G'_h

Algorithm 4 details the trade-off hierarchical pruning process, which retains the original neighbors for the layer L_t and prunes all other layers similar to Algorithm 1.

Algorithm 5: Trade-off Search on Hierarchically Pruned HNSW

Input: Partially pruned graph G'_h, query q, number of query result k, beam width ef, trade-off layer L_t
Output: k nearest neighbors to q

1 $ep \leftarrow G'_h$.entry_point
2 $L_{max} \leftarrow G'_h$.max_layer
3 $V \leftarrow \{ep\}$ `// Set of visited nodes`
4 $W \leftarrow \{ep\}$ `// Set of top candidates`
5 **for** $l = L_{max}$ **to** $L_t + 1$ **do**
6 $W \leftarrow \text{SearchLayer}(G'_h, q, l, W, 1, V)$
7 **for** $l = \min(L_t, L_{max})$ **to** 0 **do**
8 $W \leftarrow \text{SearchLayer}(G'_h, q, l, W, ef, V)$
9 **return** *Top k elements of W*

Algorithm 5 presents the corresponding search algorithm on the hierarchically pruned graph G'_h returned by Algorithm 4. The search starts at the entry point of the graph and proceeds through the layers. For layers higher than L_t, it performs a greedy search to find the best entry point for the next layer (lines 5–6); while for layers equal to or lower than L_t, it performs a full beam search using the candidates from the previous layer (lines 7–8).

The typical settings and their implications are described as:

- **Maximum Memory Savings ($L_t = L_{max}$):** It sets L_t to the maximum layer L_{max} will prune all layers. This configuration achieves the highest memory savings by removing all redundant edges across the entire hierarchy. However, it incurs the highest search overhead, as every layer requires a full beam search with a beam width of ef, and the search candidates must be passed down and initialized at each layer transition.
- **Maximum Search Speed ($L_t = 0$):** Setting L_t to 0 retains the original neighbors for the base layer while pruning all upper layers. This configuration minimizes search latency by only requiring the search in the base layer to be a full beam search and all the upper layers to be greedy searches. This setting is ideal for scenarios where search efficiency is critical, but it sacrifices memory savings by retaining the base layer, which is the densest layer and contains the most redundant edges.
- **Balanced Performance ($L_t = 1$):** Setting L_t to 1 retains the original neighbors for the layer 1 while pruning all the upper layers and also the most dense base layer. In this setting, the search in layers 0 and 1 is a full beam search, while the search in layers higher than 1 is greedy. It achieves a balance between memory savings and search latency.

C SPACE COMPLEXITY ANALYSIS

HNSW-Slim is designed to reduce the memory footprint associated with standard HNSW. Thus, we first provide the space complexity of HNSW and then that of HNSW-Slim.

C.1 Space Complexity of HNSW

The space complexity of an HNSW graph is determined by the total number of neighbor pointers stored across all layers. Let n be the total number of data vectors, and the connectivity is governed by two key parameters: M, the maximum number of neighbors for nodes in layers $l > 0$, and M_0, the maximum for the base layer ($l = 0$), which is conventionally set to $2M$. Analogous to skip lists, HNSW assigns each node a maximum layer l_{max} randomly from a geometric distribution with a decay factor $p \in (0, 1)$. Consequently, the expected number of nodes present in any given layer l, denoted $\mathbb{E}[N_l]$, is $O(n \cdot p^l)$.

The total expected space, $\mathbb{E}[S_{\text{HNSW}}]$, is the sum of the expected space cost of the base layer, $\mathbb{E}[S_0]$, and that of all upper layers, $\mathbb{E}[S_{>0}]$. For the base layer ($l = 0$), the space cost is:

$$\mathbb{E}[S_0] = O(n \cdot M_0) = O(2nM) \tag{13}$$

For the upper layers ($l > 0$), the expected space cost is:

$$\mathbb{E}[S_{>0}] = \sum_{l=1}^{L_{\max}} \mathbb{E}[N_l] \cdot M = O\left(nM \sum_{l=1}^{L_{\max}} p^l\right) \tag{14}$$

where $L_{\max}$ is the highest layer. As $\sum_{l=1}^{\infty} p^l$ converges to $\frac{p}{1-p}$, the total expected space complexity is:

$$\mathbb{E}[S_{\text{HNSW}}] = \mathbb{E}[S_0] + \mathbb{E}[S_{>0}] = O\left(nM\left(2 + \frac{p}{1-p}\right)\right) \quad (15)$$

Equation 15 highlights that the space cost of HNSW depends on hyperparameters M and p. The term $M(2+\frac{p}{1-p})$ acts as a significant constant factor, and our HNSW-Slim aims to reduce this factor.

C.2 Space Complexity of HNSW-Slim

HNSW-Slim's space efficiency stems from the two-stage pruning strategy that simultaneously reduces inter-layer and intra-layer edge density. We analyze its effect on the expected space complexity.

Small-World Pruning. The uniform neighbor limit M is replaced by two distinct limits based on the degree: M_h for the top α fraction of high-degree nodes (hubs) and M_l for the remaining $(1-\alpha)$ fraction of low-degree nodes. In the base layer ($l = 0$), they are doubled to $2M_h$ and $2M_l$ for simplicity.

Hierarchical Pruning. To balance memory and performance, hierarchical pruning is applied to all layers except for the specified trade-off layer L_t. For any pruned layer l, neighbors with the maximum layer of l are retained.

Let $\bar{M} = \alpha M_h + (1-\alpha)M_l$ be the average neighbor number in the upper layers, and $\bar{M}_0 = 2\bar{M}$ be the average neighbor number in the base layer after small-world pruning. Given the expected space of HNSW (Equation 15), the expected space after small-world pruning, denoted $\mathbb{E}[S_{\text{SW}}]$, is as:

$$\mathbb{E}[S_{\text{SW}}] = O\left(n\bar{M}\left(2 + \frac{p}{1-p}\right)\right) \quad (16)$$

After applying hierarchical pruning, the expected space is further reduced. To estimate this reduction, we make a reasonable assumption that the layer distribution of any node's neighbors is independent and identically distributed (i.i.d.) to the global layer distribution of all nodes in the dataset. Based on this assumption, for a given node, its neighbors consist of two parts: nodes are the same as those in the higher layer with a probability p (i.e., the decay factor of HNSW) and nodes in the same layer with a probability 1-p. Thus, the expected space after small-world and hierarchical pruning, denoted $\mathbb{E}[S_{\text{H}}]$, is as:

$$\mathbb{E}[S_{\text{SW+H}}] = O\left(n\bar{M}(1-p)\left(2 + \frac{p}{1-p}\right)\right) = O(n\bar{M}(2-p)) \quad (17)$$

Considering the trade-off layer L_t (a specific layer is not hierarchically pruned), when $L_t > 0$, the expected space of HNSW-Slim, denoted $\mathbb{E}[S_{\text{Slim}_{L_t>0}}]$, is as:

$$\begin{aligned}
\mathbb{E}[S_{\text{Slim}_{L_t>0}}] &= \mathbb{E}[S_{\text{SW+H}}] - O(\bar{M}(1-p)p^{L_t}n) + O(\bar{M}p^{L_t}n) \\
&= \mathbb{E}[S_{\text{SW+H}}] + O(\bar{M}p^{L_t+1}n) \\
&= O(n\bar{M}(2-p) + \bar{M}p^{L_t+1}n) \\
&= O(n\bar{M}(2-p+p^{L_t+1})) \\
&= O(n(\alpha M_h + (1-\alpha)M_l)(2-p+p^{L_t+1}))
\end{aligned} \quad (18)$$

Equation 18 provides the expected memory cost of HNSW-Slim, where $2-p+p^{L_t+1}$ is reduction via the hierarchical pruning and $\bar{M} = (\alpha M_h + (1-\alpha)M_l)$ is the reduction via the small-world pruning, which are smaller than $2+\frac{p}{1-p}$ and M in Equation 15. Specifically, since the limit of neighbors in the base layer is doubled, the expected space when the trade-off layer is the base layer (i.e., $L_t = 0$), the expected space complexity is $O(n\bar{M}(2+p))$. Therefore, the expected space of HNSW-Slim, denoted $\mathbb{E}[S_{\text{Slim}}]$ is as:

$$\mathbb{E}[S_{\text{Slim}}] = \begin{cases} O\big(n(\alpha M_h + (1-\alpha)M_l)(2-p+p^{L_t+1})\big), & L_t > 0 \\ O\big(n(\alpha M_h + (1-\alpha)M_l)(2+p)\big), & L_t = 0 \end{cases} \quad (19)$$

An interesting insight is that for fixed values of $\bar{M}$ and $L_t > 0$, there exists an optimal value for the decay factor p that minimizes the expected space. To find this optimum, we analyze the function $f(p) = 2 - p + p^{L_t+1}$ for $p \in (0,1)$. By taking the first derivative with respect to p and setting it to zero, we find the critical point:

$$f'(p) = -1 + (L_t+1)p^{L_t} = 0 \implies p^* = \left(\frac{1}{L_t+1}\right)^{\frac{1}{L_t}} \quad (20)$$

The second derivative, $f''(p) = (L_t+1)L_t p^{L_t-1}$, is positive for all $p \in (0,1)$ and $L_t > 0$, confirming that p^* corresponds to a local minimum. By substituting this optimal value back into Equation 18, we can determine the minimum expected space for a given L_t:

$$\begin{aligned}
\min \mathbb{E}[S_{\text{Slim}}] &= O\left(n\bar{M}\left(2 - p^* + p^{*L_t+1}\right)\right) \\
&= O\left(n\bar{M}\left(2 - \left(\frac{1}{L_t+1}\right)^{\frac{1}{L_t}} + \left(\frac{1}{L_t+1}\right)^{\frac{L_t+1}{L_t}}\right)\right) \\
&= O\left(n\bar{M}\left(2 - \frac{L_t}{L_t+1}\left(\frac{1}{L_t+1}\right)^{\frac{1}{L_t}}\right)\right)
\end{aligned} \quad (21)$$

Equation 21 provides a theoretical minimum space cost of HNSW-Slim for a selected trade-off layer L_t, offering a clear quantitative guide for hyperparameter tuning.

D PARAMETER SETUP

While Appendix B discusses the typical settings and implications of the hierarchical pruning parameter L_t, it is more practical to determine suitable values in advance within a specific memory budget, avoiding multiple rounds of pruning. In addition, the small-world pruning has many parameters (e.g., α, M_h, M_l), which are hard to set without empirical studies. This section provides a practical workflow for parameter setup of HNSW-Slim.

As discussed in Section 6.1, the index size can be estimated given the dataset size and the pruning parameters via Equation 6.

To validate the accuracy of this estimation, experiments are conducted to compare the estimation with the actual index size by setting p, L_t, α, M_h, and M_l as stated in Section 7, and varying the data number n. As illustrated in Figure 11, the theoretical estimation of the index size is very close to the real size but still has some discrepancies. However, the discrepancies observed are primarily attributed to the connectivity enhancement as discussed in Section 6.1. Since the purpose of connectivity enhancement is to prevent the in-degree of nodes from being too low, we provide an adjustment named HNSW-SlimZero for better estimation. During small world pruning, we first check neighbors' in-degree, and retain neighbors with in-degree less than the threshold to avoid additional reverse addition and edge refinement. The experimental results show that this adjustment can significantly reduce the gap

between estimation and empirical measurement, as shown in Figure 11. Here, the in-degree threshold for the base layer and other layers was set to half the corresponding low-degree threshold (i.e., M_{l_0}, M_l). This guarantees that, in the worst-case scenario, half of the neighbors of low-degree nodes originate from their original nearest neighbors, while the other half come from nodes with in-degrees below the threshold. This approach impairs the node's navigability, while also avoids the retention of too few low-in-degree nodes.

To further validate HNSW-SlimZero, more general experiments are conducted in comparison with HNSW-Slim and baselines. It is ensured that all indexes are kept at the same values, simulating real-world scenarios with specified memory budgets. To ensure fairness, the sizes of the other solutions are maintained as closely as possible to that of the HNSW-SlimZero. As illustrated in Figure 12, the query performance of indexes of equivalent size is demonstrated. HNSW-SlimZero is comparable to HNSW-Slim, yet it demonstrates superiority over other baselines, suggesting that the adjustment does not compromise the quality of the index. LEANN lags behind them, but still performs better than HNSW, further demonstrating that pruning on HNSW yields better performance than directly constructing a smaller HNSW.

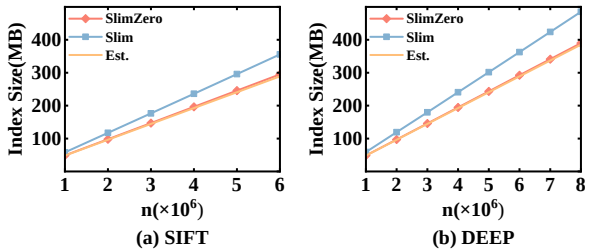


Figure 11: Index size estimation

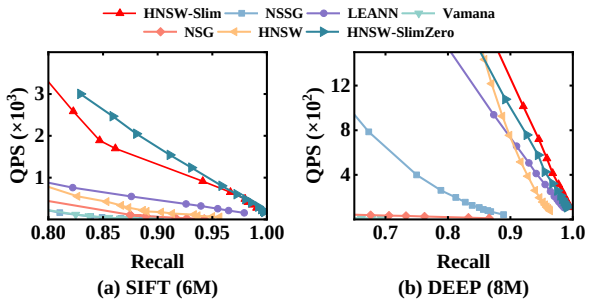


Figure 12: Search performance under the same index size

E BUILD-AND-PRUNE PIPELINE RATIONALE

In this section, we discuss the rationale behind why HNSW-Slim adopts a two-step build-and-prune pipeline, rather than interleaving pruning with construction. While it is theoretically possible to perform pruning immediately after each node insertion, such an approach would significantly degrade both efficiency and accuracy. The reasons are as follows:

(1) Small-world pruning depends on per-layer degree histograms and thresholds that stabilize only after the graph is fully constructed. During incremental graph construction, the degree distribution of nodes may differ substantially from the final graph, leading to premature or suboptimal pruning that can negatively affect the ultimate search performance.
(2) The CHAL format supports efficient node-level updates, but does not support efficient neighbor-level updates for space reduction. Interleaving pruning with construction would require frequent neighbor-level updates that trigger repeated reallocation of variable-length adjacency lists and pointer updates, substantially degrading efficiency.
(3) Pruning inherently introduces loss in search accuracy, and if pruning is interleaved with graph construction, this loss can accumulate and be amplified, resulting in a substantial degradation of the final index quality.
(4) Mainstream pruning algorithms (e.g., NSG/NSSG) follow the same two-step build-then-prune pipeline, which has been empirically validated in practice.

This rationale explains why HNSW-Slim adopts a build-and-prune pipeline to ensure both efficiency and accuracy.

F NODE DELETION AND FRAGMENTATION

HNSW-Slim builds upon HNSW, which supports deletion by marking nodes as deleted and reusing their IDs. Upon subsequent insertions, these recycled IDs are preferentially assigned, and the incoming node's metadata is written back to the existing slot in the node array. Hence, the space in the node array is reusable and not permanently wasted. Since HNSW-Slim preserves this behavior, deletions do not inherently force periodic rebuilds due to unreusable fragmentations.

In CHAL, each node's neighbor list is a contiguous memory block that can be freed and reallocated independently. Unlike the fixed blocks in native HNSW, CHAL's neighbor lists are compact and variable-sized, which can not be reused for other nodes. When a node is deleted, its neighbor list is freed, and new insertions need to allocate a new neighbor list. Over long runs, this may lead to heap fragmentation. However, ANNS workloads primarily involve graph traversal, and node accesses are usually random, while accesses to a node's neighbors are sequential. CHAL is dedicated to the ANNS index, ensuring that the access pattern consists of random access to nodes and sequential access to each node's neighbors; thus, the overall impact of fragmentation is negligible.

In the implementation, HNSW-Slim also utilizes TCMalloc for memory allocation and management, which incorporates several advanced strategies to mitigate memory fragmentation effectively. Therefore, only in cases with heavy dynamic updates, periodic index rebuilding is considered as a maintenance operation.

To validate these claims, we conduct a simulation experiment on long-running updates. Specifically, the update batch constitutes 1% of the dataset. For each insertion, we provide a predefined probability of deleting previously inserted nodes, which are finally reinserted. By varying the deletion probability, we can control the degree of memory fragmentation and then evaluate both the extent of fragmentation and the search throughput under different recalls. As shown in Table 5, our results indicate that although memory fragmentation exists, the difference in search performance is very marginal under different deletion probabilities, showing the resilience of long-term fragmentation.

Table 5: Impact of deletion probability on performance

SIFT				
Prob.	**Frag.**	**90% QPS**	**98% QPS**	**99% QPS**
0.0	2.1%	9690	2783	1898
0.1	6.4%	10331	2963	1917
0.3	6.0%	10320	2921	1881
0.4	7.6%	10352	2919	1941
0.5	6.6%	10050	2760	1874
GIST				
Prob.	**Frag.**	**90% QPS**	**98% QPS**	**99% QPS**
0.0	3.5%	1984	530	299
0.1	4.5%	1433	560	287
0.3	6.7%	1733	552	286
0.4	6.4%	1633	504	287
0.5	7.4%	1724	552	282

G QUANTIZATION-GRAPH INDEX SETUP

- **Glass.** Since Glass does not employ FastScan and reranking, we don't modify it and use NSG with the same configurations in Section 7, and employ PQ8 quantization.
- **NGT-QG.** To minimize memory usage, we employ PQ4 and do not store raw vectors in the index, and disable the reranking during the search. Since NGT-QG requires storing quantized codes for each neighbor, the memory consumption for edges is much higher. To mitigate this, we limit the maximum number of neighbors to 20.
- **SymphonyQG.** The original implementation of SymphonyQG mandates the use of the AVX512 instruction set, which is typically available only on high-end server CPUs and is not supported by most of client-side hardware, including ours (Intel Core i7-12700). To address this, we implemented an AVX2 version of SymphonyQG. Similar to NGT-QG, we do not store raw vectors and disable reranking. The original SymphonyQG only supports 1-bit RaBitQ quantization, which results in very low recall without reranking. To overcome this limitation, we migrate the Extended RaBitQ [10] to SymphonyQG, enabling multi-bit RaBitQ quantization. In our experiments, we select 3-bit quantization. Furthermore, SymphonyQG requires storing additional distance factors besides neighbor quantization codes, necessitating a limit on the number of neighbors. The original implementation requires the number of neighbors to be a multiple of 32 to fully utilize AVX512 for FastScan. We adapt this to use SSE instructions, allowing the neighbor count to be a multiple of 8, and set the number of neighbors to 8 for the client environment.

It is worth noting that these modifications, while necessary for the resource-constrained setting, inevitably compromise the performance of these methods, which are originally optimized for high-end server environments with abundant memory.

H QUANTIZATION INTEGRATION

To evaluate the complementarity between HNSW-Slim and vector quantization, we implement HNSW-SlimQ, directly integrating the SOTA vector quantization RaBitQ [10] to HNSW-Slim. We compare HNSW-SlimQ with HNSW-RaBitQ and HNSW-PQ, which integrate RaBitQ and PQ into HNSW. The experiment is conducted on a client, where HNSW is deployed to the client by applying vector quantization, and we also compare HNSW-Slim with raw vectors. All the quantization techniques were configured to ensure equivalence to storing 4 bits per dimension of the raw vector.

Table 6 demonstrates the index construction performance with quantization integration. Vector quantization has been shown to result in significant memory savings, thereby enabling the deployment of oversized HNSW in memory-constrained environments. While PQ exhibits a significant delay in the index construction, RaBitQ has a negligible impact on it, but stores extra distance and padding to accelerate search performance, causing more memory usage. HNSW-SlimQ reduces the total size by 27.7%-56.8% compared to HNSW-RaBitQ and HNSW-PQ, due to a significantly smaller index. Furthermore, it reduces the total size by 65.3%-85.6% compared to HNSW-Slim, due to highly compressed vector quantization. Note that the index size of HNSW-SlimQ is marginally larger than that of HNSW-Slim. This discrepancy can be attributed to the presence of more nearest neighbors during construction, which is a consequence of the reduced distinction distance resulting from quantization. The combination of graph pruning and vector quantization has been demonstrated to result in a substantial reduction in the memory usage of an ANNS index.

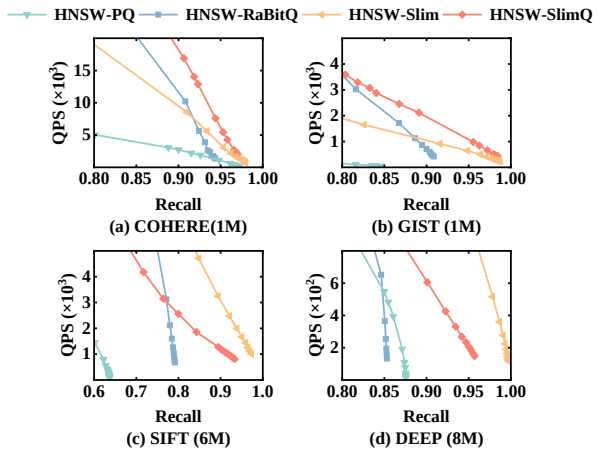


Figure 13: Search performance with quantization integration.

Figure 13 illustrates the search performance with quantization integration. Since quantization is lossy, quantization–graph methods inevitably trade recall for memory, and their recall ceilings are below HNSW-Slim. Among the quantization-integrated methods, HNSW-SlimQ consistently achieves the highest recall. Its pruning step filters ambiguous neighbors that share identical quantized distances, partially compensating for the reduced distance discrimination caused by quantization. HNSW-SlimQ also delivers the highest QPS among the quantization-integrated baselines, approaching HNSW-Slim and even surpassing it on COHERE and GIST. By contrast, HNSW-PQ exhibits unstable behavior and markedly lower recall ceiling on SIFT and GIST. HNSW-RaBitQ improves throughput over HNSW-PQ, but still trails HNSW-SlimQ. Overall, these results confirm that HNSW-Slim and vector quantization are complementary: HNSW-SlimQ retains competitive efficiency at a much smaller memory footprint, while raw HNSW-Slim remains

Table 6: Index construction performance with quantization integration. Index build time (s) and memory (MB).

Dataset	COHERE				GIST				SIFT				DEEP			
Method	**Time**	**Total**	**Index**	**Vector**	**Time**	**Total**	**Index**	**Vector**	**Time**	**Total**	**Index**	**Vector**	**Time**	**Total**	**Index**	**Vector**
HNSW-PQ	237.6	626.5	259.5	**366.2**	277.9	718.3	259.5	**457.8**	303.6	1923.5	1557.2	**366.2**	293.0	2442.5	2076.2	**366.2**
HNSW-RabitQ	87.5	648.8	263.6	385.3	**107.1**	740.4	263.6	476.8	**139.1**	2061.6	1581.0	480.7	171.5	2748.8	2107.9	640.9
HNSW-Slim	**82.8+1.6**	3022.2	**46.3**	2976.0	131.1+3.7	3713.2	**51.1**	3662.1	141.6+9.2	3293.7	**364.0**	2929.7	**158.9+11.7**	3426.3	**496.6**	2929.7
HNSW-SlimQ	87.5+1.5	**439.9**	54.6	385.3	107.1+1.6	**535.2**	58.3	476.8	139.1+7.9	**881.2**	400.5	480.7	171.5+10.2	**1188.5**	547.7	640.9

preferable when the highest recall is required. Although the current implementation of HNSW-SlimQ is a direct integration, the results already demonstrate its significant potential. In future work, we plan to explore deeper optimizations to better integrate vector quantization with the pruned graph structure, further enhancing both the efficiency and accuracy.

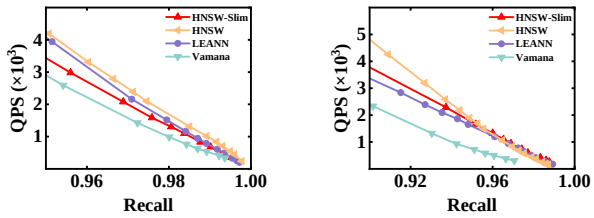
I SCALABILITY EVALUATIONS

We evaluate HNSW-Slim on two large-scale 100M datasets, i.e., DEEP-100M and MSTuring [17], to evaluate the scalability. Following Section 7.1.5, all methods are *built and queried on the server*.

Table 7: Indexing performance on large-scale datasets.

Dataset	DEEP-100M		MSTuring	
Algorithm	**Size (GB)**	**Time (min)**	**Size (GB)**	**Time (min)**
HNSW-Slim	**5.9**	53.6+**2.3**	**5.4**	66.1+**2.5**
LEANN	12.5	53.6+57.3	12.2	66.1+57.8
NSG	—	—	—	—
NSSG	—	—	—	—
Vamana	11.4	**43.6**	11.7	**38.0**
HNSW	28.1	53.6	28.1	66.1

Table 7 shows the indexing performance on large-scale datasets. Note that the results of NSG and NSSG are omitted, as they are out of memory during graph construction on large datasets. HNSW-Slim achieves the smallest index size on both 100M datasets and reduces memory consumption by 3.8×–4.2× compared to HNSW and 93%–151% compared to other methods. It also has the shortest pruning time: only 3.7%-4.2% of the HNSW construction time. Although LEANN can prune HNSW, its pruning time exceeds the construction time. NSG and NSSG fail to build due to excessive peak memory usage during construction. Vamana has the shortest indexing time, but its index size is double that of HNSW-Slim.



(a) DEEP (100M) (b) MSTuring (100M)

Figure 14: Search performance on large-scale datasets.

Figure 14 shows the search performance on large-scale datasets. Since all methods run on the server rather than the client, different from our previous experimental results, HNSW delivers the highest throughput (without the data transfer delay between server and client). HNSW-Slim and LEANN are very close and consistently outperform the remaining graph baselines, while HNSW-Slim maintains clear advantages in pruning time and index size. Note that HNSW-Slim is designed for resource-limited devices; within the same memory, it can index more data.