Efficient Vector Index Merging in Vector Databases

CHENZHE JIN, Purdue University, USA
YUNAN ZHANG, Purdue University, USA
JIAYI LIU, Purdue University, USA
JIANGUO WANG, Purdue University, USA

Vector databases have become a cornerstone of modern data science and AI applications, powering recommendation systems, semantic search, retrieval-augmented generation, and more. This paper focuses on vector index merging (particularly HNSW merging), which merges two (or more) vector indexes. This is a key operation in vector databases with many use cases in vector index construction and vector index updates. While there are a few early approaches to solve the problem, the index merging performance remains slow. In this work, we propose HNSW-Merger, a new algorithm for merging two (or more) HNSW indexes that fully exploits the proximity information in existing indexes. It is a novel two-stage, search-based algorithm that relies on forward HNSW search and lazy backward direct-connect to efficiently connect potential edges. HNSW-Merger is optimized for multi-core parallelism and memory efficiency. It also supports efficient merging of multiple indexes. Extensive experiments show that HNSW-Merger achieves significantly faster merging performance than prior approaches while maintaining similar or even higher index quality.

CCS Concepts: • **Information systems → Data management systems**.

Additional Key Words and Phrases: Vector Databases, Vector Index Merging, HNSW Index

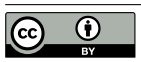
ACM Reference Format:
Chenzhe Jin, Yunan Zhang, Jiayi Liu, and Jianguo Wang. 2026. Efficient Vector Index Merging in Vector Databases. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 31 (February 2026), 26 pages. https://doi.org/10.1145/3786645

1 Introduction

In today's data-driven world, the demand for scalable vector databases has grown rapidly [31, 37]. Vector databases are used in numerous data science and AI applications, including recommendation systems, semantic search engines, retrieval-augmented generation (RAG), natural language processing systems, and even bioinformatics [23, 26, 31]. This surge in demand has been further amplified by the recent explosive growth of large language models (LLMs), such as ChatGPT, which rely on vector databases to address challenges such as hallucination and the lack of real-time or domain-specific information [31, 34, 44].

At the core of vector databases is the ability to perform fast vector searches over large-scale datasets through efficient vector indexing techniques. Among the various types of vector indexes, HNSW (Hierarchical Navigable Small World) [27] is one of the most popular and is widely adopted by major vector databases, including Pinecone [12], Milvus [38], Weaviate [13], Oracle Vector

Authors' Contact Information: Chenzhe Jin, Purdue University, West Lafayette, USA, jin467@purdue.edu; Yunan Zhang, Purdue University, West Lafayette, USA, zhan4404@purdue.edu; Jiayi Liu, Purdue University, West Lafayette, USA, liu4127@purdue.edu; Jianguo Wang, Purdue University, West Lafayette, USA, csjgwang@purdue.edu.



ACM 2836-6573/2026/2-ART31
https://doi.org/10.1145/3786645

Search [10], Azure SQL [41], PostgreSQL pgvector [11], PostgreSQL-V [25], AlloyDB [1], SingleStore-V [18], and Elasticsearch [6]. HNSW achieves both high performance and high accuracy in large-scale vector search by organizing vectors into a multi-layer proximity graph and leveraging small-world properties to guide search traversal.

Current vector databases mainly focus on optimizing vector search performance, but a critical yet overlooked issue is *vector index merging*, which combines two or more vector indexes into a single one. Prior studies [14, 15, 32] have shown that performing vector search on a merged index can significantly improve performance and recall, due to the sublinear nature of vector search and the optimized index layout.[1]

We highlight two important use cases of vector index merging.

Use Case #1: Vector Index Construction. Building a vector index for a large-scale dataset can be very time-consuming. For example, constructing a billion-scale HNSW index may take days or even weeks [28, 33]. A potential solution is to partition the dataset across multiple threads or machines, with each independently building a sub-index in parallel, followed by merging these sub-indexes into a single index [15, 18, 32, 43]. Thus, vector index merging plays an important role in enabling efficient large-scale index construction.

Use Case #2: Vector Index Updates. Updating vector indexes directly (e.g., when new vectors arrive) is challenging in vector databases. Thus, many systems (e.g., Milvus [38], SingleStore-V [18], AnalyticDB-V [40], BlendHouse [29] and Elasticsearch [15]) adopt a segment-based architecture combined with an LSM-style out-of-place update strategy to balance write performance and query efficiency. Specifically, incoming vector data is stored in a new segment. Once the segment exceeds a predefined threshold, it is marked as immutable, and a new index is built for it. A background thread periodically merges smaller segments into larger ones, during which their corresponding vector indexes are also merged.

Existing Work and Limitations. In the literature, a few early works have explored merging HNSW indexes, including Elasticsearch [15], SingleStore-V [18], Milvus [38], and Ponomarenko [32]. These approaches can be broadly classified into three categories. (1) *Rebuild-based*: This is the most straightforward approach, which rebuilds a new index from scratch using the vectors from the source indexes. For example, Milvus [38] adopts this strategy. (2) *Insert-based*: This method selects one existing index as the target and incrementally inserts vectors from the other index into it to produce the merged index. Examples include SingleStore-V [18] and Elasticsearch [15]. Notably, Elasticsearch employs an optimized version of this strategy by first selecting a subset of vectors from the smaller index for direct insertion into the larger one, followed by a constrained beam search to insert the remaining vectors. (3) *Search-based*: Ponomarenko [32] presents three search-based merging strategies (NGM, IGTM, and CGTM) that perform a two-way HNSW search process to connect potential edges. The details of these approaches can be found in Section 9.

However, all of the above approaches fail to fully leverage the proximity information in existing HNSW indexes, resulting in either poor merging performance or low quality.[2]

Overview of This Work. In this paper, we present HNSW-Merger, a novel algorithm for merging two (or more) HNSW indexes that achieves significantly faster merging performance than prior approaches while maintaining similar or even higher index quality.

[1] One might argue that multiple threads can search separate indexes, but the same level of parallelism is achievable when searching the merged index [14, 15].

[2] The quality of an HNSW index is defined by its vector search performance and recall rate, which together measure how well the index supports vector search.

At the core of HNSW-Merger is a novel two-stage search-based algorithm that first performs a forward search to identify necessary forward edges, followed by a backward direct-connect mechanism that smartly adds backward edges without performing an explicit backward search. Unlike the methods in [32], HNSW-Merger eliminates backward search and also develops an optimized forward search strategy. HNSW-Merger supports efficient parallelism by leveraging the out-of-place nature of the merge policy to significantly reduce synchronization overhead, allowing each thread to run independently by searching immutable graphs and connecting edges. HNSW-Merger is optimized for memory efficiency and does not require all indexes to fit into memory during the merging process. It loads only necessary working sets into memory and writes back intermediate results to disk during the merge process. Lastly, HNSW-Merger supports efficient merging of multiple indexes via a sequence of pairwise merge operations. It develops an optimal ordering for merging multiple indexes to improve performance.

In this work, we focus on merging the HNSW index due to its widespread adoption in modern vector databases. However, we believe that many of the designs can be applied to other graph-based indexes such as Vamana [35] and NSG [19]. For instance, the forward search and backward direct-connect mechanism, out-of-place merge policy, parallel execution strategy, and memory-efficient design can all be extended to these index structures (see Section 7.1).

Experimental Overview. We compare HNSW-Merger against prior approaches for merging HNSW indexes using the following widely used datasets (SIFT10M, SIFT100M, SIFT1B, Deep10M, Turing10M, GloVe25, and Cohere10M) under default parameters. (1) Compared with the rebuild method, HNSW-Merger achieves 9.6∼11.5× faster merge time with 90.1%∼115.4% of the index quality. (2) Compared to the Elasticsearch approach, HNSW-Merger achieves 2.2∼4.8× faster merge time with 90.8%∼116.8% of the index quality. (3) Compared to the insert-based approach, HNSW-Merger achieves 5.6∼6.6× faster merge time with 90.1%∼115.4% of the index quality. (4) Compared to the three algorithms (NGM, IGTM, and CGTM) in [32], HNSW-Merger achieves 4.3∼9.4× faster merge time with 90.6%∼166.3% of the index quality.

Contributions. The main contribution of this work is HNSW-Merger, a novel algorithm that efficiently merges HNSW indexes in vector databases. Compared to prior solutions, HNSW-Merger dramatically improves index merging performance while maintaining similar or even higher index quality.

Open-Source. https://github.com/Kimchuls/HNSWMerger.

2 Background

2.1 Vector Search

Vector search has been explored comprehensively in the past [16, 20–22, 24, 30, 34–36, 39, 42] as a fundamental task that retrieves top-k data points nearest to a given query under a specific distance metric. To support vector search, many indexes are proposed. Among these, HNSW [27] stands out for its multi-layer small-world graph topology to achieve logarithmic search complexity, offering an exceptional balance of accuracy and query speed [2, 19].

2.2 HNSW Index

Hierarchical Navigable Small World (HNSW) [27] is a graph-based vector index that enables efficient vector search in high-dimensional space. For the index construction in HNSW, when a new vector point is inserted into the index, the algorithm begins by randomly selecting an integer l to determine the highest level at which this point will be inserted. The insertion procedure then proceeds top-down, performing greedy searches at each level using Algorithm 1 to locate the top-ef

Algorithm 1: SearchLayer

Input: Query vector p, Graph $G = (V, E)$, entry point e, result list size ef
Output: Set of ef approximate nearest neighbors of p

```
top ← max-heap of visited points;
candidates ← min-heap of visited points;
visited ← {e};
d ← distance(p, e);
top.insert(e, d), candidates.insert(e, d);
while candidates is not empty do
    v ← candidates.pop();
    d_max ← top.max_distance();
    if d_max < distance(v, p) then
        break;
    foreach neighbor q of v in G do
        if q ∉ visited then
            visited ← visited ∪ {q};
            d_q ← distance(p, q);
            if |top| < ef or d_q < d_max then
                top.insert(q, d_q);
                candidates.insert(q, d_q);
                if |top| > ef then
                    top.pop();
                d_max ← top.max_distance();
return sorted top;
```

nearest neighbors. At each layer, the closest neighbor from the current layer is selected as the entry point for the next lower level, and the process continues recursively. For all layers above l, the algorithm sets $ef = 1$ and performs no edge insertions – only identifying entry points for the next layer. Once the traversal reaches level l or below, the search parameter ef is raised to efc, which is a parameter indicating the size of the candidate neighbor list maintained during the greedy search when inserting new vectors. This enables the algorithm to collect sufficient candidates for the new vector to establish connections. To ensure sparsity and structural diversity, a pruning procedure (Algorithm 2) is applied, which selects at most M well-separated neighbors from the candidate set. The pruning criterion, governed by the threshold α (default 1.0), filters out redundant or overly similar neighbors to preserve the navigability of the index. After pruning selects up to M neighbors for the new vector, each of those neighbor points inserts the new vector into its own neighbor list and prunes as necessary, ensuring bidirectional connections.

When performing HNSW search, as illustrated in Algorithm 3, the procedure consists of two different phases. In the first phase, the search traverses each layer l, starting from the highest layer down to layer 1, identifying a single best entry point at each level to guide the descent. This process yields a final entry point at the bottom layer. In the second phase, the algorithm performs a search operation at the bottom layer to retrieve the top-efs nearest neighbors, where the search parameter ef is explicitly set to efs within the function. After obtaining the candidate set, the top-k nearest

Algorithm 2: RobustPrune

Input: Candidate set Q (max-heap by decreasing distance), degree bound M, distance threshold α
Output: A pruned max-heap containing at most M neighbors

```
begin
    Convert Q to a sorted list in ascending order of distance;
    L ← ∅
    while Q is not empty and |L| < M do
        v ← the closest point in Q;
        Remove v from Q;
        flag ← true;
        for u ∈ L do
            if α · distance(v, u) < distance(v, query) then
                flag ← false;
                break;
        if flag then
            Insert v into L;
    Convert L into a max-heap and return it;
```

Algorithm 3: Search

Input: Query point p, index I, entry point e, highest level l_{high}, lowest level l_{low}, search parameter ef
Output: Top-ef nearest neighbors of p at level l_{low}

```
e_curr ← e;
for l ← l_high to l_low + 1 do
    Let graph G = (V, E) on layer l;
    e_curr ← SearchLayer(p, G, e_curr, 1);
Let graph G = (V, E) on layer l_low;
candidates ← SearchLayer(p, G, e_curr, ef);
return candidates;
```

neighbors of the query point are selected by extracting the k closest elements, depending on the desired number of results.

3 HNSW-Merger

Assume we have two HNSW indexes, I_1 and I_2. Our objective is to construct a merged index, denoted as I_m, based on I_1 and I_2. Since an HNSW index is a hierarchical graph with multiple layers, at a given layer l, let the corresponding graphs be $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, where $V_1 \cap V_2 = \emptyset$. Let $G_m = (V_m, E_m)$ be the merged graph at layer l. The point set V_m at this layer is defined as the union of the points from the two input layers: $V_m = V_1 \cup V_2$.

Goal: Fast Index Merging and High Index Quality. Our goal is to efficiently merge HNSW indexes while maintaining high index quality – comparable to a freshly rebuilt index. We measure the quality of an HNSW by its search performance and recall rate.

Opportunity. We observe that each input HNSW index encodes rich proximity information through its graph connectivity, offering a new opportunity to accelerate index merging. These pre-established connections are not only easily accessible but also semantically meaningful: if two points are connected in the source index, they are likely to remain close neighbors (though not necessarily the closest) in the merged index. Unlike rebuild-based methods that discard such structure and start from scratch, we can fully leverage the existing graph topology. This can reduce redundant distance computations, speed up the merging process, and preserve local neighborhoods that are crucial for maintaining search quality.

3.1 Main Idea of HNSW-Merger

The high-level idea of HNSW-Merger is to merge the two indexes, I_1 and I_2, layer by layer, due to the multi-layer nature of HNSW. If one index has more layers than the other, the extra layers are directly copied to the merged index to avoid redundant computation. The key challenge, however, lies in how to merge two graphs on the same layer, i.e., $G_1(V_1, E_1)$ from I_1 and $G_2(E_2, V_2)$ from I_2.

A straightforward approach is to perform a **two-stage forward-backward search**, in which, at each stage, each point from one graph searches the other graph to identify potential neighbors for connection. **Stage 1: Forward Search**. For each point $p \in V_1$, one can perform a forward HNSW search on G_2 to retrieve its top-efc nearest neighbors in V_2 as we have shown in Algorithm 1. Here, efc is the construction parameter used in HNSW that indicates the number of candidate neighbors retrieved. These candidates, together with p's existing neighbors in G_1, form a candidate set for p's neighbors in the merged graph. This set is then pruned using HNSW's construction rules to retain at most M connections (as shown in Algorithm 2), where M is the maximum number of edges a point is allowed to establish in the graph at each layer of the index. The resulting pruned neighbors become p's neighbor list in the merged index. **Stage 2: Backward Search**. Similar to the forward search, we can perform a backward HNSW search for each point $q \in V_2$, which queries G_1 for neighbors.

Although the above approach works, HNSW-Merger identifies *two new opportunities* that can significantly improve index merging performance while maintaining similar index quality. (1) We can eliminate backward search and rely solely on forward search, effectively reducing the number of vector searches by half. (2) Even for forward search, we observe that we only need to retrieve far fewer neighbors to yield high index quality.

Observation 1: No Backward Search. We observe that we only need to perform forward search and add backward edges based on the information obtained during the forward search. Our intuition is that, if a point $p \in V_1$ finds one of the nearest neighbors $q \in V_2$ via a forward HNSW search, then p is likely to be one of the nearest neighbors of q as well, or one of the nearest neighbors of q can be reachable quickly via a few hops from p's neighbors – for example, usually one or two hops as shown in Section 3.2. Thus, we can directly add a backward edge from q to p. We show the verification of the intuition in Section 3.2.

However, a key research question is deciding *when* to add the backward edges. A straightforward solution is the eager connect approach: whenever a point $p \in V_1$ discovers a neighbor $q \in V_2$ through an HNSW search on G_2, a backward edge from q to p is added immediately. However, this eager connect strategy can result in multiple invocations of the pruning function on the same point q and repeated updates to q's neighbor list in the merged index. These redundant operations incur substantial computational cost. In contrast, HNSW-Merger uses a **lazy connect approach** that

defers backward connections and batch-processes all source points of incoming edges. Specifically, if a point $q \in V_2$ appears in the nearest neighbors of R ($R \geq 1$) points during forward search, all such points, together with q's original neighbors in G_2, are treated as candidates for q's neighbor list in the merged graph G_m. A pruning step is then applied to this union to enforce the maximum out-degree constraint M. This design leverages incoming connections from V_1 to q, using their sources as approximate indicators of q's relevant cross-index neighbors. As a result, our approach significantly reduces redundant distance computations while preserving the structural integrity of the merged graph.

Observation 2: Forward Search with Fewer Edges. Another important factor is the number of neighbors (denoted as λ) that are retrieved during forward search. A straightforward method is to set λ equal to the default construction parameter efc used in HNSW. However, we find that this value is often too large in practice, since only the top-M neighbors (with $M < efc$) are ultimately retained during the pruning process. While retrieving a larger set of neighbors (i.e., larger λ) during search can improve the accuracy of the merged index, it substantially increases the merging cost and enlarges the candidate pool. This, in turn, increases pruning overhead, as only the top-M neighbors are preserved.

Rather using the parameter efc, we find that a much smaller parameter λ is sufficient to collect the nearest neighbors for each vector in G_1 and G_2, while still maintaining strong navigability in the merged index. This is because an indirect path can be found from one point to its neighbor in the vector space.

The downside of using a small λ is that it might slightly affect the index quality. This is because the resulting neighbor list in the merged index may omit some of p's true nearest neighbors across the full dataset, thereby requiring more hops to reach high-quality results. As a result, vector search latency can increase due to longer traversal paths and additional distance computations. However, we believe that such a tradeoff is worthwhile because our experiments show that this choice speeds up merge time significantly while the vector search performance degrades by less than 10% (Section 8.2). In the experiment (Section 8.7), we investigate how varying the value of λ can affect merge performance and index quality.

Therefore, HNSW-Merger adopts a **forward search, backward direct-connect** approach to merge two layers. **Stage 1: Forward Search with Fewer Edges.** In the first stage, for each point $p \in V_1$, we perform a forward HNSW search on G_2 to retrieve its top-λ nearest neighbors. Once the search completes, we compute p's neighbor list in the merged index as follows: merge these λ neighbors with p's original neighbor list from G_1, and call HNSW's prune function (Algorithm 2) to keep at most M connections. These M neighbors will be p's neighbor list in the merged index and will be written to the merged index. Moreover, we also need to prepare the necessary information for the second stage in order to compute q's neighbor list (i.e., the *lazy connect approach*): for each neighbor q in the top-λ list, we insert p into q's candidate neighbor list. **Stage 2: Backward Direct-Connect (No Backward Search).** In the second phase, for each point $q \in V_2$, we skip the backward search and instead use the candidates recorded during the forward stage. Specifically, for each vector $q \in V_2$, we combine the neighbor list recorded during the forward stage with the original neighbor list from G_2. Then, we call HNSW's prune function to select up to M connections and write them into the merged index.

Example. Figure 1 shows an example of merging two graphs. Figure 1a represents G_1 with blue points, and Figure 1b represents G_2 with green points. For point g in G_1 showing in 1a, the original neighbors are points d, f, and h, connect with red solid lines in the figure. By performing a forward HNSW search in G_2, we find three additional neighbors for g: k, p, and q, connect with red dotted lines in the figure 1a. These six points form the candidate neighbor set for g in the merged graph.

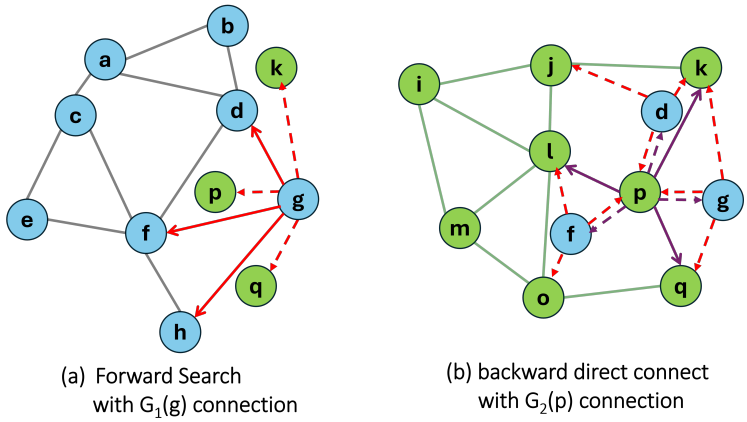


Fig. 1. Example of Forward Search and Backward Direct-connect in HNSW-Merger

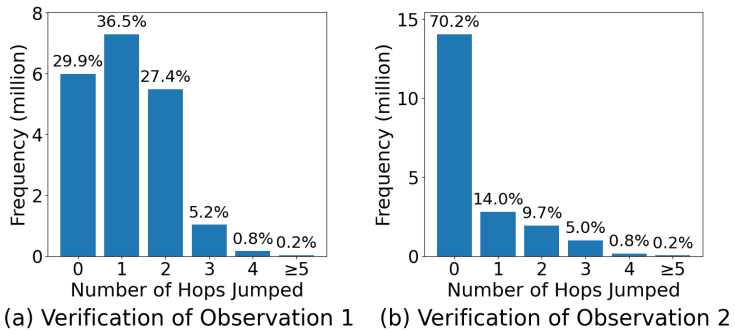


Fig. 2. Verification of Observations

Conversely, points k, p, and q also treat g as a candidate neighbor in the merged index. Specifically, consider point p in Figure 1b. During a forward HNSW search for points in G_1, points d, f, and g identify p as one of their nearest neighbors. As a result, during the backward direct-connect stage, p collects d, f, and g (connected with purple dotted line), along with its original neighbors k, l, and q (connected with purple solid line), as candidate neighbors in the merged graph.

3.2 Verification of Observations

To validate our two observations, we design two experiments. We focus on vector search in the bottom layer of HNSW, which contains all data points and dominates search performance. Using the SIFT10M dataset [5], we split the 10 million vectors evenly into two subsets of 5 million each and build two independent HNSW indexes (I_1 and I_2; parameters in default). Let $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ be the bottom layers of I_1 and I_2.

For the first experiment, we want to show that forward search alone still preserves high index quality. We design an experiment to measure how often true nearest neighbors can be reached in just a few graph hops using only our backward direct-connect step, without any explicit backward search. We set λ as 4. For every $p \in G_1$, we retrieve its top-4 neighbors in G_2 to form Q_p. Then for each $q \in Q_p$, we perform a backward HNSW search in G_1 to obtain its top-4 neighbor set Q_q. We record, for each pair (p, q), the number of graph hops from p to any $p' \in Q_q$, and summarize these hop-count statistics in Figure 2a.

As shown, in the total 20 million (p, q)-pair cases, for 93.8 % of them, the graph-distance from p to some member of Q_q is at most two hops (and over 99 % within three hops). This tight locality indicates that for the vast majority of points $q \in V_2$, the corresponding proxy $p \in V_1$ lies extremely close to q's true neighbor set. Such proximity is largely due to the transitive nature of HNSW's connectivity – many disconnected neighbors can be reached via indirect paths even under a limited search budget. For example, if $p_1 \in Q_q \subset V_1$ is one of p's retained neighbors, then the path $q \to p \to p_1$ suffices to connect q with one of its nearest neighbors p_1. This demonstrates that, even when we omit an explicit backward search, our backward direct-connect mechanism effectively achieves the same result: it retrieves accurate or at least very close nearest neighbors by leveraging the candidates recorded during forward search. Hence, we verified the intuition in Observation 1.

To validate Observation 2, we compare forward-search results under different values of λ. We wish to confirm that with a small λ, the search results closely match those obtained with a larger λ, or that indirect connections enable rapid traverse to the larger-λ neighbors from the small-λ results. Specifically, we set $\lambda = 4$ and $\lambda = 10$, and for every $p \in V_1$ perform forward searches on G_2 to obtain the top-4 neighbor list Q_4 and the top-10 neighbor list Q_{10}. For a fair comparison, we select the four closest points from Q_{10} to form Q'_4. Then, for each $q \in Q_4$, we measure how many graph hops are required to reach any point in Q'_4. The results in Figure 2b report the distribution of graph-hop counts required for each (p, q) pair to reach any point in Q'_4.

We can see that, for searching 70.2 % of (p, q) pairs, q is in Q'_4, which indicates that most of our forward search candidates are in a good accuracy. And for 93.9% of the (p, q) pairs, the graph-distance from q to some member of Q'_4 is at most two hops. Only 1.1 % of (p, q) pairs have a bad accuracy that needs to jump more than 3 hops to reach p's true nearest neighbors. This distribution confirms that even with a small λ for HNSW search, we can capture the most important nearest neighbors.

3.3 Merging Policy

In HNSW-Merger, we optimize the merging order and leverage an out-of-place merge strategy to improve performance.

Merging Order. Another factor influencing merge efficiency is the order in which the two indexes are merged. When $|V_1| \leq |V_2|$, performing forward vector searches from the smaller graph into the larger one is beneficial, as the complexity of HNSW search scales logarithmically with the graph size, i.e., $O(\log|V|)$ [7]. Forward searches therefore cost $O(|V_1| \log|V_2|)$, which is lower than the $O(|V_2| \log|V_1|)$ cost incurred by searching from the larger into the smaller graph. To exploit this property, we compare $|V_1|$ and $|V_2|$ and swap the indexes so that I_1 always refers to the smaller index.

Out-of-place Merge Strategy. HNSW-Merger leverages an out-of-place strategy by constructing the merged index in a separate structure, which is different from the in-place merge strategy used in [6, 18]. This design ensures that during the forward search phase, the search graph remains static – its size does not grow with newly added connections – thereby maintaining consistent search efficiency throughout the merging process. Moreover, the decoupling of read and write operations enables an efficient multi-threading design: while one thread performs neighbor search on the input graph, others can simultaneously write results to the merged index without locking. This separation facilitates highly efficient parallelism and reduces synchronization overhead (see Section 4). Moreover, by avoiding in-place modifications, the algorithm achieves better memory efficiency, as only partial data needs to be loaded into memory at a time (see Section 5).

Algorithm 4: HNSW-Merger

Input: Two HNSW indexes I_1 and I_2, merge parameters (M, λ, α)
Output: Merged HNSW index I_{m}

1 **begin**
2 Swap I_1 and I_2 if $|I_1| > |I_2|$;
3 Initialize a new empty HNSW index I_{m} and copy index metadata from inputs;
4 Copy all layers that exist only in one index into I_{m};
5 **foreach** *layer l that exists in both indexes I_1 and I_2* **do**
6 Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be the graphs at layer l of I_1 and I_2;
// Stage 1: Forward search
7 **foreach** $p \in V_1$ **do**
8 $e \leftarrow$ *entry point* of I_2;
9 $l_{\text{high}} \leftarrow$ max level of I_2 // top-λ search;
10 $Q \leftarrow \text{Search}(p, G_2, e, l_{\text{high}}, l, \lambda)$;
11 For each $q \in Q$, insert p into q's candidate neighbor list;
12 $C_p \leftarrow Q \cup \text{Neighbors}(p, E_1)$;
13 **if** $|C_p| > M$ **then**
14 $C_p \leftarrow \text{RobustPrune}(C_p, M, \alpha)$;
15 Set p's neighbor list in I_{m} as C_p;
// Stage 2: Backward Direct-Connect
16 **foreach** $q \in V_2$ **do**
17 // lazy connect;
18 $P_q \leftarrow q$'s candidate neighbor list ;
19 $C_q \leftarrow P_q \cup \text{Neighbors}(q, E_2)$;
20 **if** $|C_q| > M$ **then**
21 $C_q \leftarrow \text{RobustPrune}(C_q, M, \alpha)$;
22 Set q's neighbor list in I_{m} as C_q;
23 **return** I_m;

3.4 Pseudocode

Algorithm 4 presents the full procedure of HNSW-Merger that merges two HNSW indexes. The algorithm initializes index order and a new empty index, and copies all metadata from both inputs (line 2~3). If one index contains layers that the other does not, those exclusive layers are directly reused in the merged index (line 4). It then iterates over each common layer, performing a forward top-λ HNSW search from every point $p \in V_1$ into G_2, retrieving a small candidate set (line 10) and maintaining a backward record is maintained (line 11). Then, p's new candidate neighbors in G_2 and original neighbors in G_1 are merged and pruned if necessary (line 14), and the final neighbor list is written into the merged graph (line 15). In the backward direct-connect stage, each point $q \in V_2$ is revisited to aggregate its candidate neighbors from the original neighbor list and the candidates recorded (line 19). A pruning step is then applied if necessary (line 21), and the resulting neighbors are inserted into the merged index (line 22).

Complexity Analysis. The HNSW-Merger consists of a forward merge stage for all points in I_1 and a backward direct-connect stage for all points in I_2. Let $N_1 = |I_1|$, $N_2 = |I_2|$. According to [27], each

point is assigned $O(1)$ layers on average, so we omit any layer-count factor from our complexity bounds. In the forward search stage, each $p \in I_1$ incurs the following steps: (1) searching through the index I_2 at $O(\log N_2)$ [7, 27], (2) selecting its candidate neighbors at $O(M \cdot \lambda)$ and maintain a priority queue at $O(\lambda \log \lambda)$, (3) pruning edges at $O((\lambda+M) \cdot M)$ with maximum M new edges and maximum $(\lambda+M)$ candidates. Thus the forward merge stage cost is $O\big(N_1 \cdot \big(\log N_2 + M \cdot \lambda + \lambda \log \lambda + \big(\lambda+M\big) \cdot M\big)\big)$.

The backward direct-connect stage examines up to R recorded candidates and maximum M original edges per $q \in I_2$ and prunes backward edges in $O\big((R+M) \cdot M\big)$. Hence the overall cost is $O\big(N_1 \left(\log N_2 + M \cdot \lambda + \lambda \log \lambda + \big(\lambda + M\big) \cdot M\right) + N_2\big(\big(R+M\big) \cdot M\big)\big)$.

We estimate the average number of forward-search connections per point in I_2 as $R \approx N_1\lambda/N_2$. This is because each of the N_1 points in I_1 creates λ forward edges and these edges are distributed over the N_2 points in I_2, yielding an average of $N_1\lambda/N_2$ edges per point. Since M is constant here, the time complexity of merging algorithm can be regarded as $O\big(N_1 \left(\log N_2 + \lambda + \lambda \log \lambda\right)\big)$. If we recognize λ as a constant, the time complexity becomes $O\big(N_1 \cdot \log N_2\big)$.

4 Parallel Merging

In this section, we present how to make HNSW-Merger support multiple threads to improve merging performance. As modern servers are equipped with many CPU cores, our merging algorithm should be easily parallelizable and scalable with the number of cores.

We observe that the *out-of-place* nature of HNSW-Merger (explained in Section 3.3) is highly suitable for the multi-threaded environment. Unlike existing insert-based merging approaches [6, 18], which introduce high synchronization overhead as multiple threads concurrently modify the same index structure in place, HNSW-Merger adopts an out-of-place merge policy that can significantly reduce these synchronization constraints, allowing each thread to run independently by searching immutable graphs and connecting edges (with minimal locking).

Recall that HNSW-Merger has two stages. We now describe how to parallelize each one.

Parallel Forward Search. In the forward search stage, each point $p \in V_1$ is assigned to a thread, which performs a forward HNSW search on G_2 to identify its top-λ nearest neighbors. Because this stage only reads from G_2, there are no conflicts. But multiple threads of different points in V_1 may concurrently discover the same point $q \in V_2$ as a candidate neighbor. If each thread were to immediately update q's neighbor list in the merged index I_m during the forward search stage, this would trigger redundant pruning with incur extra distance computations and write locks. To avoid this, we give each q a lightweight candidate container for storing incoming neighbor IDs. A simple lock on that container ensures safe insertion of each backward-connect candidate, without the overhead of recomputing distances or locking the full neighbor list. The lock is released immediately once recording completes. After pruning p's top-λ nearest neighbors and its original neighbors in G_1, we write its new neighbor list into the merged index, without any read or write conflict on neighbor list between threads because of our out-of-place merge structure. Once a thread finishes processing point p, it fetches another unprocessed point for this thread, and repeats until all forward searches are complete.

Parallel Backward Direct-Connect. In the backward direct-connect stage, we employ a lazy-connect approach to process the candidates recorded during the forward search stage, thereby isolating operations on each point and enabling independent neighbor selection in each thread. Each $q \in V_2$ is assigned to a thread, which retrieves its set of backward-connect candidates and its original neighbor list from G_2. Within each thread, q reads from a read-only data space, which is independent on its own, ensuring no conflicts between different threads. We then combine the neighbor lists, perform a pruning step on the merged list, and write the result into the merged index. As in the forward search stage, our out-of-place merge structure prevents any read or write conflicts

on neighbor lists across threads. This design greatly reduces redundant pruning and eliminates synchronization overhead from concurrent writes, resulting in a more efficient merge.

5 Optimizing for Memory Efficiency

In Section 3 and Section 4, we follow the common assumption used in prior index merging algorithms [15, 18, 32, 38]: both indexes, I_1 and I_2, are fully loaded into memory, and the merged index is also kept in memory during the merge process. However, this approach can lead to excessive memory usage, particularly for large datasets or on servers with limited memory capacity. In this section, we describe how to reduce memory consumption in HNSW-Merger, albeit at the cost of longer merge time.

The high-level idea is to leverage the out-of-place nature of HNSW-Merger to load only minimal working sets into memory and write back intermediate results to disk during the merge process. Since HNSW-Merger merges two indexes layer by layer, we observe that only the active layers (assuming they are graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$) are required to be in memory. To further reduce memory consumption, only one graph needs to be fully loaded into memory. The other graph, as well as the merged index, can be streamed to disk during the merge process.

However, the challenge is how to maintain high index merging performance. An important issue is how to perform forward search efficiently while keeping only active layers in memory. Recall that in HNSW search, a query point p begins at the topmost layer of the index, locates its nearest neighbor in that layer, and then uses that neighbor as the entry point for the next lower layer. Each time point p is queried, the search operation must repeatedly start from the topmost layer. However, if we only keep active layers in memory, the higher layers are not resident in memory.

Caching Traversal Entry Points. To avoid repeatedly loading these layers, we cache the entry points for each $p \in V_1$ in the higher layers of I_2. Let p's highest active layer be l, and let I_2 have $L_{\max}$ layers. During the merge of layers $L_{\max}$ down to $l + 1$, we perform an HNSW search for p at each layer, record the closest point found, and cache it as the entry point for the next lower-layer search. This strategy significantly reduces redundant traversals in our layer-by-layer merging process. By merging each lower layer only after its corresponding upper layer has been fully processed, it can accelerate the overall merge. Importantly, this cache mechanism aligns well with our out-of-place merge design: since we construct the merged index in a separate structure, the content of I_2 remains unchanged throughout, ensuring that entry points remain valid and reusable. This stability avoids re-searching from scratch and further reduces merge latency. Thus, we can bypass the top-layer search and directly retrieve the entry point from the cached array.

Memory-Efficient Forward Search. Recall that in the forward search stage of HNSW-Merger, for each point $p \in V_1$, HNSW-Merger performs a forward HNSW search on G_2. Instead of keeping both G_1 and G_2 in memory, we retain only G_2 in memory[3] and stream the points in G_1 from disk. After retrieving the top-λ neighbors of p in G_2, we load p's neighbors from G_1 (on disk), combine the two sets and prune if necessary. This merged neighbor list is then flushed directly to its designated location on disk, and all in-memory data associated with p is released. This design fully leverages the advantages of the out-of-place merge policy.

To prepare for the backward direct-connect phase, we also need to store candidate neighbor lists. Specifically, let $q \in G_2$ be one of the top-λ neighbors of $p \in G_1$ found through the forward search, then we need to insert p into q's neighbor list queue. To minimize memory consumption, we store only the vector ID of p and fetch its full vector during the pruning process. Once all the points in

[3]Note that it is also possible to store G_1 in memory and G_2 on disk, or even to store both G_1 and G_2 on disk, but the merging performance may be significantly degraded.

G_1 are processed, we can release G_2 and other corresponding memory, retaining only recorded candidate neighbor lists in memory which require minimal space relative to the full graph.

Memory-Efficient Backward Direct-Connect. In this stage, for each vector $q \in G_2$, we load its original neighbor list from G_2 on disk and retrieve its recorded candidate neighbor list. Then, we prune the union of these two lists as the final neighbor list for q and immediately flush it to its designated disk location in the merged index, avoiding loading the full merged index into memory. Once q's merged neighbors are written, we can release the memory space. Note that in this stage, G_2 is not required to be stored in memory.

6 Extension to Merging Multiple Indexes

In this section, we extend HNSW-Merger to support merging multiple HNSW indexes, which is important in certain scenarios. For example, during index construction of a large-scale vector dataset where data is partitioned into multiple shards and indexed independently, or during continuous updates where multiple LSM segments are created and indexed separately.

Our high-level idea is to perform a sequence of pairwise merge operations. Each step takes two indexes as input and produces a new merged index by calling HNSW-Merger. This process is repeated until a single, fully merged index remains. In this way, we can reuse all the optimizations in HNSW-Merger.

The key challenge is the merge *ordering*, since each step needs to decide *which two indexes* to merge, leading to many different options. Lemma 6.1 shows that there are $(2s-3)!!$ unique ways to merge a sequence of s indexes. The notation "!!" denotes the double factorial, and $(2s-3)!!$ is defined as $(2s-3) \times (2s-5) \times \cdots \times 3 \times 1$.

Lemma 6.1. *Assume there are s HNSW indexes $I_1, I_2, \ldots, I_s$, with sizes $N_1 \leq N_2 \leq \cdots \leq N_s$. Then, there are $(2s-3)!!$ unique ways to merge them pairwise until only one index remains.*

Due to space limitations, we omit the full proof. The main idea of the proof is based on a structural induction argument: each merge operation corresponds to inserting a new leaf into a full binary tree, and each insertion into a tree with s leaves yields $2s-1$ valid positions.

Lemma 6.1 highlights the combinatorial complexity of multi-index merging. As s grows, enumerating and evaluating all possible merge orders becomes infeasible. Next, we present a merge ordering that optimizes merge time while maintaining high index quality.

We analyze from the time complexity of HNSW-Merger. From the cost model described in Section 3.4, merging two indexes of sizes $N_i \leq N_j$ takes the following cost:

$$O\left(N_i \left(\log N_j + \lambda + \lambda \log \lambda\right)\right).$$

As the size of merging index increases, to keep the merged index quality, it is desired to increase λ over time. An intuition is to change the λ based on the size of merging index, especially the size of the larger input index, i.e., N_j. In a uniformly distributed dataset, as the search index size grows, a larger fraction of its points falls within a closer neighborhood of the query point. Since the time complexity of HNSW search is $O(\log N)$ [7, 27], where N is the size of the index, the number of index points visited during a query grows proportionally with $\log N$. To match the candidate list size to the number of graph points visited during a query, we therefore set $\lambda = O(\log N)$. More details can be found in Section 7.2. Empirical evaluation shows that this choice yields excellent index quality for the multi-way merge. Therefore, our HNSW-Merger time complexity simplifies to:

$$O\left(N_i \log N_j \left(1 + \log \log N_j\right)\right).$$

Next, we show in Theorem 6.2 that large-first ordering is optimal.

Table 1. Statistics of Datasets

Dataset	# Dimensions	# Vectors	# Queries
GloVe25 [2]	25	1,183,514	10,000
SIFT10M [5]	128	10,000,000	10,000
Deep10M [17]	96	10,000,000	10,000
Turing10M [3]	100	10,000,000	10,000
Cohere10M [4]	768	10,000,000	1,000
SIFT100M [5]	128	100,000,000	10,000
SIFT1B [5]	128	1,000,000,000	10,000

Theorem 6.2. *Assume there are s HNSW indexes $I_1, I_2, \ldots, I_s$, with sizes $N_1 \leq N_2 \leq \cdots \leq N_s$. We assume the forward search parameter scales as $\lambda = O(\log N)$, where N is the size of the larger index during two-index merge.*[4] *Then, the optimal merge ordering that minimizes total merge time in a sequence of pairwise merges is to repeatedly merge the* ***two largest indexes*** *at each step.*

We can prove Theorem 6.2 by mathematical induction. Due to space limitations, we omit the full proof. Our intuitive idea comes from the complexity standpoint: it is far more important to reduce the linear N_i term than to reduce the logarithmic $\log N_j$ term, which makes the large-first order the most efficient. We first establish the base case for $s = 3$, confirming that the greedy strategy yields the minimal merge cost. For the inductive step, we assume the strategy is optimal for any $s = m$ indexes. To prove it remains optimal for $s = m + 1$ indexes, we consider all possible ways of selecting the first pair of indexes to merge. For each such pair, we compute the total merge cost and compare it against the cost incurred by merging the two largest indexes. By showing that merging the two largest yields no greater (and often strictly lower) total cost than any alternative, we validate that the greedy choice maintains optimality as the number of indexes increases. This ensures the global optimality of our merge order across all possible merge trees.

Therefore, we employ a greedy, large-first, pairwise chaining strategy. At each step, we select the two largest indexes by size and merge them using HNSW-Merger's two-index merge procedure. The result replaces the original pair in the pool, and we repeat the process until only one index remains. This approach minimizes the total merge cost by handling the larger indexes earlier and leveraging the merged index in subsequent steps.

7 Discussion

7.1 Extending to Other Graph-based Vector Indexes

Though this work focuses on merging HNSW indexes, we believe that the same idea can be extended to merge Vamana [35] indexes. Specifically, we can treat the Vamana graph as a single-layer HNSW index. When merging two Vamana indexes $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, during the forward search stage, we iterate over each node $p \in V_1$ and identify its λ nearest neighbors in G_2, which serve as candidate neighbors. We then combine these candidates with the existing neighbors of p in G_1 and apply the pruning function to remove redundant edges when necessary. The selected candidate neighbors from the forward search stage are retained for establishing backward connections in the backward-direct connection stage. For each node $p' \in V_2$, we merge its neighbor list with any corresponding candidate neighbors (if available) and prune the list if it exceeds the predefined threshold.

[4]For example, if we merge I_1 and I_2, where I_2 is larger, then N is $|I_2|$.

Table 2. Parameters and Default Values

Parameters	Meaning and Default Value
k	The number of results selected from vector similarity search
	Default Value: 100
M	The maximum degree in HNSW
	Default Value: 32
efc	The queue length in HNSW build
	Default Value: 64
efs	The queue length in HNSW search
	Default Value: 200
λ	The number of nearest neighbors retrieved in the forward search of HNSW-Merger
	Default Value: 4

We can also extend our algorithm to merge NSG [19] graphs in the same way as described above for merging Vamana graphs.

7.2 Guideline of Tuning λ

As HNSW-Merger introduces a parameter λ to indicate the number of neighbors retrieved during the forward search, we next provide guidelines for tuning this parameter. We differentiate between two cases: (1) merging two HNSW indexes; and (2) merging multiple HNSW indexes.

Case 1: Merging two HNSW indexes. We use a fixed λ in this case and suggest setting $\lambda = 4$ in practice. This is because we found that a small λ (e.g., 4) consistently works well across different datasets, including GloVe25 [2], SIFT10M [5], Deep10M [17], Turing10M [3], Cohere10M [4], SIFT100M [5], and SIFT1B [5]. We also studied the impact of λ in Figure 8 (ranging from 1 to 20). The results show that larger λ values yield better merged index quality, but $\lambda = 4$ provides a good balance between merge time and quality.

However, after the index has been serving vector search for some time, if the application requires a higher recall rate, a larger λ can be used. In that case, we suggest keeping λ below 10 (based on the observations in Figure 8), and it should never exceed the M parameter (which is usually 32). Here, M is a build-time parameter of HNSW that specifies the maximum number of edges a point can establish in the graph at each layer of the index.

Case 2: Merging multiple HNSW indexes. We use an adaptive λ in this case because after multiple merges, a fixed λ can lead to accumulated quality loss. We conducted an experiment shown in Figure 9 to demonstrate that using a fixed $\lambda = 4$ to merge 10 HNSW indexes results in a 40% quality loss. We designed an adaptive λ (described below) to determine the value of λ based on the current sizes of the indexes and their initial sizes before applying our algorithm. Figure 9 shows that using an adaptive λ determined by our mechanism can achieve comparable index quality (e.g., 92.3%∼97.3% of the QPS) to the rebuild baseline even after merging 10 times, which demonstrates that our adaptive λ strategy effectively mitigates the performance degradation caused by repeated merges.

We next describe our approach for adaptively setting λ. Assume there are s HNSW indexes to be merged: $I_1, I_2, \ldots, I_s$. We begin by merging I_1 and I_2 using the default $\lambda = 4$, as mentioned above (denoted as λ_0). For subsequent merges, we design a linear function $F(|I_i|, |I_j|, |I_1|, |I_2|)$ to set λ,

where I_i and I_j are the current indexes being merged, and I_1 and I_2 are the initial ones. The intuition is that as more indexes are merged, a larger λ should be used.

To determine this linear function, we observe that the growth of λ correlates with the search complexity of HNSW, which follows a logarithmic trend with respect to index size during the forward search. Therefore, we express $F(|I_i|, |I_j|, |I_1|, |I_2|)$ as $F(\log(\max(|I_i|, |I_j|)), \log(\max(|I_1|, |I_2|)))$, since the forward search always starts from the smaller index toward the larger one (as described in Section 3). To determine the slope of the linear function, we use two boundary points, with intermediate values obtained via interpolation. The first point is (λ_0, N_0), where $N_0 = \max(|I_1|, |I_2|)$, representing the initial merge. The second point is $(\lambda_{\max}, N_{\max})$, where $\lambda_{\max}$ represents the maximum value (i.e., M), and $N_{\max} = \lambda_{\max} \times N_0$. The intuition is that if the merged index becomes too large (e.g., larger than $N_{\max}$), we set λ to M. Note that even when λ reaches M, our algorithm still significantly outperforms the rebuild-based approach. Moreover, when λ reaches M, it indicates that the merged indexes have achieved a quality equivalent to a newly built index, which means that λ can be reset to λ_0 for the next merge. we also reset N_0 at this time. With this function, all the intermediate values can be obtained via interpolation. As mentioned earlier, we have conducted an experiment in Figure 9 to verify the effectiveness of this approach.

8 Experiments

8.1 Experiment Setup

We implement our HNSW-Merger based on HNSWlib [8], a widely adopted C++ library for HNSW indexing.

Datasets. We use the following standard datasets commonly used in the area of vector databases: SIFT10M, 100M, and 1B [5]; Deep10M [17]; Turing10M [3]; GloVe25 [2]; and Cohere10M [4]. Three real-world datasets are involved. Deep10M is an image descriptor dataset pretrained on the ImageNet classification task [17]. Turing10M consists of Bing queries encoded to capture similarity of intent in web search queries [3]. Cohere10M [4] contains a preprocessed version of Wikipedia data suitable for semantic search. Table 1 summarizes the data statistics.

Experimental Platform. All 10 M-scale experiments – including baseline comparisons, parallelism and memory-efficiency evaluations, multi-index merging, and ablation studies – are performed on a dual-socket server with 76 physical cores (152 threads) of Intel Xeon Platinum 8368 CPUs at 2.40 GHz and 188 GB DRAM. The system runs Ubuntu 20.04, and all data and indexes reside on local SSDs. For the 100 M- and 1 B-scale scalability tests, we use a separate quad-socket machine with 96 physical cores (192 threads) of Intel Xeon Platinum 8168 CPUs at 2.70 GHz and 3 TB DRAM, also running Ubuntu 20.04, to provide a high-memory environment.

Competitors. We compare HNSW-Merger with six baselines: (1) rebuilding a new index from scratch (used in Milvus [38]), (2) inserting vectors into an existing index (used in SingleStore-V [18]), (3) Elasticsearch's index merge algorithm [15], (4) Naive Graph Merge (NGM) [32], (5) Intra Graph Traversal Merge (IGTM) [32], and (6) Cross Graph Traversal Merge (CGTM) [32]. For Elasticsearch, we reimplement its Java Lucene [9] code[5] based on HNSWlib (C++) to ensure a fair comparison. We also translate Python code[6] of NGM, IGTM, and CGTM [32] into C++ reusing HNSWlib's codebase. We open-sourced our implementation.

Parameters. We adopt the standard HNSW configuration and, unless otherwise noted, use the default values listed in Table 2. These include the maximum number of neighbors per point (M), the sizes of construction priority queues (efc), as well as the merge-specific parameter λ used in

[5]https://github.com/apache/lucene/tree/main/lucene/core/src/java/org/apache/lucene/util/hnsw
[6]https://github.com/aponom84/merging-navigable-graphs

forward search. In the index quality experiment, our efs varies from 100 to 1000 to demonstrate the search performance.

Evaluation Metrics. We evaluate different index merging algorithms using two metrics: index merge time and index quality. Index quality measures how well the merged index supports vector search, including vector search performance (QPS) and recall rate, under a single-threaded environment.

Compiler flags. We compile all algorithms with the `O3` for optimization and the `fopenmp` flag to enable parallel execution.

8.2 Comparison on Different Datasets

In this experiment, we compare our merging algorithm with six baseline methods under single-threaded settings. Experiments are conducted on five datasets (SIFT10M, Deep10M, Turing10M, GloVe25 and Cohere10M). For each run, we evenly split the dataset into two parts, build an index on each part using default construction parameters, and then merge the two indexes using different algorithms.

8.2.1 Merge Time. We first compare merge times across algorithms to evaluate merge performance. Figures 3a, 3b, 3c, 3d, and 3e report merge times on SIFT10M, Deep10M, Turing10M, GloVe25 and Cohere10M, respectively. In all cases, HNSW-Merger delivers the fastest performance, achieving speedups of $4.8 \sim 11.6\times$ over baselines on SIFT10M, $3.5 \sim 9.6\times$ on Deep10M, $4.2 \sim 10.7\times$ on Turing10M, $4.4 \sim 10.4\times$ on GloVe25, and $2.2 \sim 9.2\times$ on Cohere10M.

The superior merge performance of HNSW-Merger comes from two aspects: it reuses the existing neighbor connections in the input indexes to avoid redundant graph construction, and it limits every HNSW search to a fixed-size set of top-λ candidates, greatly reducing unnecessary distance computations and search overhead. Unlike NGM [32], IGTM [32], and CGTM [32] – which at each layer perform two-way nearest-neighbor searches between the two graphs – our HNSW-Merger uses a one-way forward HNSW search only. Moreover, the forward search in HNSW-Merger uses a small search parameter λ, rather than a value near or significantly higher than efc used in those approaches.

8.2.2 Index Quality.

To evaluate index quality, we conducted experiments to evaluate how well the constructed indexes from different approaches support vector search. We plotted the QPS and recall rates achieved under varying efs settings.

Figures 3h, 3i, 3j, 3k, and 3l compare the vector search performance of HNSW-Merger against six baselines on SIFT10M, Deep10M, Turing10M, GloVe25, and Cohere10M datasets. On SIFT10M, HNSW-Merger achieves approximately $92.8\% \sim 94.7\%$ of the rebuild approach's QPS with $11.6\times$ faster merging speed, and $90.8\% \sim 92.9\%$ of the highest-performing NGM baseline with $8.8\times$ faster merge speed. On Deep10M, HNSW-Merger sustains roughly $90.1\% \sim 91.5\%$ of rebuild QPS with $9.6\times$ faster merging speed. On GloVe25, HNSW-Merger sustains roughly $90.0\% \sim 93.0\%$ of rebuild QPS with $10.4\times$ faster merging speed. On Cohere10M, HNSW-Merger sustains roughly $94.8\% \sim 105.8\%$ of Elasticsearch QPS with $2.2\times$ faster merging speed. Interestingly, on Turing10M, HNSW-Merger even slightly outperforms rebuild, achieving $102.1\% \sim 109.2\%$ the QPS as well as $10.7\times$ the merge speed, while matching $90.6\% \sim 93.4\%$ of NGM QPS and $8.6\times$ the merge speed.

The small performance gaps reflect our design choice of a relatively small forward-search parameter λ, which we tune to balance merge time and index quality. As detailed in Section 8.7.3, increasing λ yields higher recall and throughput at the cost of longer merge time. Overall, these

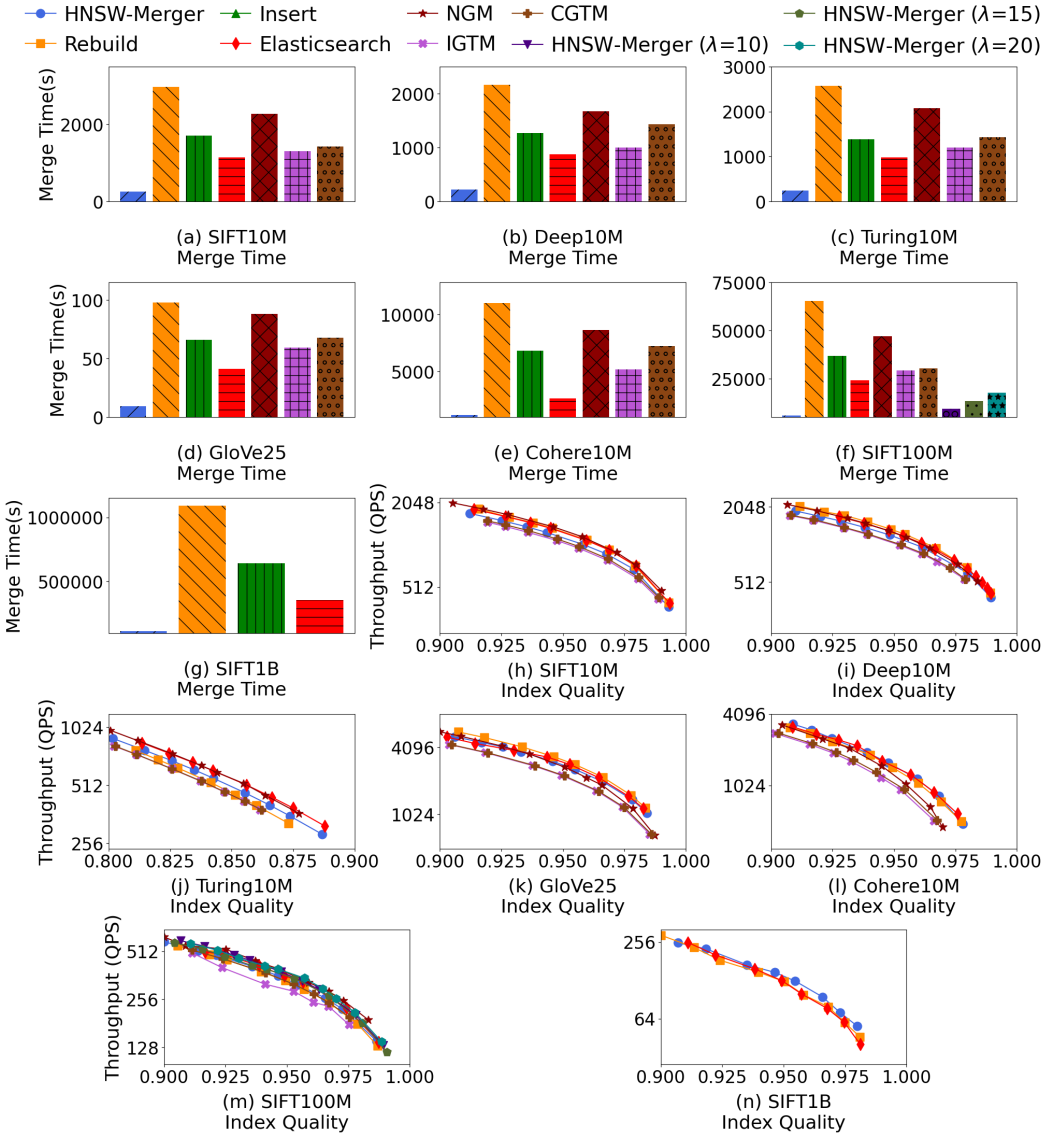


Fig. 3. Comparing Merging Algorithms

results demonstrate that HNSW-Merger delivers search performance on par with rebuilding and other baselines, while offering substantially faster merge times.

8.3 Experiments on Larger Dataset

We further assess our merge algorithm on larger datasets, SIFT100M and SIFT1B, which include 100 million and 1 billion vectors. For each trial, we split the dataset evenly into two subsets, build separate HNSW indexes with the default construction parameters, and then merge them using each merge method. We evaluate merge time and index quality using the same metrics as in the 10M experiments. The results are presented in Figure 3f, 3g, 3m, and 3n.

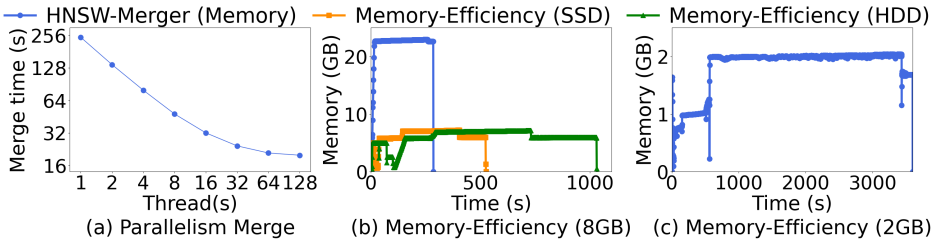


Fig. 4. Evaluating HNSW-Merger on Parallelism and Memory-Efficiency

Figure 3f reports merge times: HNSW-Merger ($\lambda = 4$) completes merging 11.0× faster than rebuild, 3.6× faster than Elasticsearch, and 7.9× faster than NGM. Even HNSW-Merger with $\lambda = 20$ shows 1.4~3.7× speedups compared to other baselines. There are a few reasons for the slight performance gap, as HNSW-Merger is optimized for fast index merging. (1) HNSW-Merger uses one-way search (i.e., forward search only), while ES and NGM use two-way search (i.e., both forward and backward searches) to connect edges. (2) Even for the forward search, HNSW-Merger uses a small λ to select the top nearest neighbors to connect. These two factors might slightly affect the index quality, but the benefit is a significant improvement in index merge performance. We believe that such a tradeoff is worthwhile. Figure 3m compares search performance: HNSW-Merger ($\lambda = 20$) sustains 108.2%~117.5% of the QPS of the rebuild baseline, 102.9%~112.9% of the Elasticsearch approach, and 96.4%~103.6% of the NGM method. These figures show that our merge strategy produces indexes of **comparable or even better search quality** even at large scale.

For the 1-billion-scale experiment, Figure 3g shows that, HNSW-Merger completes merging 9.7× faster than rebuild and 3.1× faster than Elasticsearch. Figure 3n compares search performance: HNSW-Merger sustains 97.1%~115.4% of the QPS of the rebuild baseline and 95.9%~116.8% of the Elasticsearch approach.

This performance pattern mirrors our 10M results, showing that HNSW-Merger consistently achieves substantial speedups without sacrificing index quality, even on large-scale datasets.

8.4 Evaluation of Parallelism

We evaluate the parallel performance of HNSW-Merger on the SIFT10M dataset using 1, 2, 4, 8, 16, 32, 64, and 128 threads. As shown in Figure 4a, HNSW-Merger exhibits excellent parallel efficiency with near-linear speedup.

8.5 Evaluation of Memory-efficient Design

To quantify memory consumption, we track the process's physical DRAM footprint over time for our memory-efficient merge strategy versus a basic all-in-memory approach.

We set the total memory usage (including both RSS and OS page cache) to 8 GB by default and 2 GB for the constrained experiment, using the command `systemd-run -scope -p MemoryMax=xGB`. No temporary files are used.

Figure 4b illustrates the results on 8GB limit working on SSD and HDD. At the beginning, both methods perform an initial loading phase, but the all-in-memory method quickly rises to ~23 GB as it holds both source indexes and the merged index entirely in memory for reads, writes, and computation. This allocation persists throughout merging and is only released after the merged index is flushed to disk. By contrast, our memory-efficient method loads only the active layers of the indexes and constructs the merged index in memory, peaking at ~5 GB before immediately

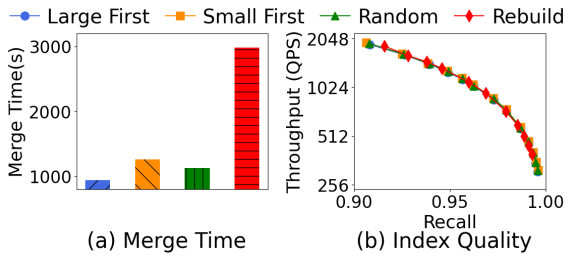


Fig. 5. Comparing Multi-Index Merge Strategies on SIFT10M

releasing most memory once the merged index is initialized. During the merge process, the memory-efficient method focuses on one layer of the graph at a time from disk, loading potentially usable vector data into memory, where memory usage rises to ~7.2 GB. After executing each stage on any layer, the algorithm releases all unnecessary data, causing the RSS usage to drop in the figure. Overall, our optimization reduces peak RSS usage by ~3.3× compared to the all-in-memory baseline, showing its suitability for resource-constrained environments. The total read volume of 30.4 GB and a write volume of 14.7 GB, measured by summing the total number of bytes read or written through `ifstream` and `ofstream`. Although merge time increases due to additional disk I/O, memory-efficient merge strategy on SSD still achieves a 2.2× speedup over the fastest baseline (Elasticsearch), showing that it retains its performance advantage even under memory-constrained settings. Compared with the SSD experiment, the HDD experiment takes longer to merge, as the HDD is slower than SSD. It confirms that our algorithm works on HDD as well.

For the constrained experiment, which limits the memory budget to 2 GB, Figure 4c confirms that our HNSW-Merger still works. The merge time is about 7× longer than when the memory budget is limited to 8 GB due to more frequent disk reads and writes, which is expected. The performance pattern mirrors the previous 8 GB experiment. The peak RSS reaches 2.0 GB, with a read volume of 92.8 GB and a write volume of 14.7 GB. No temporary files were used in this setting either.

8.6 Evaluation of Merging Multiple Indexes

We compare our large-first multi-index merge strategy against three alternative merge orders – random merge, small-first merge (merging the two smallest indexes at each step), and full rebuild from scratch – evaluating both merge time and index quality. We conduct these comparisons on unbalanced datasets, where the indexes to be merged have sizes of 1M, 1M, 1M, 2M, and 5M. We run our experiments under the default configuration.

Figure 5a presents the merge times: large-first achieves a 1.3× speedup over small-first, a 1.2× speedup over random merge, and a 3.2× speedup over full rebuild. Figure 5b shows the QPS-recall trade-off curves for the four methods. These curves are almost overlapping, confirming that our merging method is comparable to other methods in terms of index quality. Overall, this empirical validation confirms our complexity-based proof of the superior efficiency of the large-first merge strategy.

8.7 Ablation Study

To assess the effectiveness of key optimizations, we conduct ablation study on the design choice of one-way search, impact on merging clustered and unbalanced indexes , number of candidate neighbors λ during forward search, and Vamana graph extension. We perform all ablation experiments on the SIFT10M dataset under the default configuration.

8.7.1 Impact on Forward Search and Backward Direct-Connect.

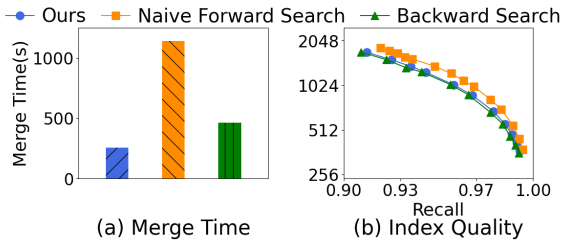


Fig. 6. Comparing Our Approach with Naive Forward Search and Backward Search on SIFT10M

We compare naive forward search with our optimized forward search and evaluate both index quality and merge time. Figure 6 reports that naive forward search achieves 113.5%~118.0% index quality while increasing merge time by approximately 4.4×. We also compare backward direct-connect with backward search and evaluate both index quality and merge time. This figure also reports that backward direct-connect achieves the same index quality while reducing merge time by approximately 1.8×. These two experiments show that our forward-search and backward-direct-connect design is good for merging HNSW indexes.

8.7.2 Impact on Clustered and Unbalanced Index Merging.

Next, we conduct experiments on extreme cases, including merging two indexes with non-overlapping regions and extremely unbalanced indexes.

In the first experiment, we partitioned a dataset into two halves of the same size, V_1 and V_2, using K-means clustering. In this way, V_1 and V_2 are disjoint and non-overlapping regions. Then, we built an index I_1 for V_1 and I_2 for V_2 and merged them. We compared our approach with other baselines under this scenario. Figure 7a and 7c shows the results. It shows that our HNSW-Merger achieves speedups of 5.3~13.7× over the baselines and achieves approximately 95.8%~98.2% of the rebuild approach's QPS and 92.5%~94.7% of the highest-performing Elasticsearch baseline. This confirms that our approach also works on sparse or non-overlapping regions.

In the second experiment on merging highly unbalanced indexes, we split the SIFT10M dataset into two subsets, 9M and 1M, and built indexes based on the two subsets, then merged them together. The results are shown in Figures 7b and 7d. HNSW-Merger achieves 2.7~35.0× speedups compared to other baselines and reaches 91.8%~94.1% of the QPS of the highest-performing rebuild/insert baseline and 93.7%~95.8% of the Elasticsearch approach. This confirms that our approach works in unbalanced scenarios.

8.7.3 Impact on λ Selection. We vary λ over $\{1, 2, \ldots, 10, 15, 20\}$ to investigate how the size of the forward-search parameter influences the QPS-recall trade-off. As λ increases, Figure 8a shows that merge time grows nearly linearly with λ, while Figure 8b demonstrates improvements in the QPS-recall curve. The reason for this is obvious: the larger the λ, the higher the search cost, but the more accurate the selection of neighbors for each point in the merged index. Comparing our HNSW-Merger with $\lambda = 20$ with rebuild from scratch SIFT10M dataset, our algorithm achieves 98.3%~106.6% index quality while 4.5× index merge speed. These confirm that larger λ values enhance index quality at the expense of higher construction cost. Based on this trade-off, we select $\lambda = 4$ as the default in our two-index merge experiments to balance search effectiveness and merge efficiency. Depending on application requirements, users may opt for a larger λ to prioritize index quality or a smaller λ to minimize merge time.

We also evaluate our dynamic λ strategy (described in Section 7) by comparing it with a fixed λ and the rebuild-based approach under the scenario of merging ten 1M indexes. Figure 9 shows

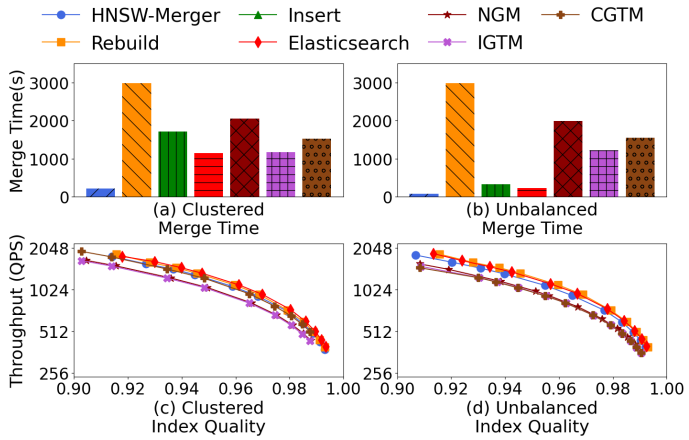


Fig. 7. Comparing Clustered or Unbalanced Indexes on SIFT10M Dataset

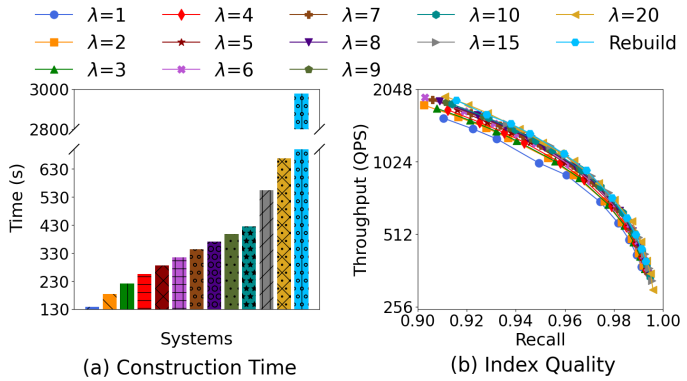


Fig. 8. Impact of λ on SIFT10M

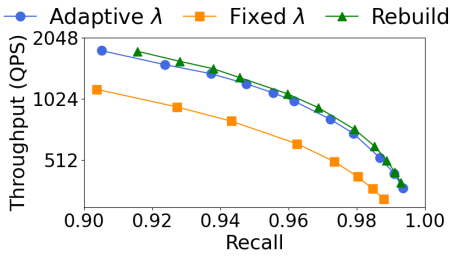


Fig. 9. Comparison Between Fixed and Adaptive λ

the results. We can see that using a fixed λ leads to a 40% quality drift, while using our dynamic λ achieves comparable index quality to the rebuild approach.

8.7.4 Extending to Vamana Graph.

We implement the design and add an experiment to compare the rebuilding approach with our approach for merging two Vamana indexes. Figure 10 shows the results. It shows that our approach is 3.2× faster while achieving comparable index quality.

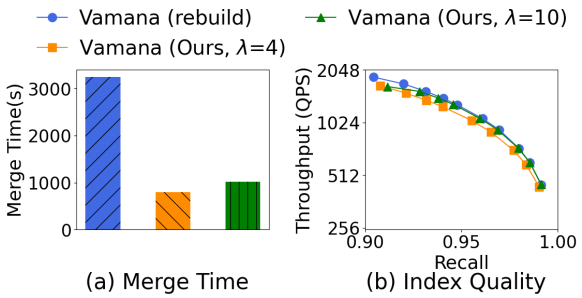


Fig. 10. Extending Our Approach to Vamana Graph

9 Related Work

9.1 Merging Approaches for HNSW Indexes

Rebuild-based Methods. A straightforward approach rebuilds the HNSW index entirely from scratch by merging all source vectors. Milvus [38] applies this strategy during segment compaction, rebuilding each segment's HNSW index from its stored vectors. However, it does not leverage the existing graph proximity information from input indexes that HNSW-Merger heavily uses.

Insert-based Methods. Insert-based merge approaches add points from one index into another using standard HNSW insertion procedure. They do not fully leverage the existing graph edges when handling merges. Systems like SingleStore-V [18] and Elasticsearch [15] employ this strategy. SingleStore-V copies one index and incrementally inserts the other's points [18]. Elasticsearch [15] uses an optimized method. The high-level goal is to avoid running the full, multi-layer HNSW insertion for every point from the smaller graph. It fully inserts only a small set of selected points $J \subseteq G_1$ into G_2 using the standard HNSW insertion procedure across all layers, and uses a more efficient method for the rest. For each remaining point $u \in G_1 \setminus J$, it reuses u's neighbors already merged into G_2 (either from J or by ID order) and these neighbors' connections as entry points for a greedy beam search at the bottom layer, adding edges based on the search results, while applying the normal insertion procedure for higher layers. This reduces the number of expensive full-index insertions while still leveraging existing graph edges to find good connections for the rest of the points. However, computing J can be costly, and the method still performs extensive searches for points not in J because the candidate set for beam search and result list is large, which burden the cost of search during insertion phase. Also, this algorithm does not fully exploit small-world connectivity or the existing edge structure. In contrast, our HNSW-Merger reuses every existing neighbor list in both G_1 and G_2 and performs only small top-λ forward searches – never recalculating all the neighbor lists of points from scratch – thereby cutting all redundant distance computations in HNSW search.

Search-based Methods. Ponomarenko [32] views merging two HNSW indexes G_1 and G_2 as an iterative, layer-by-layer two-way neighbor search, by building efficient connections between indexes to merge indexes. They proposes three algorithms – NGM, IGTM, and CGTM – that differ only in traversal order. In NGM, at each layer a point in G_1 (or G_2) refines its neighbor list by merging its own edges with the results of a full nearest-neighbor search in the other graph, then prunes to satisfy the degree limit. IGTM and CGTM reduce repeated full-graph searches by warm-starting: IGTM picks the next point near the last updated point within the same graph, while CGTM selects the next point globally across both graphs. Despite these optimizations, each method still performs a two-way nearest-neighbor search from every point in one graph to the other at

every layer, leading to heavy search overhead. In contrast, HNSW-Merger conducts forward HNSW search only in G_2 for points in G_1, uses a lazy backward direct-connect to connect from G_2 back to G_1 without any explicit backward searches, and limits each search to the top-λ candidates rather than the full efc set, dramatically cutting traversal and distance-computation costs.

9.2 Other Related Work

Zhao et al. [43] propose algorithms for merging k-NN graphs. However, their approaches are not applicable to merging HNSW graphs due to fundamental differences in graph structure. For example, k-NN graphs require edges to represent the nearest neighbors, whereas HNSW graphs include both long-range and short-range edges that are not necessarily nearest neighbors.

10 Conclusion

In this work, we introduced HNSW-Merger, an out-of-place, two-stage algorithm for merging HNSW indexes that combines a lightweight forward search with a lazy backward direct connect mechanism, achieving significant speedup over prior approaches while maintaining comparable or higher index quality. Building on this core, we developed a parallel merging strategy and a memory-efficient design to improve performance and reduce resources. We also extended our method to multi-index scenario.

Acknowledgments

This work was partially supported by the National Science Foundation under Grant Number 2337806 and by a gift funding from Google through the Google ML and Systems Junior Faculty Award.

References

[1] [n. d.]. AlloyDB AI (https://cloud.google.com/alloydb/ai).
[2] [n. d.]. ANN-Benchmarks (https://ann-benchmarks.com/index.html).
[3] [n. d.]. Billion-Scale Approximate Nearest Neighbor Search Challenge (https://big-ann-benchmarks.com/).
[4] [n. d.]. Cohere (https://huggingface.co/datasets/Cohere/wikipedia-22-12/tree/main/en/).
[5] [n. d.]. Datasets for Approximate Nearest Neighbor Search (http://corpus-texmex.irisa.fr/).
[6] [n. d.]. Elasticsearch Vector Search (https://www.elastic.co/lp/vector-database).
[7] [n. d.]. HNSW Search Complexity (https://milvus.io/ai-quick-reference/what-is-the-typical-time-complexity-of-popular-ann-approximate-nearest-neighbor-search-algorithms-and-how-does-this-complexity-translate-to-practical-search-speed-as-the-dataset-grows).
[8] [n. d.]. HNSWlib (https://github.com/nmslib/hnswlib).
[9] [n. d.]. Lucene (https://github.com/apache/lucene).
[10] [n. d.]. Overview of Oracle AI Vector Search (https://docs.oracle.com/en/database/oracle/oracle-database/23/vecse/overview-ai-vector-search.html).
[11] [n. d.]. pgvector (https://github.com/pgvector/pgvector).
[12] [n. d.]. Pinecone (https://www.pinecone.io/).
[13] [n. d.]. Weaviate (https://weaviate.io/).
[14] 2024. Speeding Up Multi-graph Vector Search (https://www.elastic.co/search-labs/blog/multi-graph-vector-search).
[15] 2025. Speeding Up Merging of HNSW Graphs (https://www.elastic.co/search-labs/blog/hnsw-graphs-speed-up-merging).
[16] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. Elpis: Graph-Based Similarity Search for Scalable Data Science. *Proceedings of the VLDB Endowment (PVLDB)* 16, 6 (2023), 1548–1559.
[17] Artem Babenko and Victor S. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2055–2063.
[18] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proceedings of the VLDB Endowment (PVLDB)* 17, 12 (2024), 3772–3785.
[19] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proceedings of the VLDB Endowment (PVLDB)* 12, 5 (2019), 461–474.

[20] 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. *Proceedings of the ACM on Management of Data (PACMMOD)* 3, 3 (2025), 202:1–202:26.

[21] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *International Conference on Very Large Data Bases (VLDB)*. 518–529.

[22] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 33, 1 (2011), 117–128.

[23] Shiwei Li, Huifeng Guo, Xing Tang, Ruiming Tang, Lu Hou, Ruixuan Li, and Rui Zhang. 2024. Embedding Compression in Recommender Systems: A Survey. *ACM Computing Surveys (CSUR)* 56, 5 (2024), 130:1–130:21.

[24] Yuliang Li, Jianguo Wang, Benjamin S. Pullman, Nuno Bandeira, and Yannis Papakonstantinou. 2019. Index-Based, High-Dimensional, Cosine Threshold Querying with Optimality Guarantees. In *International Conference on Database Theory (ICDT)*. 11:1–11:20.

[25] Jiayi Liu, Yunan Zhang, Chenzhe Jin, Aditya Gupta, Shige Liu, and Jianguo Wang. 2026. Fast Vector Search in PostgreSQL: A Decoupled Approach. In *Conference on Innovative Data Systems Research (CIDR)*.

[26] 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 *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 553–565.

[27] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 42, 4 (2020), 824–836.

[28] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. Parlayann: Scalable and Deterministic Parallel Graph-based Approximate Nearest Neighbor Search Algorithms. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 270–285.

[29] Zhaojie Niu, Xinhui Tian, Xindong Peng, and Xing Chen. 2025. BlendHouse: A Cloud-Native Vector Database System in ByteHouse. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 4332–4345.

[30] James Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 597–604.

[31] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of Vector Database Management Systems. *VLDB Journal* 33, 5 (2024), 1591–1615.

[32] Alexander Ponomarenko. 2025. Three Algorithms for Merging Hierarchical Navigable Small World Graphs. *CoRR* abs/2505.16064 (2025).

[33] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. 2021. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. In *NeurIPS Competitions and Demonstrations Track*. 177–189.

[34] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. 2024. Vexless: A Serverless Vector Data Management System Using Cloud Functions. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 187:1–187:26.

[35] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 13748–13758.

[36] Sairaj Voruganti and M. Tamer Özsu. 2025. MIRAGE-ANNS: Mixed Approach Graph-based Indexing for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data (PACMMOD)* 3, 3 (2025), 188:1–188:27.

[37] Jianguo Wang, Shasank Chavan, Guoliang Li, Yannis Papakonstantinou, and Charles Xie. 2024. Vector Databases: What's Really New and What's Next? *Proceedings of the VLDB Endowment (PVLDB)* 17, 12 (2024), 4505–4506.

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

[39] 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. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 1 (2024), 14:1–14:27.

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

[41] Muazma Zahid. 2023. Vector Search with Azure SQL Database (https://devblogs.microsoft.com/azure-sql/vector-search-with-azure-sql-database/).
[42] Yunan Zhang, Shige Liu, and Jianguo Wang. 2024. Are There Fundamental Limitations in Supporting Vector Data Management in Relational Databases? A Case Study of PostgreSQL. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 2835–2849.
[43] Wan-Lei Zhao, Hui Wang, Peng-Cheng Lin, and Chong-Wah Ngo. 2022. On the Merge of k-NN Graph. *IEEE Transactions on Big Data (TBD)* 8, 6 (2022), 1496–1510.
[44] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2024. Chat2Data: An Interactive Data Analysis System with RAG, Vector Databases and LLMs. *Proceedings of the VLDB Endowment (PVLDB)* 17, 12 (2024), 4481–4484.

Received July 2025; revised October 2025; accepted November 2025