

ANNLlib: A Development Framework for Efficient Approximate Nearest Neighbor Search

Zheqi Shen
UC Riverside
Riverside CA, USA
zshen055@ucr.edu

Jingbo Su
William & Mary
Williamsburg VA, USA
jsu02@wm.edu

Zijin Wan
UC Riverside
Riverside CA, USA
zwan019@ucr.edu

Yan Gu
UC Riverside
Riverside CA, USA
ygu@cs.ucr.edu

Yihan Sun
UC Riverside
Riverside CA, USA
yihans@cs.ucr.edu

Abstract

Approximate Nearest Neighbor Search (ANNS) plays a pivotal role in modern deep learning pipelines. Recently, many ANNS systems have been proposed to provide broad, flexible functionalities or achieve high performance. However, it is inherently difficult to achieve both. We propose **ANNLlib** to address this gap. ANNLlib is a library that provides a programming framework to achieve high performance and flexible functionalities for ANNS systems, based on popular graph-based ANNS algorithms. We carefully decouple and independently optimize both the *algorithm* and the *data structure* components in an ANNS system. In addition, we integrate state-of-the-art algorithms and data structures as modules in ANNLlib, as well as our new designs. Users can choose combinations of components to support sophisticated settings with high performance, such as filtered search, fully dynamic updates, and historical queries on snapshots. Our experiments show that our new solution provides a simple interface for various applications, and achieves comparable or even better performance to previous work specifically for each application.

VLDB Workshop Reference Format:

Zheqi Shen, Jingbo Su, Zijin Wan, Yan Gu, Yihan Sun. ANNLlib: A Development Framework for Efficient ANNS. VLDB 2026 Workshop: The 2nd Workshop on Vector Databases.

VLDB Workshop Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/ucrparyl/ANNLlib-pub>.

1 Introduction

Approximate nearest neighbor search (ANNS) is essential for efficient data retrieval in modern deep learning pipelines. Typically, models embed objects into high-dimensional spaces, casting diverse downstream queries as *k-nearest neighbor searches* (*k-NN search*). Formally, given a dataset $P \subseteq \mathbb{R}^d$ and a query point $q \in \mathbb{R}^d$, a *k-NN search* returns the *k* closest points to q in P under a given distance metric.

Given the inherent difficulty of searching in high dimensions, approximate methods are used for practical efficiency. Over the past decade, ANNS has emerged as a prominent research focus. It has driven the development of many algorithms [16, 24, 37, 42, 46, 47, 54], most notably graph-based approaches, and enabled a

wide range of applications [3, 12, 25, 31, 34, 41, 49, 53, 56, 57, 66, 69]. Graph-based methods index the dataset as a proximity graph, where vertices (data points) are connected to their spatial neighbors, alongside some “long edges” to maintain global connectivity. An ANNS query traverses the graph from an entry point using a *beam search*, greedily moving toward the query vector to identify the *k* nearest neighbors. Graph-based algorithms have become the dominant choice for ANNS systems due to their ability to achieve high recall (the fraction of true *k*-NNs identified) while maintaining efficient throughput (queries per second, or QPS). **In this paper, we study graph-based ANNS systems that combine a broad range of functionalities with strong performance.**

Given its broad applicability, ANNS is widely deployed [21, 65]. Beyond the standard static setting, different scenarios demand specialized *functionalities*. For example, search and recommendation engines are required to handle frequent *updates* to the index [35, 63, 70]; shopping platforms require *filters* over product attributes for item search [28, 30, 59]; analytical workloads may need historical *snapshots* for temporal queries [61]. As a result, a developer must consider *both the choice of algorithm and whether (and how) it can be adapted to different functionalities in the target application.*

Beyond functionalities, the ever-growing scale and dimensionality of modern datasets (e.g., the 1,536-dimensional OpenAI [43] and the billion-scale BIGANN [7] in Table 1) make *performance* increasingly critical for ANNS. This demand applies not only to queries but also to construction, updates, and other functionalities mentioned above, all of which often require *parallelism*. Consequently, a developer must consider *both high-level algorithmic choices and low-level, performance-critical implementation details.*

As ANNS continues to attract attention, research efforts have largely focused on either broadening functionality or boosting performance. Despite significant progress on both fronts, few systems excel at both. Feature-rich vector databases [2, 4, 5, 24, 61] prioritize an easy-to-use interface over efficiency, but underperform specialized algorithms. Conversely, performance-optimized systems [1, 10, 23, 40, 42, 52, 54] consist of code refined over years with sophisticated optimizations, making them hard to adapt to new functionalities, even for experts.

In this paper, we propose **ANNLlib** to address this gap (Figure 1). ANNLlib is a C++ header-only library providing a programming framework to achieve high performance and flexible functionalities for ANNS systems, based on popular graph-based ANNS algorithms. The design goal of ANNLlib is to *ease the development* of an ANN system while *maintaining high performance*. Importantly, ANNLlib encapsulates core algorithmic primitives for ANNS, all natively supporting *high parallelism*. By design, ANNLlib decouples, modularizes, and independently optimizes the *algorithm* and *data*

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment. ISSN 2150-8097.

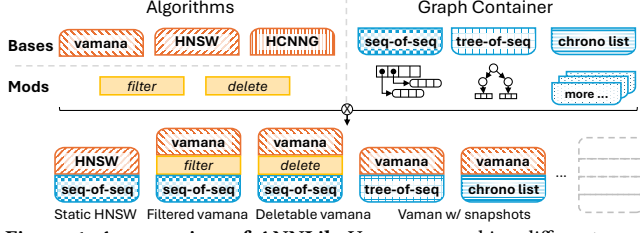


Figure 1: An overview of ANNLib. Users can combine different components, including the base algorithm, algorithm modules and the graph container data structure, as needed by their applications.

structure components of an ANNS system, enabling flexible combinations and extensions based on application requirements. Each component offers several optimized building blocks and provides multi-level interfaces for adequate customization. For algorithms, ANNLib provides basic primitives such as parallel beam search and batch insertions, which are useful in various graph-based ANNS. We include three pre-built graph-based algorithms — *Vamana* [54], HNSW [37] and HCNNG [42]. For data structures, ANNLib abstracts a concise interface for algorithms to interact with the underlying graph. We implemented several pre-built data structures for the graph index in different use scenarios, such as Compressed Sparse Row (CSR) for static settings, and functional trees [19] for streaming and/or historical snapshot settings. We also proposed a new data structure optimized for the features of ANNS graphs with dynamic updates.

With the primitives and data structures in ANNLib, complex ANNS functionalities can be easily constructed through modular combinations. Notably, it allows general algorithmic ideas to be implemented with minimal effort while enabling natural parallelism. To demonstrate ANNLib’s flexibility and effectiveness, we build four applications: (1) the *fundamental, regular ANNS*; (2) *fully dynamic updates* with batch insertions and deletions [52]; (3) *filtered search* by implementing *Stitched-Vamana* and *Filtered-Vamana* [28]; and (4) *streaming updates* with historical snapshots, where we compare several graph containers and their space/time trade-offs. Each needs only minimal code atop ANNLib, showing how easily advanced algorithms and custom data structures can be built.

We evaluate ANNLib’s implementations against existing work specifically optimized for these applications. In most cases, ANNLib achieves more competitive or superior performance (construction time and query throughput) and quality (query recall) compared to existing solutions, while allowing for a much more concise implementation. We note that based on our design, ANNLib can also support more combinations of applications. For example, using the algorithms for filtered search and the data structure for historical snapshots, one can also perform filtered queries on a snapshot of a dynamically changed dataset. We release our code at [6].

2 Preliminaries and Background on ANNS

2.1 Problem Definitions

Given a set $P \subseteq \mathbb{R}^d$ and a query $q \in \mathbb{R}^d$, the k -nearest neighbor search (k -NNS) is to find a subset $K \subseteq P$ of size k s.t.

$$\min_{p \in P \setminus K} d(p, q) \geq \max_{p \in K} d(p, q)$$

where $d(p, q)$ is the distance between p and q in $\mathbb{R}^d$. Since computing exact k -NNS is expensive in high dimensions, *approximate*

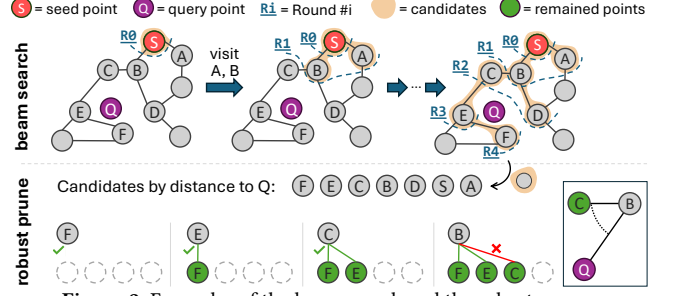


Figure 2: Examples of the beam search and the robust prune.

nearest neighbor search (ANNS) is widely used. We measure ANNS quality using standard $k@k'$ recall, $r_Q = \frac{1}{|Q|} \sum_{q \in Q} \frac{|K \cap K'|}{|K|}$ for the query set Q , where K and K' are the true and retrieved neighbor sets (we use $k = k'$ by default). Efficiency is measured by query *throughput*, defined as $|Q|/t$.

2.2 Graph-based Algorithms for ANNS

ANNLib focuses on efficient and general implementations of graph-based ANNS, where a graph is constructed to facilitate search. Each vertex corresponds to a point in the base set, and edges represent proximity relationships. To keep the graph size under control, the degree of each vertex is capped at r , called the *degree bound*. Both the graph construction and the query algorithms build on a common primitive, *beam search*, discussed below.

Beam Search. A beam search traverses a graph $G = (V, E)$ from a starting point $s \in V$ towards the query point q . As shown in Alg. 1, it maintains a candidate queue C with the capacity of beam width L , which initially only contains s and keeps updating iteratively. At each iteration, the algorithm retrieves the nearest point to q among the candidates in C , and visits its unvisited neighbors (line 3). Each newly visited point u is added to C . If the size of C exceeds L , only the L closest points to q are kept (line 6).

Algorithm 1: BeamSearch(G, s, q, L)

Input: Graph $G = (V, E)$, starting point s , query vertex q , and beam width L

Output: the L closest vertices to q visited during search

```

1  $\mathcal{V} \leftarrow \emptyset; C \leftarrow \{s\}$ 
2 while  $C \setminus \mathcal{V} \neq \emptyset$  do
3    $u \leftarrow \arg \min_{p \in C \setminus \mathcal{V}} d(p, q)$ 
4    $\mathcal{V} \leftarrow \mathcal{V} \cup \{u\}$ 
5    $C \leftarrow C \cup \{v \mid (u, v) \in E\}$ 
6   if  $|C| > L$  then keep  $L$  closest vertices to  $q$ 
7 return  $\mathcal{V}$ 
```

Pruning. During graph construction, newly inserted vertices may add new edges to existing vertices. In this case, a *pruning* algorithm is typically adopted to refine the neighborhoods of vertices. Given a candidate set C with $|C| \geq r$, the pruning algorithm selects r neighbors for a vertex $u \in V$. We present the algorithm in Alg. 2. The algorithm determines whether to prune each point in C in ascending order of distance to u (lines 1–3). Each pruning decision is made by a predicate (line 6), based on the distances to u and to the already-selected neighbors.

A simple strategy [37], known as *re-ranking*, considers only the distance to u and selects the r nearest points. It is typically used to

determine the final answers to a query. In this case, *pred* always returns *false*. Another commonly used heuristic, *robustPrune* [54], optimizes the neighbor distribution to improve the index quality. In *robustPrune*, the current point p^* is pruned if it is closer to any already-selected point p' than to u , i.e., *pred* holds when $d(p^*, p') < d(p^*, u)$. The intuition is that, since p' is preserved and p^* is closer to p' , the search is likely to reach p^* through p' , making p^* redundant. Other predicate variants exist, such as [27, 28].

Algorithm 2: *prune*($u, C, r, pred$)

Input: Point u , candidate set C , degree bound r , predicate *pred*
Output: Selected neighbors N

```

1 sort  $C$  by distance to  $u$  in an ascending order
2  $N \leftarrow \emptyset$ 
3 for  $p^* \in C$  (in sorted order) do
4    $N \leftarrow N \cup \{p^*\}$ 
5   for  $p' \in N$  do
6     if pred( $u, p', p^*$ ) then remove  $p^*$  from  $N$  and break
7   if  $|N| = r$  then break
8 return  $N$ 

```

Figure 2 illustrates this process for determining the neighbors of Q . First, a beam search starting from seed S visits its neighbors A and B , which forms the candidate set. Next, it picks the current nearest point B and visits B 's neighbors C and D , and so on. The beam search ends at round 4 and outputs the candidates sorted by their distances to Q . *robustPrune* then selects the final neighbors from the candidates by degree bound $r = 4$, where F , E , and C are preserved, while B is pruned since it fails the predicate against C .

In the next section, we introduce our techniques, which encapsulate the algorithms reviewed above and new data structures under a unified abstraction, enabling adaptation to more scenarios with minimal effort.

3 The ANNLlib Design

3.1 Overview

ANNLlib is designed to decouple ANNS components into modular, plug-and-play units. However, achieving such decoupling without compromising flexibility or performance is challenging, especially when supporting multi-level customization (for both developers and users) across complex applications.

To address this, we partition ANNLlib into two core components: *algorithms* and *data structures*. The algorithm component includes *base algorithms* for index construction (integrating Vamana [16, 54], HNSW [37], and HCNNG [42], while allowing user extensions) and *algorithm modules* for advanced functionalities (e.g., filtered queries and updates). The *data structure* manages the underlying graph, providing unified interfaces for reading and modifying it.

Listing 1 demonstrates component selection and customization: line 7 instantiates HNSW using specifications in *desc* (e.g., point type, distance metric, and graph container). Users can swap the base algorithm (line 8) or graph container independently without modifying other code.

```

1 struct desc{
2   using point_t = ...;
3   using graph_t = ANN::graph::adj_seq;
4   auto dist_func = ...;
5   ... };

```

```

6 std::vector<point_t> ps = ...;
7 ANN::algo::HNSW<desc> index;
8 // Alternatively, ANN::vamana<desc> index;
9 index.insert(ps); // batch insertion
10 index.search(q, 10); // 10-NN search regarding q

```

Listing 1: Example of invoking the pre-built HNSW algorithm, using the *adj_seq* graph and the direct mapping.

Furthermore, if lower-level logic needs customization, one can partially redefine existing building blocks, as long as they follow the interface definition (see details below).

Listing 2 shows how to support filtered search by customizing the predicate on top of Vamana. On line 2, the original Vamana invokes the regular pruning algorithm during index construction. To develop *Filtered-Vamana*, one can inherit from Vamana and redefine insert to pass a custom predicate that considers labels.

```

1 // in vamana::insert
2 auto nbhs = prune(cands, pred_reg, ...);
3
4 // in filtered_vamana::insert
5 auto pred_filter = [&](...){/*consider labels*/};
6 auto nbhs = prune(cands, pred_filter, ...);

```

Listing 2: Example of implementing filtered search.

3.2 Algorithms: Common Logic Abstraction

The algorithm component is the core of ANNS index construction and query, and where most development effort goes. ANNLlib minimizes this effort by letting users implement only the part unique to their algorithm, while abstracting away the common logic.

```

1 auto collect(f_nbhs, f_dist, eps, r)
2 auto prune(cand, rsize, f_dist, pred)
3 // Insert points along with their neighbors into the graph
4 void insert(auto points){
5   auto new_nbhs[];
6   (parallel-)for(p : points){
7     new_nbhs[p] = comp_nbhs(p, max_deg);
8     // auto cands = collect(p);
9     // new_nbhs[p] = prune(cands, max_deg);
10  }
11  graph.set_edges(new_nbhs); }

```

Listing 3: A typical insertion procedure.

3.2.1 Batch Insertion as an Example of Algorithmic Abstraction. We present the abstract insertion procedure as an example — an important subroutine for various ANNS algorithms [15, 37, 42, 54, 67] that build indexes via incremental insertion. Listing 3 shows a typical insertion workflow, with the key logic highlighted inside the for-loop: determining the neighbors of the newly inserted point p . We also support parallel batch computation (using parallel for-loops), as many applications require it.

Computing a point's neighbors usually takes two steps: collecting candidates (function *collect*, line 8) and selecting final neighbors (function *prune*, line 9). These two steps can be solved by combinations of primitives *collect* and *prune*, introduced in Section 2. For example, HNSW [37] performs beam search to p at each layer to

collect candidates and then applies either a simple sort or a heuristic to prune neighbors. HCNNG [42] uses 3-MST at leaf nodes to compute candidate edges in a single tree, then merges them across trees with pruning strategies. To express all of these in a unified interface, we encapsulate collect and prune and adapt them across multiple ANNS algorithms and variants (detailed in Section 4). Such extensibility is achieved by adhering to the actual data requirements of collect and prune: across algorithms, only distance computation and the neighbor list are needed. We therefore expose them through two parameters, `f_nbhs` and `f_dist`, on the collect and prune interfaces. `f_nbhs(u) → {v : (u, v) ∈ E}` is a callable object that returns the reachable neighbors of u . `f_dist(u): u → d(u, q)` is also callable and returns the distance between u and the query point; an additional overload `f_dist(u, v)` returns the distance between u and v . `pred(p', p*) → bool` accepts a candidate $p^* ∈ \text{cand}$ and a selected neighbor p' , deciding whether p^* should be selected with respect to p' . We now present how these objects are defined for regular search.

```
1 f_nbhs = [g](u){return g.get_edges(u);};
2 f_dist = [g,q](u){return g.dist(u,q);};
3 pred = [(pp,ps){return false;};
```

Listing 4: Examples of `f_nbhs`, `f_dist`, and `pred`

Instead of explicitly passing the query point q as a parameter [1, 38, 54], our design embeds q inside `f_dist` to handle the cases where q is virtual, as in filtered search. By encapsulating g 's information separately in `f_nbhs` and `f_dist`, ANNLib does not require the vertices used for neighbor traversal and for distance computation to come from the same graph, supporting multi-layer algorithms such as HNSW, where only the bottom layer stores coordinates. Besides, the indirection of `f_nbhs` makes it easy to manipulate the neighbor list for extended applications. For example, deletion is supported by filtering out tombstone-marked neighbors on the fly. The remaining parameters for collect and prune include the number of returned edges (degree bound) r , the entry points `eps`, and the candidates `cand`. With ANNLib, developers can reuse this shared logic to implement and optimize multiple algorithms and strategies by simply plugging in these parameters and helper functions.

3.2.2 Base Algorithms and Algorithm Modules. ANNLib supports a variety of *base algorithms* that can be easily plugged in, including built-in *Vamana* [54], HNSW [37] and HCNNG [42]. Many are carefully selected from state-of-the-art solutions with proven efficiency. They can all be combined with additional functionalities, which we call *algorithm modules*, including construction, filtered search, deletions, and snapshots. For example, our construction module builds on the prefix-doubling scheme [38], which calls the insert algorithm in increasing batch sizes, enabling parallel insertion of all points in a race-free manner. We further introduce other modules in Section 4.

3.3 Data Structures: Graph Containers

The graph container lets developers choose the data structure that best fits their scenario. For example, a nested array suits static, insertion-only workloads for its good locality, while a tree-embedded container suits highly dynamic, snapshotted ones. ANNLib supports both behind a uniform interface. Below, we present the

container interfaces and how the cases above use them.

```
1 using vid_t;
2 using struct vertex_ptr;
3 // return an extended-range type
4 agent get_edges(vid_t/vertex_ptr u);
5 void set_edges(seq<(vid, edges)>);
6 // vertex operations
7 vertex_ptr get_vertex(vid_t u);
8 void add_vertices(seq vertices);
9 void remove_vertices(seq vids);
```

Listing 5: Interfaces of a graph container

Listing 5 lists the minimal interfaces to support diverse ANNS algorithms. To enable batch operations, `add_vertices` and `remove_vertices` accept multiple vertices, and `set_edges` accepts a sequence of vertex-edges pairs, each indicating a vertex to update and its associated edges. The interfaces require only basic graph operations by design for ease of use, but the challenge is to support complex use cases efficiently across varying containers. ANNLib addresses this by separating container-specific details from the interface definition and encapsulating them into an *edge agent*.

3.3.1 Edge Agent. The *edge agent* is an intermediate structure that lets each container apply its own graph-specific optimizations. The key idea is to push container-specific details down into the *edge agent*, which each container implements for maximal performance, so that algorithm developers see a uniform, simple interface. The *edge agent* for a vertex u is produced by `get_edges(u)`. It acts as a view over u 's edges while also holding temporary ownership of them during its lifetime. Below, we illustrate the use of *edge agent* with examples that insert back-edges across graph containers.

3.3.2 Back Edge Insertion. Back-edge insertion is a subroutine of batch insertion in many algorithms [38]. Given a vertex u , an algorithm attempts to add a set of vertices es as new neighbors of u . In most cases, this requires jointly evaluating es and the existing neighbors of u , and re-selecting at most r of them as u 's new neighbors, where r is the degree bound. The primary steps are shown in listing 6.

```
1 // For each forward edge (u, v), insert (v, u) to the graph in batch
2 updates = rev_and_semisort(forward_edges)
3 auto edge_agents[];
4 parallel_for(u, es : updates){
5     agent = graph.get_edges(u);
6     old = move_to_seq(agent);
7     agent = prune(old + es);
8     edge_agents = move(agent); }
9 graph.set_edges(edge_agents);
```

Listing 6: The primary steps of back edge insertion.

Here, `updates` is the batch containing vertices and their new edges to be added. On lines 5–7, for each vertex u , the existing edges are retrieved, merged with the new edges, and the final neighbors are written back via the agent. The same code snippet works across both nested-array and tree-embedded graph containers because each provides its own *edge agent* backend.

In the following sections, we introduce the built-in containers in ANNLib, each backed by a different data structure.

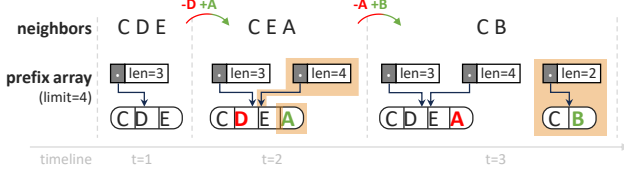


Figure 3: Examples of preserving all changes on a chrono list.

3.3.3 Nested array. A nested array stores each vertex’s edges in an array, and the agent behaves like a reference to that array. During updates, since u will replace its edges with a new array, there is no need to preserve the array referenced by agent; on line 6, it is trivially moved into old because the agent owns it. Moreover, the agent directly forwards the write on line 7, completing the insertion early; consequently, lines 8 and 9 are no-ops. The nested array provides good locality and space efficiency for accessing the graph, but is expensive to update.

3.3.4 Functional Tree. To support snapshottable graphs, we use functional trees. Using functional trees for dynamic graphs with snapshots was first proposed in [20]. We use a follow-up work called CPAM [19]. In general, functional trees use copy-on-write (CoW) upon updates, thus preserving their previous version. A graph is represented by a two-level nested tree. The outer tree, or the *vertex tree*, has each entry as a vertex, associated with an *inner tree*, which stores all its neighbors. Both levels are functional and are snapshottable. In this case, the agent holds the tree’s root plus an internal array to stage updated edges. Since the edge tree nodes can be shared due to CoW, `move_to_seq` on line 6 effectively flattens the tree into an array. The assignment to agent on line 7 writes the final neighbors into the internal array, rather than the tree itself, to avoid immediate mutations that would ripple into the outer vertex tree and cause data races. Agents are then gathered (line 8) and submitted as a batch (line 9) to apply changes top-down. Using CPAM, one can effectively run historical queries on a previous snapshot.

3.3.5 The Chrono Prefix Array. Although CPAM provides a direct solution for historical ANNS queries, it is not always efficient. Since the degree bound is usually small (30–100) relative to the block size (typically 64), each inner tree holds only a few nodes, and at such small size the tree’s structural overhead outweighs its benefits. We design a new data structure for ANNS graphs with historical queries, called the *chrono prefix array*. For each vertex, it maintains a linked list of its edge versions over time, where edges are stored in a space-saving structure, *prefix-shared array*. A *prefix-shared array* compresses chronologically consecutive edge versions together by sharing their prefix. It consists of a reference-counted buffer R and multiple cursors $c_1 \dots c_k$, where k is the number of versions. Each combination of a cursor and the buffer (c_i, R) represents an effective version of edges in $R_{1\dots c_i}$. When new edges E arrive, it computes the increment $M = E \setminus R_{1\dots c_k}$ and appends M to R . A new version $(c_{k+1}=|R|, R)$ is thereby created for the changes of E . Note that $R_{1\dots c_{k+1}}$ may contain more edges than E , since points appearing in $R_{1\dots c_k}$ but not in E are also included. We argue that it is usually safe to contain these extra edges because such deviations are small in practice and extra edges in beam search do not impair the query recall; they just slightly increase the search time. R has a size limit. Once the number of new edges exceeds the limit, a new array is

allocated.

Figure 3 shows an example: a new version appends only the increment (e.g., A at $t=2$) and may retain a harmless extra edge (e.g., D). Once the buffer hits its size limit, a fresh array is allocated (at $t=3$).

4 Applications and Implementations

This section presents four applications built on ANNLlib to demonstrate how developers can easily implement advanced features with minimal modification. These applications are evaluated in Section 5.

4.1 Parallel Batch Insertion on Vamana

We use the parallel batch insertion of *Vamana* as an example of how algorithm-specific logic can be plugged into ANNLlib. For each newly inserted point p , it performs a collect on the existing graph targeting p , then applies *robustPrune* to the search results, using `f_nbhs` at line 2 and `f_dist` at line 5. The back edges of *Vamana* are then inserted following the technique of ParlayANN [38]. The `group_by` operation is customized to a parallel implementation without changing the invocation at line 15, due to the native parallel design. The `nbh_rev[·]` is an array of *edge agents*, inserted in the same form as the forward edges.

```

1 auto get_f_nbhs(){
2   return [&](nid_t u){ return g.get_edges(u); }; }
3 auto get_f_dist(nid_t v){
4   return [&](nid_t u){
5     return dist(g.get_node(u), g.get_node(v)); };
6 }
7 void insert(auto pts){
8   auto nbh_fwd[];
9   foreach(p : pts){
10    auto cands = beamSearch(p, get_f_nbhs(), get_f_dist(p));
11    nbh_fwd[p] = prune(cands, max_deg, get_f_nbhs()); }
12   set_edges(nbh_fwd);
13   auto nbh_rev[];
14   auto edge_rev = {(v,u) | (u,v) in nbh_fwd};
15   auto groups = group_by(edge_rev);
16   foreach([v, es] : groups){
17     nbh_rev[v] = prune(es + get_edges(v), ...); }
18   set_edges(nbh_rev); }

```

Listing 7: *Vamana* Batch Insertion

4.2 Batch Deletion

As mentioned, while most ANNS algorithms naturally support insertion, deletion is more challenging since removed vertices and edges may disconnect the graph. ANNLlib implements the dynamic setting based on the deletion algorithm in FreshDiskANN [52], consisting of two steps: 1) mark deleted points and 2) consolidate marked neighbors with edge repairing. We deem deletion marks as a label and pass in a filter to skip marked points during search. To perform consolidation, we use ANNLlib’s helper to traverse the points in parallel and update neighbors through *edge agents*, reusing the same code path. The full batch deletion algorithm and pseudocode are available in the full version of this paper [50].

4.3 Filtered Search

Filtered ANNS is a widely-used application of constrained retrieval during vector search. Let $\mathcal{F}$ denote the universe of labels.

Given an input set P where each point is associated with *labels* defined by $F : P \rightarrow 2^{\mathcal{F}}$, the filtered ANNS aims to find k -NNs of a query point q with specified labels F_q that satisfy $F_q \subseteq F_u$ for each $u \in P$ in the result. Filtered ANNS is inherently difficult, especially on skewed distributions of labels.

We provide filtered ANNS as a built-in block in ANNLib based on two algorithms in Filtered-DiskANN [28]: *Stitched-Vamana* and *Filtered-Vamana*. *Stitched-Vamana* consists of two steps that: 1) build a regular *Vamana* graph over points with each label; and 2) merge all the sub-graphs using *filtered prune*. By inheriting from *Vamana*, *Stitched-Vamana* easily reuses the insertion implementation on the first step. For the second step, we add a new function `g.merge(h)` for a *Stitched-Vamana* graph g that merges (add all vertices and edges) from a graph h . For vertices existing both in g and h , the edges from two sources are merged by *filtered prune*, which is implemented by a custom `f_nbhs`. We also improve the performance by parallelizing the per-label graph merge process, where the original algorithm is constrained by the bottleneck of sequential merging.

```
1 auto get_f_nbhs(seq labels){
2   return [&](nid_t u, nid_t w){
3     auto es = get_edges(u);
4     auto check = filter([&](auto e){
5       auto (u,v) = e;
6       return u.label ∩ labels ≠ ∅; });
7     return es | check;  }; }
8 void merge(t){
9   for_each(v in t.graph){
10    edges = t.graph.get_edges(v);
11    if(v not in this.graph) new_nbh[v] = move(edges);
12    else
13      new_nbh[v] = prune(
14        edges + this.graph.get_edges(v),
15        get_f_nbhs(this_label)); }
16   ids.merge(t.ids); }
```

Listing 8: Stitched-Vamana Merge

Filtered-Vamana is also based on *Vamana* with a variant of pruning algorithms. The algorithm relaxes the pruning conditions and additionally preserves a candidate v as the final neighbor of the current vertex u if $F_v \cup F_w \subseteq F_u$ where w is a vertex already preserved. This logic is implemented by customizing the `f_nbhs(u, w)` function as shown in listing 9.

```
1 auto get_f_nbhs(seq labels){
2   return [&](nid_t u){
3     auto es = get_edges(u);
4     auto check = filter([&](auto e){
5       auto (u,v) = e;
6       return u.label ∩ labels ≠ ∅; });
7     return es | check;  }; }
```

Listing 9: Filtered Vamana Build

The search for either *Stitched-Vamana* or *Filtered-Vamana* is based on the regular *beam search* by customizing the `f_nbhs` to filter for neighbors with specific labels, as presented in listing 8.

4.4 Snapshots

We develop the snapshot functionality, which saves the whole ANN index for future use. This is achieved by making the graph con-

tainer snapshottable, without changing the algorithms. As demonstrated in listing 10, an ANN snapshot is conceptually a historical copy, where the underlying data movements are managed by space-efficient data structures.

```
1 snapshot1 = (index, tick++);
2 index.insert(points);
3 snapshot2 = (index, tick++);
4 ...
```

Listing 10: Create a snapshot

We use two structures to maintain the snapshots, *chrono prefix array* and *tree of prefix array*. The latter also utilizes *prefix-shared array* to compress edges but employs a PAM tree [55] to manage the vertices, where each vertex is a tree node associated with a *prefix-shared array*. Compared to *chrono prefix array*, the PAM-based structure supports the snapshot branching and concurrent access in a single-writer multiple-reader manner, bringing more functionality at a higher cost. We compare their performance in Section 5.5.

Implementing either structure requires a shim layer for the graph interface, and any prebuilt algorithm such as *Vamana* can be adapted afterwards without modification.

5 Experimental Evaluations

We evaluate ANNLib and show that the flexibility for development comes without compromising performance. Our results show that the performance of ANNLib is generally comparable to or better than that of its monolithic counterparts. ANNLib is implemented as a C++ header-only library and uses ParlayLib [9] as its default parallel framework to support work-stealing parallelism.

We run experiments on a machine equipped with four-way Intel® Xeon® Gold 6252 CPUs at 2.1GHz, and 1.5TiB main memory in 24 channels at 2933MT/s. The billion-scale experiments in Section 5.2 instead ran on a machine equipped with two-way Intel® Xeon® Gold 6438Y CPUs at 2.0GHz and 1TiB main memory. The datasets we used in our experiments are summarized in Table 1. For filtered search, we use two datasets with native labels and two with synthetic ones. *MS MARCO Web Search (MARCO)* [14] builds on ClueWeb22 [44]’s web pages; we derive 10 distinct labels from its semantic annotations. *Yahoo Flickr Creative Commons (YFCC)* [58] comprises Flickr image embeddings; we take the first 10M points with 200,363 distinct labels, following the NeurIPS’23 Big-ANN Competition [51]. For synthetic labels, we draw 10M points each from *BIGANN* and *Cohere* and assign 50 distinct values in a Zipfian distribution ($\alpha = 1$).

Name	Dim.	Size	Metrics	Labels		
				Type	Num.	Label/pt.
BIGANN [7]	128	100M/1B	L2	Synthetic	50	3.39
DEEP [8]	96	100M/1B	L2	—	—	—
Cohere [17]	768	10M	cosine	Synthetic	50	3.39
OpenAI [43]	1,536	5M	cosine	—	—	—
MARCO [14]	768	10M	L2	Native	10	6.13
YFCC [58]	192	10M	L2	Native	200K	10.82

Table 1: Datasets used for experiments. For labels, *Num.* is the number of distinct labels; *Lbl/pt.* average # of labels for each point.

Baselines. We compare ANNLib against *Vamana* and *HNSW* in ParlayANN [38] on construction and batch insertions. ParlayANN is known for fast construction and high query throughput, but does

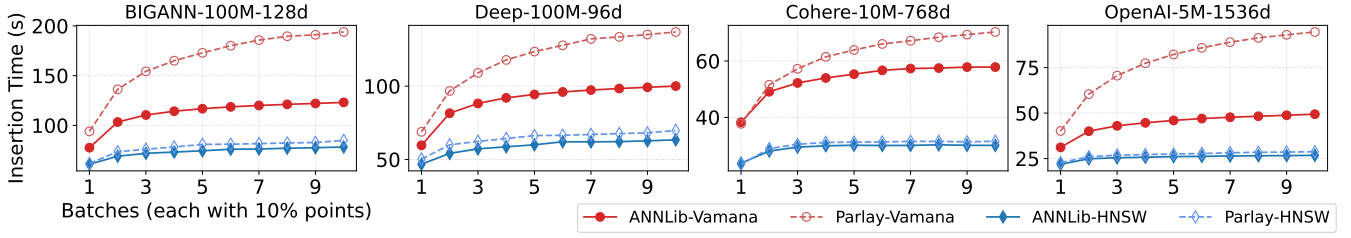


Figure 4: Incremental Insertion Time. Lower is better. x-axis indicates ten batches to insert; y-axis shows the per-batch insertion time.

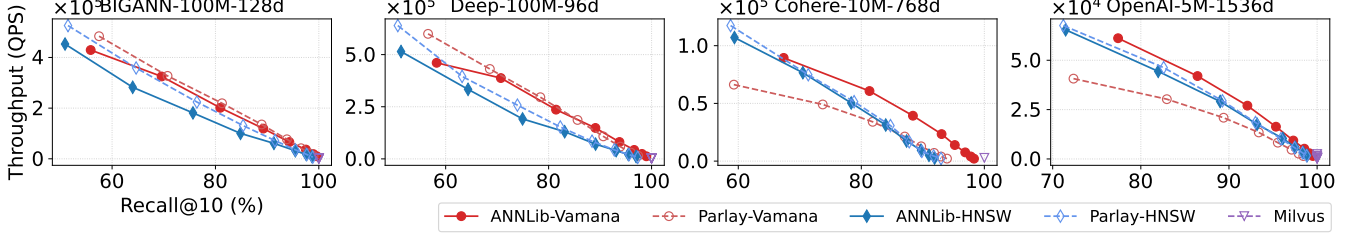


Figure 5: QPS-recall of regular query. Top-right is better. The optimal throughput-recall tradeoff over parameter sweeping.

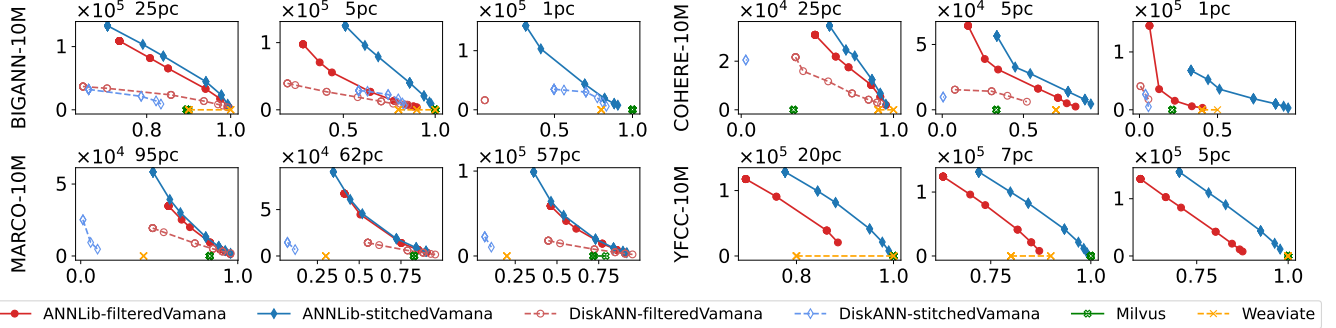


Figure 6: QPS-recall of Filtered Search. The throughput-recall curves performed on varying datasets and labels; x-axis for recall and y-axis for QPS. Each dataset uses three labels for query, indicating the specificity on the top. 25pc means 25% of base points contain the queried label.

not support deletion or versioning. We compare batch deletion time with FreshDiskANN [52]. For filtered search, we compare to DiskANN [54]’s original implementation, and two industrial vector databases, Milvus [59] and Weaviate [22]. In all experiments, we focus only on the in-memory setting and exclude disk-related operations for both ANNLib and baselines.

5.1 Evaluating Insertion

We evaluate batch insertion for both *Vamana* and *HNSW* compared with *ParlayANN*. Since these indices are built by incremental insertion, insertion time also reflects build performance. We split each dataset into ten equal batches, insert them into an initially empty index, and plot per-batch insertion time in Figure 4. On all datasets, ANNLib achieves shorter total time than *ParlayANN* (itself known for fast builds) for both *Vamana* and *HNSW*. Our *Vamana* is 1.4 $\times$ faster on average, up to 1.9 $\times$ on the last batch of *OpenAI*. The *HNSW* on ANNLib keeps runtime comparable to *Parlay-HNSW*, with a 1.06 $\times$ speedup on average. This improvement primarily stems from copy elision: our interfaces adhere to the true data requirements and avoid unnecessary copies.

5.2 Evaluating Regular Vector Search

We evaluate both graph quality and search efficiency by running standard vector search over the built *Vamana* and *HNSW* indices.

Figure 5 presents QPS-recall trade-offs for 10-NN search. On *Cohere* and *OpenAI*, ANNLib-Vamana is consistently faster than Parlay-Vamana, while their *HNSW* curves are similar. On *BIGANN* and *DEEP*, ANNLib shows lower throughput in low-recall regions but converges to comparable QPS as recall approaches 1.0. The low-recall throughput gap is due to a specialized optimizer in *ParlayANN* that caps the number of node visits when the search parameter is small. We also include Milvus in these tests, which is a vector database optimized for high query quality. Across parameter sweeps, Milvus consistently achieves near-perfect recall, but at the cost of significantly lower throughput.

Figure 7 reports billion-scale results. All indices build in a few hours on a single machine. On DEEP-1B ANNLib builds 1.99 $\times$ faster for *Vamana* and 1.63 $\times$ faster for *HNSW*. On BIGANN-1B it is 1.16 $\times$ faster for *Vamana*. For search, ANNLib leads throughout on DEEP-1B, where ANNLib-HNSW achieves 1.6–1.9 $\times$ the throughput of Parlay-HNSW, and ANNLib-Vamana reaches 1.4 $\times$ at 90.5% recall and over 2 $\times$ beyond 94%. On BIGANN-1B ANNLib-HNSW keeps 1.0–1.6 $\times$, while the *Vamana* curves cross. ParlayANN’s visited-node cap yields 1.35 $\times$ in 74%–93% recall, since at billion scale every avoided visit is a likely cache miss, but it also flattens the curve at small beams and caps recall earlier, so ANNLib leads above 95% and reaches 98.8% against 97.6%.

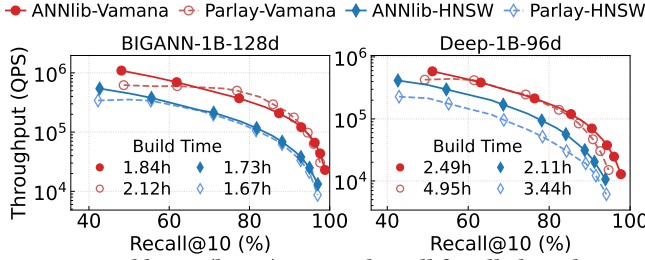


Figure 7: Build time (hours), QPS and recall for all algorithms on billion-size datasets.

5.3 Evaluating Deletion

We use the algorithm in FreshDiskANN [52], and compare our implementation to their code and other baselines. For page limit, we present the performance evaluation in the full version of this paper [50] and summarize the main conclusions here: our batch deletion algorithm built upon ANNLlib’s general interface achieves similar performance (both in deletion time and consequent query quality) to the original highly-optimized algorithm in FreshDiskANN, and are much better than other baselines.

5.4 Evaluating Filtered Search

We present filtered search results in comparison with the state-of-the-arts. Figure 6 shows QPS-recall curves across four common vector datasets under varying specificity. Here, specificity is defined as the fraction of points carrying a given label relative to the total number of points, where a lower value reflects more difficulty in the filtered search. For each dataset, we select three specificities, shown as percent, to evaluate the performance pattern.

Figure 6 shows ANNLlib’s competitive performance: it matches or outperforms the evaluated monolithic solutions in most cases, despite ANNLlib’s primary goal of simplifying ANNS system development. We follow the experimental settings of Filtered-DiskANN. We evaluate filtered search on a multi-label base set with a single filter per query. Detailed setups are available in the full version [50].

5.5 Evaluating Snapshots

The techniques in Section 3.3 reduce memory usage by eliminating duplication across snapshots. To evaluate this, we incrementally construct an index preserving all historical versions and report both total build time and memory usage in Figure 8. Across diverse datasets and settings, our prefix-sharing structure cuts memory usage by up to 70.4% over a chronological-list scheme. The benefit is especially pronounced for datasets with larger degree bounds, where avoiding full edge copies matters more. As shown in Figure 8, pairing a parallel tree with the prefix array saves further memory at the cost of build time: enabling the tree structure increases build time by 1.13× to 3.33× over the naive scheme, and enabling snapshots adds another 1.22×. Due to ANNLlib’s modular design, developers can easily swap graph components to adapt to target scenarios and achieves various trade-offs.

6 Related Work

ANNS Algorithms. A broad range of studies has addressed ANNS algorithms, including DiskANN [54], Stars [11], SPFresh [63], ParlayANN [38], and other graph-based approaches [26, 27, 29, 36, 37, 39, 42, 47, 48, 67, 70]. These state-of-the-art systems handle billion-

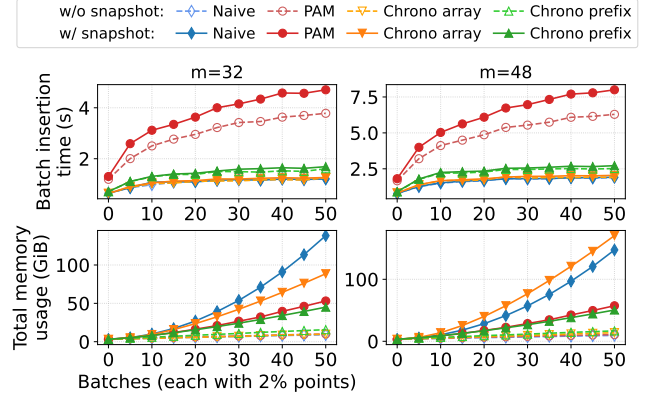


Figure 8: Build time and memory usage for inserting 50 batches (2% points each) on BIGANN-10M with/without snapshots. Lower is better. “w/o snapshot”: only keep the last version. “w/ snapshot”: keep all historical versions. “Naive”: baseline solutions, which updates an array in-place if no snapshot is taken, otherwise performs a deep copy for each snapshot. “Chrono array”: replace the prefix-shared array with a regular one in the chrono prefix array structure.

scale datasets on a single node, but their codebases have grown to over 10,000 lines of code [54], making it hard to locate and modify relevant parts to port features and tailor functions. Recently, kAN-Nolo [18] introduced a Rust library with composable abstractions for ANNS with varying vector density and quantization methods.

Vector Databases. ANNS is widely used in databases for vector queries, known as vector databases. Vector databases either extend traditional engines [33, 68] or build specialized systems [59]. They typically wrap a few fixed ANNS algorithms, requiring substantial code modifications to integrate peripheral database capabilities.

Filtered ANNS. Filtered ANNS extends traditional ANN search by additionally satisfying a set of metadata predicates, and existing work falls into two categories. The first treats filtering as an independent step that leaves the index unchanged, applying it either before (pre-filtering) [13, 22, 45, 59, 61] or after (post-filtering) [24, 32, 64] the vector search; both degrade under highly selective filters. The second builds filter-aware indexes that incorporate filtering information into the ANNS index, either by modifying construction and search algorithms to handle vector similarity and metadata constraints simultaneously, as in Filtered-DiskANN [28], or by combining attribute constraints within the index itself, as in Native Hybrid Query (NHQ) [60] and MA-NSW [62].

7 Conclusions

We propose ANNLlib, a library for graph-based ANNS development that achieves both **high performance** and **flexible functionality** by decoupling and independently optimizing the *algorithm* and the *data structure* components in an ANNS system. Both allow for easy integration of state-of-the-art solutions as modules, as well as our new designs, and compose naturally to address sophisticated settings such as filtered search, fully dynamic updates, and historical queries on snapshots. Our experiments show that solutions built atop ANNLlib’s general interfaces match or outperform previous work designed specifically for each application. Future directions include extending ANNLlib to disk-based indexing and integrating quantization techniques.

References

- [1] 2019. Hnswlib - fast approximate nearest neighbor search. Webpage. Retrieved December 16, 2019 from <https://github.com/nmslib/hnswlib>
- [2] 2023. Apache Lucene. <https://lucene.apache.org/>
- [3] 2023. Microsoft Bing Search Engine. <https://www.bing.com/new>
- [4] 2023. Pinecone: Vector Database for Vector Search. <https://www.pinecone.io/>
- [5] 2023. Weaviate: The AI Native Vector Database. <https://weaviate.io/>
- [6] 2025. ANNlib: A Development Framework for Efficient Approximate Nearest Neighbor Search. <https://github.com/ucparlay/ANNlib-pub>.
- [7] Laurent Amsaleg and Hervé Jegou. 2010. Datasets for approximate nearest neighbor search. <http://corpus-textmex.irisa.fr/>. [Online; accessed 20-May-2018].
- [8] 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.
- [9] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 507–509. <https://doi.org/10.1145/3350755.3400254>
- [10] Guy E. Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208. <https://doi.org/10.1137/1.9781611977042.16>
- [11] CJ Carey, Jonathan Halcrow, Rajesh Jayaram, Vahab Mirrokni, Warren Schudy, and Peilin Zhong. 2022. Stars: Tera-Scale Graph Building for Clustering and Learning. In *Advances in Neural Information Processing Systems*, Vol. 35. 21470–21481.
- [12] Harrison Chase. 2023. Vector DB text generation. https://python.langchain.com/en/latest/modules/chains/index_examples/vector_db_text_generation.html
- [13] 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* 17, 12 (2024), 3772–3785.
- [14] Qi Chen, Xiubo Geng, Corby Rosset, Carolyn Buracton, Jingwen Lu, Tao Shen, Kun Zhou, Chenyan Xiong, Yeyun Gong, Paul Bennett, et al. 2024. MS MARCO Web Search: a Large-scale Information-rich Web Dataset with Millions of Real Click Labels. In *Companion Proceedings of the ACM on Web Conference 2024*. 292–301.
- [15] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. SPTAG: A library for fast approximate nearest neighbor search. <https://github.com/Microsoft/SPTAG>
- [16] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 5199–5212.
- [17] Cohere. 2023. Wikipedia (en) embedded with cohere.ai multilingual-22-12 encoder. <https://huggingface.co/datasets/Cohere/wikipedia-22-12-en-embeddings>. [Online; accessed 22-January-2025].
- [18] Leonardo Delfino, Domenico Erriquez, Silvio Martinico, Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. 2025. kANNolo: Sweet and Smooth Approximate k-Nearest Neighbors Search. Springer Nature Switzerland, 400–406. https://doi.org/10.1007/978-3-031-88717-8_29
- [19] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-trees: supporting parallel and compressed purely-functional collections (PLDI 2022). Association for Computing Machinery, New York, NY, USA, 108–121. <https://doi.org/10.1145/3519939.3523733>
- [20] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees (PLDI 2019). Association for Computing Machinery, 918–934. <https://doi.org/10.1145/3314221.3314598>
- [21] Laxman Dhulipala, Majid Hadian, Rajesh Jayaram, Jason Lee, and Vahab Mirrokni. 2024. MUVERA: Multi-Vector Retrieval via Fixed Dimensional Encodings. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [22] Etienne Dlocker, Bob van Luijt, Byron Voorbach, Mohd Shukri Hasan, Abdel Rodriguez, Dirk Alexander Kulawiak, Marcin Antas, and Parker Duckworth. 2024. Weaviate. Webpage. <https://github.com/weaviate/weaviate>
- [23] Xiangyun Ding, Rohin Garg, Yan Gu, and Yihan Sun. 2026. Parallel Metric Skip Lists and Nearest Neighbor Search. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [24] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jegou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).
- [25] Magdalini Eiriniaki, Jerry Gao, Iraklis Varlamis, and Konstantinos Tserpes. 2018. Recommender systems for large-scale social networks: A review of challenges and solutions. , 413–418 pages.
- [26] Cong Fu, Changxu Wang, and Deng Cai. 2022. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.* 44, 8 (2022), 4139–4150.
- [27] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.
- [28] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *Proceedings of the ACM Web Conference 2023 (WWW '23)*. 3406–3416.
- [29] Lars Gottesbüren, Laxman Dhulipala, Rajesh Jayaram, and Jakub Łącki. 2025. Unleashing Graph Partitioning for Large-Scale Nearest Neighbor Search. *Proc. VLDB Endow.* 18, 6 (2025), 1649–1662.
- [30] 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. *VLDB Endowment* (2022), 3548.
- [31] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based retrieval in facebook search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2553–2561.
- [32] Jeff Johnson, Matthijs Douze, and Hervé Jegou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.
- [33] Andrew Kane. 2022. pgvector. Webpage. Retrieved July, 2023 from <https://github.com/pgvector/pgvector/>
- [34] Yann LeCun, Yoshua Bengio, et al. 1995. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks* 3361, 10 (1995), 1995.
- [35] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. 2018. The Design and Implementation of a Real Time Visual Search System on JD E-commerce Platform. In *Proceedings of the 19th International Middleware Conference Industry*. Association for Computing Machinery, 9–16. <https://doi.org/10.1145/3284028.3284030>
- [36] Jingjia Luo, Mingxing Zhang, Kang Chen, Xia Liao, Yingdi Shan, Jinlei Jiang, and Yongwei Wu. 2025. Efficient Graph-Based Approximate Nearest Neighbor Search Achieving: Low Latency Without Throughput Loss. *arXiv:2504.20461*.
- [37] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.
- [38] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '24)*. 270–285.
- [39] Leland McInnes. 2020. PyNNDescend for Fast Approximate Nearest Neighbors. Webpage. Retrieved December 15, 2022 from <https://pynndescent.readthedocs.io/en/latest/>
- [40] Ziyang Men, Zheqi Shen, Yan Gu, and Yihan Sun. 2025. Parallel kd-tree with Batch Updates. *Proc. ACM Manag. Data* 3, 1 (2025).
- [41] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).
- [42] Javier Alvaro Vargas Muñoz, Marcos André Gonçalves, Zanon Dias, and Ricardo da Silva Torres. 2019. Hierarchical Clustering-Based Graphs for Large Scale Approximate Nearest Neighbor Search. *Pattern Recognit.* 96 (2019).
- [43] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005* (2022).
- [44] Arnold Overwijk, Chenyan Xiong, and Jamie Callan. 2022. ClueWeb22: 10 billion web documents with rich information. In *Proceedings of the 45th international ACM SIGIR conference on research and development in information retrieval*. 3360–3362.
- [45] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. Acorn: Performant and predicate-agnostic search over vector embeddings and structured data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.
- [46] Ninh Pham and Tao Liu. 2022. Falconn++: A Locality-sensitive Filtering Approach for Approximate Nearest Neighbor Search. *CoRR abs/2206.01382* (2022). <https://doi.org/10.48550/arXiv.2206.01382> *arXiv:2206.01382*
- [47] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).
- [48] Tobias Rubel, Richard Wen, Laxman Dhulipala, Lars Gottesbüren, Rajesh Jayaram, and Jakub Łącki. 2026. PiPNN: Ultra-Scalable Graph-Based Nearest Neighbor Indexing. *arXiv:2602.21247*.
- [49] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.
- [50] Zheqi Shen, Jingbo Su, Zijin Wan, Yan Gu, and Yihan Sun. 2026. ANNLib: A Development Framework for Efficient Approximate Nearest Neighbor Search. *arXiv:2607.17582* [cs.LG] <https://arxiv.org/abs/2607.17582>

- [51] Harsha Vardhan Simhadri, Martin Aumüller, Amir Ingber, Matthijs Douze, George Williams, Magdalen Dobson Manohar, Dmitry Baranchuk, Edo Liberty, Frank Liu, Ben Landrum, et al. 2024. Results of the Big ANN: NeurIPS'23 competition. *arXiv preprint arXiv:2409.17424* (2024).
- [52] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *CoRR abs/2105.09613* (2021). arXiv:2105.09613 <https://arxiv.org/abs/2105.09613>
- [53] Colette Stallbaumer. 2023. Introducing Microsoft 365 copilot. <https://www.microsoft.com/en-us/microsoft-365/blog/2023/03/16/introducing-microsoft-365-copilot-a-whole-new-way-to-work/>
- [54] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 13748–13758.
- [55] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [56] Yukihiro Tagami. 2017. Annexml: Approximate nearest neighbor search for extreme multi-label classification. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 455–464.
- [57] tawalke. 2023. PR: SK Vectorsdb Connector Work - Merging forked branch; PR from Fork to SK Branch by Tawalke · pull request 83 · Microsoft/Semantic-Kernel. <https://github.com/microsoft/semantic-kernel/pull/83>
- [58] Bart Thomee, David A Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. 2016. Yfcc100m: The new data in multimedia research. *Commun. ACM* 59, 2 (2016), 64–73.
- [59] 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 *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.
- [60] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongzhang Ni. 2023. An efficient and robust framework for approximate nearest neighbor search with attribute constraint. *Advances in Neural Information Processing Systems* 36 (2023), 15738–15751.
- [61] 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* 13, 12 (2020), 3152–3165.
- [62] Xiaoliang Xu, Chang Li, Yuxiang Wang, and Yixing Xia. 2020. Multiattribute approximate nearest neighbor search based on navigable small world graph. *Concurrency and Computation: Practice and Experience* 32, 24 (2020), e5970.
- [63] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 545–561.
- [64] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension (*SIGMOD '20*). 2241–2253.
- [65] Shangdi Yu, Joshua Engels, Yihao Huang, and Julian Shun. 2025. PECANN: Parallel Efficient Clustering with Graph-Based Approximate Nearest Neighbor Search. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*.
- [66] Jianjin Zhang, Zheng Liu, Weihao Han, Shitao Xiao, Ruicheng Zheng, Yingxia Shao, Hao Sun, Hanqing Zhu, Premkumar Srinivasan, Weiwei Deng, et al. 2022. Uni-retriever: Towards learning the unified embedding based retriever in bing sponsored search. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4493–4501.
- [67] Jiaru Zhang, Ruhui Ma, Tao Song, Yang Hua, Zhengui Xue, Chenyang Guan, and Haibing Guan. 2022. Hierarchical Satellite System Graph for Approximate Nearest Neighbor Search on Big Data. *ACM/IMS Trans. Data Sci.* 2, 4 (2022).
- [68] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 377–395.
- [69] Yanhao Zhang, Pan Pan, Yun Zheng, Kang Zhao, Yingya Zhang, Xiaofeng Ren, and Rong Jin. 2018. Visual search at alibaba. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 993–1001.
- [70] Ziyu Zhang, Yuanhao Wei, Joshua Engels, and Julian Shun. 2025. CleANN: Efficient Full Dynamism in Graph-based Approximate Nearest Neighbor Search. arXiv:2507.19802.