CFFI Working Paper No.26-03

Accelerating Ephemeral Approximate Nearest Neighbor Search by Progressive Index Construction

Minhui Xie, Enrui Zhao, Yaxin Ma, Puqing Wu,
Baotong Lu[†], Yuanhui Luo, Yongqiang Xiong[†], Jing Wang, Yunpeng Chai*
Renmin University of China [†]*Microsoft Research*

Center for Future Financial Innovation
Renmin University of China

This paper is available for free download from the electronic paper repository of the Center for Future Financial Innovation, Renmin University of China.
http://cffi.ruc.edu.cn/kycg/gzlw/

Accelerating Ephemeral Approximate Nearest Neighbor Search by Progressive Index Construction

Minhui Xie, Enrui Zhao, Yaxin Ma, Puqing Wu,
Baotong Lu[†], Yuanhui Luo, Yongqiang Xiong[†], Jing Wang, Yunpeng Chai*
Renmin University of China [†]*Microsoft Research*

[1]

Abstract

Emerging applications like AI chatbots, code assistants, and agentic workflows have created a growing need for ephemeral Approximate Nearest Neighbor Search (ANNS), where an ANN index must be constructed online over pre-unknown, ad-hoc, short-lived datasets. Traditional ANNS methods, designed for offline index construction on pre-known datasets, are ill-suited for such scenario: the monolithic, upfront index construction process imposes substantial latency on the user's critical path, degrading the interactive experience.

This paper presents FLEETANN, a system that accelerates ephemeral ANNS by pioneering a progressive index construction paradigm. FLEETANN logically partitions the dataset into an already-indexed component (*I-component*) and an unindexed brute-force component (*BF-component*), separated by a conceptual *cursor*. In the background, FLEETANN continuously advances this cursor by migrating vectors from BF-component into I-component, incrementally building the index. In the foreground, FLEETANN can serve user queries immediately via a hybrid retrieval strategy, ensuring theoretically guaranteed recalls even with a partially constructed index. To mitigate the initial high cost of brute-force search, FLEETANN introduces a history-guided pruning technique that exploits distance information from past queries to avoid unnecessary computations. Evaluation shows that FLEETANN can avoid costly initial construction stall (up to hundreds of seconds) while ultimately achieving the same or even better query performance as a full ANNS index.

1 Introduction

Approximate Nearest Neighbor Search (ANNS) is a fundamental technique for efficiently identifying data points close to a given query in high-dimensional spaces. Due to its efficiency and scalability, ANNS has been widely adopted in diverse applications, including recommendation [1], NLP [2, 3], and search engines [4, 5]. In the current era of LLM [2], its importance is further magnified. As the underlying engine of Retrieval-Augmented Generation (RAG), ANNS alleviates the hallucination [6] and outdated [7] problem by retrieving knowledge from external up-to-date documents.

The workflow of ANNS mainly contains two distinct phases: index constructing and index searching (as shown in Figure 1a). The former builds a certain index (e.g., cluster-based [8, 9] or graph-based [10, 11] data structures), while the latter uses the constructed index to speedup all subsequent ANNS queries. Aligned with these two parts, existing works on ANNS can be roughly classified into accelerating index constructing [12–14] and index searching [15–18].

Traditionally, ANNS indexes are constructed *offline* in advance for a *pre-known* dataset [9, 11, 17–24]. These indexes are designed to be long-lived and *persistent*, serving queries over corresponding candidate sets such as product feeds in recommendation systems or large-scale image/video collections in image/video retrieval.

In contrast, a new wave of emerging ANNS applications [25,26] catalyzes a strong demand for *online* constructed ANNS index over a *previously-unseen*, *ephemeral* dataset (Figure 1b). In this paper, we term such scenario **ephemeral ANNS**. Typical examples include AI chatbots (e.g., ChatGPT [27]) and agentic workflows (e.g., Deep Research [28]). In AI chatbot applications (Figure 1c), a user may upload multiple private documents and expect the assistant to answer subsequent questions based exclusively on the provided contents. To reduce inference costs, the system typically first retrieves the relevant information from lengthy documents before invoking the LLM. However, as the document corpus is ad-hoc and session-specific, a pre-built index is infeasible. Consequently, a temporary but efficient ANNS index must be built "just-in-time" for the user's session and can be disposed of afterward. Similarly, in agentic research workflows, AI agents autonomously crawl, analyze, and synthesize hundreds of sources into a research-analyst-level report. In such scenarios, building a temporary ANNS index over the crawled documents enables efficient resource organization and rapid

[1]Corresponding author (chaiyp@ruc.edu.cn).

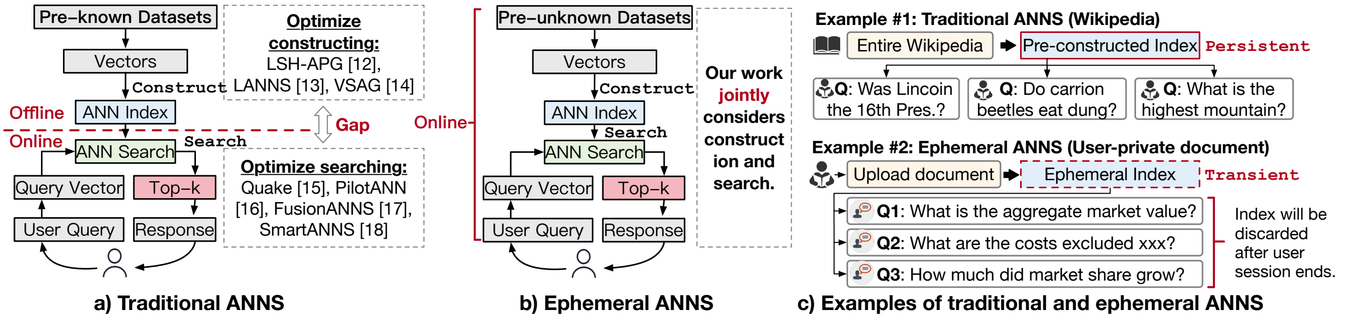


Figure 1: Comparison of traditional ANNS and ephemeral ANNS.

content retrieval. Finally, emerging interest in RAG over the data lake [29] with massive unstructured data further amplifies this need. Maintaining indexes for all data is neither feasible nor cost-effective, making on-demand, query-driven index construction increasingly compelling.

Ephemeral ANNS presents a unique set of challenges that distinguish it from traditional offline ANNS. First, different from offline ANNS, the overhead of index construction is entirely exposed online on the user's critical path, thereby degrading the performance perceived by upper-layer applications. According to our evaluation on a popular open-source ChatGPT-like chatbot framework, OpenWebUI [30], the online index construction in an RAG query accounts for 54% of the user-facing latency (see §2.2 for detailed configurations). Second, despite this high cost, the index may serve only a few queries before being discarded, causing severe resource waste. This is due to the unpredictable number of queries a user may issue to the ephemeral ANN index.

To address these challenges, this paper introduces FleetANN, a system designed for ephemeral ANNS. The key idea of FleetANN is *progressive index construction* (§3.2). Instead of a monolithic, blocking construction phase, FleetANN decomposes index construction into a series of lightweight, incremental steps that execute in parallel with query processing. This allows the system to begin serving queries almost instantly, with search performance progressively improving as the index is incrementally built. Specifically, FleetANN logically partitions the dataset into two disjoint components (*I-component* and *BF-component*), separated by a conceptual *cursor*. I-component contains vectors that have already been organized into an ANN index, while BF-component holds the remaining unindexed vectors. In the background, FleetANN continuously advances the cursor by migrating vectors from BF-component into I-component.

With these two components, FleetANN introduces a *hybrid retrieval method* (§3.2), to serve queries by performing both an efficient ANNS search on I-component and a brute-force search on BF-component, merging the results to guarantee completeness (a property we refer to as *Hybrid Retrieval Completeness*). This guarantee ensures that the average recall of the combined result is at least better than that of the original index retrieval. We validate this property through both *theoretical proofs* and empirical evaluation.

To mitigate the high cost of the brute-force search, which dominates latency in the early stages, we design a *history-guided search pruning* mechanism (§3.3). This technique leverages the distance information from historical queries to establish a lower bound on the distances to vectors in BF-component. By structuring these historical distances within our proposed HistTree, FleetANN can hierarchically prunes search space without costly distance calculations, significantly accelerating brute-force search.

FleetANN employs background threads for *advancing the cursor* (§3.4). They continuously migrate vectors from BF-component to I-component, progressively improving search performance. To preclude concurrent conflicts caused by the omission of being-moved vectors that fail to be retrieved in both components, we carefully coordinate the background migration and foreground search to ensure data consistency.

The policy of FleetANN is general to different ANN indexes. We implement FleetANN on two mainstream indexes (graph-based and cluster-based) and evaluate on four real-world ephemeral ANNS applications, including AI chatbots, code assistants, data lake retrieval, and agentic workflows (DeepResearch). Compared with existing widely-used strategies, including construct-then-search (EagerBuild, used by [30, 31]) and brute-force search (BruteOnly, used by [3]), FleetANN not only avoids the significant initial index construction time (e.g., hundred-second stall) as in BruteOnly, but also achieves comparable query performance as EagerBuild ultimately. For the end-to-end performance, FleetANN can reduce overall completion time by up to 2.6x and lower tail latency by orders of magnitude (by avoiding the initial stall), while *incurring no loss in recalls*.

In summary, this paper makes the following contribution.

- We identify and characterize the problem of ephemeral ANNS, a new and critical paradigm of ANNS driven by emerging AI applications.
- We design FleetANN, a system embodying the progressive index construction paradigm.

- We evaluate it on four real-world ephemeral ANNS applications to show its efficiency and effectiveness.

2 Background and Motivation

2.1 Approximate Nearest Neighbor Search

Finding the exact nearest neighbors to a query vector in a high-dimensional space is a computationally demanding task. The complexity of a naive scan is directly proportional to the dataset size, which makes exact search impractical for the massive datasets common in modern applications.

Approximate Nearest Neighbor Search (ANNS) addresses this challenge by relaxing the requirement for perfect accuracy. The common practice of ANNS is to maintain a certain data structure (called ANNS index) to encode proximity relationships among vectors. These structures—such as graph-based (e.g., HNSW [10]), cluster-based (e.g., IVF [8]), tree-based (e.g., K-D trees [32]), and hash-based structures (e.g., LSH [33]) —enable the search process to quickly eliminate large portions of the search space, reducing both computational cost and memory access overhead.

The canonical ANNS workflow has two distinct phases: offline index construction and online index search.

- Index construction (offline): A specialized data structure (index) is built over the entire dataset. This is a computationally intensive, one-time process performed in advance, typically on a static, pre-known dataset.
- Index search (online): Once the index is built, it can be used to speedup queries. A user query typically consists of two parameters: a query vector v and a parameter top-k. With the prebuilt index, the search algorithm efficiently narrows the search space to identify approximate nearest neighbors without scanning the entire dataset.

This conventional decoupling of construction and search stems from the traditional use cases for ANNS, which typically serve a vast, persistent dataset in applications like information retrieval and recommendation systems. The significant initial cost of index construction is amortized across a high volume of queries over a long lifespan. In contrast, the ephemeral ANNS scenario targeted in our work presents a different set of challenges.

2.2 Ephemeral ANNS

As stated in §1, unlike the conventional approach of constructing durable and persistent indexes, emerging applications call for a fundamentally different approach, which we term *Ephemeral ANNS*. In ephemeral ANNS, both index construction and search occur online within a user's active session. As a result, the index construction overhead, which was traditionally an offline pre-computation cost, is now fully exposed to users and becomes a critical component of end-to-end latency.

Real-world examples of ephemeral ANNS:

- #1: AI chatbots. As shown in Figure 1c, users may upload private documents (e.g., financial reports, confidential documents) and ask questions based on their content. Because these documents can be lengthy, an ANNS index should be created on the fly to reduce the inference cost and avoid hitting context-window limits. The index is only needed for the lifetime of the session.
- #2: Coding assistant. For developers, AI assistants (e.g., Copilot) can provide relevant code completions or answer questions by understanding the context of the current project workspace. The database is the local collection of source files, which is unique and changes frequently. An ephemeral index built over this local context provides the necessary specificity and freshness.
- #3: Retrieval on data lakes [29, 34]. In data lakes, analysts often perform exploratory searches on new or infrequently accessed datasets. Given the scale of modern data lakes and the skewed query patterns, building and maintaining persistent, pre-computed indexes for every potential dataset is computationally prohibitive and impractical.
- #4: Agentic workflows. Agentic workflows (like Deep Research [28]) also require ephemeral ANNS. Besides relying on web searches for knowledge retrieval, agentic applications also perform deep research over local corpora such as enterprise documents or crawled web pages [35,36] to synthesize research-analyst-level reports. In such scenarios, dynamically building a temporary ANNS index over these documents enables efficient organization of resources and rapid content retrieval.

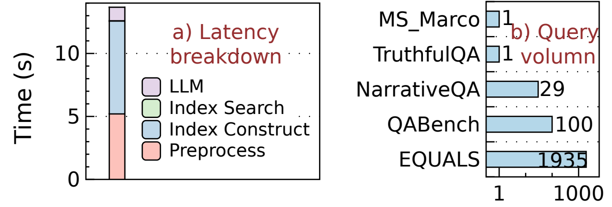


Figure 2: Characteristics of ephemeral ANNS.

Characteristics of ephemeral ANNS. Among these examples, ephemeral ANNS exhibits characteristics fundamentally distinct from those of traditional ANNS, which we summarize in the following three aspects:

- *Index construction on the critical path.* Ephemeral ANNS constructs the index online, often during the same session in which it is queried. This construction cost is no longer amortized over a long time but fully exposed to the user's end-to-end latency, making it critical. Figure 2a shows a breakdown of user-facing latency in RAG query scenario of AI chatbots (using Llama-3 8B); see §4 for configuration details. Even with a moderate capacity (20k vectors), the index construction still accounts for ~54% of latency.
- *Transient and session-specific lifespan.* Unlike traditional ANNS indexes which are long-lived and persist across days or years [11], ephemeral ANNS indexes are short-lived, often tied to a single user session (e.g., a ChatGPT session with uploaded documents). Once the session ends, the

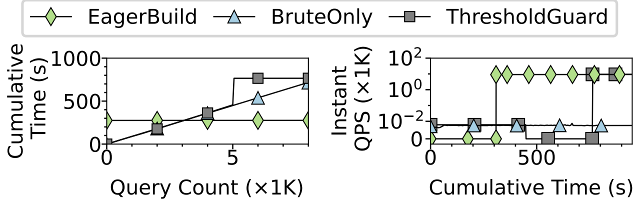


Figure 3: Cumulative time and throughput of different baseline strategies by varying query volume.

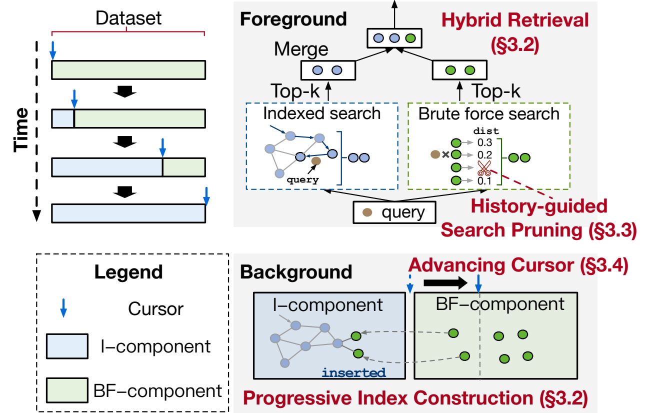


Figure 4: FLEETANN overview.

index is typically discarded. This requires us to carefully assess the cost of index construction.

- *High-variant query volume.* In Ephemeral ANNS, the number of queries served by indexes over their lifetimes exhibits particularly high variance. Figure 2b shows the query count distribution in several RAG datasets. The number of queries served can vary by orders of magnitude between the least and most utilized indexes. This extreme variance makes it challenging to assess the cost-effectiveness of performing index construction in advance.

2.3 Motivation

The unique characteristics of ephemeral ANNS render conventional ANNS solutions inadequate. To illustrate this gap, we examine three representative strawman strategies, which we name as *EagerBuild*, *BruteOnly*, and *ThresholdGuard*.

- *Always constructing (EagerBuild).* This approach constructs a complete ANNS index at the beginning of each session, regardless of how many queries the session will actually receive. It is the default policy used in well-known systems like OpenWebUI [30] and LangFlow [31]. While it guarantees optimal query performance, it incurs high, upfront construction overhead that may be wasted if the session is short-lived or receives few queries.
- *Never constructing (BruteOnly).* At the opposite extreme, some systems like IKS [3] avoid index construction entirely, relying solely on brute-force search to answer queries. While this avoids the indexing cost, it sacrifices query efficiency—especially as the query count grows—leading to unacceptable query latencies.
- *Deferred constructing (ThresholdGuard).* To strike a balance, we also introduce a deferred construction strategy recommended by the official Faiss guidelines [37]. It triggers index building only after the number of incoming queries exceeds a predefined threshold. This approach is adaptive to some extent, avoiding unnecessary indexing for short sessions while benefiting long ones.

Figure 3 illustrates the cumulative time and query processing throughput of each strategy. Both *EagerBuild* and *BruteOnly* strategies naturally lack adaptability: while *EagerBuild* is favorable with high query volume and *BruteOnly* performs well when the query volume is low, they fail to deliver robust performance across varying conditions. Although *ThresholdGuard* adapts better, it still suffers from two limitations: 1) spiking query stall when reaching a certain threshold (caused by the construction process), and 2) tuning difficulty, since handcraft thresholds may not generalize across workloads, leading to either premature indexing or missed optimization opportunities.

In summary, none of these strategies strikes an effective balance between construction overhead and query efficiency under the highly unpredictable workloads of ephemeral ANNS. This motivates the need for a new design that is both adaptive to query volume and efficient in end-to-end performance.

Our key idea: Progressive index construction. To this end, our key idea is to shift from monolithic, upfront index construction to progressive index construction. It decomposes the index construction process into small, incremental construction pieces, each of which inserts a subset of vectors into the index. These phases are interleaved with query execution, allowing the system to serve queries immediately. With progressive index construction, FLEETANN can serve user queries immediately, initially via brute-force search and progressively accelerate them as more vectors are indexed.

This progressive construction paradigm enjoys two benefits: 1) Temporal amortization: By spreading the construction across multiple queries, FLEETANN eliminates the stall of the initial query. 2) Adaptivity to query volume: If a session is short-lived, FLEETANN avoids wasting compute resources on unnecessary indexing. If query volume is high, the system can naturally build a high-quality index in the background.

3 Design

3.1 Overview

Figure 4 presents an overview of FLEETANN, which contains two parts: foreground and background.

FLEETANN progressively constructs the index in the background (§3.2). With a consistently advancing *cursor* (§3.4), FLEETANN logically partitions the dataset into two disjoint components: already indexed vectors (called *I-component*)

and unindexed vectors (called *BF-component*). The background threads continuously migrate vectors from BF-component to I-component.

In the foreground, FleetANN handles incoming user queries through a *hybrid retrieval* method (§3.2) that combines indexed search on I-component and brute-force search on BF-component. We theoretically prove that the combined search results remain correct and complete (a property we refer to as *Hybrid Retrieval Completeness*), even when some vectors remain unindexed.

In the early stages of construction, most retrievals resort to a brute-force search on BF-component. This incurs a high overhead with O(N) time complexity for each query. For example, even when half of the dataset has already been indexed, brute-force search still accounts for 97% of the total query time (Figure 9). To this end, FleetANN introduces *history-guided search pruning* (§3.3) which leverages distance information of historical queries to prune unnecessary computation in BF-component.

3.2 Progressive Index Construction

The central design goal of FleetANN is to eliminate the substantial stall of upfront index construction that directly impacts users in ephemeral cases. We achieve this through *progressive index construction*.

The key observation is that the mainstream (or widely-used) ANNS indexes commonly follow an incremental construction process, which inherently consists of a sequence of vector insertion operations. The partially-constructed index can be used to serve ANNS immediately on all inserted vectors. For example, SPFresh [9] (a cluster-based index based on SPANN [19]) supports directly inserting vectors into different clusters, with only some lightweight modifications like cluster's local re-balancing. HNSW [10] and DiskANN [11, 38] (graph-based indexes) are constructed by iteratively inserting each data vector through traversal of the existing graph, which naturally supports incremental insertions.

System architecture. Based on this observation, we decompose the construction process into multiple fine-grained pieces interleaved with queries, thereby amortizing the construction overhead. Specifically, FleetANN logically partitions the dataset into two disjoint components, *I-component* and *BF-component*, separated by the abstraction of *cursor*:

- The *I-component* consists of vectors that have already been inserted into the approximate nearest neighbor index. These vectors are organized with a certain data structure (according to ANNS algorithms) to support fast indexed search. FleetANN does not require any modification to existing ANNS algorithm implementations. Any commonly-used incremental ANNS indexes (e.g., HNSW [10], Vamana graph [11], IVF [9]) can be wrapped into I-component to serve incrementally inserted vectors.
- The *BF-component* holds the remaining vectors that have not yet been indexed and must be searched via brute-force.
- The *cursor* dynamically tracks the boundary between I-component and BF-component. The data is fully stored in the BF-component initially. If the dataset is sufficiently large to warrant ANN indexing, FleetANN advances its cursor by moving vectors from BF-component into I-component concurrently, or between user queries.

Hybrid Retrieval. Upon receiving a top-k query, FleetANN performs two searches with the same k parameter in parallel: 1) an efficient ANN search on I-component, and 2) an exhaustive search on BF-component. The results from both searches are then merged to produce the final, complete top-k. As more vectors are indexed, the cursor moves forward, ensuring that the index is progressively built over time. In the meanwhile, the search performance of FleetANN gracefully improves over the session's lifetime.

Completeness of Hybrid Retrieval. Intuitively, combining search results from I-component and BF-component can yield a top-k result set that is, on average, *at least as good as* performing ANNS over the full dataset. The reason is that retrieval on a partially-constructed index (I-component) only depends on the vectors already inserted and is independent of BF-component. Concurrently, BF-component guarantees perfect recall (100%) through brute-force searching. Therefore, by merging the results from both components, the overall recall of hybrid retrieval is well guaranteed. Here, we **theoretically prove** this property, which is further validated by an empirical study in Exp #4.

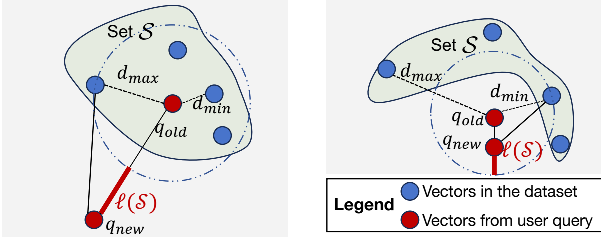
Proof. For the space limit, we mainly consider the graph-based index as I-component; the proof of cluster-based index will be included in the extended paper. For graph-based indexes (e.g., HNSW and Vamana), by borrowing the key idea from [12], we treat the final search radius (i.e., the radius at which the query terminates) as an indicator of the final search quality. Then, we try to theoretically demonstrate that this radius is irrelevant to the graph's capacity (denoted as N). If such independence holds, it implies that graphs constructed at different times, with varying capacities, can still achieve similar final search quality.

For a given query q, a typical query process employs an iteratively best-first search strategy: it iteratively explores neighbors of the current best candidate, adds them to a candidate set, and always proceeds from the candidate closest to the query. We define the final search radius $s(r)$ as a function of r, where r indicates that the search starts from the current point that is at a distance r from the query.

Retrieval always starts from the entry point (called en_0). Assume that the search proceeds to the t-hop point (called en_t), which has a distance of r_t from q. Let en_{t+1} denote the closest point to all queries in the candidate set, with a distance of r_{t+1} from q.

$$s(r_t) = p(r_t) \times r_t + [1 - p(r_t)] \times s(r_t - \delta(r_t))$$

where $\delta(r_t) = r_t - r_{t+1}$ and $p(r_t)$ denotes the probability that the search terminates at en_t.



case I: $dist(q_{old}, q_{new}) \geq d_{max}$ **case II**: $dist(q_{old}, q_{new}) \leq d_{min}$

Figure 5: Basic principle of pruning. *This figure visualizes Equation (3). The red line shows $\ell(\mathcal{S})$, the lower bound of the distance from the new query to all vectors in the set $\mathcal{S}$, which can be used for pruning.*

We have: $p(r_t) = Pr[r_{t+1} > r_t] = Pr[\delta(r_t) < 0]$. Given that $\delta(r_t)$ is determined by the distribution of a point's neighbors. In proximity-graph-based methods, these neighbors are selected based on local information—they are points in close proximity—rather than from the entire dataset [12]. Consequently, the value of $\delta(r_t)$ and also $s(r)$ is irrelevant to the total number of vectors, N. Thus, the final search quality is irrelevant to the capacity. □

3.3 History-guided Search Pruning

In the early stages of FleetANN (when BF-component contains more vectors), hybrid retrieval may degenerate into brute-force search, which is inefficient due to its $O(N)$ time complexity. In our experiments, even when half of the dataset has already been indexed, brute-force search still accounts for 97% of the total query time; see Exp #5 in §4.

We observe that once a user query has been processed via brute-force search, the distances between that query and all vectors in the dataset become known. These historical distance computations, accumulated across previous queries, offer valuable information that can be exploited to efficiently prune redundant searches on BF-component.

In this section, we first present the underlying mathematical underpinnings for pruning based on historical information (§3.3.1). As the cursor advances, the set of vectors associated with the distance information of historical queries is constantly evolving. Thus, we then introduce how to organize them into a hierarchical index efficiently (§3.3.2). Specifically, we build a lightweight auxiliary data structure (named *HistTree*) for BF-component.

3.3.1 Basic Principle of Pruning with Historical Information

We first show the underling mathematical foundation of search pruning. Let q_i and v_j represents the ith user query vector and jth vector in the database. The function $dist(a,b)$ denotes the distance between two vectors, a and b.

For a given new query q_{new}, the basic idea of search pruning is to give a lower bound of distance calculation, $\ell(v) = \inf(dist(q_{new}, v))$, for any vector v in a given corpus $\mathcal{S}$. If this lower bound already exceeds the largest distance among the current candidates, then the corresponding vector v will certainly not be included in the final top-k result set. Thus, we can safely prune all the calculations of $dist(q_{new}, v)$.

Next, we show how to efficiently estimate all $\ell(v) = \inf(dist(q_{new}, v))$ for any given vector v by exploiting the historic information.

For any given vector v and past user query q_{old}, according to the triangle inequality [39][2], the following formula holds:

$$\begin{cases} dist(q_{new}, v) \geq |dist(q_{old}, q_{new}) - dist(q_{old}, v)| \\ dist(q_{new}, v) \leq |dist(q_{old}, q_{new}) + dist(q_{old}, v)| \end{cases} \tag{1}$$

We mainly focus on the upper part of Equation (1) and get

$$\ell(v) = |dist(q_{old}, q_{new}) - dist(q_{old}, v)| \tag{2}$$

The computation of $\ell(v)$ introduces little additional overhead, since $dist(q_{\text{old}}, q_{\text{new}})$ can be shared across different v, and $dist(q_{\text{old}}, v)$ is retrieved from the historical information.

Furthermore, Equation (2) can be extended to a set-based formulation. For a given set $\mathcal{S}$ (e.g., a historical BF-component or a piece of BF-component), since the distances from the previous query to all vectors in this set are known, there must exist a nearest and a farthest vector, with their respective distances to q_{old} denoted as $d_{\min}$ and $d_{\max}$.

According to Equation (1), we can get the distance lower bound of a given set $\mathcal{S}$:

$$\forall\, v \in \mathcal{S}, dist(q_{new}, v) \geq \ell(\mathcal{S})$$

$$\ell(\mathcal{S}) = \begin{cases} dist(q_{old}, q_{new}) - d_{\max}, & \text{if } dist(q_{old}, q_{new}) \geq d_{\max} \\ d_{\min} - dist(q_{old}, q_{new}), & \text{if } dist(q_{old}, q_{new}) \leq d_{\min} \end{cases} \tag{3}$$

Similarly, if $\ell(\mathcal{S})$ already exceeds the maximum value in the candidate set, the computation for the entire set $\mathcal{S}$ can be safely skipped.

Details of pruning. So far, all our discussion has been limited to one specific historical query labeled *old*. In fact, the search pruning can benefit from multiple historical queries (i.e., all $old \in [1, new-1]$), and the only additional calculation required is for the new query q_{new} and each historical query q_{old}. For multiple historical queries, we can get multiple corresponding lower bounds $\ell(\mathcal{S})$ according to Equation (3); the minimum $\ell(\mathcal{S})_{min}$ will be chosen as the final tight lower bound for a more efficient pruning. If this $\ell(\mathcal{S})_{min}$ already exceeds the largest distance in the top-k candidate set, the distance calculation of the given set can be safely excluded.

More specifically, since each historical query corresponds to a different set $\mathcal{S}$ (as the cursor keeps advancing), we need a data structure to organize multiple lower bounds of the overlapping parts across these queries, and then calculate the minimum. We will discuss this issue in §3.3.2.

[2] Our current pruning strategy targets similarity metrics that satisfy the triangle inequality, such as Euclidean distance and cosine similarity. We leave the extension of pruning on other metrics (e.g., inner product) as future work.

Correctness of pruning. It is worth noting that our history-guided search pruning does not introduce any loss in the recall for BF-component, as Equation (2) provides a strict lower bound $\ell(v)$ on distance of the vector v. When this bound already exceeds the largest distance in the top-k candidate set, this vector v can be safely excluded.

3.3.2 Hierarchical Pruning with HistTree

According to §3.3.1, for any historical query q_{old} and any given vector set $\mathcal{S}$ (where the distances between q_{old} and every vector in $\mathcal{S}$ are known), the corresponding lower bound $\ell(\mathcal{S})$ can be obtained according to Equation (3). A historical query can generate multiple $\mathcal{S}$ and $\ell(\mathcal{S})$ for pruning.

A natural question is how to determine the size of the set $\mathcal{S}$. If $\mathcal{S}$ is too coarse-grained (in the extreme case, the entire BF-component), the lower bound $\ell(\mathcal{S})$ will be too loose, which hinders effective pruning. Conversely, if $\mathcal{S}$ is too fine-grained, although pruning opportunities increase, a single query will generate too many sets $\mathcal{S}$, and multiple historical queries will further fragment them, resulting in significant overhead from numerous set intersection operations.

To address this issue, we design a *hierarchical pruning scheme*, which recursively partitions the entire BF-component using a tree structure named *HistTree*. Each tree node corresponds to the set $\mathcal{S}$ of a historical query, which brings a pruning opportunity. Node n_c is a child of another node n_p if and only if the set $\mathcal{S}_c$ corresponding to node n_c is a subset of the set $\mathcal{S}_p$ corresponding to node n_p. Thus, if an upper-level node n is pruned, the entire subtree rooted at n is also pruned.

Our method avoids the problem of the $\mathcal{S}$ granularity. Upper-level nodes correspond to sets with more vectors (looser bounds); once pruning occurs at this level, the benefit is more substantial. Conversely, lower-level nodes correspond to sets with fewer vectors (tighter bounds), offering more pruning opportunities.

Basic structure of HistTree. The HistTree organizes the unindexed vectors (BF-component) in a tree where each tree node t stores partial historical information and brings a chance that could be pruned. More specifically, as shown in Figure 6, an inner node maintains: 1) the associated historical query (we call it *pivot vector*), 2) the bounds d_{max} and d_{min}, and 3) the $\mathcal{S}$ set[3]. For leaf nodes, in addition to maintaining the same information as inner nodes, it is also necessary to store the distances between all vectors in set $\mathcal{S}$ and the pivot vector. These distances can be obtained from historical information. All leaf nodes form a partition of all vectors in BF-component, where each pair of leaf nodes is mutually disjoint.

Next, we present how we retrieve nearest neighbors on BF-component with HistTree, and how the HistTree grows with new query information to bring more pruning opportunities.

Serving nearest neighbor search. Since the HistTree provides a hierarchical decomposition of BF-component, the search process traverses the HistTree and retrieve nearest neighbors by efficiently pruning large portions of the search space. At each visited node, the pruning condition from Equation (3) is evaluated. If the calculated lower bound $\ell(\mathcal{S})$ for the node's vector set $\mathcal{S}$ exceeds the largest distance in the current top-k candidate set, the entire subtree rooted at this node is pruned from the search. If the traversal reaches a leaf node and it cannot be pruned, FleetANN performs brute-force search over its stored vectors in $\mathcal{S}$. Upon completion of the tree traversal, the candidate set constitutes the final results.

Node splitting in HistTree. HistTree grows dynamically as user queries are processed. With more queries, accumulated historical distance information deepens the HistTree and creates additional pruning opportunities. The evolution of the tree is governed by node splitting. Specifically, whenever a new query arrives and is routed to a leaf node, a decision is made on whether to split the node according to predefined criteria (e.g., when the cardinality of $\mathcal{S}$ in the leaf exceeds a threshold). If a split occurs, the leaf node is converted into an inner node with two new children leaf nodes. The new query and the original node's pivot become the pivots for these two new children, respectively. Each vector in the original $\mathcal{S}$ is assigned to the child node whose pivot is closest to it; the relative proximity to the children's pivots can be efficiently derived using historical information. Also, two children's corresponding d_{max} and d_{min} bounds are also updated accordingly. Through this adaptive splitting, the tree incrementally refines its partitions in response to the query workload, thereby enhancing the effectiveness of the pruning strategy over time.

Optimization. For ease of understanding, the above description assumes that all inner nodes store the vector set $\mathcal{S}$. In practice, this is unnecessary: distance computations between vectors are performed only at leaf nodes, while inner-node operations rely solely on d_{min} and d_{max} to do pruning. This can also be understood intuitively: pruning at an inner node only requires the hypersphere defined by the pivot and d_{min}, d_{max}, without knowledge of the points inside. Thus, the vector sets at inner nodes can be omitted to reduce the memory footprint of the HistTree. With this optimization, the whole HistTree is extremely memory-efficient (e.g., only 8MB for a million-scale dataset).

Example. Figure 6 shows an concentrate example (finding top-2). It contains 6 vectors in the dataset ($v_1 - v_6$) and serves 3 queries ($q_1 - q_3$).

a) For query q_1, ❶ initially, the HistTree has only one single root node that also serves as a leaf, containing all vectors in BF-component. ❷ Upon the arrival of query q_1, since no historical distance information is available at this point, no pruning occurs. FleetANN performs brute-force distance computations between q_1 and each v, and returns the top-2 results. ❸ Subsequently, q_1 is designated as the pivot for N_1. The distances from this pivot to each point (d_{11}-d_{16}) are stored to facilitate future pruning. d_{min} and d_{max} are also updated.

[3] $\mathcal{S}$ can be further eliminated with the optimization in §3.3.2; we retain it now to facilitate understanding.

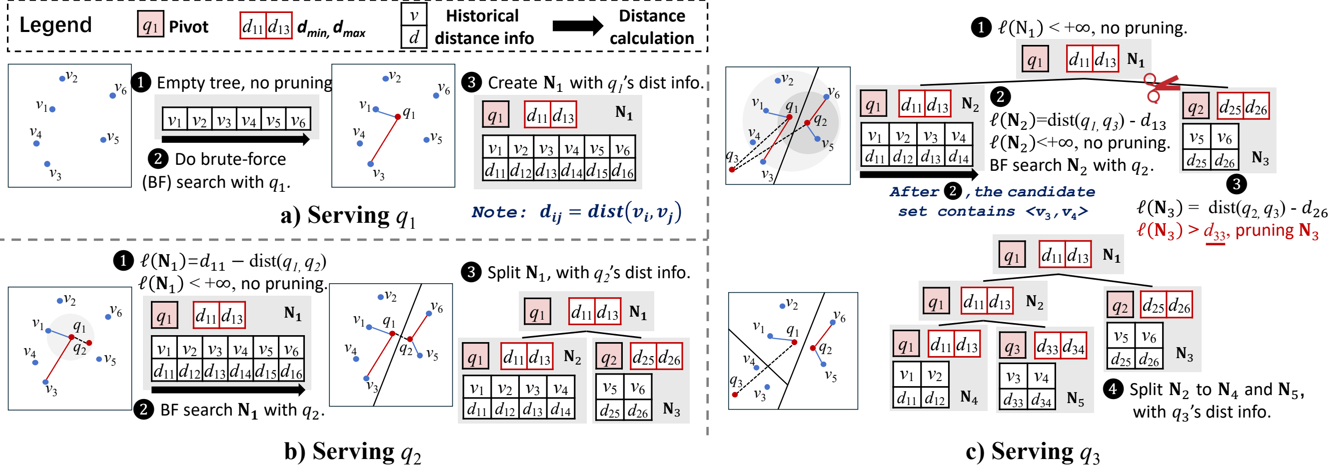


Figure 6: Example of history-guided search pruning. *We try to find top-2 vectors of each query. d_{xy} denotes the distance between q_x and v_y. The blue and red line shows d_{min} and d_{max} respectively. The bold black arrows indicate all operations involving distance computations. It is noteworthy that all structure modification operations, including **a❸**, **b❸**, **c❹**, do not require any distance computation. The red underlined $\underline{d_{33}}$ denotes the distance of the top element (i.e., v_3) in the candidate set.*

Note that the creation of N_1 does not incur any distance calculation; it only needs to reuse the distance computation results obtained from q_1.

b) For q_2, ❶❷ is similar to q_1. ❸ Since the cardinality of N_1 has reached the threshold, it is split to N_2 and N_3 by partitioning points based on proximity to q_2 versus the existing pivot q_1. The original N_1 is converted into an internal node: its v and d arrays are split to N_2 and N_3, while retaining only its d_{min} and d_{max} values for a coarse-grained prune. From a visualization perspective, this split can be interpreted as a partitioning of the hyperspace associated with N_1.

c) For q_3, ❶❷ is similar to q_1 and q_2. After do the BF search in N_2, the candidate set contains v_3 and v_4. ❸ Thus, we can find that $\ell(N_3)$, the lower bound of N_3, exceeds the distances in the current candidate set, leading to pruning of the entire N_3. q_3's ❹ is similar to ❸ of q_2.

3.4 Advancing the Cursor

In the background, FleetANN gradually builds the ANN index by advancing the cursor (migrating vectors from BF-component to I-component). Implementing such progressive index construction involves three steps: 1) efficiently delete vectors from HistTree, since BF-component is implemented as HistTree, 2) efficiently insert to I-component, and 3) ensure consistency between two components, preventing temporary inaccessibility of vectors during relocation.

We begin by describing the first two steps individually, and subsequently elaborate on how they are coordinated to guarantee consistency.

Deleting vectors from BF-component (i.e., HistTree). FleetANN employs a batch-deletion strategy to both enhance insertion parallelism in I-component and amortize the structural maintenance overhead of the HistTree. The maximum number of nodes migrated during each operation is a tunable hyperparameter; our evaluation identifies 1% of the dataset as the sweet spot.

The deletion process can be broken down into three phases: 1) select to-be-migrated vectors, 2) remove them and update two bounds, and 3) restructure the tree.

First, to select vectors for deletion and migration, we observe that vectors near historical query points are more likely to be retrieved by future queries (i.e., hotspots). By prioritizing these hotspots for early insertion to I-component, we achieve better overall performance. Thus, we always migrate vectors from the leaf nodes recently visited by user queries.

Second, when deleting vectors from leaf nodes, we use mutex locks to prevent conflicts with concurrent node-splitting operations. If a deleted vector previously defined the node's d_{min} and d_{max}, we also need update these two bounds.

Third, if a leaf node becomes empty after deletion, it should be removed to maintain the tree structure. The empty leaf, its parent, and its sibling are compacted into a new node containing the original sibling's content. All structural modifications are protected by node-level locks. Note that this compaction does not entail actual data movement, but merely lightweight metadata operations on pointers.

Inserting vectors to I-component. Since our system requires the ANN index in I-component to support incremental insertions, concurrency safety between insertions and queries is guaranteed by the index itself. Notably, widely used ANN indexes, including cluster-based (e.g., SPFresh [9]) and graph-based indexes (e.g., Vamana [11]), commonly support concurrent incremental insertions.

Maintaining consistency between I-component and BF-component. The above two parts describe in detail how vectors are deleted from BF-component and how they are inserted

into I-component. However, the background migration must be carefully coordinated with foreground search to ensure data consistency. To this end, background threads always insert a migrating vector into I-component before deleting it from BF-component. On the search side, foreground queries are required to probe BF-component first and then search I-component, after which the two top-k result sets are merged. This ordering guarantees that no migrating vector becomes invisible: a query may see the same vector in both components, but never in neither. Any duplicates that appear across the components are eliminated during the final merge.

4 Evaluation

4.1 Experimental Setup

Testbed. All experiments are conducted on a server equipped with Intel Xeon Gold 6242 CPU, 374 GB DRAM. It is also equipped with 4 NVIDIA H20 GPUs for neural network tasks (such as generating embeddings or running LLMs). For all experiments, we use LLama-3.2 8B for LLM.

Compared systems. We compare FLEETANN against different systems, including BruteOnly and two mainstream EagerBuild indexes (one cluster-based and one graph-based).

- *BruteOnly*: Brute force search adopted by Faiss.
- *HNSW* [37]: A popular graph-based index (HNSW) implemented in Faiss. It natively supports incremental insertion.
- *SPFresh* [9]: A updatable cluster-based index based on SPANN [19]. We use its built-in DRAM-only version.

We also evaluate variants of HNSW and SPFresh integrated with the ThresholdGuard strategy, denoted as *HNSW-T* and *SPFresh-T*. All these strategies are widely-used strategies in existing ANNS or RAG systems. For example, EagerBuild is used by OpenWebUI [30] and LangFlow [31]. BruteOnly is used in work [3]. ThresholdGuard is a strategy recommended in the official Faiss guidelines [37].

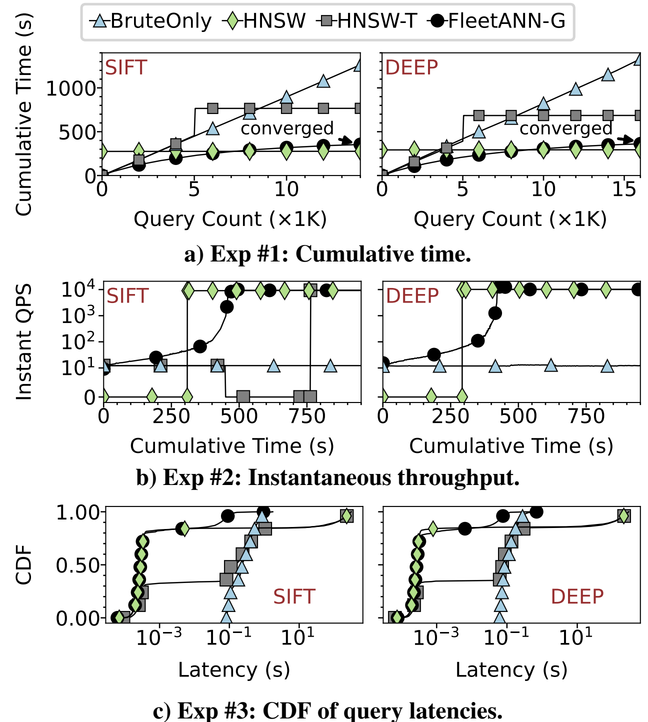
We integrate two indexes (HNSW and SPFresh) as the I-component of FLEETANN and create two FLEETANN variants, named FLEETANN-G (graph-based) and FLEETANN-C (cluster-based) respectively. Our work can also be applied to other updatable indexes, with similar findings.

Datasets. For microbenchmark, since ephemeral ANNS typically targets relatively small-scale datasets, we adopt the following two widely-used 1-million-scale datasets:

- SIFT-1M [8]: It consists of 1 million 128-dimensional vectors of 32-bit floats. The k value of top-k is 10.
- DEEP-1M [40]: It consists of 1 million 96-dimensional vectors of 32-bit floats. k is also 10.

Aside from microbenchmark, we also test 4 applications of empheral ANNS in §4.5, including AI chatbots, code assistant, data lakes and agentic workflows. The datasets in different scenarios will be detailedly described in §4.5 respectively.

Metrics. We compare different systems using four metrics: instantaneous throughput, per-query latency, cumulative time, and accuracy (recall@10). Among them, cumulative time



a) Exp #1: Cumulative time.

b) Exp #2: Instantaneous throughput.

c) Exp #3: CDF of query latencies.

Figure 7: Microbenchmark.

measures the total processing time for a given number of queries. Recall@10 captures the recall of the top 10 results.

Due to space limit and to ensure the clarity of all figures, we primarily present the results of graph-based indexes (HNSW, HNSW-T, and FLEETANN-G). Cluster-based indexes exhibit similar findings, and §4.4 will provide a detailed description of them. Unless otherwise specified, FLEETANN refers to FLEETANN-G, and the default dataset used is SIFT.

4.2 Microbenchmark

Since ephemeral ANNS exhibits extremely high variability in query volume, to evaluate system performance under different query loads, we continuously issue query requests to each system and record various metrics at different query counts.

(Exp #1): Cumulative time. Figure 7a shows the cumulative time of different systems on SIFT-1M and DEEP-1M. We can make the following observations:

1) As expected, BruteOnly and EagerBuild (HNSW) respectively perform well only at low and high query counts, while FLEETANN adapts to varying query volumes. At low volumes (e.g., <100 queries), FLEETANN amortizes the index construction cost across the query stream, enabling immediate query processing similar to BruteOnly, whereas HNSW incur a hundred-second stall for construction before serving queries. In high-volume scenarios (e.g., 8,000 queries), its performance approaches that of EagerBuild, making it significantly faster than BruteOnly. These results highlight

FLEETANN's efficiency for ephemeral ANNS.

2) Even at low query volumes, FLEETANN outperforms BruteOnly due to its progressive construction and history-guided pruning mechanism. By eliminating massive vector computations (see Exp #5 for techniques' contribution), FLEETANN fully covers the additional index construction overhead.

3) At high query volumes, FLEETANN's throughput progressively converges to the performance of EagerBuild. However, it still falls short of EagerBuild, primarily because the early phase of FLEETANN incurs heavy brute-force computation, during which its retrieval latency is inevitably higher than that of index-based search.

4) Although ThresholdGuard (HNSW-T) achieves a certain degree of adaptivity, it yields the longest end-to-end time. This is due to its delayed responsiveness, which suffers from both the overhead of brute-force search in the early stage and index construction in the later stage.

(Exp #2): Instantaneous throughput. Figure 7b shows the instantaneous throughput of different systems. Except in the early stage of queries, where FLEETANN shows slightly lower throughput than BruteOnly and ThresholdGuard (HNSW-T) due to overhead of inserting to the I-component, FLEETANN consistently surpasses all baselines.

BruteOnly sustains constant but low throughput, limited by its fixed per-query cost. ThresholdGuard (HNSW-T) initially matches BruteOnly, drops to zero during index construction, and then converges to EagerBuild (HNSW) once the index is ready. Compared with EagerBuild, FLEETANN maintains throughput even while construction is ongoing, and eventually converges to the same level as EagerBuild. Through progressive construction, its throughput gradually increases as the build progresses. By interleaving construction with query processing, FLEETANN achieves robust, adaptive throughput across workloads, a key advantage in ephemeral search.

(Exp #3): Latency distribution. Figure 7c presents the latency CDF. FLEETANN demonstrates the most favorable latency profile. EagerBuild and ThresholdGuard experience orders of magnitude higher tail latency (over 300s) due to the initial construction stall. While BruteOnly provides stable latency (~1.7s), most of its queries are processed more slowly than FLEETANN (~0.9s). We can also observe that: in a small fraction of cases (<7%), FLEETANN incurs slightly higher latency (15%) than BruteOnly, due to the additional index construction overhead.

(Exp #4): Recall. Figure 8 shows the recalls of different systems on SIFT; DEEP has a similar result. Since ThresholdGuard (HNSW-T/SPFresh-T) is a combination of BruteOnly and EagerBuild (HNSW/SPFresh), its result is omitted. BruteOnly consistently achieves perfect recall (100%) by scanning the entire dataset. FLEETANN (both -G/-C), while slightly below BruteOnly, consistently matches or outperforms the EagerBuild indexes. This experiment also empirically validates the hybrid retrieval's completeness, as proven in §3.2.

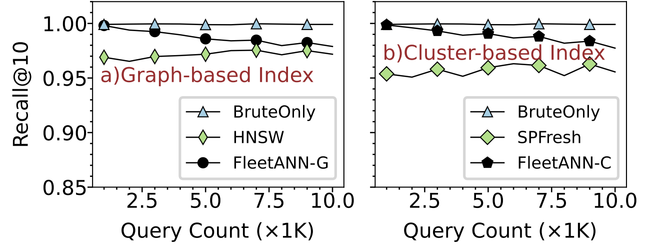


Figure 8: (Exp #4) Recall on two types of indexes.

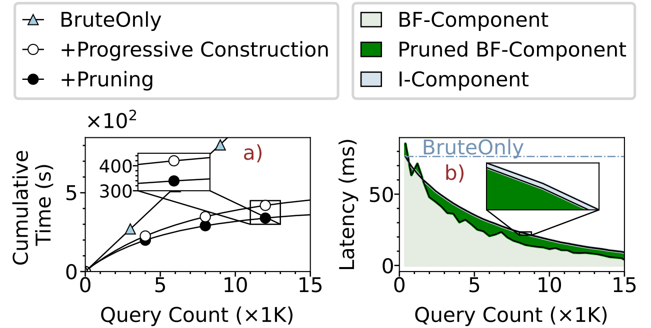


Figure 9: (Exp #5) Breakdown analysis. *a) shows the contribution of each technique. b) breaks down the query latency into the latencies of two components in* FLEETANN.

4.3 Techniques

(Exp #5): Breakdown analysis. We breakdown the performance gap between BruteOnly and FLEETANN, to present the contributions of each technique. Figure 9a shows the result. Techniques are added cumulatively. Figure 9b also further decomposes the query time of two components in FLEETANN.

- *BruteOnly*. With a time complexity of $O(N)$, brute-force search incurs high query latency and cumulative time.
- *+Progressive construction*. Since ANNS scales sub-linearly, by adding progressive construction, the ANN index absorbs the exhaustive BF computation (interpreted as the area below the dashed blue line), reducing the end-to-end cumulative time by up to 4.5x. As queries accumulate, the time contributed by the BF-component progressively decreases, leading to a corresponding reduction in the total query time. However, before the index is fully built, the BF-component still tends to dominate—for example, at 8,000 queries it contributes over 98%. This indicates the necessity of our pruning.
- *+History-guided search pruning*. By further adding the pruning technique, a significant proportion (in average 29%) of BF component is pruned; it can be visualized as the dark green area in Figure 9b. As a result, pruning also reduces the cumulative processing time by 1.3x further.

(Exp #6): Effect of history-guided search pruning. Figure 10 shows the number of distance computations in BF-component with or without pruning. We can see that over

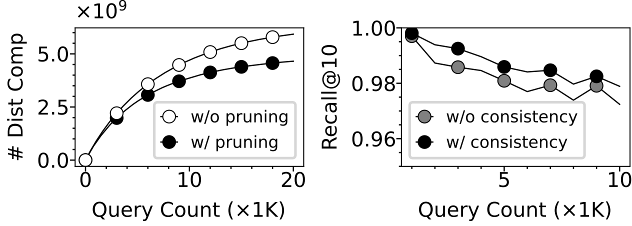


Figure 10: (Exp #6) Effect of history-guided pruning.

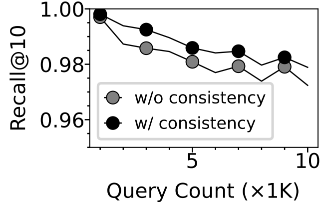


Figure 11: (Exp #7) Effect of maintaining consistency.

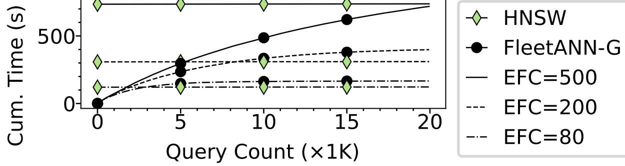


Figure 12: (Exp #8) Different configurations of HNSW.

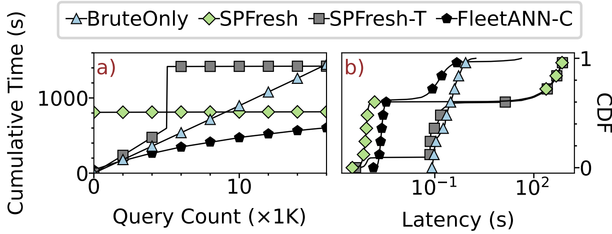


Figure 13: (Exp #9) Results of cluster-based indexes.

21% of distance calculations are pruned. With a growing number of queries, the amount of exploitable historical distance information also increases, thereby enabling a higher pruning rate (some queries can even prune 90%).

(Exp #7): Effect of cursor advancing consistency. As the cursor advances, vectors are continuously moved from BF-component to I-component. We evaluate an extreme case where the moving vectors are all missed, as illustrated in Figure 11. It would lead to over 2% (significant) degraded recall. This highlights the importance of our two-phase migration.

4.4 Different I-component configurations

(Exp #8): Different configurations of graph-based I-component. Figure 12 characterizes the impact of I-component's internal configurations in FLEETANN-G on the performance. We vary efConstruction (EFC), a pivotal parameter governing the quality of constructed HNSW. With increasing graph construction complexity, FLEETANN-G demonstrates a wider margin of performance advantage, stemming from HNSW's increased startup stall time.

(Exp #9): Cluster-based I-component. To demonstrate FLEETANN's generality, we apply it to SPFresh, a SOTA cluster-based index, and get the variant FLEETANN-C. Figure 13 shows the cumulative time and query latency CDF of SPFresh, SPFresh-T, and FLEETANN-C. The results mirror those ob-

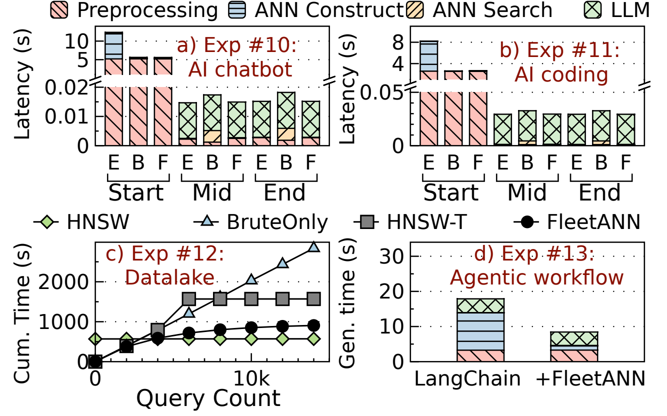


Figure 14: Real ephemeral ANNS application evaluation. *E:EagleBuild, B:BruteOnly, F:*FLEETANN. *Preprocessing involves embedding documents or queries into vectors.*

served with FLEETANN-G: FLEETANN-C effectively eliminates the initial construction stall and achieves performance comparable to SPFresh. However, different from FLEETANN-G, FLEETANN-C incurs slightly higher latency than SPFresh (EagerBuild). This is because eager built SPFresh utilizes BKT [19] to compute a near-optimal clustering partition scheme (with a longer initialization time), whereas the partition scheme obtained through incremental insertions deviates from this optimal scheme. We observe that this problem can be mitigated by allowing SPFresh's background threads to perform reassignment after a period. In cumulative time, FLEETANN-C still outperforms both BruteOnly and SPFresh.

4.5 Application Performance

(Exp #10): AI chatbots. We embed FLEETANN into a famous AI chatbot framework, OpenWebUI [30], and evaluate it using the Enron [41] email dataset. We do email-related Q&A by retrieving top-5 related emails. In real-world scenarios, queries typically arrive at irregular intervals, making it inappropriate to continuously issue queries as in microbenchmarks. Thus, we focus three representative queries: Start (the first query), Mid (when I-component is halfway built), and End (after I-component construction). Figure 14a shows the TTFT results. EagerBuild involves significant initial overhead (~7s). Although BruteOnly avoids this initial cost, its ANNS component becomes expensive (~23%) for the mid and end queries. In contrast, FLEETANN exhibits consistently superior performance from start to end.

(Exp #11): AI-assist coding. We plug FLEETANN as the retrieval component in CodeRAG-Bench [26]. This benchmark contains 1000 numpy or pandas related questions based on code snippets crawled from Stack Overflow. For each user question, it finds the top-5 relevant snippets to generate codes. Figure 14b shows the breakdown of TTFT. In code generation scenarios, the context length is longer, leading to a higher

proportion of LLM. Nevertheless, FleetANN achieves good latency across three representative queries.

(Exp #12): Exploratory search in datalake. We use the LAION-1M [42] dataset, which encodes 1 million images and texts into vectors using the OpenAI CLIP model [43]. For evaluation, we conduct multiple exploratory searches by retrieving top-10 nearest images for each query text. Figure 14c shows that FleetANN achieves 2–4x lower cumulative time than BruteOnly. The basic findings are similar as microbenchmark. We make further observations: FleetANN shows higher cumulative time than EagerBuild at large query counts. This is because CLIP produces higher-dimensional vectors, making the early brute-force computations more expensive.

(Exp #13): Agentic workflows. We take research workflow as a representative case. Since OpenAI Deep Research is closed-source, we use an open-source variant [44] built on LangChain [45] to reproduce a similar workflow. This workflow first builds a vector store (HNSW) by collecting 100 papers related to a given topic, and then iteratively generates research questions, retrieves relevant content, summarizes findings, and reflects on whether to proceed to the next round, ultimately producing a research report. Figure 14d shows the final report generation time. Compared with the default EagerBuild policy, FleetANN can reduce the end-to-end time by 49%, due to the reduced time of ANNS-related time.

5 Limitations

1) *Performance with extremely high/low query volume*. While FleetANN adapts well to varying query volumes, its performance is slightly inferior when the query count is extremely high/low (e.g., >8k or <10). This limitation arises from the unavoidable overhead (although is alleviated by pruning) of the initial brute-force phase. Consequently, for workloads with extremely high/low query volumes known a priori, EagerBuild/BruteOnly may yield better performance.

2) *Index compatibility*. FleetANN currently requires the underlying index of I-component to natively support incremental insertions. Although mainstream indexes (HNSW, IVF and DiskANN) support this prerequisite, still some indexes (like NSG [46]) do not. Broadening the framework to accommodate all variety of index types remains our future work.

6 Related Work

We categorize related work into two primary areas: accelerating index construction and accelerating index query.

Accelerating index construction. Research in accelerating index construction can be broadly divided into two paradigms: 1) optimizing construction for existing widely-used index structures (e.g., HNSW, IVF), and 2) customizing new index structures that facilitates efficient construction.

The first paradigm retains existing ANN index structures while accelerating their construction. Flash [47] optimizes a compact coding strategy to cater to modern CPU SIMD architecture and fasten distance calculation. LSH-APG [12] leverages an additional lightweight hash-based index (e.g., LSH) to quickly identify superior entry points of the original graph-based index and reduce costly graph navigation. Other works [48–50] leverage GPUs' massive parallelism to accelerate index construction.

The second paradigm focuses on developing entirely new index structrues. Inspired by RNG [51], NSG [46] introduces Monotonic RNG and lowers its indexing complexity by relaxing the strictness of graph connectivity. DiskANN [11] introduces Vamana, a graph variant complemented by a pruning strategy meticulously optimized for disk-based access patterns. CAGRA [21] tailors GPU-friendly index structures from scratch to maximize GPU utilization for construction.

Though with these innovative techniques, the construction stall of an ANNS index is still costly. Such circumstances underscore the significance of FleetANN.

Accelerating index query. Prior efforts to accelerate ANN queries fall into two categories: hardware-aware system designs and algorithmic optimizations.

Given a fixed index algorithm, many works leverage heterogeneous hardware to speed up. GPUs are widely adopted due to massive parallelism and high memory bandwidth [16, 17, 20, 22, 49, 52–54]. Among them, BANG [20] is constrained by the limited capacity of GPU memory. RUMMY [22] and PilotANN [16] extend it to the host DRAM by minimizing data swap, while FusionANNS [17] further pushes the boundary by enabling GPU access to indices stored on SSDs. Beyond GPUs, existing works also explore domain-specific hardware to accelerate query. Examples include FPGA [23, 55], RTX cores [56], and near-storage computing devices [18, 24, 57, 58]. These works are orthogonal to FleetANN.

Another line of work employs heuristic algorithms to reduce query time. METIS [59] adapts RAG configuration knobs to speedup per-query RAG time. CrackIVF [29] leverages the observation that more clusters can lower the query latency, and heuristically splits more clusters to adapt to a query-heavy workload. SPANN [19], LAET [60], Auncel [61] and Quake [15] dynamically prunes the probing of some clusters based on heuristic rule [19], learned cost model [15, 60] or geometric model [61]. Such studies face two limitations: 1) they lack theoretical guarantees of recalls, and 2) they typically target a specific index type, often necessitating a redesign of its internal structures. In contrast, FleetANN preserves recalls while remaining general to any index type; the only requirement is supporting incremental insertions, a capability commonly provided by mainstream indexes.

7 Conclusion

This paper identifies and addresses the problem of ephemeral ANNS, an emerging ANNS scenario in modern AI applications, where traditional ANNS methods fall short. We present FleetANN, a system that embracing the progressive index construction paradigm. Extensive evaluation demon-

strates that FleetANN outperforms traditional strategies on microbenchmark and real applications.

References

[1] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry P. Heck. Learning deep structured semantic models for web search using clickthrough data. In Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi, editors, *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 2333–2338. ACM, 2013.

[2] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology*, 15(3):1–45, 2024.

[3] Derrick Quinn, Mohammad Nouri, Neel Patel, John Salihu, Alireza Salemi, Sukhan Lee, Hamed Zamani, and Mohammad Alian. Accelerating retrieval-augmented generation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 15–32, New York, NY, USA, 2025. Association for Computing Machinery.

[4] Yanhao Zhang, Pan Pan, Yun Zheng, Kang Zhao, Yingya Zhang, Xiaofeng Ren, and Rong Jin. Visual search at alibaba. *CoRR*, abs/2102.04674, 2021.

[5] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. Embedding-based retrieval in facebook search. In Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash, editors, *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 2553–2561. ACM, 2020.

[6] Jiarui Li, Ye Yuan, and Zehua Zhang. Enhancing LLM factual accuracy with RAG to counter hallucinations: A case study on domain-specific queries in private knowledge-bases. *arXiv preprint arXiv:2403.10446*, 2024.

[7] Shailja Gupta, Rajesh Ranjan, and Surya Narayan Singh. A comprehensive survey of retrieval-augmented generation (RAG): Evolution, current landscape and future directions. *arXiv preprint arXiv:2410.12837*, 2024.

[8] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864. IEEE, 2011.

[9] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. SPfresh: Incremental in-place update for billion-scale vector search. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 545–561, 2023.

[10] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.

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

[12] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *Proceedings of the VLDB Endowment*, 16(8):1979–1991, 2023.

[13] Ishita Doshi, Dhritiman Das, Ashish Bhutani, Rajeev Kumar, Rushi Bhatt, and Niranjan Balasubramanian. LANNS: a web-scale approximate nearest neighbor lookup system. *arXiv preprint arXiv:2010.09426*, 2020.

[14] Xiaoyao Zhong, Haotian Li, Jiabao Jin, Mingyu Yang, Deming Chu, Xiangyu Wang, Zhitao Shen, Wei Jia, George Gu, Yi Xie, et al. VSAG: An optimized search framework for graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2503.17911*, 2025.

[15] Jason Mohoney, Devesh Sarda, Mengze Tang, Shihabur Rahman Chowdhury, Anil Pacaci, Ihab F Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. Quake: Adaptive indexing for vector search. *arXiv preprint arXiv:2506.03437*, 2025.

[16] Yuntao Gui, Peiqi Yin, Xiao Yan, Chaorui Zhang, Weixi Zhang, and James Cheng. PilotANN: Memory-bounded GPU acceleration for vector search. *arXiv preprint arXiv:2503.21206*, 2025.

[17] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Hai Jin, Xuecang Zhang, Junhua Zhu, and Yu Zhang. Towards high-throughput and low-latency billion-scale vector search via CPU/GPU collaborative filtering and re-ranking. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 171–185, 2025.

[18] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. Scalable billion-point approximate

nearest neighbor search using SmartSSDs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1135–1150, 2024.

[19] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems*, 34:5199–5212, 2021.

[20] Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, Jyothi Vedurada, et al. BANG: Billion-scale approximate nearest neighbor search using a single gpu. *arXiv preprint arXiv:2401.11324*, 2024.

[21] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. Cagra: Highly parallel graph construction and approximate nearest neighbor search for GPUs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 4236–4247. IEEE, 2024.

[22] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast vector query processing for large datasets beyond GPU memory with reordered pipelining. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 23–40, 2024.

[23] Wei Yuan and Xi Jin. FANNS: An FPGA-based approximate nearest-neighbor search accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2025.

[24] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. CXL-ANNS: Software-hardware collaborative memory disaggregation and computation for billion-scale approximate nearest neighbor search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 585–600, Boston, MA, July 2023. USENIX Association.

[25] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[26] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. CodeRAG-bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497*, 2024.

[27] ChatGPT. https://chatgpt.com/. [Accessed 20-06-2025].

[28] Introducing deep research | openai. [Online; accessed 2025-05-26].

[29] Vasilis Mageirakos, Bowen Wu, and Gustavo Alonso. Cracking vector search indexes. *arXiv preprint arXiv:2503.01823*, 2025.

[30] GitHub - open-webui/open-webui: User-friendly AI Interface. https://github.com/open-webui/open-webui. [Accessed 20-06-2025].

[31] Langflow AI Team. Langflow: UI for LangChain. https://github.com/langflow-ai/langflow, 2023. Accessed: 2024-05-24.

[32] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, September 1977.

[33] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[34] Rihan Hai, Christoph Quix, and Matthias Jarke. Data lake concept and systems: a survey. *CoRR*, abs/2106.09592, 2021.

[35] Zilliz. Deepresearcher, 2024.

[36] Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. Agentic retrieval-augmented generation: A survey on agentic RAG. *CoRR*, abs/2501.09136, 2025.

[37] Facebook AI Research. Guidelines to choose an index. https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index, 2022. Accessed: 2024-05-24.

[38] Jiongkang Ni, Xiaoliang Xu, Yuxiang Wang, Can Li, Jiajie Yao, Shihai Xiao, and Xuecang Zhang. DiskANN++: Efficient page-based search over isomorphic mapped graph index using query-sensitivity entry vertex. *arXiv preprint arXiv:2310.00402*, 2023.

[39] Gilbert Strang. *Introduction to linear algebra*. SIAM, 2022.

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

[41] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *European conference on machine learning*, pages 217–226. Springer, 2004.

[42] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. Laion-400M: Open dataset of clip-filtered 400 million image-text pairs. *arXiv preprint arXiv:2111.02114*, 2021.

[43] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR, 2021.

[44] langchain-ai/local-deep-researcher: Fully local web research and report writing assistant. [Online; accessed 2025-08-01].

[45] Langchain. [Online; accessed 2025-08-01].

[46] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

[47] Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Yifan Zhu, and Wenchao Zhou. Accelerating graph indexing for anns on modern cpus. *Proceedings of the ACM on Management of Data*, 3(3):1–29, 2025.

[48] Hui Wang, Wan-Lei Zhao, Xiangxiang Zeng, and Jianye Yang. Fast k-NN graph construction by GPU based nn-descent. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, CIKM '21, page 1929–1938, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. GGNN: Graph-based gpu nearest neighbor search. *IEEE Transactions on Big Data*, 9(1):267–279, February 2023.

[50] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. GPU-accelerated proximity graph approximate nearest neighbor search and construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 552–564, 2022.

[51] Godfried T Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern recognition*, 12(4):261–268, 1980.

[52] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[53] Jia Pan and Dinesh Manocha. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 211–220, 2011.

[54] Haodi Jiang, Hao Guo, Minhui Xie, Jiwu Shu, and Youyou Lu. High-throughput, cost-effective billion-scale vector search with a single GPU. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '26. ACM, 2026.

[55] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, et al. DF-GAS: a distributed FPGA-as-a-service architecture towards billion-scale graph-based approximate nearest neighbor search. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 283–296, 2023.

[56] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. Juno: Optimizing high-dimensional approximate nearest neighbour search with sparsity-aware algorithm and ray-tracing core mapping. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 549–565, 2024.

[57] Puqing Wu, Minhui Xie, Enrui Zhao, Dafang Zhang, Jing Wang, Xiao Liang, Kai Ren, and Yunpeng Chai. Turbocharge ANNS on real processing-in-memory by enabling fine-grained per-pim-core scheduling. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 1223–1241, 2025.

[58] Shengwen Liang, Ying Wang, Ziming Yuan, Cheng Liu, Huawei Li, and Xiaowei Li. VStore: in-storage graph based vector search accelerator. In *Proceedings of the 59th ACM/IEEE DesiND Automation Conference*, pages 997–1002, 2022.

[59] Siddhant Ray, Rui Pan, Zhuohan Gu, Kuntai Du, Shaoting Feng, Ganesh Ananthanarayanan, Ravi Netravali, and Junchen Jiang. METIS: fast quality-aware RAG systems with configuration adaptation. In Youjip Won, Youngjin Kwon, Ding Yuan, and Rebecca Isaacs, editors, *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*, pages 606–622. ACM, 2025.

[60] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving approximate nearest neighbor search through learned adaptive early termination.

In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2539–2554, New York, NY, USA, 2020. Association for Computing Machinery.

[61] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. Fast, approximate vector queries on very large unstructured datasets. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 995–1011, Boston, MA, April 2023. USENIX Association.