PANNS: Enhancing Graph-based Approximate Nearest Neighbor Search through Recency-aware Construction and Parameterized Search

Xizhe Yin
University of California, Riverside
Riverside, California, USA
xyin014@ucr.edu

Chao Gao
University of California, Riverside
Riverside, California, USA
cgao037@ucr.edu

Zhijia Zhao
University of California, Riverside
Riverside, California, USA
zhijia@cs.ucr.edu

Rajiv Gupta
rajivg@ucr.edu
University of California, Riverside
Riverside, California, USA

Abstract

Approximate Nearest-Neighbor Search (ANNS) has become the standard querying method in vector databases, especially with the recent surge in large-scale, high-dimensional data driven by LLM-based applications. Recently, graph-based ANNS has shown improved throughput by constructing a graph from the dataset, with edges representing the distances between data points, and using best-first or beam search algorithms for query evaluation.

This work highlights that key aspects of the design space for both graph construction and search in graph-based ANNS remain under-explored. Specifically, the construction phase often neglects the potential temporal correlation between input queries and their results, while the search phase lacks a thorough exploration of beam search parameterization. To address these gaps, we present PANNS, a system that embeds temporal information in the proximity graph construction to capture query-result correlations. Moreover, it introduces a fully parameterized beam search algorithm, enabling extensive performance optimization. PANNS achieves up to a 3.7× improvement in query throughput over state-of-the-art graph-based ANNS methods, while maintaining equivalent recall levels. Furthermore, it reduces graph size by up to 30% without degrading query quality.

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**; • **Information systems** → **Retrieval tasks and goals**.

Keywords: nearest neighbor search, vector search, graph processing

ACM Reference Format:
Xizhe Yin, Chao Gao, Zhijia Zhao, and Rajiv Gupta. 2025. PANNS: Enhancing Graph-based Approximate Nearest Neighbor Search through Recency-aware Construction and Parameterized Search. In *The 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '25), March 1–5, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3710848.3710867

1 Introduction

Similarity search in high-dimensional datasets has become an important part of modern deep learning applications, including recommendation systems, retrieval augmented generation, content filtering, large language models (LLMs), and others. These datasets usually contain millions or billions of high-dimensional vector representations (a.k.a. embeddings) of documents, images, videos, and user content generated by pre-trained deep neural networks. The algorithm used to find vectors similar to user input from datasets is known as the k-nearest neighbor search, where the most similar k embeddings for a query are returned.

Searching for the exact k-nearest vectors can be very costly in a high-dimensional space, and most real-world applications can tolerate small errors. Therefore, *Approximate Nearest Neighbor Search* (ANNS) has been widely adopted in production environments, offering better query latency and throughput while maintaining good search quality.

Various approaches exist for solving the ANNS problem. Inverted File Indexing (IVF) methods [3, 29, 49] partition vectors into buckets, allowing queries to search a portion of the dataset instead of the entire space. More recently, graph-based ANNS solutions have demonstrated superior performance while achieving high recall. They construct a proximity graph from the vector dataset. Queries are then evaluated on the constructed graph using *beam search*, which terminates when all vertices in a fixed-size buffer (the beam) have been visited. Finally, the top-k vertices (vectors) are returned as the query results.

By carefully profiling and characterizing the workload of ANNS, we have identified two major limitations in existing ANNS solutions:

- ***Unexploited parameter space.*** For graph-based ANNS solutions, once the proximity graph is constructed, the recall-QPS (queries per second) tradeoff is controlled at a coarse-grained level with one parameter called *beam size*, which is the maximum queue size of the beam search. The larger the beam size, the higher the recall and the worse the QPS. For high query throughput, the parameter space of beam search is worth exploiting but has not been explored by existing works.
- ***Unexploited nearest neighbors correlation.*** In some real-world datasets (images or ad videos), user queries often exhibit consistent temporal distributions of their nearest neighbors over periods of time. For example, more recent data are more likely to be close to the query (the recency pattern), or nearest neighbors are more likely to appear in some specific period of time (the seasonal pattern). The correlation phenomenon was first described as "content drift" [6]. Unfortunately, such temporal information is not embedded in the vector and is not exploited for query optimization.

To address the above limitations, we propose PANNS[1], a graph-based ANNS system with parameterized search and learned query-data correlations, which improve both the graph construction and search efficiency.

First, we observed that the beam search can be roughly categorized into two phases according to the hardness of improving the recall score: **The first phase** is a brief period during which a substantial portion of accurate top-k nearest neighbors to the query vector are quickly identified (80% to 90% according to our profiling); **The second phase** is a significantly slower period that continues until the beam search algorithm terminates, that is, when no unvisited vertices remain in the queue. Since there is no explicit condition to guide early termination when a target recall score is achieved, the search continues, resulting in numerous vertex visits and distance computations during the second phase. On the other hand, the second phase yields only marginal improvements in the recall score, a behavior we describe as 'finding needles in a haystack' in the ANNS problem."

Based on the above observation, we propose to improve the search efficiency by introducing new parameters for the two phases, including *step sizes* that control the greediness of exploration at each iteration of the search (i.e., how many vertices are expanded per iteration) and a *cut-off factor* for discarding unpromising candidates. Since these two phases are artificially divided and exhibit different recall gain efficiencies (fast or slow), we assign distinct parameter values to each phase. Typically, the first phase requires a smaller step size (e.g., 1) to ensure convergence without compromising search quality. In contrast, the second phase can adopt a more aggressive approach with a larger step size. Similarly, the cut-off factor should vary between phases, distinguishing our approach from previous solutions [26, 39], which apply a uniform cut-off throughout the entire search.

Second, we demonstrate that it is possible to construct and refine the proximity graph by leveraging the temporal correlation between queries and their results. Specifically, we present a new algorithm for ANNS graph construction, where vertices are adaptively connected. By integrating correlation information, the algorithm produces a smaller and sparser graph while enabling faster navigation to the nearest neighbors. To the best of our knowledge, these techniques represent the first methods designed to optimize graph queries under skewed distributions.

Furthermore, we propose system-level optimizations with vector data rearrangement based on the query-data correlation. The rearrangement enables *truncated edge traversal*, effectively reducing memory footprint without harming the search quality. The edge list per vertex is sorted based on the query-data correlation during the graph construction phase. Such reordering enables the early termination at the iteration level in the search phase. Instead of scanning all out-going neighbors of a vertex, the traversal can stop early according to a threshold specific to the query vector.

Overall, this work makes the following contributions:

- We provide an in-depth analysis of the graph-based ANNS workload, highlighting its optimization space. Our findings enable a more detailed parameterization of the search algorithm.
- To incorporate hidden dimensions not embedded into vectors (e.g., temporal information), we propose a new proximity graph construction algorithm and a graph memory layout optimization.
- Our extensive experiments show that, to achieve the same recall level, PANNS can improve query throughput by up to 1.8× when query-data correlation exists, while reducing the constructed graph size by approximately 30%. Moreover, our techniques introduce little to no slowdown for queries that do not exhibit correlation with their results.

2 Background

2.1 Approximate Nearest Neighbor Search (ANNS)

We first introduce the problem definition of k-NNS. Given a dataset P of n points (vectors) in d-dimension space and a query point q, the k nearest neighbor search returns a set $\mathcal{K}$ that contains k points, such that $max_{p\in\mathcal{K}}||p,q|| \leq min_{p\in\mathcal{P}\setminus\mathcal{K}}||p,q||$. Note that the distance between two points $p,q \in \mathbb{R}^d$, denoted as $||p,q||$, can be either Euclidean distance (L_2 norm) or cosine distance. The Euclidean distance is often preferred for most real-world datasets.

[1] *P* stands for *praesens* (or *present*) and *parameterized.*

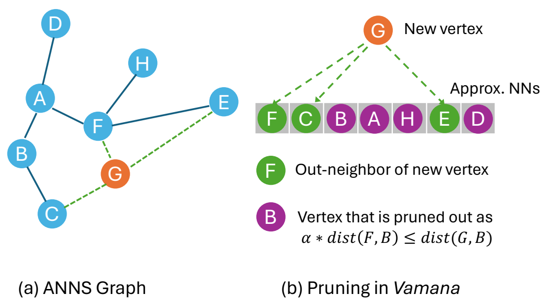

D
H
A
E
F
B
G
C
New vertex
Approx. NNs
F C B A H E D
Out-neighbor of new vertex
Vertex that is pruned out as
$\alpha * dist(F, B) \le dist(G, B)$
(a) ANNS Graph
(b) Pruning in *Vamana*

Figure 1. ANNS graph construction example: inserting a new data point to the existing graph.

Algorithm 1 Beam Search

```
1:  function BEAMSEARCH($G, s, q, k, L$)
2:      $\mathcal{L} \leftarrow \{s\}$
3:      $\mathcal{V} \leftarrow \emptyset$
4:      while $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ do
5:          $p^* \leftarrow \arg\min_{p \in \mathcal{L} \setminus \mathcal{V}} d(p, q)$
6:          $\mathcal{L} \leftarrow \mathcal{L} \cup N_{out}(p^*)$
7:          $\mathcal{V} \leftarrow \mathcal{V} \cup p^*$
8:          if $|\mathcal{L}| > L$ then
9:              update $\mathcal{L}$ to retain closest $L$ points to $q$
10:     return [closest $k$ points from $\mathcal{L}$; $\mathcal{V}$]
```

k-ANNS is defined as k-approximate NNS, in which only approximate results are returned. Without any ambiguity, we use ANNS for k-ANNS throughout this paper. The most commonly used metric for measuring the accuracy of ANNS is the *recall* score. Specifically, the $k@k'$ recall of query q is defined as $\frac{|\mathcal{K} \cap \mathcal{K}'|}{|\mathcal{K}|}$, where $\mathcal{K}$ is the ground-truth set of k-nearest neighbors of q in the dataset and $\mathcal{K}'$ is the output of an k'-ANNS algorithm. In this paper, we use k' that is equal to k when calculating recall scores.

2.2 Graph-based ANNS

ANNS can be efficiently solved using graph-based approaches. In these methods, queries are evaluated on a graph built from the vector dataset, with the search being performed using a graph-traversal algorithm known as *beam search*. In the remainder of this section, we review the ANNS algorithm and the graph construction process.

Beam search. As a popular heuristic search algorithm, *beam search* efficiently explores the search space by tracking a fixed number of the most promising candidates at each step, rather than exhaustively exploring all possible candidates or expanding a single best candidate (as in the case of *best-first search*). It operates in a greedy manner by maintaining a queue called the *beam set* $\mathcal{L}$, which has a maximum capacity defined by the beam size L.

In the context of ANNS, the beam set stores points from the dataset along with their distances to the query point, sorted in ascending order. During each iteration, the algorithm expands the *first unvisited point in the beam set* that is closest to the query vector. In graph-based ANNS, expanding a point means selecting its out-neighbors, computing their distances to the query point, and adding them to the beam set, which is then re-sorted based on the distances. If the beam set reaches its maximum capacity, points with greater distances to the query are discarded. This iterative process continues until all points in the beam set have been visited, after which the top-k points are returned as the ANNS query results. The beam search algorithm is detailed in Algorithm 1. Figure 3(a) demonstrates this process per iteration. Since the beam set is always sorted, the top-k query results correspond to the first k elements in the beam.

Graph construction. For graph-based ANNS, there are various graph construction algorithms, including NSG [19], HNSW [38], DiskANN (Vamana) [28], and many others [11, 15, 18, 25, 40, 42]. Most of them build *proximity graphs* where vertices represent points in a geometric space, and edges are drawn between vertices "close" to each other according to a proximity rule. Proximity graphs facilitate fast navigation from a query point to its nearest neighbors in the dataset. For instance, the Vamana algorithm [28] incrementally builds the proximity graph by continuously inserting points and adding edges based on Algorithm 2. Basically, given a dataset $\mathcal{P}$ and a point $p \in \mathcal{P}$, the out-neighbors of p, denoted as $N_{out}(p)$, are determined through beam search and a subsequent pruning procedure (lines 2 and 3 in Algorithm 2).

Figure 1 illustrates the pruning process when inserting a new vertex G into the existing ANNS graph. First, an ANNS query is executed on vertex G, returning its approximate nearest neighbors as the candidate set for G's out-neighbors. The pruning procedure then iteratively selects candidates to be G's neighbors by adding edges until the maximum degree of G is reached or the candidate set is empty. In this example, vertex F first is chosen as G's out-neighbor as it is the closest to G. Other candidates are pruned based on the inequality $\alpha \cdot dist(p^*, p') \le dist(p, p')$, where α is the pruning factor, typically slightly greater than 1. In this case, vertices B, A, H, and D are excluded as they satisfy the inequality.

The pruning procedure selects both long and short edges as p's out-edges, helping to prevent ANNS query results from converging to local optima. The rationale is that long edges connect distant neighbors, providing fast navigation during graph traversals, while short edges ensure that once the search reaches a region close to the query point, it can converge quickly. In Figure 1, a long edge $G \longrightarrow E$ is created, even though vertex E is farther from G than vertices B, A, and H. Creating long edges is crucial as they enhance the navigability of the graph, enabling efficient traversal toward regions near the query point, while short edges ensure quick convergence once such regions are reached.

Algorithm 2 Insert

```
1: function INSERT($p, G, s, L, R$)
2:     $\mathcal{L}, \mathcal{V} \leftarrow beamSearch(p, s, L, 1)$
3:     $N_{out}(p) \leftarrow prune(p, \mathcal{V}, R)$
4:     for $v \in N_{out}(p)$ do
5:         $N_{out}(v) \leftarrow N_{out}(v) \cup \{p\}$
6:         if $|N_{out}(v)| > R$ then
7:             $N_{out}(v) \leftarrow prune(v, N_{out}(v), R)$
```

Algorithm 3 Vamana Prune

```
1: function PRUNE($p, \mathcal{V}, R$)
2:     $\mathcal{V} \leftarrow \mathcal{V} \cup N_{out}(p)$
3:     $\mathcal{L} \leftarrow \emptyset$
4:     while $\mathcal{V} \neq \emptyset$ do
5:         $p^* \leftarrow \arg\min_{p' \in \mathcal{V}} d(p, p')$
6:         $\mathcal{L} \leftarrow \mathcal{L} \cup \{p^*\}$
7:         if $|L| = R$ then
8:             break
9:         for $p' \in \mathcal{V}$ do
10:            if $\alpha \cdot dist(p^*, p') \leq dist(p, p')$ then
11:                $\mathcal{V} \leftarrow \mathcal{V} \setminus p'$
12:    return $\mathcal{L}$
```

3 ANNS Workload Characterization

In this section, we report several interesting observations from profiling the ANNS workload. Based on these findings, we identify potential optimization opportunities and propose a fully parameterized beam search algorithm.

3.1 Parallelism of ANNS queries

Graph-based ANNS queries differ significantly from other graph traversal queries (e.g., BFS, SSSP, PageRank). A key difference is in the computational workload of a single query. Vertex-specific queries like SSSP require visiting all vertices in the graph to calculate the shortest distances from the source to all other vertices, which can cause a large memory footprint and many vertex value computations within one iteration. To improve their performance, efficient parallel algorithms have been proposed, most of which exploit the intra-query and intra-iteration parallelism. In contrast, an ANNS query only visits a small portion of the graph and a few vertices are computed for their distances to the query vector. For example, on BIGANN-50M dataset, on average, each query only visits around 360 data points and calculates 7000 distances to find the top-10 nearest neighbors accurately. As a result, most ANNS systems focus on improving the query throughput (QPS) [28, 39, 44] rather than the latency.

In this work, we argue that *it is not worthwhile to exploit intra-iteration and intra-query parallelism.* As a single ANNS query is very lightweight, vertex and edge parallelisms do not help much with query evaluation performance. We have conducted profiling experiments and found that a single ANNS query is very transient – most queries can finish in a few milliseconds. We also forced a single ANNS query to be evaluated sequentially and found that this setup does not harm the overall system throughput (Section 6.5).

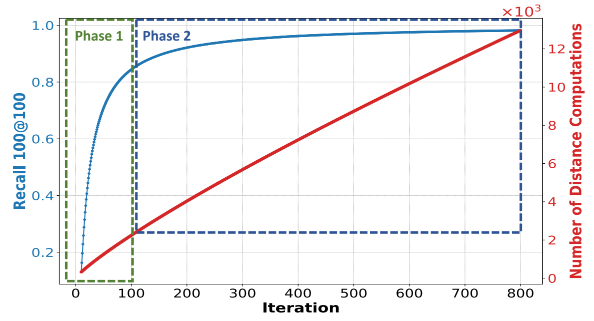


Figure 2. Profiling of ANNS queries.

Based on these observations, we propose to optimize ANNS throughput by improving the beam search algorithm rather than the parallelism of a single query evaluation.

3.2 Hardness-based ANNS Phases

Almost all graph-based ANNS solutions employ the greedy beam search outlined in Algorithm 1, with implementation details differing across methods. The algorithm executes iteratively and terminates when the beam $\mathcal{L}$ consists of no unvisited vertices. Although it has been shown to achieve superior performance and quality in solving ANNS queries, the throughput-recall tradeoff can be challenging.

Dilemma of recall and throughput. First, the quality of ANNS queries, recall of top-k results, is correlated with the *number of distance calculations* during the graph traversal of search. The more vertices in the graph have their distances to the input query computed, the higher the chance that vertices close to the query can be found. The *beam size L* determines how many distances are computed during the beam search. A larger beam size allows for exploring more of the graph, thus enhancing the recall score. However, increasing the beam size can deteriorate the system's query throughput.

To obtain a better tradeoff between query throughput and recall, DiskANN [28] and ParlayANN [39] use a parameter sweeping approach to find the best beam size for a given dataset and report the highest throughput. ParlayANN also adopts a method called $(1+\epsilon)$-cutting from Iwasaki et al. [26] for discarding unpromising candidates. In each iteration, the beam will only contain vertices with distances to the query point less than $(1+\epsilon)$ times the current k-th nearest neighbor.

Key observation. We show that it is possible to achieve higher query throughput by exploring the design space of beam search algorithm. The key idea is that *not every iteration contributes equally to the final results*, which we refer to as the *hardness* of ANNS computation. Figure 2 shows the recall score (left y-axis) for calculating the top-100 nearest neighbors in each iteration on dataset BIGANN-50M (50 million points) averaged over 10K queries. The right y-axis shows the accumulated number of distance computations. In this

Algorithm 4 Parameterized Beam Search

```
1: function BeamSearch($G, s, q, k, L, es_1, es_2, \alpha_1, \alpha_2, tr$)
2:     $\mathcal{L} \leftarrow \{s\}$
3:     $\mathcal{V} \leftarrow \emptyset$
4:     $[\alpha, es] \leftarrow [\alpha_1, es_1]$          ▷ cut-off factor and step size for phase 1
5:     check ← true
6:     while $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ do
7:         $len \leftarrow \min(es, |\mathcal{L} \setminus \mathcal{V}|)$
8:         $P \leftarrow \emptyset$
9:         for $i = 0 ... len - 1$ do
10:            $p^* \leftarrow \arg\min_{p \in \mathcal{L} \setminus \mathcal{V}} d(p, q)$
11:            if $d(p^*, q) \leq \alpha * d(\mathcal{L}^{kth}, q)$ then
12:                $P \leftarrow P \cup \{p^*\}$
13:            $\mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$
14:        for $p \in P$ do
15:            $ngh\_len \leftarrow Deg(p) - 1$
16:            for $i = 0 ... tr * ngh\_len$ do
17:                $v \leftarrow N_{out}[i]$
18:                if $d(v, q) \leq \alpha * d(\mathcal{L}^{kth}, q)$ then
19:                    $\mathcal{L} \leftarrow \mathcal{L} \cup \{v\}$
20:        if $|\mathcal{L}| > L$ then
21:            update $\mathcal{L}$ to retain closest $L$ points to $q$
22:        if check then
23:            $\mathcal{L}_k \leftarrow$ TopK($\mathcal{L}$)          ▷ k=min($|\mathcal{L} \setminus \mathcal{V}|$, max($k$,10))
24:            if $\mathcal{L}_k \setminus \mathcal{V} = \emptyset$ then
25:                check ← false
26:                $[\alpha, es] \leftarrow [\alpha_2, es_2]$          ▷ cut-off factor and step size
27:    return [closest $k$ points from $\mathcal{L}$; $\mathcal{V}$]
```

example, the progress of ANNS query evaluation accelerates rapidly within the first 100 iterations; the algorithm quickly identifies over 80% of the final answers. However, in later iterations, despite the distance calculations increase almost linearly, the recall score struggles to improve, discovering only a few new nearest neighbors.

Based on the above observation, it is natural to divide the beam search into two phases according to the hardness of ANNS computations. The first phase finds the majority of the nearest neighbors in a short time, while the second phase calculates many more distances to ensure that the query quality reaches the high recall regime. However, the benefit-cost ratio is low in the second phase, and terminating the search early usually leads to lower recalls. Since recall is a posterior metric that requires ground-truth data, which is not available a prior in real-world scenarios, the beam search algorithm lacks mechanisms to determine if a specified recall target has been achieved. As a consequence, early termination of the beam search usually leads to lower recalls.

3.3 Parameterized Beam Search

To effectively leverage the idea of ANNS phases, we propose a fully parameterized beam search algorithm by introducing separate *cut-off factors* (ϵ) and *expanding sizes* (es) for the two phases. The expanding size is the number of vertices to be expanded in one iteration (the original beam search always expands one vertex per iteration).

We now explain the rationale for introducing new parameters. First, the two phases exhibit completely different

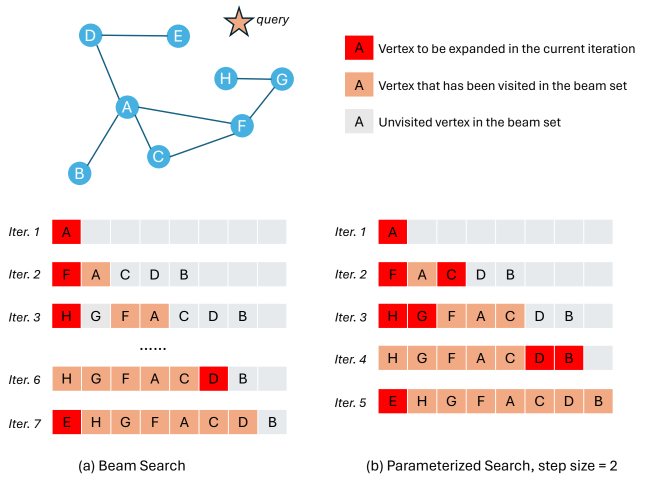

query
D
E
H
G
A
F
C
B
Vertex to be expanded in the current iteration
Vertex that has been visited in the beam set
Unvisited vertex in the beam set
Iter. 1
Iter. 2
Iter. 3
Iter. 6
Iter. 7
Iter. 4
Iter. 5
......
(a) Beam Search
(b) Parameterized Search, step size = 2

Figure 3. Beam search example.

graph traversal patterns. Phase 1 easily finds many accurate vertices due to the fast navigability of the constructed graph. In this case, long edges help locate the graph region close to the query vector. It is intuitive to think that Phase 1 should use a smaller expanding size than Phase 2 to avoid prematurely populating the beam with unpromising vertices. The cut-off factor for Phase 1 should be as large as possible, meaning little to no vertices should be discarded. In Phase 2, the search explores more vertices to improve the query quality. However, the distance differences among vertices explored in Phase 2 are subtle. This suggests that a more aggressive exploration pattern—expanding multiple vertices in one iteration, could be employed. The vertices expanded in the same iteration are quite similar in terms of their distances to the query vector, expanding them together in the same iteration gives their neighbors an equal chance to be considered, which makes the search more akin to a breadth-first search (BFS) rather than a depth-first search (DFS). This approach allows the algorithm to explore more neighbors at each level before moving deeper, reducing the likelihood of missing relevant neighbors.

To distinguish the two phases, one can use the iteration count, which, however, may vary significantly depending on the dataset and the method used to construct the graph. Instead, we propose to use a simple criterion, which is *the iteration when the first k vertices in the sorted beam are all visited*, to decide the boundary of the two phases. This can be easily examined at runtime during the query evaluation with little overhead. Algorithm 4 depicts the parameterized beam search algorithm. An additional benefit of identifying the two phases is that it allows for early termination of ANNS computation when sufficiently good results are acceptable. We found that the recall achieved by Phase 1 is only the dataset's intrinsic property and is irrelevant to query vectors. Thus, the proposed criterion that the first k vertices have been visited can be used to stop a beam search with arguably good recall scores. Figure 3 compares the parameterized beam search with the original one.

4 Recency-Aware Graph Construction

When constructing a graph from a vector dataset, traditional ANNS solutions typically overlook the temporal information associated with each data point, such as timestamps, which are oftentimes not embedded in the vectors. This omission can lead to the loss of valuable correlations between queries and their nearest neighbors, as the recency of the data may influence these relationships. Incorporating the temporal dimension, however, can enhance the relevance of search results and uncover important time-sensitive relationships within the data.

In this section, we introduce a novel approach to graph construction that is *recency-aware*, leveraging the temporal information within the dataset. By integrating this temporal aspect, the resulting graph not only maintains a smaller size (average degree) but also enhances the efficiency of ANNS query evaluation. Importantly, this approach does not compromise the quality of the query results, ensuring that the benefits of recency-awareness are realized without trade-offs in accuracy.

4.1 Temporal Query-Results Correlation

In many real-world datasets, the input queries and their ANNS results exhibit a notable temporal correlation [6]. One common pattern observed is that more recent content tends to appear more frequently in the top-k results of ANNS queries. Another frequently observed pattern is the seasonal trend, where content from the same season as the query date is more likely to be retrieved. Figure 4 illustrates this query-results correlation along the time dimension[2]. In this example, the vectors in the dataset are evenly distributed over a 36-month period. The top-100 nearest neighbors for any given query are more likely to be temporally close to the query issued date, with distant data points less frequently appearing in the nearest neighbor set.

Inspired by earlier works modeling temporal information in retrieval tasks [14, 17], we use the following likelihood model to depict the temporal correlation:

$$P(\mathbf{x}_i \mid \mathbf{q}, t_q) \;\propto\; f(d(\mathbf{q}, \mathbf{x}_i),\; t_q - t_i) \tag{1}$$

where $d(\mathbf{q}, \mathbf{x}_i)$ is a distance (or similarity) measure between query $\mathbf{q}$ and data point $\mathbf{x}_i$. The term $t_q - t_i$ represents the recency gap—how long ago $\mathbf{x}_i$ was added or updated, relative to the query time t_q. The function f combines distance and recency into a single score or probability.

For basic recency patterns (e.g., the one in Figure 4), f can be chosen to be monotonic in $(t_q - t_i)$, often represented by

[2]Synthetic timestamps are generated to illustrate this pattern since datasets used in [6] are not publicly available.

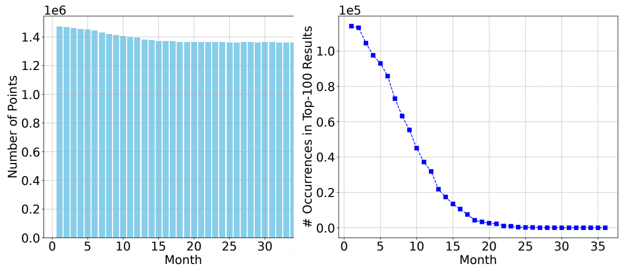


Figure 4. Vectors distribution and query-results correlation on the BIGANN-50M Dataset. The vectors are distributed evenly over 36 months (left). The query-results correlation (right) shows the total number of 10k queries' top-100 nearest neighbors in each month. The x-axis is the number of months away from the current date.

an exponential function. To account for periodic or seasonal patterns, f can be extended to include a periodic term in addition to the monotonic decay.

This correlation presents an opportunity to improve the efficiency and relevance of search results.

4.2 The Recency-Aware Construction Algorithm

To effectively exploit this correlation, we propose integrating temporal awareness into the graph construction process. The idea is that rather than pruning candidates based on a fixed scaling factor α (Line 10 in Algorithm 3), we define α as a function of time that considers query-results correlation when determining whether to create an edge between two nodes. We begin by modeling the query-results correlation, denoted by $\alpha(t)$, and then illustrate how $\alpha(t)$ can be used to guide the construction of the ANNS graph.

The pruning function $\alpha(t)$ is defined as a scaled and shifted logistic function:

$$\alpha(t) = b - \frac{b - a}{1 + e^{st - T/2}} \tag{2}$$

where a, b, s, and T are the scaling and shifting factors. The range of $\alpha(t)$ is $[a, b)$, T is the time span of data points in the dataset, s controls the steepness of $\alpha(t)$, and t is the absolute time difference between two data points ($t = |t_{p^*} - t_{p'}|$).

We now explain the rationales of using $\alpha(t)$. The original pruning procedure (Algorithm 3) removes vertex p' from p's out-neighbor candidates set if p' is too close to p^*. This ensures that both short and long edges can be added to p's outgoing edge list, resulting in a better navigability. Our design (Algorithm 5) considers the extra time dimension; it prunes more aggressively for vertices that are close in time. The pruning plays a less significant role for vertices that are far away in time — edges are inserted mainly based on distance. The overall effect of our $\alpha(t)$-based pruning for graph construction is that it provides better navigability for vertices that are close in time, that is, the search reaches the

Algorithm 5 Adaptive Pruning

```
1:  function AdaptivePrune($p$, $\mathcal{V}$, $R$)
2:      $\mathcal{V} \leftarrow \mathcal{V} \cup N_{out}(p)$
3:      $\mathcal{L} \leftarrow \emptyset$
4:      while $\mathcal{V} \neq \emptyset$ do
5:          $p^* \leftarrow \arg\min_{p' \in \mathcal{V}} d(p, p')$
6:          $\mathcal{L} \leftarrow \mathcal{L} \cup \{p^*\}$
7:          if $|L| = R$ then
8:              break
9:          for $p' \in \mathcal{V}$ do
10:             if $\alpha(|t_{p^*} - t_{p'}|) \cdot dist(p^*, p') \leq dist(p, p')$ then
11:                 $\mathcal{V} \leftarrow \mathcal{V} \setminus p'$
12:     return $\mathcal{L}$
```

region near the query point more quickly. In addition, the constructed graph tends to have a smaller average degree, reducing the memory footprint.

5 Graph Layout Optimization

In a typical graph representation, the neighbors of a vertex are continuously stored in the edge list (adjacency list) or edge array (CSR) by their vertex IDs. In graph-based ANNS, the neighbor vertex ID is used to access the vector data in the data array, which has the size of n (number of vertices), as shown in the example in Figure 5. Accessing the data array involves a random access pattern due to the irregularity of graph traversals. Unfortunately, the high dimensionality of vector space worsens this random access overhead. In ANNS computation, we found that fetching the vector data from memory (most of which are not even in the last level cache) and computing the distance between two vectors constitutes the majority of the total query evaluation time.

Selective neighbor scan. During beam search, it may be beneficial to skip unpromising neighbors while expanding a vertex. This is made possible with the insight of query-results correlation, that is, we may utilize recency information to avoid evaluating neighbors far away from the query and are likely to have little impact on the final recall score, reducing the need of vector fetching and distance calculation while still yielding satisfying query quality.

Vertex reordering. To avoid random access and simplify the selection logic, we propose a graph layout optimization. The idea is that, after the graph is constructed, we sort each vertex's out-neighbor list based on the timestamp. Then, at each iteration of the beam search, only a fraction of the vertex's out-neighbors is fetched and evaluated.

This truncated neighbor scan is achieved by setting the parameter tr (line 16 in Algorithm 4). As shown in Figure 5, when expanding vertex A, rather than calculating all of its out-neighbors' distances to the query, we use tr to control the portion of neighbors to be considered. With out-neighbors sorted by the global ordering of timestamps and the query-results correlation, the skipped neighbors are far away from the query and are likely to have little impact on the final recall score.

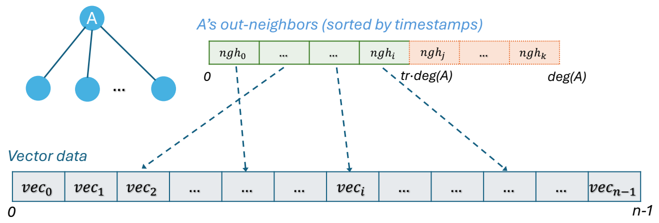


Figure 5. Truncated neighbor scan on sorted neighbor list using timestamp, up to the first $tr \cdot deg(A)$ neighbors.

This early termination of neighbor scanning has two advantages: 1) it avoids fetching unpromising vector data from the memory; 2) unpromising distance computations are also circumvented. Neighbors beyond the range of $[0, tr{*}ngh\_len)$ are skipped. Note that this early termination of neighbor scanning is distinguished from the cut-off factor (shown at Line 18 of Algorithm 4). The former happens before fetching vector data while the latter is applied after the vector data has been fetched and the distance between v and q has been calculated.

6 Evaluation

6.1 Experimental Setup

We evaluate PANNS and compare it against popular ANNS solutions, including DiskANN [28], HNSW [38], HCNNG [42], and pyNNDescent [40]. We implemented our recency-aware graph construction algorithm and parameterized beam search within the ParlayANN [39] codebase. All experiments were run on a Google Cloud c4-highmem-192 instance, which are equipped with two Intel Xeon Platinum 8581C CPUs (192 vCPUs) and 1.5TB memory.

Datasets. The datasets used in our experiments include BIGANN-1B, BIGANN-100M, DEEP-100M, SPACEV-100M, and TEXT2IMAGE-100M. Among them, the BIGANN-1B dataset [30] consists of 1 billion SIFT images embedded as 128-dimensional vectors. DEEP-100M is randomly sampled from the DEEP1B [5] dataset that consists of 1B image vector embeddings of 96 dimensions. TEXT2IMAGE-100M is randomly sampled from TEXT2IMAGE [1], which contains image embeddings produced by the SeResNet-101 model and textual queries encoded by the DSSM model, both have a dimension of 200.

Timestamps generation. Since the datasets studied by Baranchuk et al. [6] are proprietary and not publicly available, we replicate the recency pattern observed in their VideoAds dataset using the following approach: First, we uniformly assign a timestamp to each data point within a time range of 36 months. Next, we evaluate a set of sampled queries and augment the dataset by correlating the top-100 query results with their corresponding timestamps, following a half-normal distribution. The sample queries are either from the learn vectors set or the base vectors (the rest points are used as the index data). During the graph construction, we use the following adaptive pruning function $\alpha(t)$ in Equation 2 for all datasets: $\alpha(t) = 1.8 - \frac{0.8}{1+e^{0.8t-16}}$, where $a = 1.0$, $b = 1.8$, $k = 0.8$, and $T = 36$.

Algorithm parameters. Graphs constructed by Vamana algorithm have $R = 64$ (degree bound), $L = 128$ (beam size when building the graph), and $\alpha = 1.2$ (pruning parameter). For other benchmarks, we follow the setup described in ParlayANN [39]. For beam search, we perform a parameter sweep for both the baseline beam search and ours and choose the best performance for each specific recall target. Recall scores are calculated from the average of 10k queries.

6.2 Overall Performance

Figure 6 presents the QPS-recall curves for the ANNS algorithms evaluated, where 10k queries are evaluated simultaneously. PANNS, which incorporates recency-aware graph construction, parameterized beam search, and graph reordering optimization, demonstrates superior performance compared to other graph-based ANNS solutions. Specifically, PANNS achieves significantly higher throughput in the low recall regime (i.e., recall below 0.8). In the high recall regime (recall above 0.8), PANNS is 0.9× to 2.5× faster than the others on the BIGANN-1B graph, 1.2× to 3.7× on SPACEV-100M, 1.4× to 3.6× on TEXT2IMAGE-100M, and 1.0× to 2.1× on DEEP-100M.

6.3 Detailed Performance Analyses

We consider the default beam search in Algorithm 1 on graphs constructed using Vamana algorithm (Algorithms 2 and 3) as the `Baseline`. RAC employs a recency-aware graph while still using the default beam search for querying. PBS uses the graphs from Vamana algorithm, but evaluates queries via our parameterized beam search. RAC+PBS combines the parameterized beam search and the recency-aware graph construction. RAC+PBS+OPT represents the full implementation of PANNS, applying the graph reordering optimization described in Section 5, with the truncating parameter tr (line 16 in Algorithm 4) set to 0.8 for all graphs.

As illustrated in Figure 7, RAC and PBS individually offer QPS improvements over `Baseline` (1.2× and 1.3× speedup on average, respectively). Combining these two approaches in RAC+PBS results in even higher query throughput. This improvement is due to the recency-aware construction that allows the parameterized beam search to explore the graph more aggressively without compromising query quality. The graph reordering optimization further enhances performance by reducing the number of distance computations, providing additional gains on top of RAC+PBS. Figures 7d-7f present the average number of distance computations across $10k$ queries.

Table 1. Graph Constructed by Vamana and RAC

Graph	Algo.	Avg. Degree	Build Time (s)
BIGANN-100M	Vamana	55.8	753
	RAC	42.0	454
SPACEV-100M	Vamana	50.3	857
	RAC	34.1	506
TEXT2IMAGE-100M	Vamana	25.1	1615
	RAC	33.5	1776
DEEP-100M	Vamana	52.0	1528
	RAC	36.5	1023

The QPS of RAC+PBS+OPT ranges from 0.98× to 3.7× that of `Baseline`. Additionally, Figure 8 shows the QPS-recall curve for $k = 100$, which follows a similar trend for $k = 10$.

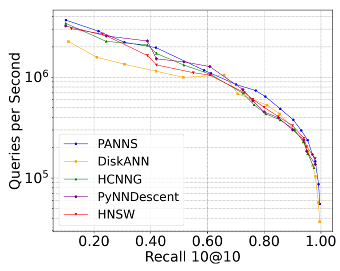
Parameter study. The parameterized beam search (PBS) enhances query throughput primarily by separating the beam search process into two phases and by fine-tuning parameters specific to each phase. Our results consistently show that the optimal parameter sweep favors a large cut-off factor (α_1) in Phase 1, which prevents any dropping of neighbor candidates, and a small step size (es_1). In contrast, Phase 2 typically selects a larger step size (es_2) and a smaller cut-off factor (α_2), resulting in a more significant dropping effect. This behavior underscores the importance of dividing the search into two distinct phases. In Phase 1, maintaining a high cut-off factor and a conservative step size helps identify the majority of the final nearest neighbors; any dropping of candidates or increase in step size could degrade the quality of this phase. Conversely, Phase 2, which is responsible for identifying the remaining few nearest neighbors, benefits from a more aggressive strategy. By expanding the candidate pool and dropping less promising candidates more rigorously, the search efficiency can be significantly improved.

6.4 Graph Construction Performance

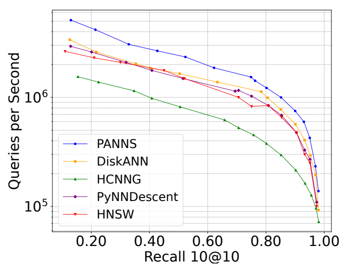
We analyze the graphs constructed using Vamana algorithm and our recency-aware construction (RAC) algorithm. The statistics for these graphs are presented in Table 1. Note that the construction time for RAC includes the time required for graph reordering. We used $R = 64$ and $L = 128$ for both algorithms. When the query-results correlation exists, the graphs constructed by RAC tend to have smaller average degrees. The only exception is TEXT2IMAGE graph, where RAC produces a slightly larger graph. Figure 7c and Figure 8c also indicate the RAC graph does not provide a performance gain over `Baseline` on TEXT2IMAGE graph.

An interesting aspect of RAC is that the graph built time is also reduced. This reduction is attributed to the adaptive pruning factor $\alpha(t)$, which results in fewer edges, thereby speeding up the graph construction process.

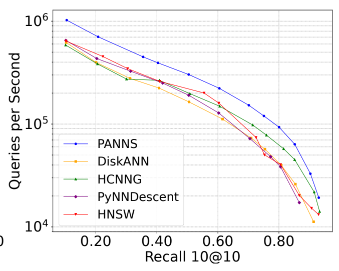
Figure 9 illustrates the degree distribution of the Vamana and RAC graphs. The X-axis represents the time span in our



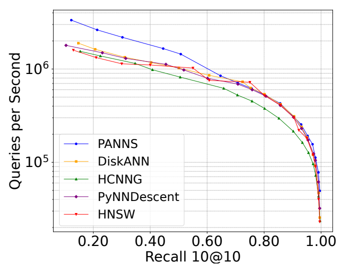
(a) BIGANN-1B QPS



(b) SPACEV-100M QPS

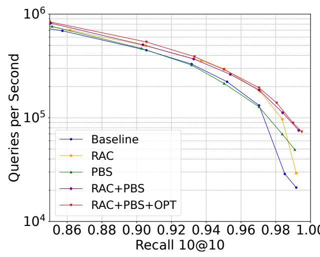


(c) TEXT2IMAGE-100M QPS

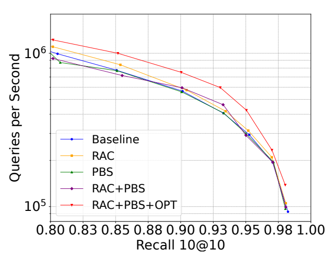


(d) DEEP-100M QPS

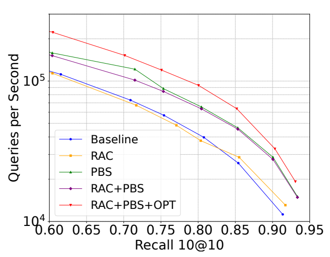
Figure 6. Overall Performance.



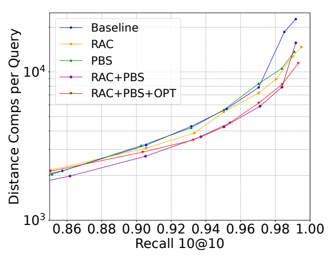
(a) BIGANN-100M QPS



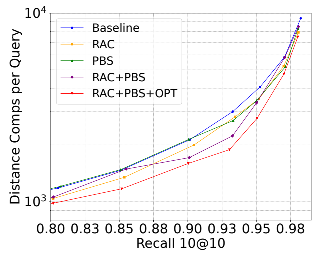
(b) SPACEV-100M QPS



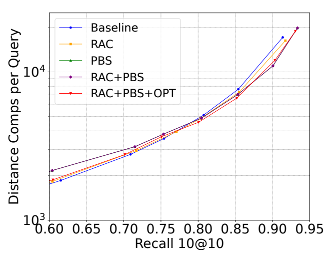
(c) TEXT2IMAGE-100M QPS



(d) BIGANN-100M Dist Comps

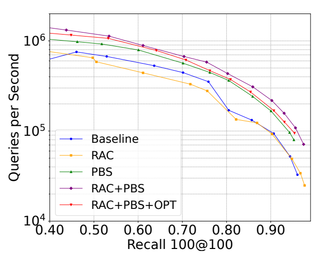


(e) SPACEV-100M Dist Comps

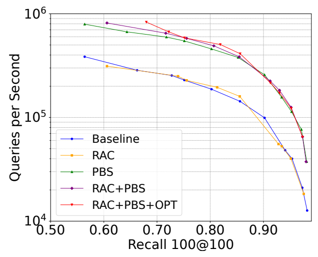


(f) TEXT2IMAGE-100M Dist Comps

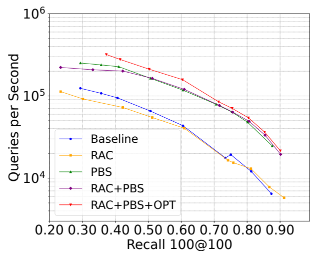
Figure 7. QPS-Recall curve when k = 10, while the second row shows the average distance computations.



(a) BIGANN-100M QPS



(b) SPACEV-100M QPS



(c) TEXT2IMAGE-100M QPS

Figure 8. QPS-Recall when k = 100.

tested datasets. Since the Vamana algorithm does not consider recency information, the degrees are evenly distributed over time. In contrast, our RAC algorithm incorporates temporal information during pruning, intentionally connecting both recent and distant vertices along the time dimension, much like how Vamana focuses on spatial distance.

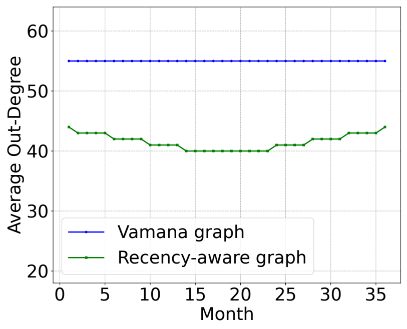


Figure 9. Degree distribution of BIGAMM-100M graph

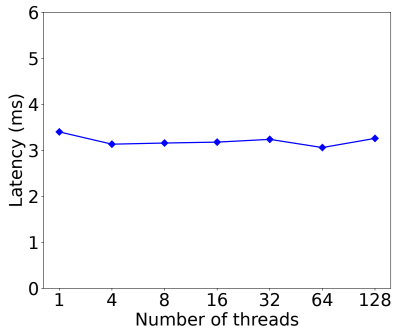
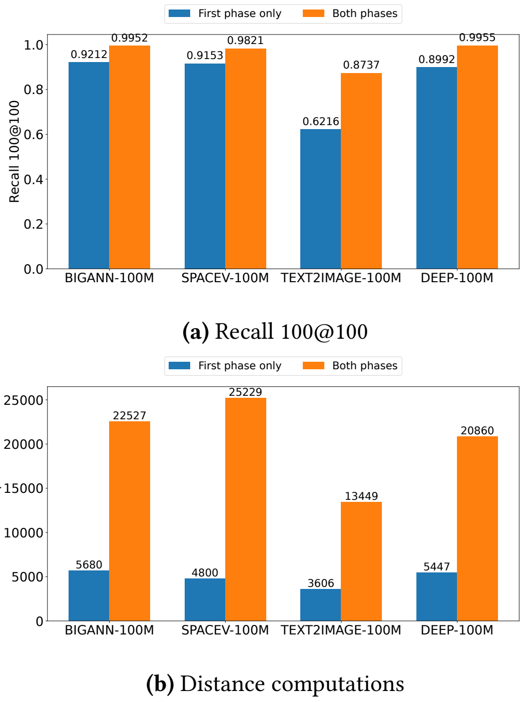


Figure 10. Scalability of beam search on BIGANN-100M with varying number of threads



(a) Recall 100@100

(b) Distance computations

Figure 11. Running first phase only vs. full algorithm.

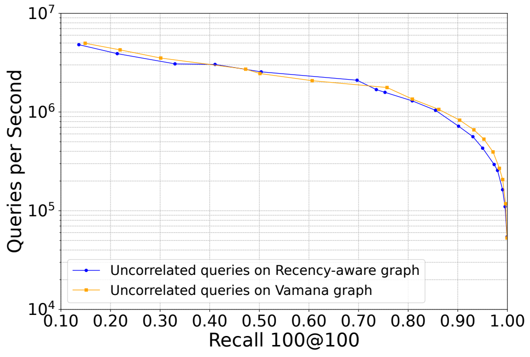


Figure 12. Query evaluation without temporal correlation.

6.5 Supporting Experiments

We present several experimental results that support our claims in earlier sections.

Inter- vs. intra-query parallelism. Figure 10 depicts the ANNS query latency as the number of threads varies (using ParlayANN implementation). In this experiment, 10k queries were evaluated sequentially (i.e., no concurrent query execution), meaning that parallelism is achieved through intra-query rather than inter-query evaluation. However, as the number of threads increases, the query latency remains largely unchanged. While edge parallelism can theoretically be leveraged at the query or iteration level, the lightweight nature of ANNS queries offers minimal opportunities for effective intra-query parallelism.

Early termination of ANNS queries. Figure 11 compares the recall and distance computations between the two-phase execution and one-phase execution—only the first phase of the parameterized search is executed. The first phase ends when the first k vertices in the sorted beam queue have all been visited, which often means that the first phase takes just k iterations. Despite the rapid completion of Phase 1, the recall scores achieved are notably high across all datasets, while the number of distance computations is significantly lower compared to Phase 2.

Performance of uncorrelated queries. We evaluated a set of queries that do not exhibit any temporal query-results correlations on both a RAC graph and a Vamana graph. As shown in Figure 12, the query throughputs on the two graphs are nearly identical. This shows that our RAC graph is robust enough to handle queries without recency correlations and does not introduce additional overhead for such queries.

6.6 Generality of the Solution

While our discussion focus on the temporal correlation, where the nearest neighbors of a query are likely to be close to the query itself in the time dimension, our recency-aware construction algorithm is versatile enough to handle 1) more complex temporal patterns, such as seasonal patterns that repeat annually or quarterly, and 2) domains beyond time series, where each data point is associated with an attribute that forms a total order, such as tags or categories associated with the data points. These more complex patterns can be easily transformed into the correlation studied in this work through total order sorting, making our approach broadly applicable without requiring significant modifications.

7 Related Work

Approximate nearest neighbor search. ANNS can be efficiently solved using different data structures and algorithms. One commonly used approach is the Inverted File Indexing (IVF) method. It partitions the dataset into multiple clusters using a clustering algorithm, often k-means. Only a few clusters need to be searched instead of the entire dataset. Product Quantization (PQ) is further proposed to reduce the vector space by quantization techniques [29]. Popular IVF-based algorithms include FAISS-IVF [16, 30], FALCONN [3], and SPANN [12]. DeDrift [6] is the first work that observed the content drift issue in ANNS.

Tree-based data structures are also used for computing nearest neighbors [9, 13], such as kd-trees [4, 53] and cover trees [8, 23]. One limitation of tree-based ANNS algorithms is that they are subject to low-dimensional space and do not scale well for high-dimensional search.

ANNS can be solved using graph-based approaches, which have been shown to provide better navigability in finding nearest neighbors by exploiting the small-world property. Graph-based ANNS consists of two main components: graph construction and search on the graph [27, 31, 32, 34, 35, 43, 57]. Existing graph construction algorithms include NSG [19], HNSW [38], DiskANN [28], among others [11, 15, 18, 25, 40, 42]. Most of them build *proximity graphs* that enable fast navigation from a query point to its closest neighbors in the dataset. iQAN [44] focuses on intra-iteration and intra-query parallelism for graph-based ANNS queries.

Graph system optimizations. In general-purpose graph processing systems, optimizations, such as reordering and scheduling techniques, have been proposed to improve graph traversal locality or reduce the computation redundancy [10, 20–22, 24, 36, 37, 41, 45–48, 51, 52, 56, 58]. For example, Glign [54] introduces inter- and intra-query alignments to reduce cache miss rates during the evaluation of concurrent queries. HAU [7] presents an accelerator that sorts streaming edges to improve the locality and accelerate the execution. CommonGraph [2] avoids repeated subcomputations in evolving graph analysis by identifying a common graph that exists across all snapshots and substituting expensive deletion updates with insertion updates.

Finally, both general-purpose graph systems [33, 50, 55] and graph-based ANNS systems can be naturally extended to support incremental processing, where results are updated based on newly arrived data. A key distinction is that ANNS systems produce the latest graph, while general-purpose graph systems directly yield the latest query results.

8 Conclusion

In this work, we propose PANNS, a graph-based ANNS system that addresses critical gaps in the design space of graph-based ANNS by incorporating temporal correlations between queries and data during graph construction and enabling parameterization in the search. Our system demonstrates significant improvements in query throughput, achieving up to a 3.7× speedup over existing methods, while also reducing graph size by up to 30% without sacrificing recall accuracy.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. This work was supported in part by the National Science Foundation Grants CCF-2226448, CCF-2106383, CCF-2028714, CCF-2002554, CCF-1813173, and CCF-1751392 to the University of California, Riverside. We acknowledge the support of UCR Ursa Major for providing computing resources used in our experiments.

References

[1] [n. d.]. Benchmarks for Billion-Scale Similarity Search. https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search Date accessed May 11, 2024 from https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search.

[2] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. Commongraph: Graph analytics on evolving data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 133–145.

[3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. *Advances in neural information processing systems* 28 (2015).

[4] Sunil Arya and David M Mount. 1993. Approximate nearest neighbor queries in fixed dimensions.. In *SODA*, Vol. 93. Citeseer, 271–280.

[5] 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.

[6] Dmitry Baranchuk, Matthijs Douze, Yash Upadhyay, and I Zeki Yalniz. 2023. Dedrift: Robust similarity search under content drift. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 11026–11035.

[7] Abanti Basak, Zheng Qu, Jilan Lin, Alaa R Alameldeen, Zeshan Chishti, Yufei Ding, and Yuan Xie. 2021. Improving streaming graph processing performance using input knowledge. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1036–1050.

[8] Alina Beygelzimer, Sham Kakade, and John Langford. 2006. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*. 97–104.

[9] Guy E Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208.

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. 587–596.

[11] 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.

[12] 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. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[13] Michael Connor and Piyush Kumar. 2008. Parallel Construction of k-Nearest Neighbor Graphs for Point Clouds.. In *VG/PBG@ SIGGRAPH*. Citeseer, 25–31.

[14] Wisam Dakka, Luis Gravano, and Panagiotis G Ipeirotis. 2008. Answering general time sensitive queries. In *Proceedings of the 17th ACM conference on Information and knowledge management*. 1437–1438.

[15] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*. 577–586.

[16] Matthijs Douze, Hervé Jégou, and Florent Perronnin. 2016. Polysemous codes. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part II 14*. Springer, 785–801.

[17] Miles Efron and Gene Golovchinsky. 2011. Estimation methods for ranking recent information. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. 495–504.

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

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

[20] Chao Gao, Mahbod Afarin, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. Mega evolving graph accelerator. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 310–323.

[21] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.

[22] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. {GraphX}: Graph processing in a distributed dataflow framework. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*. 599–613.

[23] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. 2022. Parallel cover trees and their applications. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 259–272.

[24] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE, 1–13.

[25] Ben Harwood and Tom Drummond. 2016. Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5713–5722.

[26] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355* (2018).

[27] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. {CXL-ANNS}:{Software-Hardware} collaborative memory disaggregation and computation for {Billion-Scale} approximate nearest neighbor search. In *USENIX Annual Technical Conf. (USENIX ATC)*. 585–600.

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

[29] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

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

[31] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes de Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, et al. 2023. Co-design hardware and algorithm for vector search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[32] Wenqi Jiang, Marco Zeller, Roger Waleffe, Torsten Hoefler, and Gustavo Alonso. 2023. Chameleon: a heterogeneous and disaggregated accelerator system for retrieval-augmented language models. *arXiv preprint arXiv:2310.09949* (2023).

[33] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: generalized incremental graph processing via graph triangle inequality. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 17–32.

[34] V Karthik, Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. 2024. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. *arXiv e-prints* (2024), arXiv–2401.

[35] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2024. 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*. 549–565.

[36] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2014. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041* (2014).

[37] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[38] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

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

[40] Leland McInnes. 2020. PyNNDescent for Fast Approximate Nearest Neighbors. *Webpage. Retrieved December* 15 (2020), 2022.

[41] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

IEEE, 1–14.

[42] Javier Vargas Munoz, Marcos A Gonçalves, Zanoni Dias, and Ricardo da S Torres. 2019. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition* 96 (2019), 106970.

[43] Julián Pavón, Ivan Vargas Valdivieso, Joan Marimon, Roger Figueras, Francesc Moll, Osman Unsal, Mateo Valero, and Adrian Cristal. 2023. VAQUERO: A Scratchpad-based Vector Accelerator for Query Processing. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1289–1302.

[44] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. 2023. iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 313–328.

[45] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 410–424.

[46] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[47] Semih Salihoglu and Jennifer Widom. 2013. Gps: A graph processing system. In *Proceedings of the 25th international conference on scientific and statistical database management*. 1–12.

[48] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[49] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. 2013. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1930–1941.

[50] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[51] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the edges you need: A generic {I/O} optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 507–522.

[52] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. 2015. {GraphQ}: Graph Query Processing with Abstraction {Refinement—Scalable} and Programmable Analytics over Very Large Graphs on a Single {PC}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 387–401.

[53] Rahul Yesantharao, Yiqiu Wang, Laxman Dhulipala, and Julian Shun. 2021. Parallel Batch-Dynamic k d-Trees. *arXiv preprint arXiv:2112.06188* (2021).

[54] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2022. Glign: Taming misaligned graph traversals in concurrent graph processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 78–92.

[55] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2024. IncBoost: Scaling Incremental Graph Processing for Edge Deletions and Weight Updates. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 915–932.

[56] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. 178–190.

[57] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, et al. 2023. 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*. 283–296.

[58] Jun Zhuang and Mohammad Al Hasan. 2022. Robust node classification on graphs: Jointly from bayesian label transition and topology-based label propagation. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 2795–2805.