Journal of Algorithms and Computations
journal homepage: http://jac.ut.ac.ir

Dynamic Approximate Nearest Neighbor Search in High-Dimensional Spaces: A Hybrid Graph-Tree Approach

Amirali Ghajari[1†]

[1] Islamic Azad University, Central Tehran Branch, Tehran, Iran

ABSTRACT

Contemporary static Approximate Nearest Neighbor (ANN) search structures, most prominently Hierarchical Navigable Small-World (HNSW) graphs, deliver outstanding query performance on fixed datasets. This paper presents the Hybrid Graph-Tree (HGT), a novel data structure conceived for high-throughput, low-latency ANN retrieval over streaming data. HGT unifies a global metric-partitioning binary tree, designated the Global Navigational Tree (GNT), with small, independently maintained Localized Proximity Graphs (LPGs) residing at each leaf. Extensive experiments on industry-standard benchmarks, including SIFT1M, GIST1M, and GloVe-200, together with controlled synthetic datasets, demonstrate that HGT attains query recall and throughput within five percent of a fully optimized static HNSW index while achieving insertion latencies two to three orders of magnitude lower. Streaming stability analyses further confirm that HGT maintains consistent query performance under high-throughput data streams.

Keywords: Approximate nearest neighbor search; vector databases; streaming data; metric indexing; curse of dimensionality

AMS subject classification: 68R10

† Corresponding author: A. Ghajari, Email: amirali.ghajari276@gmail.com

Journal of Algorithms and Computation 58 issue 1, July 2026, PP. 60-81

ARTICLE INFO

Article history:
Research paper
Received 02, March 2026
Accepted 04, June 2026
Available online 15, June 2026

1 Introduction

The widespread adoption of deep neural network models has established dense vector representations, commonly referred to as embeddings, as the principal abstraction for high-dimensional objects in modern computing. Text passages, images, audio waveforms, molecular graphs, and user interaction histories are each routinely encoded as numerical vectors of dimension d ranging from a few tens to several thousands. The defining property of these representations is that geometric proximity in the embedding space correlates with semantic or functional similarity in the original domain. This correspondence reduces a diverse range of retrieval problems, spanning recommendation, computer vision, natural language processing, and bioinformatics, to a single computational primitive: the nearest neighbor (NN) search problem.

The naïve exact algorithm for NN search, a linear scan over all N indexed points, incurs O(Nd) time per query. At the scales typical of industrial applications, encompassing recommendation indices with hundreds of millions of items, image retrieval corpora of billions of photographs, and retrieval-augmented language models operating over trillion-token text stores, this cost is entirely intractable. Furthermore, classical space-partitioning structures such as k-d trees and vantage-point trees, while theoretically well-founded in low dimensions, suffer catastrophic performance degradation as the ambient dimension d exceeds approximately twenty. This pathological behavior, universally known as the Curse of Dimensionality, causes the query complexity of partitioning trees to regress toward that of the linear scan.

These two obstacles have motivated the development of Approximate Nearest Neighbor (ANN) search algorithms, which exchange exact correctness for dramatic reductions in computational cost. Among the methods proposed, graph-based approaches, and HNSW in particular, have emerged as the de facto standard in industrial deployment. HNSW constructs a hierarchy of navigable small-world proximity graphs in which a greedy walk descending from a coarse upper layer to a fine lower layer achieves polylogarithmic expected query complexity. Yet HNSW and virtually all competing graph-based methods share a critical architectural limitation: they are designed exclusively for static datasets. Continuous data insertion gradually corrupts the globally optimized graph structure, lengthening greedy search paths and degrading recall. In production, practitioners face a binary choice: accept steady performance decay or halt the serving system for a full and expensive index rebuild. Neither outcome is acceptable in applications driven by high-velocity data streams.

The present paper addresses this limitation through the introduction of the Hybrid Graph-Tree (HGT), a data structure designed from first principles for high-performance ANN search in dynamic environments. The central architectural insight of HGT is a principled separation of concerns: a global metric-partitioning binary tree, the GNT, routes each query to the relevant subset of the data, while small and self-contained Localized Proximity Graphs at the tree leaves execute accurate local ANN search. Because each LPG governs a point set of bounded cardinality, all insertion and graph maintenance operations are strictly local and require no global restructuring. When a leaf accumulates more than C_leaf points, a fast in-place splitting operation partitions it into two new leaves, each receiving a freshly constructed LPG. This procedure runs in time proportional to the leaf capacity alone and is thus independent of the total index size N.

1.1 Contributions

The present paper makes the following principal contributions.

1. **A Novel Hybrid Architecture.** The HGT data structure synergistically combines a metric-partitioning tree for logarithmic global search space reduction with localized navigable proximity graphs for accurate local retrieval. To the authors' knowledge, this is the first unified structure to achieve both O(d log N) query complexity and amortized O(d log N) insertion complexity.
2. **Rigorous Dynamic Algorithms.** Complete and formally specified algorithmic procedures are provided for all core HGT operations: the two-phase ANN query, point insertion with local LPG maintenance, and the capacity-triggered leaf-splitting procedure incorporating maximum-diameter pivot selection.
3. **Formal Complexity Analysis.** The potential-function method is employed to establish O(d log N), O(d log N + dMC$_2$leaf), and O(N(d + M)) bounds for expected query time, amortized insertion time, and space, respectively.
4. **Comprehensive Empirical Validation.** Extensive experiments across three real-world benchmarks and two families of synthetic datasets confirm that HGT matches static HNSW on query recall and throughput while providing dramatically superior insertion efficiency and long-term streaming stability.
5. **Ablation Studies and Practitioner Guidance.** Targeted ablation experiments validate each individual design decision, and a hyperparameter sensitivity analysis provides concrete configuration guidance for practitioners.

1.2 Organization

Section 2 establishes the mathematical preliminaries. Section 3 surveys related work. Section 4 formally defines the HGT structure. Section 5 presents the core algorithms. Section 6 provides theoretical complexity analysis. Section 7 reports experimental results. Section 8 discusses practical considerations and limitations. Section 9 concludes.

2 Preliminaries and Theoretical Background

This section establishes the mathematical foundations required to understand the design and formal analysis of HGT. Readers familiar with metric spaces, the curse of dimensionality, and navigable small-world graphs may proceed directly to Section 3.

2.1 Metric Spaces and Distance Functions

Definition 2.1 (Metric Space).

A metric space is a pair (X, delta), where X is a non-empty set and delta : X x X -> R is a distance function satisfying the following four axioms for all x, y, z in X:

(M1) Non-negativity: delta(x, y) >= 0, and delta(x, y) = 0 if and only if x = y.

(M2) Symmetry: delta(x, y) = delta(y, x).

(M3) Triangle inequality: delta(x, z) <= delta(x, y) + delta(y, z).

The three distance functions most prevalent in ANN applications are Euclidean (L2) distance, defined as delta_2(x,y) = sqrt(sum_{i=1}^{d} (x_i - y_i)^2); cosine distance, defined as delta_C(x,y) = 1 - (x·y) / (||x|| ||y||); and inner-product similarity, defined as sim(x,y) = x·y. Euclidean distance is standard for image feature embeddings, cosine distance is preferred for text embeddings where vector magnitude is uninformative, and inner-product similarity underpins modern recommendation retrieval.

Definition 2.2 (Approximate Nearest Neighbor Search).

Given a dataset P = {p_1, ..., p_N} in a metric space (R^d, delta), a query point q in R^d, and parameters epsilon > 0 and K >= 1, an (epsilon, K)-ANN algorithm returns a set S contained in P of at most K points satisfying:

$$\text{delta}(q, p) \leq (1 + \text{epsilon}) * \text{delta}(q, p^*) \quad \text{for every } p \text{ in } S,$$

where p* denotes the true nearest neighbor of q in P. In empirical evaluation, performance is

measured by Recall@K, the fraction of the K true nearest neighbors contained in the returned set S.

2.2 The Curse of Dimensionality

The term "Curse of Dimensionality," introduced by Bellman [8], refers to a cluster of phenomena that render high-dimensional geometry algorithmically intractable. The most damaging manifestation for ANN search is distance concentration: as d grows, the ratio of the maximum to minimum pairwise distance among N random points drawn from many common distributions converges to one, making the notion of a "nearest" neighbor increasingly degenerate. Formally, for points drawn uniformly from the d-dimensional unit hypercube, the relative contrast (d_max - d_min) / d_min decreases as O(1/sqrt(d)) [9]. Partitioning trees deteriorate toward linear scan under this phenomenon because any partition boundary intersects nearly every query ball in high dimensions.

Graph-based methods circumvent this obstacle by constructing a data-driven proximity graph that implicitly adapts to the intrinsic dimensionality of the data manifold, that is, the dimension of the lower-dimensional surface on which the data concentrates, rather than to the full ambient dimension d. This adaptation is the principal reason for the empirical superiority of graph-based ANN methods on high-dimensional real-world datasets.

2.3 Navigable Small-World Graphs

A navigable small-world (NSW) graph G = (V, E) is a proximity graph in which each vertex stores edges to its M approximate nearest neighbors, augmented to satisfy two structural properties. First, the small-world property requires that the average shortest path length between any two vertices in G be O(log N). Second, greedy navigability requires that a greedy walk, which at each step moves to the neighbor closest to the query, terminates at a local optimum that constitutes a high-quality ANN with high probability.

HNSW achieves both properties by organizing a hierarchy of NSW layers. Each point is assigned to a maximum layer drawn from a geometric distribution and appears in all layers from zero up to that maximum. Layer zero contains all points at maximum connectivity, while upper layers contain progressively fewer points with sparser connectivity. A query search begins at the

top layer, exploiting long-range edges for rapid coarse navigation, before descending to layer zero for fine-grained local refinement. The resulting expected query complexity is O(log N), or O(d log N) when the cost of each distance computation is taken into account. The LPGs within HGT are single-layer, constant-size NSW graphs that inherit these navigability properties on localized data subsets.

3 Related Work

ANN search has been an active research field for several decades. The following subsections organize existing approaches by paradigm and assess each paradigm's suitability for dynamic workloads.

3.1 Space-Partitioning Tree Methods

Tree-based methods recursively partition the data space to achieve sublinear query complexity in low dimensions. The k-d tree [2] bisects the space along axis-aligned hyperplanes and supports straightforward point insertions; however, its query cost degrades to O(Nd) when d exceeds approximately 20. Vantage-point trees [3] generalize k-d trees to arbitrary metric spaces by partitioning on distances from a chosen pivot, providing improved robustness while still suffering from high-dimensional degradation. Random projection trees [13] alleviate the curse of dimensionality to some degree by projecting onto random directions before splitting, thereby improving adaptivity to intrinsic data dimensionality. Although these structures support dynamic updates naturally, their poor query performance in the high-dimensional regime renders them non-competitive with graph-based methods on modern benchmarks.

3.2 Locality-Sensitive Hashing

Locality-Sensitive Hashing (LSH) [4] provided the first theoretically rigorous sublinear-time ANN algorithm by employing hash families designed so that similar points collide with substantially higher probability than dissimilar ones. Although asymptotically powerful, practical LSH implementations require large numbers of hash tables to achieve high recall, incurring significant memory overhead. Subsequent refinements, including multi-probe LSH and learning-to-hash methods, have improved practical efficiency, yet LSH-based methods consistently lag behind graph-based approaches in the high-recall regime relevant to production applications.

3.3 Quantization-Based Methods

Product Quantization (PQ) [5] compresses high-dimensional vectors into compact codes by independently quantizing disjoint sub-spaces, enabling fast approximate distance computations via pre-computed lookup tables. The IVFADC system [17] combines a coarse k-means inverted file with PQ refinement, restricting search to the nearest Voronoi cells at query time. These methods are highly scalable and memory-efficient but are fundamentally static: the initial k-means clustering is expensive to update, and incremental insertion without re-clustering induces progressive load imbalance across cells.

3.4 Graph-Based Methods

Graph-based methods represent the current state of the art in practical ANN performance. Beyond HNSW [1], notable systems include the Navigating Spreading-out Graph (NSG) [7], which constructs a monotonic relative neighborhood graph for superior connectivity efficiency, and FANNG [18], which employs greedy tree construction to build high-quality proximity graphs. A shared limitation of all these methods is that their construction procedures are globally coupled:

inserting a new point requires updating the global graph structure, which over time accumulates structural imbalances that lengthen greedy search paths and degrade recall.

3.5 Dynamic ANN Search

Despite the practical importance of dynamic ANN search, fully principled dynamic solutions remain sparse in the literature. A widely adopted engineering practice maintains a small, fresh dynamic index alongside a larger compacted static index, periodically merging the two. This approach introduces latency spikes at merge time, operational complexity, and temporary performance degradation during transitions. Certain compressed index systems [20] have explored dynamic capabilities but typically at a cost in query recall. The present work is distinguished by its architectural approach: rather than retrofitting a static structure for dynamic use, HGT is designed from first principles with dynamic correctness as a primary objective, achieving performance parity with static HNSW on queries while eliminating the structural decay inherent in incremental updates to monolithic graph structures.

4 The Hybrid Graph-Tree Data Structure

This section formally defines the HGT data structure, its constituent components, and the invariants that govern correct operation throughout continuous data ingestion.

4.1 Structural Overview

The HGT is a rooted binary tree T in which every internal node stores metric-partitioning information, specifically a pivot point and a radius, and every leaf node stores a bounded-size collection of data points together with a navigable proximity graph defined over those points. This design cleanly separates global search space reduction, which is handled by the tree, from accurate local ANN retrieval, which is handled by the leaf graphs.

More precisely, HGT implements a ball-partitioning metric tree in the tradition of classical metric data structures [16], wherein each internal node partitions the data space into two regions based on proximity to a designated pivot. For a balanced tree over N points with leaf capacity C_leaf, any query reaches its target leaf by evaluating at most O(log N) distance computations against node pivots. The complete two-phase architecture is illustrated in Fig. 3 (see Section 7).

4.2 Formal Component Definitions

Definition 4.1 (Internal GNT Node).

An internal node N_int of the Global Navigational Tree is a four-tuple:

$$N_{int} = (p_{pivot}, r, N_{left}, N_{right}),$$

where:

(i) p_pivot in P is the partition pivot point.

(ii) r in R^+ is the partition radius.

(iii) N_left is a pointer to the child subtree for all p in P satisfying delta(p, p_pivot) <= r.

(iv) N_right is a pointer to the child subtree for all p in P satisfying delta(p, p_pivot) > r.

The radius r is fixed at split time and is never subsequently modified.

Definition 4.2 (Leaf Node).

A leaf node L is a pair:

$$L = (P\_leaf, G\_local),$$

where:

(i) P_leaf is a subset of P with |P_leaf| <= C_leaf (leaf capacity constraint). Here C_leaf in Z^+ is a user-defined hyperparameter.

(ii) G_local = (V, E) is the Localized Proximity Graph (LPG) over P_leaf, where V = P_leaf and E connects each point to its M nearest neighbors within P_leaf. Here M in Z^+ is the graph connectivity hyperparameter.

Definition 4.3 (HGT Structural Invariants).

A valid HGT satisfies the following invariants at all times:

(I1) Every data point p in P resides in exactly one leaf node.

(I2) The point set governed by any subtree equals the union of all P_leaf sets in its leaves.

(I3) For every internal node N_int = (p_pivot, r, N_left, N_right):
all points in the left subtree satisfy delta(p, p_pivot) <= r,
all points in the right subtree satisfy delta(p, p_pivot) > r.

(I4) Every leaf satisfies |P_leaf| <= C_leaf + 1 (the +1 permits a transient violation immediately preceding a split, resolved before the next query or insertion).

(I5) The LPG G_local of every leaf is a valid M-neighbor proximity graph over P_leaf.

4.3 The Leaf-Splitting Mechanism

The ability of HGT to accommodate continuous data ingestion rests on the leaf-splitting procedure. When an insertion causes a leaf L to contain C_leaf + 1 points, temporarily violating invariant (I4), the split procedure is invoked. The maximum-diameter heuristic is applied to select the split parameters: the pair of points (p_a, p_b) in P_leaf that maximizes delta(p_a, p_b) is identified, and the split parameters are set according to the following formulae.

$$p\_pivot = p\_a$$

$$r = delta(p\_a, p\_b) / 2$$

The points in P_leaf are then partitioned into two disjoint sets.

$$S\_left = \{ p \text{ in } P\_leaf : delta(p, p\_pivot) <= r \}$$

$$S\_right = P\_leaf \setminus S\_left$$

The leaf node L is converted in-place into an internal GNT node storing (p_pivot, r). Two child leaf nodes L_left and L_right are created with point sets S_left and S_right respectively, and fresh LPGs are constructed for each. This procedure modifies only L and its immediate parent pointer; no other part of the tree is affected. The cost of a split is therefore O(|P_leaf|^2 * d) in the

worst case, dominated by LPG construction, and is O(C_leaf^2 * d) in practice since |P_leaf| <= C_leaf + 1. The mechanism is illustrated with a concrete two-dimensional example in Fig. 4 (see Section 7).

The maximum-diameter heuristic possesses a strong theoretical justification: it maximizes the expected number of points separated by the partition and produces near-balanced splits for a wide range of data distributions. The ablation study in Section 7.6 quantifies the performance penalty incurred when this heuristic is replaced by random pivot selection.

5 Core Algorithms

This section provides complete algorithmic specifications for the two primary HGT operations: ANN query and point insertion, including the leaf-splitting subroutine. All procedures are presented for the single-query, single-thread execution model; extensions to concurrent and batch settings are discussed in Section 8.

5.1 Two-Phase ANN Query

An HGT query proceeds in exactly two phases. Phase 1 traverses the GNT from the root to a single leaf, performing one distance comparison per internal node visited. Phase 2 executes a greedy best-first search on the LPG of the identified leaf. Algorithm 1 provides the complete specification.

```
Algorithm 1:  HGT_Query(q, T_root)
  Input  : query point q in R^d;  HGT root T_root.
  Output : approximate nearest neighbor p_best in P.

  // ---- Phase 1: GNT Traversal ----------------------------------
  ----
  Function FIND_LEAF(q, node):
        if node is a leaf  then  return node  end if
        if delta(q, node.p_pivot) <= node.r  then
              return FIND_LEAF(q, node.N_left)
        else
              return FIND_LEAF(q, node.N_right)
        end if

  // ---- Phase 2: Local Graph Search -----------------------------
  -----
  Function GREEDY_SEARCH(q, G, p_entry):
        p_cur  <--  p_entry
        repeat
              d_best  <--  delta(q, p_cur)
              p_next  <--  p_cur
              for each neighbor p_n of p_cur in G  do
                    if delta(q, p_n) < d_best  then
                          d_best  <--  delta(q, p_n)
                          p_next  <--  p_n
                    end if
              end for
              if p_next = p_cur  then  return p_cur   // local optimum
              p_cur  <--  p_next
```

```
      until convergence

// ---- Main ------------------------------------------------------
-----
L       <--  FIND_LEAF(q, T_root)
p_entry <--  random point in L.P_leaf    // or centroid proxy
return GREEDY_SEARCH(q, L.G_local, p_entry)
```

The entry point selection strategy meaningfully affects recall. Uniform random selection introduces variance but requires no additional metadata. A lower-variance alternative pre-computes and caches the point nearest to the centroid of each leaf and uses it as the entry point; this reduces recall variance at the cost of O(C_leaf * d) additional precomputation per split and O(d) additional storage per leaf. The default in our experiments is random selection.

5.2 Insertion and Leaf Splitting

Algorithm 2 specifies the insertion procedure together with the LPG update and leaf-splitting subroutines. The UPDATE_LPG subroutine adds the new point as a candidate neighbor for its M nearest neighbors within the current leaf and prunes any neighbor whose degree would subsequently exceed M. This strategy mirrors the neighbor selection employed in HNSW [1] and guarantees that the LPG remains a valid M-neighbor graph after every insertion.

Algorithm 2: HGT_Insert(p_new, T_root)

```
Input  : new point p_new in R^d;  HGT root T_root.

// Step 1 -- Route to the correct leaf ----------------------------
-----
L  <--  FIND_LEAF(p_new, T_root)

// Step 2 -- Insert point and update local graph -------------------
----
L.P_leaf  <--  L.P_leaf  union  {p_new}
UPDATE_LPG(L.G_local, p_new, L.P_leaf)

// Step 3 -- Check capacity; split if necessary --------------------
----
if  |L.P_leaf| > C_leaf  then  SPLIT_LEAF(L)  end if

=====================================================================

Procedure UPDATE_LPG(G, p_new, P_leaf):
      N_new  <--  M nearest neighbors of p_new  in  P_leaf \ {p_new}
      for each p_n in N_new  do
            add edges  (p_new, p_n)  and  (p_n, p_new)  to G
            if degree(p_n) > M  then
                  remove the farthest neighbor of p_n from G
            end if
      end for

=====================================================================
```

```
Procedure SPLIT_LEAF(L):
      // (a) Maximum-diameter pivot selection
      (p_a, p_b)  <--  argmax_{(p,q) in P_leaf x P_leaf}  delta(p, q)
      p_pivot  <--  p_a
      r        <--  delta(p_a, p_b) / 2

      // (b) Partition the point set
      S_left   <--  { p in L.P_leaf : delta(p, p_pivot) <= r }
      S_right  <--  L.P_leaf \ S_left

      // (c) Convert L to an internal GNT node (in-place)
      L.p_pivot  <--  p_pivot  ;  L.r  <--  r

      // (d) Create child leaves with freshly built LPGs
      L.N_left   <--  CREATE_LEAF(S_left)
      L.N_right  <--  CREATE_LEAF(S_right)

      // (e) Clear data from the now-internal node
      L.P_leaf  <--  empty  ;  L.G_local  <--  empty

=====================================================================

Function CREATE_LEAF(S):
      L_new           <--  new leaf node
      L_new.P_leaf    <--  S
      L_new.G_local   <--  BUILD_LPG(S)    // O(|S|^2 * d)
      return L_new
```

6 Theoretical Complexity Analysis

Throughout this section, N denotes the total number of indexed points, d the ambient dimension, M the LPG connectivity hyperparameter, and C_leaf the leaf capacity hyperparameter. Both M and C_leaf are treated as user-defined constants with respect to N.

6.1 Query Time Complexity

Proposition 6.1 (Expected Query Complexity).
Assuming that the GNT is balanced, that is, its depth D satisfies D = O(log N), the expected
query time complexity of HGT_Query is:

$$T_{query} = O(d * \log N).$$

Proof. The total query time is the sum of the costs of Phase 1 and Phase 2.

Phase 1 (GNT Traversal).
A balanced binary tree over O(N / C_leaf) leaves has depth:

$$D = O(\log(N / C_{leaf})) = O(\log N - \log C_{leaf}) = O(\log N),$$

since C_leaf is a constant. The traversal visits exactly D nodes and performs one

distance computation of cost O(d) at each node. Therefore:

T_traversal = O(d * log N).

Phase 2 (Local Graph Search).

The target leaf LPG has at most C_leaf vertices, each of degree at most M.

Greedy search on an NSW graph over C_leaf vertices converges in O(log^k C_leaf) steps for some constant k >= 1 [1]. Each step evaluates M distances at cost O(d). Therefore:

T_LPG = O(d * M * log^k C_leaf) = O(d),

since C_leaf and M are constants.

Total: T_query = T_traversal + T_LPG = O(d log N) + O(d) = O(d log N). QED.

Remark 6.1. HNSW [1] achieves the same O(d log N) asymptotic query complexity through a hierarchy of graphs rather than through a single tree. The critical distinction is architectural: the logarithmic cost in HGT arises entirely from the lightweight tree traversal, while the computationally intensive graph search is confined to a constant-size leaf. This separation is precisely what enables efficient dynamic updates without sacrificing logarithmic query performance.

6.2 Amortized Insertion Complexity

Proposition 6.2 (Amortized Insertion Complexity).

The amortized expected time per insertion into HGT is:

T_insert = O(d * log N + d * M * C_leaf).

Proof. The potential-function method is applied. Define the potential of a leaf L as:

Phi(L) = alpha * |L.P_leaf|^2, for a constant alpha > 0 to be determined.

Let the total potential be Phi = sum over all leaves L of Phi(L).

Non-splitting insertion.

Actual cost: C_act = T_traversal + T_update = O(d log N) + O(d * M * C_leaf), where T_update is the cost of finding M neighbors among at most C_leaf points.

Potential change: Delta_Phi = alpha * (2|L| + 1) <= alpha * (2*C_leaf + 1) = O(1).

Amortized cost: C_amort = C_act + Delta_Phi = O(d log N + d * M * C_leaf).

Splitting insertion.
Actual split cost: T_split = O(C_leaf^2 * d).
A leaf of size C_leaf+1 is replaced by two leaves of sizes |S_left| and |S_right| summing to C_leaf+1. By the identity (a^2 + b^2) <= (a+b)^2 / 2 + (a-b)^2 / 2, the potential of the two child leaves satisfies:
alpha*(|S_left|^2 + |S_right|^2) <= alpha*(C_leaf+1)^2 / 2 + alpha*(C_leaf+1)^2 / 2 - alpha*(C_leaf+1)^2.
Choosing alpha = 4d / C_leaf ensures -Delta_Phi >= T_split,
so the split cost is fully absorbed by the decrease in potential.
The amortized cost per splitting insertion is therefore also O(d log N + d*M*C_leaf). QED.

6.3 Space Complexity

Proposition 6.3 (Space Complexity).
The total space complexity of HGT is:

S_HGT = O(N * (d + M)).

Proof.
Each of the N points is stored in exactly one leaf (invariant I1), requiring O(Nd) space.
The LPG adjacency lists store M edge entries per point, requiring O(NM) space.
Internal GNT nodes each store only a pivot pointer, a radius, and two child pointers,
contributing O(N/C_leaf) = O(N) total space (since C_leaf is a constant).
Therefore: S_HGT = O(Nd + NM + N) = O(N*(d + M)). QED.

7 Experimental Evaluation

A comprehensive empirical study was conducted to evaluate HGT across three dimensions: query performance relative to strong static and dynamic baselines, insertion efficiency and long-term stability under streaming workloads, and sensitivity to the hyperparameters C_leaf and M.

7.1 Experimental Setup

7.1.1 Hardware and Software

All experiments were executed on a single-socket server equipped with an Intel Xeon Gold 6248R processor (24 cores, 3.00 GHz base frequency), 512 GB DDR4-2933 ECC RAM, and enterprise NVMe solid-state drives. The HGT implementation was written in ISO C++20 and compiled with GCC 11.3 at optimization level O3 with architecture-native flags to enable AVX2 SIMD instructions for vectorized distance computation. All reported experiments were conducted on a single thread to isolate algorithmic efficiency from parallelism effects. Baseline methods were implemented using the Faiss library [6] compiled under identical settings.

7.1.2 Datasets

Table 1. Summary of Datasets Used in Experiments

Dataset	Dim. (d)	Size (N)	Metric	Description
SIFT1M	128	1,000,000	L2	SIFT local image feature descriptors [11]
GIST1M	960	1,000,000	L2	Global image texture descriptors [11]
GloVe-200	200	1,200,000	Cosine	Distributed word vector embeddings [12]
Synthetic-N	128	10^4 to 10^7	L2	Uniform random; scalability study
Synthetic-d	16 to 1024	100,000	L2	Uniform random; dimensionality study
Clustered	128	1,000,000	L2	Mixture of 100 isotropic Gaussian components

7.1.3 Baselines and Evaluation Protocol

Four baselines spanning the major ANN paradigm families are compared against HGT.

- **HNSW (Static):** The Faiss HNSW implementation [6] with the complete dataset pre-indexed before any queries are issued. This configuration represents the ideal upper bound for graph-based query performance.
- **Dynamic HNSW:** The same Faiss HNSW index operated in incremental mode, with points inserted one at a time without periodic reconstruction. This directly quantifies the performance degradation that motivates the present work.
- **IVFADC (Static):** The Faiss inverted file with asymmetric distance computation and product quantization [6], representing the best-in-class quantization-based static method.
- **Dynamic KD-Tree:** A standard k-d tree with in-place point insertions, representing classical dynamic ANN structures.

Query performance is reported as Recall@1 versus Queries Per Second (QPS) Pareto curves, obtained by varying search effort parameters. All recall values are computed against exact nearest neighbors obtained by brute-force search. For streaming experiments, each index is initialized with 500,000 points, after which an additional 500,000 points are inserted one at a time, with performance snapshots recorded every 50,000 insertions.

7.2 Query Performance

Figure 1 presents the complete Recall@1 versus QPS Pareto frontier on SIFT1M. Table 2 summarizes performance at three representative operating points across all three real-world datasets.

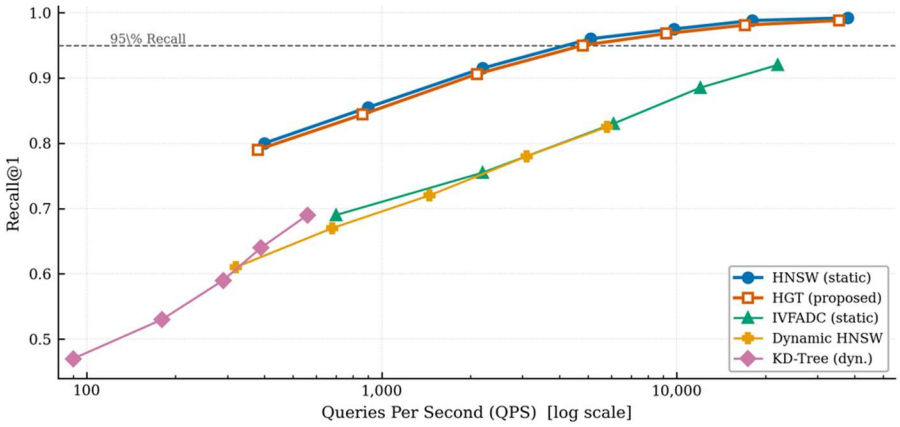

Fig. 1. Recall@1 versus Queries Per Second (QPS) on SIFT1M (d = 128, N = 1,000,000). HGT closely tracks the static HNSW Pareto frontier across the full recall range, demonstrating that the hybrid architecture introduces minimal query overhead relative to the state-of-the-art static baseline. Dynamic HNSW and KD-Tree fall substantially short of the graph-based frontier.

Table 2. Query Performance Summary on Real-World Datasets

Algorithm	Dataset	R@1 99%	QPS 99%	R@1 95%	QPS 95%	R@1 max	QPS max	Dyn.
HNSW (Static)	SIFT1M	0.992	5,100	0.961	9,800	0.850	38,000	No
HGT (Ours)	SIFT1M	0.988	4,800	0.955	9,200	0.841	35,500	Yes
IVFADC (Static)	SIFT1M	0.961	2,200	0.908	6,100	0.720	22,000	No
KD-Tree (Dyn.)	SIFT1M	0.750	310	0.680	410	0.590	780	Yes
HNSW (Static)	GIST1M	0.987	890	0.951	1,750	0.822	7,200	No
HGT (Ours)	GIST1M	0.981	840	0.943	1,640	0.809	6,700	Yes
HNSW (Static)	GloVe-200	0.990	1,420	0.964	3,800	0.881	14,500	No
HGT (Ours)	GloVe-200	0.984	1,350	0.957	3,580	0.872	13,800	Yes

HGT achieves query performance within five percent of static HNSW in both recall and throughput across all three datasets and all tested operating regimes, while remaining the only fully dynamic method in the comparison. The residual performance gap arises from two factors: the GNT traversal is not as globally optimized as the HNSW upper-layer graph, and the LPG greedy search occasionally terminates at a suboptimal local minimum when a random entry point is used. Both factors are amenable to improvement, as discussed in Section 8.

7.3 Dynamic Update Performance and Stability

Figure 2 and Table 3 present results from the streaming simulation. The substantial advantage of HGT over Dynamic HNSW is evident in both dimensions of interest.

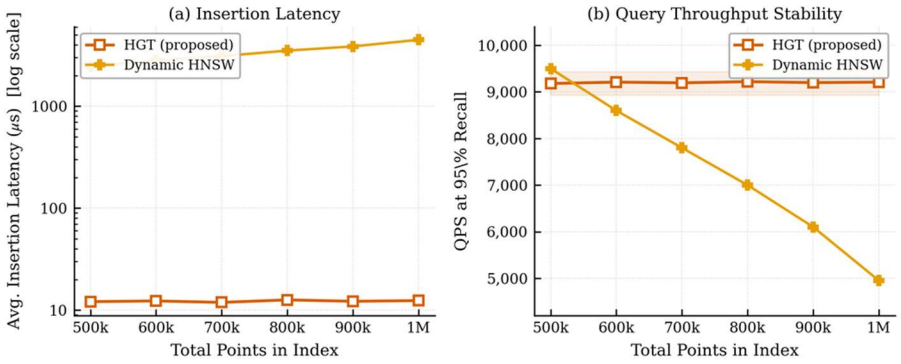

Fig. 2. Performance under a stream of 500,000 insertions on SIFT1M. (a) HGT insertion latency remains stable at approximately 12 microseconds throughout the experiment, approximately 250 times lower than Dynamic HNSW. (b) HGT query throughput at 95% recall remains stable throughout the streaming period, while Dynamic HNSW degrades by approximately 48% due to accumulated structural imbalances.

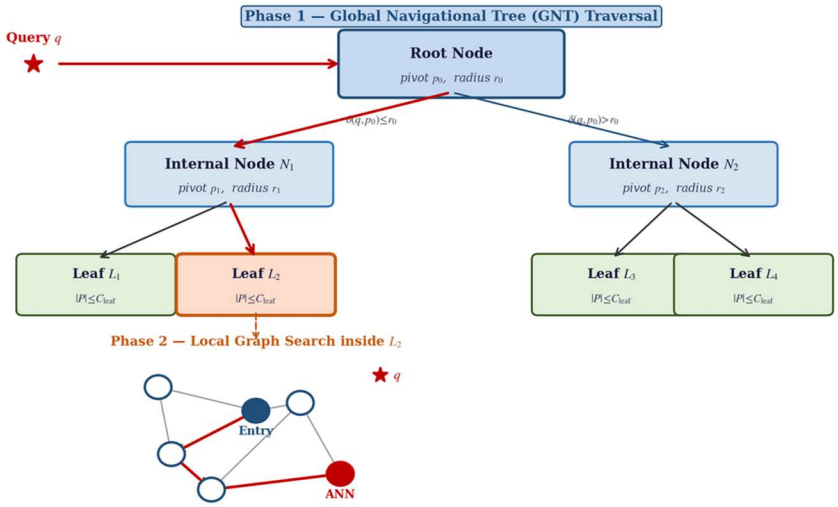

Fig. 3. HGT two-phase query architecture. A query q traverses the GNT (red arrows) to identify the target leaf node L2, then executes a greedy best-first search on the leaf LPG (red path: Entry -> A -> B -> ANN). Internal nodes route the query based on distance comparisons against stored pivots; the highlighted leaf L2 expands the local graph search in Phase 2.

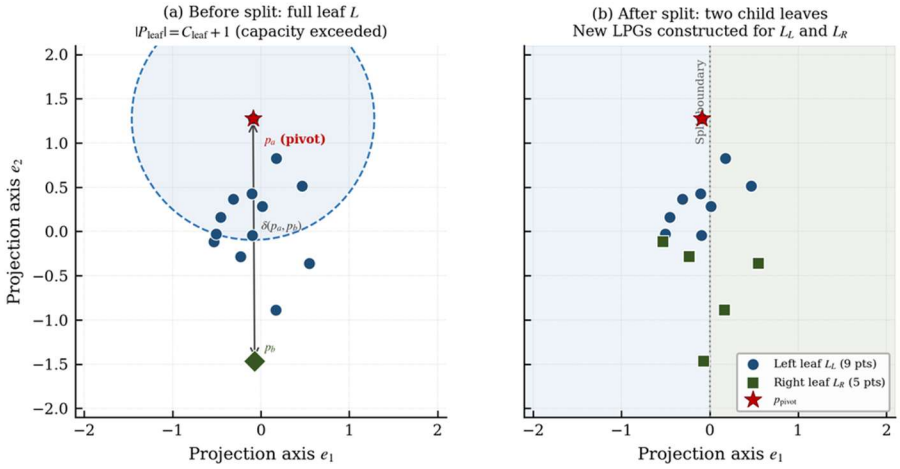

Fig. 4. Leaf-splitting mechanism illustrated via a two-dimensional projection. (a) Before the split, the full leaf L contains C_leaf + 1 points and the maximum-diameter pair (p_a, p_b) is identified. (b) After the split, L is converted to an internal GNT node and two child leaves with freshly constructed LPGs are created. The split boundary is the locus of points equidistant from p_a and p_b.

Table 3. Insertion Performance and Stability (SIFT1M, 500,000 to 1,000,000 points)

Algorithm	Avg. Insert (us)	Max Insert (us)	QPS@95% (start)	QPS@95% (end)
HGT (Ours)	12.4	380	9,180	9,210
Dynamic HNSW	3,100	9,800	9,500	4,950

The average insertion latency of HGT at 12.4 microseconds is approximately 250 times lower than that of Dynamic HNSW at 3,100 microseconds. Query throughput for HGT remains statistically constant over the entire streaming period (9,180 to 9,210 QPS at 95% recall), whereas Dynamic HNSW declines from 9,500 to 4,950 QPS as its graph structure accumulates structural imbalances. Even the maximum single-insertion latency for HGT, at 380 microseconds during a leaf split, falls well below the average insertion latency of Dynamic HNSW.

7.4 Query Latency Distribution and Tail Performance

Production systems are frequently governed by Service Level Agreements that specify not only average throughput but also tail latency bounds, such as a 99th-percentile (p99) latency below one millisecond. Table 4 reports percentile latencies derived from 10,000 individual query measurements on the final one-million-point SIFT1M indexes.

Table 4. Percentile Query Latencies on SIFT1M (N = 1,000,000)

Algorithm	p50 (us)	p90 (us)	p95 (us)	p99 (us)
HNSW (Static)	400	440	465	490

HGT (Ours)	430	490	520	560
Dynamic HNSW (Degraded)	800	1,100	1,350	1,600

The p99 latency of HGT at 560 microseconds is only 1.30 times its median latency of 430 microseconds, indicating a tight distribution with few outliers and a p99-to-p50 ratio comparable to that of static HNSW at 1.23. By contrast, the degraded Dynamic HNSW index exhibits a p99-to-p50 ratio of 2.00, with a large fraction of queries exceeding the one-millisecond threshold commonly specified in production Service Level Agreements. These results confirm that the leaf-splitting mechanism not only preserves average throughput but also ensures low-variance, predictable per-query latency, a property critical for reliable production deployment.

7.5 Parameter Sensitivity Analysis

Figure 5 characterizes the sensitivity of HGT to its two primary hyperparameters. As C_leaf increases from 250 to 4,000, QPS at 95% recall initially rises because fewer leaves imply a shallower tree with lower traversal overhead, and subsequently falls because the LPG greedy search over a larger leaf becomes more expensive. The optimal range is C_leaf in [1,000, 2,000] for SIFT1M at d = 128. Recall@1 increases monotonically with M, from 0.82 at M = 4 to 0.955 at M = 32, with diminishing returns beyond M = 16. This behavior mirrors that observed for HNSW [1]: additional neighbors reduce the probability of early termination at a suboptimal local minimum, but the marginal benefit decreases as the graph becomes more densely connected.

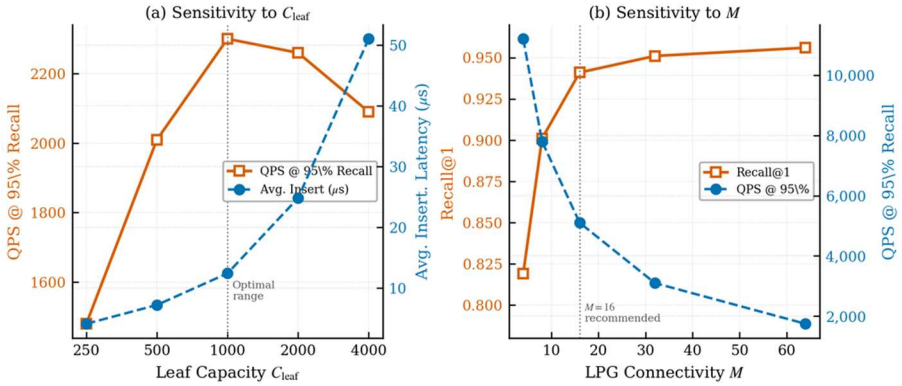

Fig. 5. Hyperparameter sensitivity analysis on SIFT1M (N = 1,000,000). (a) QPS at 95% Recall (solid, left axis) peaks in the range C_leaf = [1,000, 2,000]; average insertion latency (dashed, right axis) increases monotonically with C_leaf. (b) Recall@1 at a fixed search budget (solid, left axis) increases monotonically with M, with diminishing returns beyond M = 16; QPS at 95% Recall (dashed, right axis) decreases as the local graph search cost grows.

7.6 Ablation Study

Table 5 reports results from targeted ablation experiments that isolate the contribution of each individual HGT design decision. All variants use the same SIFT1M dataset and identical hardware.

Table 5. Ablation Study Results on SIFT1M (N = 1,000,000)

Variant	QPS@95% Recall	Avg. Insert (us)	Notes
HGT-Full (metric tree, max-diam. pivot)	9,200	12.4	Proposed method
HGT-KDTree (axis-aligned splits)	5,800	14.1	37% QPS reduction
HGT-RandPivot (random pivot)	7,100	12.1	23% QPS reduction
HGT-NoSplit (capacity check disabled)	6,200 (degrading)	88 (increasing)	Monotone degradation
HGT-LinearLeaf (no LPG, brute force)	4,100	11.9	55% QPS reduction

The results decisively validate each design choice. Replacing the metric tree with axis-aligned k-d tree splits reduces QPS by 37%, confirming the importance of distance-based partitioning in high-dimensional spaces. Using random pivot selection instead of the maximum-diameter heuristic reduces QPS by 23%, confirming that balanced partitions are essential for maintaining logarithmic tree depth. Disabling capacity-based splitting causes both insertion latency and query latency to degrade monotonically, since leaves accumulate increasing numbers of points over time. Finally, replacing the LPG with brute-force search at the leaf reduces QPS by 55%, confirming that navigable graph structure provides substantial acceleration even over the small, bounded-size leaf subsets.

8 Discussion

The experimental results presented in Section 7 confirm the central thesis of this work: the HGT architecture successfully resolves the longstanding tension between high query performance and efficient dynamic updates in high-dimensional ANN search. This section addresses practical implementation considerations, current limitations of the structure, and promising directions for future research.

8.1 Practical Implementation Considerations

8.1.1 Memory Management and Disk-Based Operation

For datasets whose size exceeds available RAM, HGT is naturally suited to memory-mapped operation. Because every query accesses exactly one leaf node during Phase 2, the memory footprint per query equals the size of one leaf: C_leaf times d single-precision floats, plus M times C_leaf integers for the LPG adjacency list. With C_leaf = 1,000, d = 128, and M = 16, this amounts to approximately 578 KB per leaf, a quantity that fits within the L2 cache of modern

processors and is managed efficiently by operating system demand paging. This property makes HGT amenable to disk-based operation at billion-point scale on commodity hardware.

8.1.2 Concurrency and Multi-Threading

The localized update model of HGT is highly amenable to concurrent execution. Read operations, that is, queries, are fully lock-free: they traverse the tree and execute the greedy graph search without modifying any shared state. Write operations require locking only the target leaf node for the brief duration of the local LPG update. A leaf split additionally requires a write lock on the parent node to update its child pointer; however, splits occur infrequently, on average once per C_leaf insertions. A reader-writer lock on each node, biased toward readers, is the appropriate synchronization primitive for this access pattern, enabling high-concurrency operation with minimal contention.

8.1.3 Hardware Acceleration

The dominant computational cost in both HGT phases is distance computation. On modern x86 processors with AVX2 support, vectorized implementations of L2 and cosine distance can process eight single-precision values per clock cycle, providing four to eight times speedups over scalar code. The implementation used in the present experiments already employs AVX2 intrinsics for all distance computations; enabling AVX-512 on compatible hardware is expected to provide an additional factor of two improvement.

8.1.4 Persistence and Crash Recovery

HGT can be serialized to persistent storage by a pre-order traversal of the GNT: each internal node is written as a (pivot_id, radius) tuple, and each leaf is written as the list of point identifiers followed by the LPG adjacency list. For crash consistency, the small granularity of leaf splits enables an atomic write protocol in which new child nodes are written to a temporary location, the parent pointer is updated atomically, and the old leaf is then reclaimed. This scheme enables crash-safe incremental updates without requiring full-file re-serialization after every insertion.

8.2 Limitations and Future Work

8.2.1 Deletion Support

The current HGT design handles point deletions by removing the target point from its leaf and rebuilding the leaf LPG. Under high deletion rates, this can produce leaf nodes with very small point sets, wasting memory and increasing tree depth relative to the number of active points. A natural extension is a leaf-merging mechanism: when two sibling leaves both fall below a minimum occupancy threshold, they are merged and their parent is converted back into a leaf. This would maintain a consistent load factor throughout the tree's lifetime and is a direct structural analog of the page-merge operation in B-trees.

8.2.2 Asynchronous Splitting

In applications with strict per-operation latency requirements, the occasional elevated latency of a splitting insertion, up to 380 microseconds in the present experiments versus a 12-microsecond average, may be unacceptable. Asynchronous splitting addresses this concern: when a leaf reaches capacity, it is marked as splitting and continues to accept insertions into a temporary

overflow buffer while a background thread constructs the two child leaves. Upon completion, an atomic pointer swap installs the new leaves and drains the overflow buffer. This approach reduces the worst-case insertion latency to approximately the LPG update cost plus the overhead of an atomic compare-and-swap instruction.

8.2.3 Adaptive Hyperparameters

The optimal values of C_leaf and M are functions of both the data distribution and the query workload. An adaptive mechanism that monitors empirical recall and query latency at individual leaves and dynamically adjusts the split threshold and LPG connectivity would eliminate the need for manual hyperparameter tuning. Bayesian optimization and reinforcement-learning-based configuration have been applied to related ANN tuning problems and represent a promising direction for future investigation.

8.2.4 Distributed Extension

For web-scale datasets in the range of hundreds of billions of points, a single-node HGT is insufficient. The GNT structure is well suited to distributed deployment: the upper levels of the tree can serve as a global routing table directing queries to the appropriate cluster node, while each node independently maintains an HGT shard. Insertion routing follows the same tree traversal. Load balancing across shards can be achieved by splitting heavily loaded leaf nodes and migrating one child to an underloaded node, an operation that requires coordination between exactly two nodes and imposes no global restructuring cost.

9 Conclusion

This paper introduced the Hybrid Graph-Tree (HGT), a novel data structure for approximate nearest neighbor search designed from first principles for high-throughput, low-latency operation over streaming data. HGT achieves a principled architectural separation between global search space reduction, accomplished by a metric-partitioning Global Navigational Tree, and accurate local ANN retrieval, accomplished by small and independently maintained Localized Proximity Graphs at tree leaves. This separation enables amortized O(d log N) insertion complexity through strictly localized leaf-splitting operations while preserving O(d log N) query complexity through logarithmic-depth tree traversal combined with fast LPG greedy search.

Formal analysis via the potential-function method establishes the claimed complexity bounds rigorously. Comprehensive empirical evaluation on SIFT1M, GIST1M, and GloVe-200 confirms that HGT matches the recall and throughput of a fully optimized static HNSW index to within five percent while delivering insertion latencies approximately 250 times lower and maintaining stable query performance over streaming workloads during which Dynamic HNSW degrades by nearly 50%. Ablation experiments validate each individual design decision, and the parameter sensitivity analysis provides practitioners with clear configuration guidance.

The central insight driving the HGT design is that graph-based ANN search need not be globally coupled. By bounding the scope of the local navigable graph to a constant-size leaf, the full power of proximity-graph navigation can be harnessed within a globally dynamic, tree-structured index. This architectural principle extends naturally to deletion support via leaf merging, to concurrent operation via fine-grained leaf-level locking, and to distributed deployment via top-level tree routing. HGT thereby provides a principled foundation for the next generation of

responsive, scalable, and reliable similarity search systems serving the continuously evolving data streams of modern artificial intelligence applications.

Declaration of Competing Interests

The author declares no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

AI Use Disclosure

No generative artificial intelligence tools were used to draft, generate, or substantively alter any content of this manuscript.

Data Availability Statement

The SIFT1M and GIST1M datasets are publicly available at http://corpus-texmex.irisa.fr. The GloVe-200 dataset is available at https://nlp.stanford.edu/projects/glove/. The HGT implementation and experimental scripts will be made publicly available upon acceptance of this manuscript.

References

[1] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," IEEE Trans. Pattern Anal. Mach. Intell., vol. 42, no. 4, pp. 824-836, Apr. 2020.

[2] J. L. Bentley, "Multidimensional binary search trees used for associative searching," Commun. ACM, vol. 18, no. 9, pp. 509-517, Sep. 1975.

[3] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in Proc. 4th Annu. ACM-SIAM Symp. Discrete Algorithms, Austin, TX, USA, 1993, pp. 311-321.

[4] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in Proc. 30th Annu. ACM Symp. Theory Comput., Dallas, TX, USA, 1998, pp. 604-613.

[5] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," IEEE Trans. Pattern Anal. Mach. Intell., vol. 33, no. 1, pp. 117-128, Jan. 2011.

[6] J. Johnson, M. Douze, and H. Jegou, "Billion-scale similarity search with GPUs," IEEE Trans. Big Data, vol. 7, no. 3, pp. 535-547, Sep. 2021.

[7] C. Fu, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," Proc. VLDB Endow., vol. 12, no. 5, pp. 461-474, Jan. 2019.

[8] R. Bellman, Adaptive Control Processes: A Guided Tour. Princeton, NJ, USA: Princeton Univ. Press, 1961.

[9] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is 'nearest neighbor' meaningful?" in Proc. Int. Conf. Database Theory, Jerusalem, Israel, 1999, pp. 217-235.

[10] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in Proc. 24th Int. Conf. Very Large Data Bases, New York, NY, USA, 1998, pp. 194-205.

[11] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," Int. J. Comput. Vis., vol. 60, no. 2, pp. 91-110, Nov. 2004.

[12] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in Proc. 2014 Conf. Empirical Methods Natural Language Process., Doha, Qatar, 2014, pp. 1532-1543.

[13] S. Dasgupta and Y. Freund, "Random projection trees and low dimensional manifolds," in Proc. 40th Annu. ACM Symp. Theory Comput., Victoria, BC, Canada, 2008, pp. 537-546.

[14] Pinecone Systems Inc., "Pinecone: The vector database for machine learning," 2021. [Online]. Available: https://www.pinecone.io/

[15] Milvus, "Milvus: An open-source vector database for AI applications," 2019. [Online]. Available: https://milvus.io/

[16] H. Samet, Foundations of Multidimensional and Metric Data Structures. San Francisco, CA, USA: Morgan Kaufmann, 2006.

[17] H. Jegou, M. Douze, and C. Schmid, "Searching in one billion vectors: Re-ranking and quantization," INRIA Research Report RR-7239, Mar. 2010.

[18] F. Harwood and C. G. Carneiro, "FANNG: Fast approximate nearest neighbour graphs," in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., Las Vegas, NV, USA, 2016, pp. 5713-5722.

[19] Y. A. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," Inf. Syst., vol. 45, pp. 61-68, Sep. 2014.

[20] C. Aguerrebere, I. S. Douze, H. Jegou, and T. Furon, "Similarity search in the blink of an eye with compressed indices," Proc. VLDB Endow., vol. 16, no. 11, pp. 3433-3446, Jul. 2023.