DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node

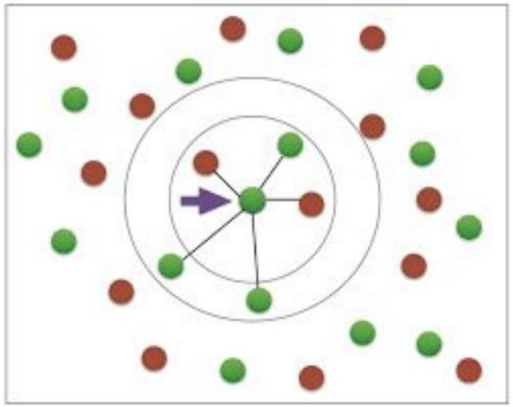
S.J. Subramanya, Devvrit, R. Kadekodi, R. Krishaswamy, H.V. Simhadri
By: Abdel Kareem Dabbas

K Nearest Neighbours

Given dataset $P \subset R^d$ and query $q \in R^d$, return the k closest points to q

Distance in this paper: $d(x,q) = \| x-q \|_2$ (Euclidean)

Target setting: billions of points, hundreds/thousands of dimensions, millisecond latency



Why approximate?

Approximate Nearest Neighbors (ANN)

- Exact NN in high dimensions often degenerates toward near-linear scan (“curse of dimensionality”)
- ANN goal: maximize recall subject to latency/compute constraints
- Modern ML embeddings → high-dim nearest neighbor search is a core primitive

Evaluation

Let G = true top-k neighbors; output X (size k)

k-recall@k: $|X \cap G| / k$

1-recall@1 (strict) vs 1-recall@100 (weaker - “did we include the true NN somewhere in a list of 100?”)

Approaches pre DiskANN

Today’s billion-scale ANN (pre-DiskANN) = two main families

Inverted Index + Compression

Sharded in-memory indices

DiskANN’s position

DiskANN aims to get: high 1-recall@1 like graph methods

with memory footprint closer to compression methods

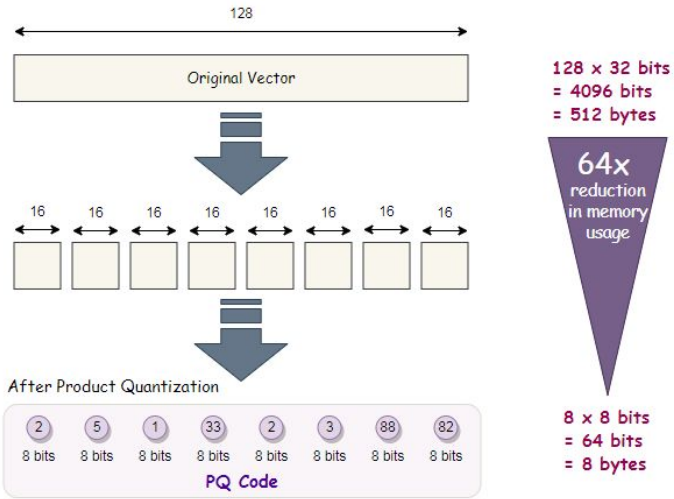
by putting the graph + full vectors on SSD and using compressed vectors in RAM

Product Quantization

Vector Compression Technique

Split each vector into blocks and quantize each block separately.

Store only centroid indices (short codes) instead of full vectors.



Approximate distances via lookup tables, trading small error for huge memory savings.

Approach #1: Inverted Index + Data Compression

Approach #1: IVF + PQ/OPQ (FAISS and IVFOADC+G+P)

- Cluster dataset into M partitions
- For query q: search only the closest m $\ll$ M partitions
- Store vectors in compressed codes (e.g., Product Quantization) so indices fit in RAM
- Compute approximate distances quickly using codes (often GPU-friendly)

Key tradeoff

- Small memory footprint, fast lookup
- But **l**ossy compression yields lower 1-recall@1 (paper reports it can be ~0.5 at this scale)

Approach #2: Sharded in-memory ANN

Approach #2: Shard the dataset + in-memory graph indices

- Split dataset into disjoint shards
- Build high-recall in-memory ANN (e.g., HNSW/NSG) per shard on uncompressed vectors
- Query routed to (often many or all) shards; merge partial results

Key tradeoff

- High recall possible
- But memory footprint is large ⇒ many machines for 10^9+ points
- Example from the paper: an NSG index for 100M (128D) can be ~75GB (index + data), implying multi-node hosting for 1B

How much does switching to SSD help?

SSD random reads take hundreds of microsecond
Latency budgets are a few milliseconds ⇒ only a small number of round trips is acceptable

Paper's central serving constraints for SSD-resident indices:

- keep random SSD accesses/query to "a few dozen"
- keep round trips under ~10 (preferably ~5)

FAISS supports searching only for RAM, even with SSDs

DiskANN Summary

- Build a graph index (Vamana)
- Store on SSD: graph + full-precision vectors
- Store in RAM: compressed vectors (PQ codes)
- Query-time:
    - Use PQ-in-RAM for cheap distance estimates to guide search
    - Fetch neighborhoods from SSD in batches (BeamSearch)
    - Rerank using full vectors “for free” (piggyback on SSD reads)

Claimed outcome

- High recall, low latency, high density on a single node

Graph ANN: Greedy Search

Graph-based ANN (search phase)

- Construct directed graph G=(P,E) over points
- Search does best-first / greedy traversal from a start node s

GreedySearch (concept)

- Maintain:
  - Candidate list L (size limit)
  - Visited set V
- Repeat:
  - Pop closest unvisited candidate to q
  - Add its out-neighbors to the candidate pool
  - Keep best L candidates

Key parameters

- L: search list size (accuracy vs compute)
- Graph degree R: memory vs navigability

Algorithm 1: GreedySearch(s, x_q, k, L)

Data: Graph G with start node s, query x_q, result size k, search list size $L \geq k$

Result: Result set $\mathcal{L}$ containing k-approx NNs, and a set $\mathcal{V}$ containing all the visited nodes

begin

- initialize sets $\mathcal{L} \leftarrow \{s\}$ and $\mathcal{V} \leftarrow \emptyset$
- **while** $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ **do**
  - let $p* \leftarrow \arg\min_{p \in \mathcal{L} \setminus \mathcal{V}} ||x_p - x_q||$
  - update $\mathcal{L} \leftarrow \mathcal{L} \cup N_{\text{out}}(p^*)$ and $\mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$
  - **if** $|\mathcal{L}| > L$ **then**
    - update $\mathcal{L}$ to retain closest L points to x_q
- return [closest k points from $\mathcal{L}$; $\mathcal{V}$]

Sparse Neighborhood Graphs

Sparse Neighborhood Graph (SNG) property (informal)

- For each point p, connect p to neighbors so that “dominated” points are pruned
- Intuition: from any starting node s, greedy steps can always move closer to the target point p until reaching p

Claim (as used in the paper)

- In an SNG, **GreedySearch(s, target, 1, 1)** converges to p from any start s

Why this is not enough for DiskANN

- SNG can still have large diameter (many hops), which is fatal on SSD

SSD-specific bottleneck

SSD latency is dominated by “hops”

- Each hop often implies at least one SSD neighborhood fetch
- Even if each hop improves distance, too many hops ⇒ too many SSD round trips

Pathological example

- Points on a line: a valid SNG can be a near-line graph
- Hop count can be O(n) in the worst case

DiskANN goal

- Enforce faster progress per hop (not just “closer”)

RobustPrune

Input:
p: point being assigned out-neighbors

V: candidate neighbor set

α≥1: “aggressiveness” threshold

R: maximum out-degree

Procedure:
1. Repeatedly pick the closest remaining candidate p* to p
2. Add edge p→p*
3. Remove candidates p′ that are “explained” by *:
 if α · d(p*,p′)≤d(p,p′)), then drop p’
4. Stop when |Nout(p)| =R

Intuition

Keeps neighbors that are both close and geometrically diverse
α>1 pushes the graph toward smaller diameter (fewer hops)

Algorithm 2: $\mathrm{RobustPrune}(p, \mathcal{V}, \alpha, R)$

Data: Graph G, point $p \in P$, candidate set $\mathcal{V}$, distance threshold $\alpha \geq 1$, degree bound R
Result: G is modified by setting at most R new out-neighbors for p

begin
- $\mathcal{V} \leftarrow (\mathcal{V} \cup N_{\mathrm{out}}(p)) \setminus \{p\}$
- $N_{\mathrm{out}}(p) \leftarrow \emptyset$
- **while** $\mathcal{V} \neq \emptyset$ **do**
  - $p^* \leftarrow \arg\min_{p' \in \mathcal{V}} d(p, p')$
  - $N_{\mathrm{out}}(p) \leftarrow N_{\mathrm{out}}(p) \cup \{p^*\}$
  - **if** $|N_{\mathrm{out}}(p)| = R$ **then**
    - `break`
  - **for** $p' \in \mathcal{V}$ **do**
    - **if** $\alpha \cdot d(p^*, p') \leq d(p, p')$ **then**
      - `remove` p' `from` $\mathcal{V}$

Why α>1 can yield logarithmic hops

Claim:

If each point's out-neighbors were computed by RobustPrune(p, P\{p}, α, n - 1) (i.e., prune against the full dataset), then for α>1:

GreedySearch(s, target, p, 1, 1) converges in **logarithmically many steps**

Reasoning

- The pruning rule aims to guarantee "multiplicative progress":
  at each step, there exists an edge that reduces remaining distance by a factor related to α\alphaα
- Repeated multiplicative decreases ⇒ logarithmic number of steps to reach the target neighborhood

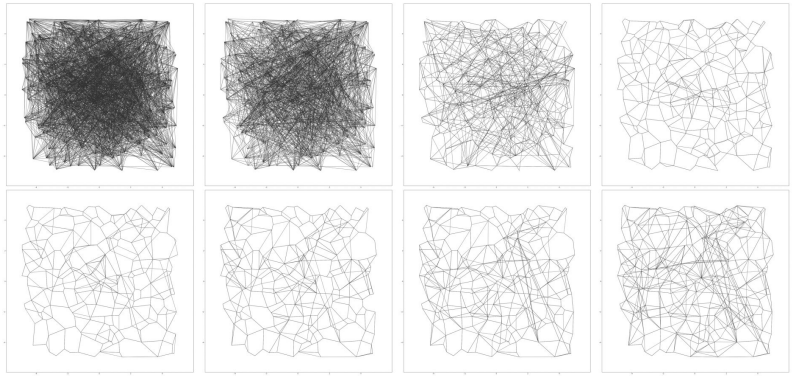
Important caveat

- Full-dataset candidate sets are infeasible (construction becomes ~O(n^2)))
- Vamana approximates this effect using a much smaller candidate set

Vamana Graph Construction Algorithm

Vamana index construction

- Initialize G as a random R-regular directed graph
- Choose a single global entry point s: the medoid of the dataset
- Iterate points p in random order:
    1. Run GreedySearch(s, p, 1, L); collect visited set Vp
    2. Run RobustPrune(p,Vp,α,R) to choose out-neighbors of p
    3. Add backward edges from neighbors back to p; if degree exceeds R, prune again
- Two passes:
    1. First pass with α=1
    2. Second pass with user-chosen α≥1 (adds longer-range edges)



What this does

- Uses a targeted candidate set Vp (visited nodes along a “path to p”) rather than all points

Vamana vs HNSW (Hierarchical Navigable Small World graphs) vs NSG (Navigating Spreading-out Graph)

- **Tunable α:**
  - Vamana: explicit α control
  - HNSW/NSG: effectively α=1 (no knob)
- **Candidate set for pruning:**
  - Vamana/NSG: prune against **all visited nodes**
  - HNSW: prune against **final candidate list** (more local), compensates via hierarchy
- **Initialization:**
  - NSG: needs approximate kNN graph (costly)
  - HNSW: starts empty
  - Vamana: starts random (paper reports better quality than empty)
- **Passes:**
  - Vamana: two passes; HNSW/NSG: one

DiskANN Storage Split

DiskANN index design

- **In RAM:** compressed vectors for all points (Product Quantization codes)
    - used for fast approximate distance comparisons during search
- **On SSD:**
    - the Vamana graph
    - the full-precision vectors x_p (stored adjacent to neighborhoods)

Why PQ is used

- Full vectors for 1B points do not fit in RAM
- PQ enables distance estimation cheaply during search, while the graph still guides region selection during build (build uses full precision)

Disk layout + implicit reranking

SSD record layout (per point iii)

- Store: full vector x_i + up to R neighbor IDs
- If degree < R: pad with zeros
- Fixed-size record ⇒ SSD offset is a simple calculation (no offset table needed in RAM)

Implicit reranking

- PQ distances are approximate (lossy)
- When fetching a node’s neighborhood from SSD, DiskANN also gets its full vector without extra reads (same sector)
- Final top-kkk results are reranked using true L2 distances on full vectors read during search

BeamSearch + Caching

BeamSearch (DiskANN search)

- Instead of expanding 1 closest frontier node, expand W closest nodes per round
- Fetch W neighborhoods in parallel (multiple reads in one round trip)
- W=1 reduces to greedy search; too large W wastes compute/bandwidth

Cache frequently visited vertices

- Cache nodes within C=3 or 4 hops of the entry point s
- Reduces SSD accesses early in the search
- C cannot be too large (cache grows quickly with hop radius)

Building a billion-point index under 64GB RAM (merge trick)

Overlapping partition + merge

- Run k-means to get k cluster centers (example used: k=40)
- Assign each point to its l nearest centers (typical l = 2)
- Build one Vamana graph per cluster (fits in memory)
- Merge graphs by union of edges into one global graph

Why overlap matters

- Overlap provides connectivity across partitions so GreedySearch can still succeed even when true neighbors span clusters

Experiments

Setup:

z840 workstation: 16 cores, 64GB RAM, SSDs (RAID-0)
Datasets for in-memory comparison: SIFT1M, GIST1M, DEEP1M and SIFT1B for large-scale

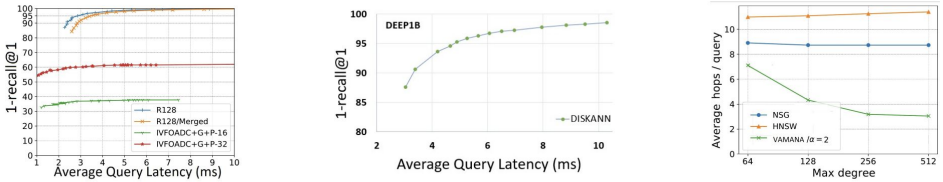


Figure 2: (a)1-recall@1 vs latency on SIFT bigann dataset. The R128 and R128/Merged series represent the one-shot and merged Vamana index constructions, respectively. (b)1-recall@1 vs latency on DEEP1B dataset. (c) Average number of hops vs maximum graph degree for achieving 98% 5-recall@5 on SIFT1M.

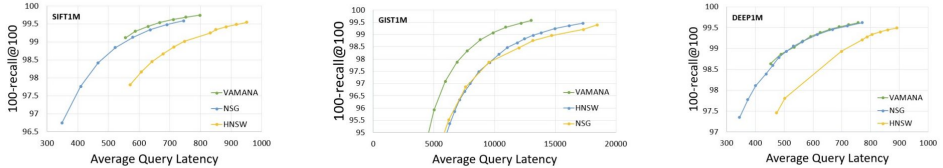


Figure 3: Latency (microseconds) vs recall plots comparing HNSW, NSG and Vamana.

Results

In-memory results

- Vamana and NSG outperform HNSW on benchmark recall–latency curves
- On GIST1M (960D), Vamana outperforms both NSG and HNSW
- Index build time example (DEEP1M): Vamana faster than HNSW and NSG

Large-Scale results

- DiskANN achieves high 1-recall@1 at a few ms latency on one node
- Merged build is close to one-shot performance, with modest latency overhead

Comparison vs IVF/PQ baselines

- IVFOADC+G+P with larger codes improves recall@1 but still plateaus well below DiskANN
- DiskANN achieves much higher 1-recall@1 at similar overall footprint

Assessment + Questions

Strengths: SSD-resident high-recall ANN; explicit diameter/degree control via α; clean end-to-end system

Weaknesses: limited formal theory beyond intuition; build time and engineering complexity; unclear dynamic updates; experiments focus on Euclidean

Future work: dynamic insert/delete; better theory for robust pruning; broader similarity metrics (cosine/IP); modern NVMe + larger RAM regimes

Questions

- When does overlap+merge fail (data distributions, connectivity collapse)?
- How sensitive are results to α,R,L,W,C? Can we auto-tune?
- What’s the right design in 2025 hardware (bigger RAM, faster NVMe)?
- How would you support streaming updates while preserving low hop count?