Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

Making DiskANN Adaptive: Navigation-Aware Search, Diversity Pruning, and Feedback-Driven Graph Refinement for Graph-Based ANN

Abhijeet Kumar[1], Krish Dange[2], Sai Jagadeesh[3]
[1,2,3]Student, Dept. of Data Science and Artificial Intelligence, Indian Institute of Technology Madras, Chennai, India

Abstract — Graph-based approximate nearest neighbour (ANN) indices such as DiskANN achieve strong recall–latency trade-offs, but the graph they search over is built once and then frozen: it never reacts to the queries it actually serves. This paper asks whether a Vamana-style index can be made to adapt as it is used, and reports an exploratory study built on a faithful re-implementation of the Vamana index. We add three mechanisms developed independently and integrated incrementally: a navigation-aware search that scores edges by how often they lead to true neighbours and biases traversal accordingly; a diversity-aware relaxation of the α-RNG pruning rule; and a periodic refinement pass that rewrites each node's neighbour list from accumulated usefulness statistics. On the SIFT1M benchmark the full system holds recall to within roughly 0.2–0.4% of the static baseline while performing up to 17.6% fewer distance computations at large beam widths. A milestone comparison shows that this reduction appears only once the refinement stage is introduced: the two-module configuration without refinement is statistically indistinguishable from the baseline in distance computations, which localises the algorithmic effect to graph refinement. We are explicit that the reduction does not currently lower wall-clock latency—the per-query bookkeeping the mechanisms require makes the adaptive system slower in absolute time under a parallel load—and we separate the hardware-independent metric (distance computations) from the implementation-dependent one (latency) throughout.

Key Words: Approximate nearest neighbour search, DiskANN, Vamana graph, adaptive indexing, edge pruning, graph refinement, vector search.

1. Introduction

Nearest neighbour search underlies recommendation, retrieval, and embedding look-up systems that now routinely index hundreds of millions to billions of high-dimensional vectors. Because exact search degrades to a linear scan in high dimensions, practical systems retrieve approximate nearest neighbours, trading a small loss in recall for a large gain in speed. Among the families of ANN methods—hashing, quantisation, trees, and graphs—graph-based indices currently offer the best recall–latency trade-offs on real datasets. DiskANN, built on the Vamana graph construction algorithm, extends this regime to billion-point datasets on a single machine by keeping the graph and full-precision vectors on SSD while caching compressed vectors in RAM [1].

A property shared by Vamana, HNSW [2], and NSG [3] is that the graph is a static artefact. It is constructed from the geometry of the base set and then never changes, regardless of which regions of the space queries actually fall in. Edge selection is governed entirely by distance heuristics; the α-RNG pruning rule that gives Vamana its short diameter does not optimise any explicit objective and receives no feedback from search. Where the query distribution is skewed or drifts over time, this leaves something on the table: edges that are never useful are retained, and the order in which neighbours are examined is fixed at build time.

This paper documents an attempt to relax that rigidity. Working from a clean re-implementation of the Vamana index, we introduce three mechanisms that let the index observe and respond to its own query traffic: a navigation-aware search (Section 4.1), an entropy-based diversity pruning rule (Section 4.2), and an adaptive graph refinement pass (Section 4.3). The three modules were developed independently by the three authors and integrated in stages; Section 9 records the division of work, and Section 6 uses that staging to attribute the observed effects to specific mechanisms.

We are candid about what we found. On SIFT1M the mechanisms together preserve recall and reduce the number of distance computations, particularly at larger beam widths, but they increase wall-clock latency in the current implementation. We therefore frame this as an exploratory systems study rather than a claim of a uniformly faster index, and we devote Section 6 to explaining exactly where the gain originates and why latency moves as it does.

In summary, this paper contributes: (i) a navigation-aware search that learns per-edge usefulness online and uses it to bias candidate admission without disturbing the exact distances recall is measured on; (ii) a diversity-aware relaxation of α-RNG pruning that preserves long-range edges, reducing to the original rule at zero strength; (iii) a feedback-driven refinement pass that rewrites adjacency lists from accumulated statistics using per-node adaptive thresholds; (iv) a concurrency design (sharded edge-score tables) that makes the bookkeeping cheap enough to run under a parallel query load; and (v) an honest empirical study on SIFT1M that isolates, via a staged-integration comparison, the mechanism responsible for the observed reduction in distance computations, while being explicit about the accompanying latency cost.

2. Background: Vamana and DiskANN

We summarise only the parts of DiskANN our work touches; the full system is described in [1]. Let $P = \{p_1, \ldots, p_n\}$ with $x_p \in \mathbb{R}^d$ and Euclidean distance $d(p,q) = \|x_p - x_q\|_2$. For a query q and a target k, the quality of a returned set X against the true neighbours G is recall@$k = |X \cap G|/k$.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

Algorithm 1 GREEDYSEARCH(s,q,k,L)

1: $\mathscr{L} \leftarrow \{s\}$, $\mathscr{V} \leftarrow \emptyset$
2: **while** $\mathscr{L} \setminus \mathscr{V} \neq \emptyset$ **do**
3: $p^* \leftarrow \arg\min_{p \in \mathscr{L} \setminus \mathscr{V}} d(p,q)$
4: $\mathscr{L} \leftarrow \mathscr{L} \cup N_{out}(p^*)$; $\mathscr{V} \leftarrow \mathscr{V} \cup \{p^*\}$
5: **if** $|\mathscr{L}| > L$ **then** retain the L closest points to q
6: **end if**
7: **end while**
8: **return** the k closest points in $\mathscr{L}$

Among ANN families, locality-sensitive hashing partitions space with random projections, inverted-file and quantisation methods cluster the base set and scan a few cells, and tree methods recursively partition space; each trades index size and build cost against query accuracy differently. Graph methods, to which Vamana belongs, instead connect each point to a bounded set of neighbours and answer a query by walking the graph greedily toward it. On modern high-dimensional benchmarks this family gives the best recall at a given query cost, which is why DiskANN, HNSW [2], and NSG [3] are all graph-based; our work inherits that setting and asks only whether the graph, once built, has to stay fixed.

2.1 Greedy search

Vamana searches a directed graph $G = (P,E)$ by a best-first traversal. Starting from a fixed medoid s, it maintains a candidate list $\mathscr{L}$ of size L and a visited set $\mathscr{V}$; at each step it expands the closest unvisited candidate $p^* = \arg\min_{p \in \mathscr{L} \setminus \mathscr{V}} d(p,q)$, adds the out-neighbours of p^* to $\mathscr{L}$, and trims $\mathscr{L}$ back to its L closest entries. The walk stops when no unvisited candidate remains, and the top k are returned. Algorithm 1 states the procedure.

2.2 Robust pruning

To keep the graph diameter small, Vamana bounds each out-degree by R and selects neighbours with the α-RNG rule. Processing candidates in increasing distance from a node p, a candidate c is discarded if some already-selected neighbour n satisfies

$$d(p,c) > \alpha \cdot d(c,n), \qquad \alpha \geq 1, \tag{1}$$

i.e. n "covers" c. The multiplier $\alpha > 1$ forces distance to the query to drop by a factor of at least α along the search path, bounding the number of hops to roughly $O(\log n)$. Algorithm 2 gives the pruning rule; our entropy variant (Section 4.2) modifies only the threshold tested on line 5.

2.3 Disk layout, beam search, and compression

DiskANN stores the graph and full-precision vectors on SSD and keeps product-quantised (PQ) vectors in RAM for fast approximate distances [4]. A query is guided through the graph using the compressed distances, which are cheap and resident in memory, while the comparatively expensive SSD reads fetch the neighbourhoods needed to expand the frontier. To amortise the per-read latency of an SSD, DiskANN replaces the strictly greedy expansion of one node per step with beam search: it expands a small beam of W frontier nodes per hop, issuing their neighbourhood reads together so that the round-trip cost is shared. Because product quantisation is lossy, the candidate ranking it produces is approximate;

Algorithm 2 ROBUSTPRUNE$(p,\mathscr{C},\alpha,R)$

1: sort candidates $\mathscr{C}$ by increasing $d(p,\cdot)$; $S \leftarrow \emptyset$
2: **for** each $c \in \mathscr{C}$ in order **do**
3: **if** $|S| = R$ **then break**
4: **end if**
5: $keep \leftarrow$ **true**
6: **for** all $n \in S$ **do**
7: **if** $d(p,c) > \alpha \cdot d(c,n)$ **then** $keep \leftarrow$ **false**; **break**
8: **end if**
9: **end for**
10: **if** $keep$ **then** $S \leftarrow S \cup \{c\}$
11: **end if**
12: **end for**
13: $N_{out}(p) \leftarrow S$

DiskANN therefore re-ranks the final candidates using full-precision vectors, which it stores adjacent to each node's adjacency list on disk so that they arrive in the same read as the neighbourhood. Frequently visited vertices near the medoid are cached in RAM to cut the number of SSD round trips further. These storage mechanics are orthogonal to our contributions, which operate entirely on the in-memory graph and the search loop and would compose with the on-disk machinery unchanged; we describe them only to place our work correctly within the system, and our experiments (Section 5) use an in-memory configuration so that the algorithmic effects are not confounded by I/O.

3. Baseline Implementation

Our enhancements are layered on a from-scratch C++ implementation of Vamana, which serves as the static baseline throughout. The index is built by the standard incremental procedure: the graph is initialised randomly, points are inserted in a random permutation, each insertion runs a greedy search to gather a candidate set, robust pruning selects up to R neighbours, and backward edges are added with re-pruning whenever a node's degree exceeds γR. Our build makes a single pass; we did not implement the two-pass ($\alpha = 1$ then $\alpha > 1$) schedule of the original paper, which we note as a minor deviation.

Two implementation choices in the baseline matter for what follows. First, the candidate list in greedy search is a flat `std::vector` kept sorted by true distance, with the "expanded" flag stored inline in each entry, rather than the more common pair of `std::sets`. New neighbours are gathered into a small batch, sorted, and folded in with a single `std::inplace_merge` per hop; finding the next node to expand is a forward linear scan over contiguous, cache-resident memory. This is logically identical to a tree-based candidate set but avoids a heap allocation and a pointer chase on every insertion. Second, the build is parallelised with per-node mutexes. Both choices were made so that the adaptive bookkeeping added later does not have to fight an already allocation-heavy inner loop.

The build is multi-threaded: points are inserted concurrently, with each node's adjacency list protected by its own lock so that two threads modifying different neighbourhoods never serialise on a global structure. The two operations that must be atomic with respect to a node are appending a back-

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

edge and re-pruning when the degree exceeds γR; both take only that node's lock. Degree growth is bounded by the γR trigger, which re-applies robust pruning to bring an over-full neighbourhood back to R, so the average out-degree stays close to R throughout the build rather than drifting upward as back-edges accumulate. This matters for the adaptive layer because the refinement pass (Section 4.3) inherits the same per-node locking discipline and the same degree bound, which is what lets it rewrite a neighbourhood in place without a separate global reorganisation step and without violating the degree invariant the search relies on for its hop bound.

4. Proposed Enhancements

All three mechanisms are backward-compatible: with their strength parameters set to zero they reproduce the original algorithm bit-for-bit, so the static baseline is a special case of the adaptive system rather than a separate codebase.

4.1 Navigation-aware search

The intuition is that, over many queries, some edges repeatedly lie on productive paths to true neighbours while others are dead ends, yet the static search treats them identically. We attach two counters to every directed edge $(u \to v)$: $\mathrm{trav}(u,v)$, the number of times the edge has been followed, and $\mathrm{help}(u,v)$, the number of those traversals in which v ended up in the final size-L candidate set. The usefulness of an edge is the ratio

$$\mathrm{useful}(u,v) = \frac{\mathrm{help}(u,v)}{\mathrm{trav}(u,v)} \in [0,1], \tag{2}$$

defined as 0 for an edge that has never been traversed, so that a freshly loaded index behaves exactly like the baseline until evidence accumulates.

During search, the usefulness score does not alter the stored distances—which must remain exact for recall to be measured correctly—but it lowers the admission threshold for a candidate. When deciding whether a neighbour v of the current node u should enter the candidate list, the comparison uses an adjusted distance

$$d'(u,v) = d(q,v)\big(1 - \beta \cdot \mathrm{useful}(u,v)\big), \qquad \beta = 0.10, \tag{3}$$

so that a perfectly useful edge receives at most a 10% discount on the comparison only. The candidate, if admitted, is stored with its true distance $d(q,v)$. The cap $\beta = 0.10$ is deliberately small: enough to reorder near-ties in favour of historically good edges without letting learned bias override geometry.

It is worth being precise about why this cannot corrupt the recall we report. The usefulness discount in (3) is applied only to the comparison that decides whether a neighbour is admitted into the working candidate list; the value physically stored for an admitted candidate is its exact distance $d(q,v)$, and the final top-k returned to the user are selected by exact distance. The learned signal therefore changes the order in which the graph is explored — potentially admitting a historically useful node a few steps earlier — but never changes which point is reported as closer than which. In the limit the bonus can only enlarge the set of nodes the walk considers; it cannot fabricate a false neighbour, because every candidate is ultimately re-judged on its true distance. This is the same separation of concerns that lets product quantisation guide search with approximate distances while a full-precision re-rank produces the answer, and it is what makes recall a fair metric to compare across the static and adaptive systems.

An early version credited only the $k = 10$ returned neighbours as "helpful". On a million-point graph this is far too sparse a signal—almost every navigation edge goes unrewarded—so scores barely move. We instead mark every node in the final size-L candidate set as helpful, which yields a learning signal one to two orders of magnitude denser depending on L.

Because the counters are read on every hop and written once per query, under a parallel query load they are a natural contention point. Rather than a single global mutex—which an early profile showed to be the dominant cost—we shard the score table into 64 independent maps keyed on the source node, $\mathrm{shard}(u) = u \bmod 64$, each with its own mutex. All out-edges of a node live in the same shard, so one shard lock covers every score needed for a hop, and threads expanding different source nodes almost never collide. Write-back groups edges by shard and acquires one shard lock at a time, never two simultaneously, removing any possibility of deadlock.

4.2 Entropy-based diversity pruning

The α-RNG rule in (1) prunes purely on proximity, which can leave a graph that is dense locally but poorly connected across regions: a candidate bridging two clusters is easily discarded because some near neighbour already covers it. We relax the rule for candidates that genuinely reach new territory. For a candidate c and the already-selected set S we measure its diversity as the distance to its nearest selected neighbour, $\mathrm{div}(c,S) = \min_{n \in S} d(c,n)$, normalise it by the candidate's own distance to the node being pruned, and clamp it to $[0,1]$,

$$\mathrm{div}_c(c) = \min\left(1,\ \frac{\mathrm{div}(c,S)}{d(p,c)}\right), \tag{4}$$

and use it to relax α per candidate:

$$\alpha_{\mathrm{eff}}(c) = \alpha\big(1 + \lambda\, \mathrm{div}_c(c)\big). \tag{5}$$

A more diverse candidate gets a larger α_{eff}, which makes the covering condition $d(p,c) > \alpha_{\mathrm{eff}}\, d(c,n)$ harder to satisfy, so diverse edges survive. The clamp is not cosmetic: without it, an isolated candidate in an empty region could earn an unbounded threshold and displace closer, more useful neighbours. With $\lambda = 0$ the rule collapses exactly to the original α-RNG. The extra work is one minimum over pairwise distances already computed for the covering check, so the $O(R^2)$ per-node cost of pruning is unchanged. We note for reproducibility that λ is a build-time parameter whose committed default is 0; reproducing an entropy-active configuration requires setting it explicitly.

4.3 Adaptive graph refinement

The first two mechanisms change how the graph is searched and built; the third changes the graph itself, after deployment, from what search has learned. A refinement pass is triggered automatically every 1000 queries and rewrites adjacency lists from the accumulated usefulness scores. A periodic batch trigger was chosen over a per-query update so that the modulo check sits in the once-per-query search wrapper rather

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

Algorithm 3 Adaptive refinement, per node u

$N \leftarrow$ copy of u's neighbour list (under node lock, then released)
read useful(u,v) $\forall v \in N$; accumulate total traversals
if total traversals $<$ MIN_TRAVERSALS $(= 10)$ **then**
    skip u ▷ too little evidence
end if
$T_{remove} \leftarrow$ 30th percentile of $\{\text{useful}(u,v)\}$
$T_{boost} \leftarrow$ 80th percentile of $\{\text{useful}(u,v)\}$
for all $v \in N$ **do**
    if useful$(u,v) > T_{boost}$ **then** mark boosted
    else if useful$(u,v) \geq T_{remove}$ **then** mark kept
    else drop v
    end if
end for
sort boosted, kept buckets by usefulness (desc.)
emit boosted first (duplicated $\leq$ MAX_BOOST_COPIES), then kept
if result empty **then** rescue single best edge ▷ connectivity
end if
trim to $\leq R$ neighbours; write back only if changed

than the per-hop inner loop, and so that updates are stable and amortised. Algorithm 3 gives the per-node procedure.

Three design points are worth drawing out. First, the thresholds are per-node percentiles of that node's own usefulness distribution, computed by linear interpolation on the sorted scores, rather than global constants. This makes the rule scale-free: a node with uniformly mediocre edges and a node with a few excellent edges are each judged relative to themselves, which is more robust to skew than a fixed cutoff. Second, the cold-node guard (MIN_TRAVERSALS) prevents refinement from acting on statistically meaningless zeros; early in a workload most nodes are skipped, which is intended. Third, reinforcement is implemented by duplicating a high-usefulness edge (capped at two copies). Because greedy search deduplicates visits through a visited bitset, a duplicate never causes a redundant distance computation; it simply gives the edge more chances to be reached from different hops. After sorting and emitting, the list is trimmed to R, so duplication competes for the same degree budget as everything else and the degree bound is preserved. The system thus runs a closed loop, search $\rightarrow$ update scores $\rightarrow$ refine $\rightarrow$ improved search, in which the structure traversed by the next batch of queries is shaped by the last.

4.4 A worked illustration

To make the loop concrete, consider a single node u whose adjacency list holds eight out-edges. Early in a workload, queries that pass through u follow some of these edges far more often than others: suppose three edges are repeatedly on paths that reach a true neighbour (they accumulate high help/trav ratios), three are followed occasionally but rarely useful, and two are almost never traversed at all. Until u has been visited at least ten times its scores are treated as unreliable and it is left untouched. Once enough evidence has accumulated, a refinement pass computes the 30th and 80th percentiles of u's eight usefulness values: the two never-useful edges fall below the 30th percentile and are dropped, the three

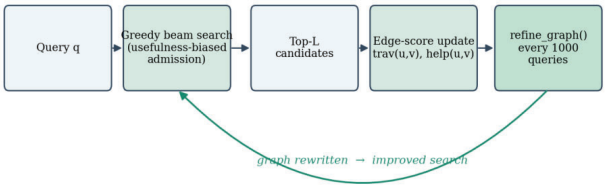


Figure 1: The closed feedback loop. A standard greedy beam search is augmented with usefulness-biased admission; the result set updates per-edge counters; and a periodic refinement pass rewrites the graph. Each module reduces to the static baseline at zero strength.

strongest rise above the 80th and the top one or two are duplicated, and the remainder are kept and re-sorted so the best edges are examined first on the next visit. The list is then trimmed back to at most R entries. The net effect on u is a smaller, better-ordered neighbourhood that the next batch of queries traverses with fewer wasted distance computations — and, occasionally, the loss of a real edge that a future query would have needed, which is the source of the small recall deficit reported in Section 6. Multiplying this picture across the warm nodes of the graph is exactly what produces the widening distance-computation gap at larger L, where more nodes have crossed the evidence threshold.

4.5 System architecture and the closed loop

Figure 1 shows how the three modules compose into a single feedback loop around an otherwise standard DiskANN search path. A query enters the greedy beam search, whose candidate-admission test is biased by the learned edge usefulness (Section 4.1). The top-L result set is then used to update the per-edge counters, and once every REFINE_INTERVAL queries the accumulated scores drive a refinement pass (Section 4.3) that rewrites the graph the next batch of queries will traverse. Entropy pruning (Section 4.2) sits at build time and at every refinement, shaping which edges are eligible to survive. Crucially, every module degenerates to the original DiskANN behaviour when its strength parameter is zero, so the loop is an augmentation of the standard search path rather than a replacement: the static index is recovered exactly by setting $\beta = \lambda = 0$ and disabling the refinement trigger.

4.6 Complexity and overhead

We summarise the cost each module adds over the static baseline. *Memory.* The score table stores two 32-bit counters per distinct traversed edge. In the worst case this approaches the edge count of the graph, $O(nR)$, but in practice only edges actually followed by queries are materialised, so the footprint tracks the union of visited subgraphs rather than the whole graph. *Per-query search overhead.* Each hop performs one extra shard-locked lookup per neighbour for the usefulness score and one multiply–subtract for the adjusted threshold in (3); both are $O(R)$ per hop and do not change the asymptotic search cost, which remains dominated by the $O(\text{hops} \cdot R)$ distance computations. Score write-back after a query is $O(|\text{traversed edges}|)$, grouped into at most NUM_EDGE_SHARDS lock acquisitions. *Refinement.* A refinement pass visits each node once; per node the dominant cost is sorting its $\leq R$ usefulness scores and rebuilding the adjacency list, i.e. $O(R \log R)$, for an $O(nR \log R)$ total that

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

is amortised over REFINE_INTERVAL queries. The pruning change of Section 4.2 reuses distances already computed for the α-RNG test, leaving the $O(R^2)$ per-node pruning cost unchanged. None of the additions change the asymptotic complexity of build or search; they add constant factors whose wall-clock effect is the subject of Section 6.3.

4.7 Parameter choices

The mechanisms introduce four tunable constants beyond the standard Vamana parameters; we record the values used and the reasoning, while noting plainly that we did not perform a systematic sweep. The priority-bonus cap $\beta = 0.10$ limits the learned discount to at most ten percent so that history can reorder near-ties but never override geometry; values much larger risk admitting genuinely distant nodes on the strength of stale statistics. The entropy strength λ is left at its backward-compatible default of 0 in the committed build. The refinement percentiles (30th for removal, 80th for boosting) were chosen so that a clear minority of weak edges is pruned while only the strongest are reinforced, keeping each pass conservative; we skip any node with fewer than 10 accumulated traversals (the cold-node guard) and trigger a refinement pass every 1000 queries, trading statistical reliability against adaptation speed. A proper sensitivity analysis over these constants is left to future work (Section 8); the present results should be read as one reasonable operating point, not a tuned optimum.

5. Experimental Setup

We evaluate on SIFT1M [7]: one million 128-dimensional base vectors, 10,000 queries, and provided ground truth, under Euclidean distance. The index is built with $R = 32$, build list $L = 75$, $\alpha = 1.2$, and $\gamma = 1.5$. Search is evaluated for $k = 10$ over beam widths $L \in \{10, 20, 30, 50, 75, 100, 150, 200\}$, reporting mean recall@10, mean distance computations per query, and mean and 99th-percentile latency. Queries within each L setting are executed in parallel.

Metrics. We report three quantities. *Recall@10* is the fraction of a query's true ten nearest neighbours that appear in the returned set, averaged over the 10,000 queries; it measures answer quality. *Average distance computations* counts, per query, the number of times the full 128-dimensional Euclidean distance is evaluated during search, averaged over all queries; it is a deterministic, hardware-independent measure of the algorithmic work performed, and is the metric on which graph ANN methods are most fairly compared. *Mean and 99th-percentile latency* report the wall-clock time per query in microseconds; these depend on the machine, compiler, memory system, and—because we measure under a parallel load—on lock contention between concurrent queries, so we treat them as implementation-dependent and secondary. Recall and distance computations are functions only of the algorithm and the graph state, so they are reproducible across machines; latency is not.

We evaluate three configurations, corresponding to the staged integration of the modules: the static baseline; a two-module system combining navigation-aware search and diversity pruning (our Milestone 2 configuration, without refinement); and the full three-module system that adds adaptive refinement. This staging is what lets us attribute effects to individual mechanisms in Section 6. Because distance computations and recall are deterministic functions of the algorithm and the graph, they are comparable across runs and machines; latency is not, and we treat it accordingly. We also note that refinement fires during the query sweep and state carries across L settings, so the three-module graph mutates as it is benchmarked—a point we return to in Section 8.

6. Results and Discussion

Table 1 reports the static baseline and the full three-module system side by side, and Figure 2 plots the three metrics.

6.1 Recall is preserved

Across every beam width the adaptive system stays within 0.0017 to 0.0042 of the baseline recall, always slightly below it (Figure 2a). At $L = 200$ both exceed 0.994. For practical purposes the retrieval quality is unchanged; the small, consistent deficit is itself informative and we return to it in Section 6.5.

6.2 Distance computations and the milestone progression

The clearest positive result is in distance computations (Figure 2b). At small L the two systems are level—-0.2% at $L = 10$—but the gap grows monotonically: 7.2% fewer at $L = 50$, 13.7% at $L = 100$, and 17.6% at $L = 200$. Distance-computation count is the standard hardware-independent proxy for the algorithmic work a graph ANN method performs.

The staged integration of the modules lets us say which mechanism produces this reduction. Table 2 and Figure 3 place the two-module configuration (navigation-aware search and diversity pruning, our Milestone 2 system) between the baseline and the full system. The two-module counts are statistically indistinguishable from the baseline—within 0.1–0.4 computations per query across all L—and the reduction emerges only when the adaptive refinement stage is added. The effect is therefore localised to graph refinement: navigation scoring re-orders traversal without removing work, and diversity pruning (with its committed default $\lambda = 0$) does not alter the search graph, so neither changes the comparison count. This is a useful confirmation that the reduction is not an artefact of one of the lighter-touch modules.

6.3 Latency increases

Wall-clock latency tells the opposite story (Figure 2c, last columns of Table 1): the adaptive system is slower at every setting, by factors of roughly 1.5 to 2$\times$ in the mean, and its 99th-percentile latency is both higher and far noisier (for example 16,440 μs at $L = 75$ against 1826 μs for the baseline). We do not present this as a positive result. It is the direct cost of the bookkeeping the mechanisms require—score snapshots on every hop, score write-back per query, periodic refinement, and the lock traffic all of this generates under a parallel load. The fact that fewer distance computations coincide with higher latency is precisely why we keep the two metrics separate.

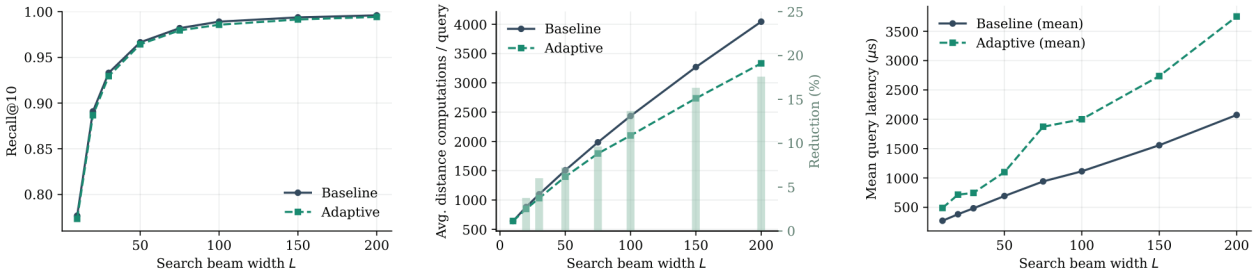
6.4 The recall–cost trade-off

Reporting each metric against L can flatter or penalise a method depending on where the curves happen to sit, so we also plot the two systems in the space that actually matters for an ANN index: recall against the work needed to achieve

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

Table 1: Static baseline vs. the full three-module adaptive system on SIFT1M, $k = 10$. Recall is essentially unchanged; distance computations fall, increasingly so at larger L; mean latency rises.

	Recall@10			Avg. distance computations			Mean latency (μs)		
L	Baseline	Adaptive	Δ	Baseline	Adaptive	Reduction	Baseline	Adaptive	Ratio
10	0.7768	0.7734	−0.0034	640.8	641.8	−0.2%	270.6	488.5	1.81×
20	0.8909	0.8867	−0.0042	880.9	847.6	3.8%	381.7	715.7	1.88×
30	0.9331	0.9295	−0.0036	1098.1	1032.0	6.0%	483.2	745.1	1.54×
50	0.9665	0.9643	−0.0022	1507.5	1398.5	7.2%	692.2	1099.2	1.59×
75	0.9820	0.9796	−0.0024	1985.3	1794.0	9.6%	941.2	1873.0	1.99×
100	0.9891	0.9857	−0.0034	2434.9	2101.9	13.7%	1113.6	2000.4	1.80×
150	0.9938	0.9915	−0.0023	3269.7	2735.9	16.3%	1557.8	2736.1	1.76×
200	0.9960	0.9943	−0.0017	4044.6	3332.8	17.6%	2073.8	3751.5	1.81×



(a) Recall@10 **(b)** Distance computations (bars: reduction) **(c)** Mean query latency

Figure 2: SIFT1M, $k = 10$. The recall curves are nearly indistinguishable; the distance-computation gap widens with L; mean latency is consistently higher for the adaptive system.

Table 2: Distance computations per query across the staged integration. The two-module system tracks the baseline; the reduction appears only with adaptive refinement.

L	Baseline	Two-module	Three-module
10	640.8	643.0	641.8
20	880.9	884.2	847.6
50	1507.5	1511.4	1398.5
100	2434.9	2438.1	2101.9
200	4044.6	4048.1	3332.8

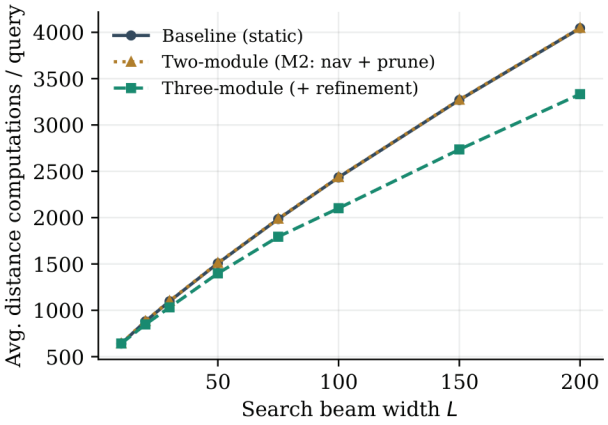


Figure 3: Distance computations across the staged integration. The baseline and two-module curves overlap; only the three-module system, with adaptive refinement, separates from them, and increasingly so at larger L.

it. Figure 4 shows recall@10 versus average distance computations, zoomed to the high-recall regime that applications care about. The two curves lie almost on top of one another. This is the most honest single picture of what the adaptive system does: it does not move the index off the baseline's recall-versus-work frontier so much as it slides along it, trading a small amount of recall for a commensurate reduction in distance computations. At a matched recall the adaptive system is at best marginally cheaper and at worst indistinguishable, which is exactly what the sparsification account in Section 6.5 predicts. We read this as a neutral-to-slightly-favourable result on the hardware-independent axis, and we resist the temptation to crop the axes to manufacture a larger-looking gap.

6.5 Where the reduction comes from

It is tempting to read the reduction as the search having learned to find the same neighbours along shorter paths. The mechanism is more prosaic, and being honest about it matters. Refinement removes the bottom 30% of each refined node's edges by usefulness; over the sweep this sparsifies the graph, so refined nodes expose fewer out-neighbours and each hop computes fewer distances. That directly lowers the count, and it explains the shape of the curve—negligible at $L = 10$ (few queries have elapsed, few nodes are warm enough to refine) and largest at $L = 200$ (the graph has by then been refined dozens of times). The same sparsification produces the small recall deficit: pruning real edges occasionally removes a path to a true neighbour. The adaptive system is therefore buying a modest reduction in work with a modest reduction in recall, mediated by graph sparsity, with the navigation bonus steering the beam toward the edges refinement has chosen to keep.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

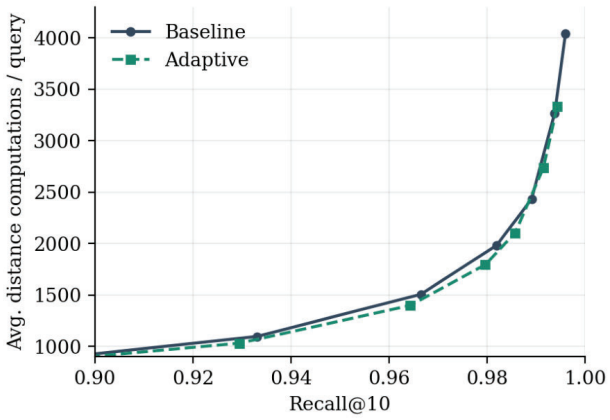


Figure 4: Recall@10 versus average distance computations per query (high-recall regime). The adaptive system slides along the baseline's recall–cost frontier rather than dominating it: a matched recall costs roughly the same number of comparisons in both systems.

This is a reasonable trade to surface, but it is not a free lunch.

As for latency: at these scales the per-hop work is small and cache-resident, so the constant overhead added—hashing edge keys, taking shard locks, snapshotting scores, writing counters back—is comparable to or larger than the distance arithmetic saved, and lock contention under the parallel benchmark adds the variance visible in the inflated tail. The sharded design makes this far cheaper than a global lock, but cheaper than catastrophic is not negligible. We believe the overhead is reducible—lock-free counters, sampling rather than recording every traversal, and decoupling refinement from the serving path are all plausible—but we have not done that work and will not claim a latency improvement we did not measure.

6.6 When should the loop help?

The recall–cost picture (Section 6.4) and the sparsification account together suggest where an adaptive loop of this kind is and is not worth its overhead. It should help most when three conditions hold. First, the query distribution is skewed or non-stationary, so that a meaningful fraction of edges are genuinely useless for the queries actually served and can be pruned without hurting the queries that matter; on a uniform benchmark like SIFT1M this condition is barely met, which is consistent with the near-coincident recall–cost curves we observe. Second, the application is recall-tolerant, able to accept a few tenths of a percent less recall in exchange for less work — many retrieval-augmented and recommendation settings are, since a missed tenth neighbour rarely changes a downstream result. Third, the deployment is comparison-bound rather than overhead-bound: where each distance evaluation is expensive (very high dimension, or on-disk full-precision re-ranking) the distance-computation savings dominate the bookkeeping cost, inverting the latency picture we measured for cheap in-memory distances. Conversely, the loop is a poor fit for a stationary workload with cheap distances and a strict tail-latency budget, which is close to the regime our own benchmark occupies and is why we are careful not to over-sell the result. Stating these conditions explicitly is also a falsifiable prediction: the warm-then-freeze, skewed-workload experiment of Section 8 is precisely the test of whether the loop delivers when its preconditions are satisfied.

7. Related Work

Graph-based ANN is dominated by HNSW [2], NSG [3], and Vamana/DiskANN [1]; all build a navigable graph from base-set geometry, search it greedily, and produce a static index. DiskANN's contribution is to make such an index disk-resident at billion scale through product quantisation [4] and beam search, and re-ranking with full-precision vectors is also used by Zoom [6].

Our work sits within an active line on query-aware and learned graph search, and we position it honestly against that line. Baranchuk et al. [8] learn a routing function over a similarity graph by attaching learned vertex representations, directly targeting the local-minima problem our navigation bonus also addresses, but with an offline-trained model rather than online counters. Feng et al. [9] apply reinforcement learning to route on proximity graphs for recommendation, which is close in spirit to our edge-usefulness reinforcement, and Li et al. [10] learn adaptive early-termination criteria for graph search from query features. On the construction side, RoarGraph [11] builds a graph under the explicit guidance of a historical query distribution and reports large speed-ups on out-of-distribution workloads, and follow-up work [12] fixes query-region graph defects dynamically. Relative to these, our contribution is narrower and lighter: a fully online, in-place feedback loop that augments an existing Vamana index with $O(1)$-per-edge counters and a periodic local rewrite, requiring no learned model, no offline query set, and no graph reconstruction. The trade-off is that we do not yet demonstrate the search-time wins those heavier methods achieve—our present gain is a reduction in distance computations attributable to refinement-driven sparsification, not a net latency improvement. Closing that gap, and connecting the percentile heuristic to an explicit objective of the kind a learned index optimises, is the most promising direction beyond this study. FreshDiskANN [5] addresses a complementary axis, adapting the graph to a changing dataset via streaming insertions and deletions, whereas we adapt to a changing query distribution over a fixed dataset.

Table 3 places these approaches on the axes that matter for our question: what the index adapts to, what signal drives adaptation, whether adaptation happens online without retraining, and whether it needs an offline query set. Our method occupies a deliberately lightweight corner—online, counter-driven, no training and no held-out queries—which is what makes it cheap to bolt onto an existing index, and also why its gains are smaller than methods that pay a heavier construction cost.

8. Future Work

The honest conclusion of Section 6—comparable recall, fewer distance computations, but no net latency win—points to a clear research agenda, and we set it out concretely so the work can be picked up.

A clean evaluation protocol. The most pressing fix is methodological: warm the index on a held-out query stream, freeze it, and only then benchmark, so that the reported numbers describe a fixed graph rather than one mutating

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

Table 3: Where this work sits among adaptive / query-aware graph ANN methods.

Method	Adapts to	Online	No query set
HNSW / NSG / DiskANN	— (static)	—	✓
Learning to Route [8]	graph routing	no (train)	no
RL routing [9]	graph routing	no (train)	no
Learned term. [10]	stop rule	no (train)	no
RoarGraph [11]	query distrib.	no (build)	no
FreshDiskANN [5]	dataset	✓	✓
This work	query distrib.	✓	✓

mid-measurement. Repeated trials on documented hardware would turn the indicative latency figures into defensible ones.

An ablation. Navigation scoring, entropy pruning, and refinement should be toggled independently across the eight on/off combinations, so that each module's contribution to recall, distance computations, and latency can be attributed rather than inferred. Our milestone comparison is a first step but covers only two of those points.

The workload the method is built for. SIFT1M queries are roughly stationary, which is precisely the regime in which an adaptive index has least to gain. The central hypothesis—that the loop helps when the query distribution is skewed or drifts—needs a non-stationary or out-of-distribution workload, of the kind RoarGraph [11] uses, to be tested at all.

Making the overhead pay for itself. Turning fewer comparisons into faster queries is an engineering problem with several plausible attacks: lock-free or sharded-atomic counters, sampling a fraction of traversals rather than recording every edge, moving refinement off the serving path onto a background thread, and compacting the score table so it stays in cache. A method that demonstrably reduces wall-clock latency at equal recall would change the headline result.

A principled objective. The percentile thresholds and the fixed bonus are deliberately simple heuristics. Replacing them with an explicit objective—for instance, learning edge weights that minimise expected hops to the true neighbours, in the spirit of [8, 9]—would connect this work to the learned-index literature and might let refinement add useful long-range edges rather than only re-weighting existing ones. Evaluation on GIST1M and DEEP1M, and at billion scale on the SSD-resident configuration DiskANN targets, would establish whether the loop survives outside a single in-memory benchmark.

9. Conclusions

We set out to make a Vamana-style ANN index respond to its own query traffic, and built three mechanisms toward that end: navigation-aware search, diversity-aware pruning, and feedback-driven graph refinement, all designed to reduce to the static baseline at zero strength. On SIFT1M the resulting system holds recall to within a few tenths of a percent of the baseline while performing up to 17.6% fewer distance computations at large beam widths, at the cost of higher wall-clock latency in its current, instrumentation-heavy form. A milestone comparison localised the distance-computation reduction to the refinement stage, and we traced that reduction to graph sparsification rather than to a smarter search. The honest summary is that adaptive feedback in a graph ANN index is cheap enough to be worth pursuing and does not, on this evidence, harm recall—but that turning "fewer comparisons" into "faster queries" is an engineering problem we have not yet solved. Immediate next steps are an ablation isolating each mechanism, a clean warm-then-freeze evaluation protocol, a genuinely skewed query workload, and a lighter-weight scoring path.

Two takeaways generalise beyond this specific system. First, distance computations and wall-clock latency can move in opposite directions, so reporting only one of them — as is common — can paint a misleading picture; an honest study of any indexing optimisation should separate the algorithmic-work metric from the implementation-dependent one, as we have tried to do throughout. Second, a staged integration is a cheap and underused way to attribute an observed effect to a specific component without a full factorial ablation: because our two-module configuration was measured independently, we could localise the entire distance-computation reduction to the refinement stage with confidence, even though a complete ablation remains future work. We hope both the positive finding (adaptive feedback need not cost recall) and the negative one (it did not, here, buy speed) are useful to others building feedback into graph indices.

Author Contributions

The three modules were developed independently and integrated in stages. Krish Dange designed and implemented the navigation-aware search (Section 4.1): the edge-usefulness scoring, the bounded priority bonus, the redesigned flat candidate list, the sharded concurrent score table, and the overall search path. Sai Jagadeesh designed and implemented the entropy-based diversity pruning (Section 4.2): the diversity metric, the clamped normalisation, and the diversity-adjusted effective threshold within robust pruning. Abhijeet Kumar designed and implemented the adaptive graph refinement (Section 4.3): the per-node percentile thresholding, the boost/keep/remove classification with connectivity rescue, the degree-bound preservation, and the periodic refinement trigger. The experimental evaluation, the milestone comparison, and the writing were carried out jointly. Authors are listed alphabetically by first name.

Acknowledgement

We thank the instructors and teaching staff of the Algorithms for Data Science course at IIT Madras for providing the baseline GraphANN/Vamana framework on which this work builds, and for their guidance during the project.

10. Implementation and Reproducibility

The system is implemented in C++ on top of a from-scratch Vamana index; build and search are parallelised with OpenMP. Edge-usefulness state is held in NUM_EDGE_SHARDS $= 64$ hash maps, each guarded by its own mutex and keyed on the 64-bit packing of a directed edge $(u \to v)$; hop counts are maintained with atomic counters. The build takes the standard Vamana parameters ($R = 32$, build $L = 75$, $\alpha = 1.2$, $\gamma = 1.5$) and an additional `entropy_scale` (λ) argument that defaults to 0, so an unmodified build reproduces the static baseline exactly. The search entry point exposes the beam width L and target k; the bonus cap β and the refinement constants are compile-time settings.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

All code, the build and run scripts, the parameter settings, and the raw result tables that back every figure are available in the project repository.[1] The SIFT1M benchmark can be reproduced end to end with a single driver script that downloads the dataset, builds the index, and runs the search sweep over $L \in \{10, \ldots, 200\}$. We restate, for honesty, two facts a reader reproducing our numbers should know: first, the committed build leaves $\lambda = 0$, so the entropy-pruning module is inactive in the reported runs and the headline reduction is produced by the navigation and refinement modules; second, the refinement trigger fires during the evaluation sweep and state carries across L settings, so the three-module numbers describe a graph that evolves over the run rather than a single frozen index. Both points are discussed in Section 5, and a clean warm-then-freeze protocol (Section 8) would remove the second confound.

References

[1] S. J. Subramanya, Devvrit, R. Kadekodi, R. Krishnaswamy, and H. V. Simhadri, "DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[2] 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.*, 2018.

[3] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph," *Proc. VLDB Endowment*, vol. 12, no. 5, pp. 461–474, 2019.

[4] H. Jégou, M. Douze, and C. Schmid, "Product Quantization for Nearest Neighbor Search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 1, pp. 117–128, 2011.

[5] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri, "FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search," arXiv:2105.09613, 2021.

[6] M. Zhang and Y. He, "Zoom: SSD-based Vector Search for Optimizing Accuracy, Latency and Memory," arXiv:1809.04067, 2018.

[7] L. Amsaleg and H. Jégou, "Datasets for Approximate Nearest Neighbor Search (TEXMEX corpus)," http://corpus-texmex.irisa.fr/, 2010.

[8] D. Baranchuk, D. Persiyanov, A. Sinitsin, and A. Babenko, "Learning to Route in Similarity Graphs," in *Proc. 36th Int. Conf. on Machine Learning (ICML)*, PMLR 97, pp. 475–484, 2019.

[9] C. Feng, D. Lian, X. Wang, Z. Liu, X. Xie, and E. Chen, "Reinforcement Routing on Proximity Graph for Efficient Recommendation," *ACM Trans. Inf. Syst.*, vol. 41, no. 1, pp. 1–27, 2023.

[10] C. Li, M. Zhang, D. G. Andersen, and Y. He, "Improving Approximate Nearest Neighbor Search Through Learned Adaptive Early Termination," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 2539–2554, 2020.

[11] M. Chen, K. Zhang, Z. He, Y. Jing, and X. S. Wang, "RoarGraph: A Projected Bipartite Graph for Efficient Cross-Modal Approximate Nearest Neighbor Search," *Proc. VLDB Endowment*, vol. 17, no. 11, pp. 2735–2749, 2024.

[12] "Dynamically Detect and Fix Hardness for Efficient Approximate Nearest Neighbor Search," arXiv:2510.22316, 2025.

[1] `https://github.com/krishdange27/graphann`