

NemoANN: I/O-Efficient Filtered Vector Search on SSDs with Graph Tunneling

Nakyung Lee*

Sungshin Women's University
Seoul, Republic of Korea
220254009@sungshin.ac.kr

Jiwoong Park

Sungshin Women's University
Seoul, Republic of Korea
jwpark12@sungshin.ac.kr

Soobin Cho*

Sungshin Women's University
Seoul, Republic of Korea
20231163@sungshin.ac.kr

Gyuyeong Kim

Yonsei University
Seoul, Republic of Korea
gyuyeong.kim@yonsei.ac.kr

Abstract

We introduce NemoANN, an I/O-efficient SSD-based Approximate Nearest Neighbor Search (ANNS) system for filtered search. Existing SSD-based systems either waste I/O on post-filtering or require expensive filter-aware index rebuilds. Our key insight is that a non-matching node can never be a result, so the search needs only its small neighbor list, not the full-precision vector on SSD. NemoANN therefore checks each predicate before the SSD read and tunnels through non-matching nodes using in-memory routing metadata, preserving graph connectivity without modifying the index. On a 100M-vector benchmark, NemoANN reduces SSD reads by about 10× and improves throughput by 8.0× over post-filtering and by 5.0× over PipeANN-Filter, a state-of-the-art filtered SSD system, at 10% selectivity and 90% recall. The gains over post-filtering grow at lower selectivity, reaching 13.1× at 5% selectivity.

CCS Concepts

• Information systems → Information storage systems; Information retrieval.

Keywords

filtered vector search, approximate nearest neighbor search, SSD

ACM Reference Format:

Nakyung Lee, Soobin Cho, Jiwoong Park, and Gyuyeong Kim. 2026. NemoANN: I/O-Efficient Filtered Vector Search on SSDs with Graph Tunneling. In *ACM Symposium on Cloud Computing Volume 2 (SoCC '26 Vol 2.)*, November 18–20, 2026, Singapore, Singapore. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3828161.3857852>

1 Introduction

High-dimensional vectors have become a universal representation for unstructured data across diverse applications, from recommendation systems to retrieval-augmented generation (RAG) [8, 16,

20, 32]. At the billion-scale, exact search is prohibitively expensive, so approximate nearest neighbor search (ANNS) with graph-based indexes is widely adopted for its superior latency-recall trade-off [10, 13, 16, 18]. Recently, to scale beyond memory capacity, SSD-based ANNS systems (e.g., DiskANN [23], PipeANN [5]) have been proposed, making large-scale vector search practical on a single commodity server. These systems store full-precision vectors and graph topology on SSDs, retaining only compressed vectors in memory for approximate distance computation.

In practice, vector search queries rarely target the entire dataset. Instead, they commonly restrict results using metadata predicates such as document type, time range, or product category [9, 26]. General-purpose SSD-based ANNS systems such as DiskANN and PipeANN handle such *filtered search* through post-filtering. They fetch candidate nodes from SSD, compute exact distances, and then discard nodes that do not satisfy the predicate. This approach preserves the original graph traversal but wastes I/O and computation on non-matching nodes. For example, at 10% selectivity, up to 90% of SSD reads and exact-distance computations may be spent on vectors that cannot appear in the final result.

Filtering before SSD access could eliminate this waste, but directly skipping non-matching nodes (naïve joint-filtering) breaks graph traversal [14, 36]. A skipped node cannot be expanded, even though its neighbors may provide essential paths to matching results. Filtered-ANNS systems restore this connectivity by changing the index itself. Filter-aware graphs such as F-DiskANN [4] build label-specific edges and entry points, so they avoid reads of non-matching nodes, but they support only the label space fixed at build time and must be rebuilt when the filter schema changes. Predicate-agnostic designs instead densify the graph with additional edges so that the search can skip most non-matching nodes without losing connectivity [7, 19]. On SSD, PipeANN-Filter [7] adds 2-hop neighbors to each node record and speculatively expands nodes that pass in-memory probabilistic filters. This keeps predicates flexible with little memory, but enlarges the on-disk index and still reads non-matching nodes, both false positives and bridges, from SSD. Existing systems therefore pay for filtering in one of three places: wasted SSD reads (post-filtering), predicate flexibility and rebuild cost (filter-aware graphs), or a larger index with speculative reads (densified graphs). In this context, we ask: *Can filtered vector search avoid unnecessary SSD I/O on an unmodified graph index and on-disk layout while leaving the predicate to query time?*

*Nakyung Lee and Soobin Cho contributed equally to this work.



This work is licensed under a Creative Commons Attribution 4.0 International License. SoCC '26 Vol 2., Singapore, Singapore

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2816-7/2026/11

<https://doi.org/10.1145/3828161.3857852>

To answer this question, we present NemoANN, an I/O-efficient SSD-based graph ANNS system for filtered search. Our key insight is that the search needs a non-matching node's neighbors, not its vector. Such a node cannot appear in the results, so the search visits it only as a waypoint to its neighbors. The visit needs only the node's neighbor list and an approximate distance, and both fit in memory. Existing SSD-based systems co-locate a node's vector and neighbor list so that a single aligned read supports both graph traversal and exact-distance computation; as a result, even a waypoint visit costs a full SSD read.

NemoANN therefore keeps a compact copy of the graph's neighbor lists in memory as the neighbor store. SSD-based ANNS systems already keep the compressed vectors in memory, so the neighbor store completes the in-memory information for traversing a non-matching node. During traversal, NemoANN checks the filter before issuing an SSD read. A matching node is fetched for exact-distance computation. A non-matching node is not fetched; NemoANN moves the search through it using its in-memory neighbor list and approximate distance. We call this *graph tunneling*. Unlike densified graphs, tunneling routes over the original edges and leaves the index and its on-disk layout unchanged. Unlike speculative filtering, it checks the exact predicate before I/O, so non-matching nodes still bridge the search but never cost an SSD read. NemoANN thus trades memory for SSD I/O; for 100M vectors, the neighbor store requires 12.3 GB at the default setting and can be reduced to about 3.4 GB while still outperforming post-filtering baselines.

We evaluate NemoANN on datasets ranging from 10M to 1B vectors with both synthetic and real multi-label metadata [22]. On BigANN-100M, NemoANN reduces SSD I/Os by about 10 $\times$ relative to post-filter baselines and improves throughput by 8.0 $\times$ at 10% selectivity and 90% recall. As filters become more selective, the gains grow further, reaching 13.1 $\times$ at 5% selectivity. Against PipeANN-Filter, NemoANN achieves 5.0 $\times$ higher throughput at 10% selectivity while keeping the original index, which is 14 $\times$ smaller on SSD than PipeANN-Filter's densified index. NemoANN also closes the gap with in-memory search; it matches or surpasses Vamana in single-thread latency despite using an SSD and 3.4 $\times$ less memory.

In summary, this paper makes the following contributions:

- We characterize the I/O dilemma in filtered SSD-based graph search, where post-filtering wastes SSD I/O and computation while naïve joint-filtering breaks graph connectivity and severely degrades recall (§2).
- We present NemoANN, which turns a pre-I/O filter failure into a tunneling decision that serves the node from its in-memory neighbor list, enabling I/O-efficient filtered search on an unmodified graph index (§3).
- We evaluate NemoANN on datasets from 10M to 1B vectors with synthetic and real metadata, showing about 10 $\times$ fewer SSD I/Os and 8.0 $\times$ higher throughput than SSD-based post-filter baselines at 10% selectivity and matched recall (13.1 $\times$ at 5% selectivity), and 5.0 $\times$ higher throughput than PipeANN-Filter (§5).

2 Background and Motivation

We begin by reviewing SSD-based graph ANNS (§2.1), then explain the I/O dilemma in filtered vector search (§2.2). We next motivate

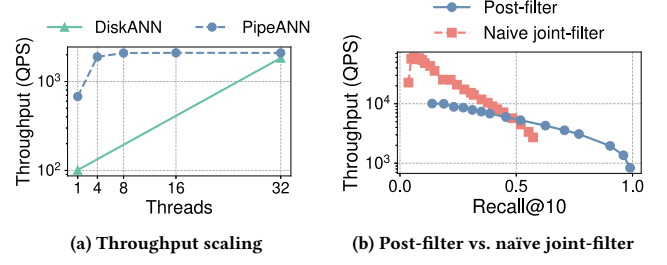


Figure 1: Motivating experiments on BigANN-100M with 10% selectivity. (a) Post-filtering systems plateau early because of wasted per-node work. (b) Naïve joint-filtering destroys graph connectivity, degrading throughput and recall.

the need for I/O-efficient joint-filtering on unmodified graph indexes (§2.3).

2.1 SSD-Based Graph ANNS

Graph-based approximate nearest neighbor search (ANNS) organizes vectors as a proximity graph, where each vector is a node and edges connect approximate neighbors. Given a query, the search starts from an entry point and greedily traverses toward nodes that appear progressively closer to the query, returning the closest visited nodes as approximate nearest neighbors [5, 16, 23].

At the billion scale, storing full-precision vectors in memory is often infeasible on a commodity server [10, 24, 25, 29, 31]. For example, storing 128-dimensional float32 vectors together with a degree-64 graph at a scale of one billion points requires approximately 768 GB of memory, far exceeding the memory capacity of a commodity server. This motivates SSD-based graph ANNS systems such as DiskANN [23] that place node records on NVMe SSDs while keeping only compressed vectors in memory. Product Quantization (PQ) encodes each vector as a short code (e.g., 32 bytes), enabling approximate distance computation without an SSD read. During search, these PQ distances prioritize which candidates to expand, while exact distances from the full-precision vectors on SSD determine the final results. In DiskANN, each node occupies a 4 KB-aligned record containing its raw vector and neighbor list. Search proceeds in rounds. The system selects a beam of promising candidates, issues SSD reads for them, waits for the batch to finish, and then continues.

PipeANN [5] replaces this batch-and-wait design by pipelining reads with `io_uring`. It continuously submits reads, polls completions, and processes nodes with many I/Os in flight. This overlap substantially improves throughput. However, in both systems, graph traversal still incurs a random SSD read for every expanded node, and each read costs tens to hundreds of microseconds. As a result, per-node SSD I/O remains the dominant cost.

2.2 The I/O Dilemma in Filtered Search

Filtered vector search is common. Production queries rarely search the entire dataset. Enterprise RAG restricts by document type and access control, legal search filters by jurisdiction and time range, and e-commerce search narrows by product category. Such

metadata predicates are pervasive in practice [9, 19, 26, 35]. The *selectivity* s of a filter is the fraction of the dataset that satisfies the filter. Selectivities in the 1–10% range are common in both benchmark and production settings [11, 17, 38].

Post-filtering wastes resources. Existing SSD-based graph ANNS systems such as DiskANN and PipeANN handle filters by *post-filtering*. They read every node selected for expansion from SSD and only then check the predicate. This approach is general, but it pays full per-node cost (SSD I/O, exact distance computation, and candidate management) regardless of whether the node matches the filter. This waste becomes costly under multi-threads. In Figure 1 (a), PipeANN initially outperforms DiskANN by overlapping I/O and computation. However, as concurrency increases, they converge because both ultimately spend most of their budget processing nodes that fail the filter.

Naïve joint-filtering breaks graph search. A natural alternative is to push the predicate before I/O and skip non-matching nodes during traversal, which is called *joint-filtering* (also known as in-filtering [7]). However, this is fundamentally problematic for graph search, because non-matching nodes are not only poor result candidates but also routing waypoints that connect the search to other regions of the graph. Skipping their expansion prevents the search from reaching their neighbors, even when those neighbors match the predicate. Because many paths between matching nodes pass through non-matching nodes, naïve joint-filtering on an unmodified graph fragments the set of matching nodes into many small disconnected components, particularly at low selectivity.

Figure 1(b) confirms this. At the same throughput, naïve joint-filtering achieves much lower recall than post-filtering, and its maximum recall plateaus at ~57% even at 2.7K QPS, compared to >99% for post-filtering. Increasing the search list size L cannot overcome this limitation since unreachable components remain invisible regardless of how many candidates the traversal explores.

Filter-aware indexes sacrifice flexibility. Prior work on filtered search incorporates label information into the index structure. For example, Filtered-DiskANN (F-DiskANN) [4] augments the graph with label-aware connectivity and entry points, eliminating I/O for non-matching nodes. However, they require the label space to be known at build time. Supporting a new predicate or changing the taxonomy requires rebuilding the index, which can take days at the billion scale. They also cannot handle ad-hoc predicates such as subset predicates over multi-label metadata or range predicates over continuous attributes. In-memory systems follow the same direction by fusing attributes into the graph or by maintaining filter-specific subindexes [1, 11, 15, 27].

Densified graphs trade index size for connectivity. Predicate-agnostic designs keep a single index for all predicates and instead add edges so that the search can skip most non-matching nodes without losing connectivity. ACORN [19] densifies an in-memory graph for this purpose. PipeANN-Filter [7] appends a random sample of 2-hop neighbors to each on-disk record, identifies a superset of matching nodes with in-memory probabilistic filters such as per-node Bloom filters, and verifies the exact attributes stored in each record when results are reranked. A per-query cost model chooses among this speculative in-filtering, pre-filtering, and post-filtering. This design keeps predicates flexible and its in-memory filter state small, but it pays in two ways. First, each densified record is larger

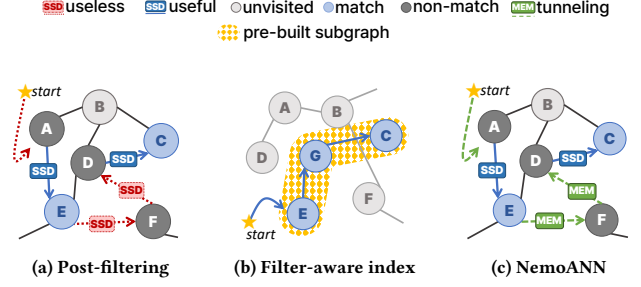


Figure 2: Different approaches for filtered search.

than the original one, which enlarges both the on-disk index and every read. Second, the search still reads non-matching nodes from SSD, namely false positives of the probabilistic filters and the non-matching neighbors it deliberately explores as bridges when too few matching neighbors are found.

2.3 Toward I/O-Efficient Joint-filtering

Rethinking why joint-filtering seems impractical. Checking the filter before I/O is not a new goal in itself; F-DiskANN avoids reads for non-matching nodes by rebuilding the graph around a fixed label space, and PipeANN-Filter avoids most of them by densifying every records. On an unmodified graph and on-disk layout, however, skipping a non-matching node discards a useful waypoint, so a system must either read the node anyway or accept the loss of connectivity.

The problem is not joint-filtering itself, but the coupling between traversal and I/O in existing systems. Specifically, a node's neighbor list is stored alongside its vector in a single SSD record; unless that record is cached in memory, discovering the node's neighbors requires reading it from the SSD. This means that even determining the next traversal hop requires a disk I/O, regardless of whether the node matches the filter.

Insight: the missing piece is small. Graph traversal requires only two pieces of per-node information, an approximate distance and a neighbor list. Existing SSD-based systems already keep compressed vectors in memory to estimate distances, but store each neighbor list on SSD together with the full-precision vector. Thus, to traverse a node without an SSD read, the only missing information is the neighbor list. Because neighbor lists are much smaller than full-precision vectors, a compact copy of the graph adjacency can be kept in memory at a practical cost.

Our approach. Based on this observation, NemoANN maintains a compact in-memory copy of each node's neighbor list. Before issuing an SSD read, NemoANN evaluates the filter predicate using in-memory metadata. If the node matches the predicate, NemoANN fetches its full-precision vector from SSD and computes its exact distance. Otherwise, NemoANN skips the SSD read but continues traversal using the node's in-memory compressed vector and neighbor list. We call this routing through non-matching nodes without SSD access *graph tunneling*. Unlike speculative filtering, the check is exact, so a non-matching node never costs an SSD read, whether or not the search routes through it. Because NemoANN uses the

Table 1: Comparison with existing works.

	PipeANN-Filter [7]	F-DiskANN [4]	NemoANN
Filter type	Speculative-filtering	Filter-aware Index	Joint-filtering
I/O-efficient	×	✓	✓
Any filter predicates	✓	×	✓
No index rebuild	✓	×	✓

original graph without rebuilding or modifying it for specific filters, it supports arbitrary predicates known only at query time.

Comparison to existing works. Figure 2 and Table 1 summarize how NemoANN differs from prior work. Post-filtering [5, 23] reads every traversed node from SSD and applies the predicate only afterward, while a filter-aware index [4] eliminates I/O for non-matching nodes but fixes the label space at build time and requires a rebuild for new predicates. PipeANN-Filter [7] keeps predicates flexible with small in-memory state, but enlarges every on-disk record and still reads false positives and bridge nodes. NemoANN checks the exact predicate before I/O and traverses non-matching nodes in memory, preserving connectivity and eliminating their SSD reads without restricting the predicate or changing the index; the cost is the DRAM for the neighbor and filter stores.

3 NemoANN Design

3.1 Overview

Figure 3 shows the architecture of NemoANN. The central design principle is to serve a node’s two roles separately. Whether a node’s full-precision vector is fetched from SSD should depend on whether the node can contribute to the query result, not on whether the node lies on the search path. A node that fails the filter retains only its routing role, and NemoANN serves that role entirely from memory.

NemoANN realizes this by inserting a filter-aware dispatch stage into the asynchronous search pipeline. Each candidate is dispatched to one of two paths based on a lightweight in-memory predicate check. The two paths differ only in how a node is processed. Both feed candidates into the same sorted frontier, so the rest of the search algorithm is unaffected. This preserves the asynchronous I/O pipeline [5] intact while ensuring that every SSD read serves a node that can appear in the final result. The following subsections describe each component in detail.

3.2 Key Components

NemoANN relies on two in-memory structures. The *filter store* decides which path a candidate takes, and the *neighbor store* makes the second path possible.

Filter store. The *filter store* is a memory-resident structure that holds each node’s filter metadata and evaluates the predicate after an $O(1)$ lookup by node ID. The evaluation is a direct check on the node’s stored metadata rather than a probabilistic test such as a Bloom filter; exactness matters because an erroneous pass would waste an SSD read and an erroneous fail would silently exclude a valid result. An equality filter compares a single fixed-width label in constant time, while a multi-label subset filter tests each query tag against the node’s in-memory tag set, so its cost grows with the number of query tags. Even this most expensive case is a short in-memory loop, far cheaper than the SSD read it avoids. Its output

is not an admission test but a routing decision. A failing node is not dropped but handed to the tunneling path, which is what lets NemoANN check the predicate before any I/O without losing that node as a waypoint. The check and tunneling therefore depend on each other; checking the filter before I/O without tunneling is exactly the naïve joint-filtering that breaks connectivity (§2).

The filter store is intentionally decoupled from the graph index. It is loaded from a separate metadata file and can be replaced independently. This separation is what enables NemoANN to support arbitrary predicates (equality, range, multi-label subset, or conjunctions thereof) without rebuilding the graph. Adding a new predicate type only requires providing the corresponding metadata; the graph index and the search algorithm remain unchanged. Its memory cost scales with metadata size rather than vector dimensionality (Table 2).

NemoANN places this gate at the earliest point in the search pipeline, before both hot-node cache lookup and SSD submission. This placement is safe only because the gate has two exits. If the check dropped failing nodes instead of routing them, it would remove their routing role along with their result role. A node that fails therefore incurs neither SSD I/O nor exact distance computation, and bypasses the entire per-read processing chain, not just the disk access.

Neighbor store. The *neighbor store* replicates each node’s adjacency list from the on-disk graph into memory. Its purpose is to enable tunneling. When a node fails the filter, the search needs to access the node’s neighbors to continue traversal, but the neighbors are normally embedded in the node’s SSD record alongside the full-precision vector. The neighbor store extracts just the adjacency information (a list of up to $R_{\max}$ neighbor IDs per node) so that tunneling can proceed without any SSD read.

Like the filter store, the neighbor store is built at load time from the existing on-disk index and does not modify it. Because each node in the Vamana graph stores neighbors in order of proximity, retaining the first $R_{\max}$ entries preserves the closest and most useful neighbors for routing. The structure supports $O(1)$ lookup by node ID, ensuring that tunneling adds negligible latency compared to the SSD read it replaces.

3.3 Graph Tunneling

A filter-failing node may still be needed for traversal, so NemoANN tunnels through it in memory rather than fetching its vector from SSD. The key observation behind tunneling is that each node in a graph search serves two distinct roles, as a *result candidate* that may appear in the final top- K and as a *routing waypoint* whose edges connect the search to other regions of the graph. When a filter predicate is present, a non-matching node is ineligible as a result candidate yet may still be essential as a routing waypoint; the two roles diverge. Post-filtering ignores this divergence, fetching every visited node from SSD regardless of whether it can contribute to the result. Tunneling separates the two roles. A node that fails the filter is marked as filter-failing and excluded from the final top- K , so it only needs to fulfill its waypoint role. This role requires only the node’s adjacency list and approximate (PQ) distances, both of which reside in memory.

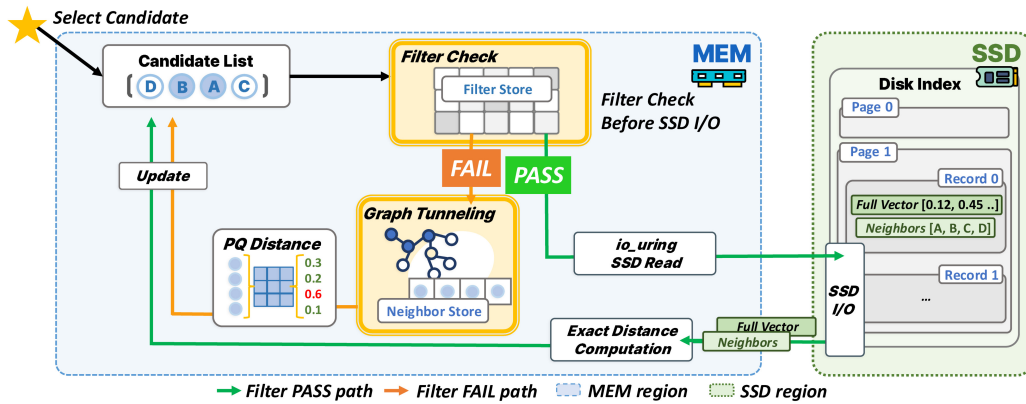


Figure 3: NemoANN overview. A candidate is first checked against the in-memory filter store. Filter-passing nodes follow the normal SSD path, an asynchronous read followed by exact distance computation. Filter-failing nodes take the graph tunneling path. The neighbor store provides the adjacency list, PQ distances are computed in memory, and promising neighbors are inserted back into the candidate list. Both paths feed into the same sorted frontier.

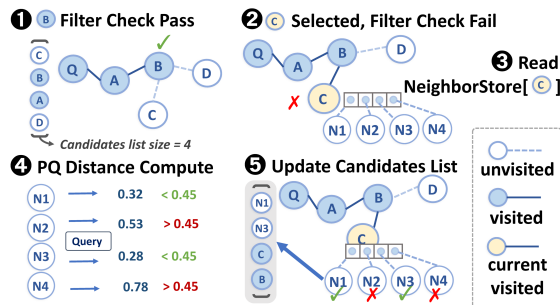


Figure 4: Graph tunneling example. ❶ Node B passes the filter and follows the normal SSD path. ❷ Node C is selected next but fails the filter check. ❸ NemoANN reads C ’s neighbors (N_1 – N_4) from the neighbor store in memory. ❹ PQ distances to the query are computed for each neighbor; neighbors below the frontier threshold (here 0.45) are kept (N_1, N_3) and the rest (N_2, N_4) are discarded. ❺ Promising neighbors are inserted into the candidate list, and C is marked visited but ineligible for results.

This separation is effective because of a large cost asymmetry between the two paths. An SSD random read takes on the order of $100\ \mu\text{s}$, while a neighbor store lookup and PQ distance computation complete in sub-microsecond time. Tunneling therefore replaces the most expensive per-node operation with one that is $\sim 100\times$ cheaper, and the benefit scales with the fraction of visited nodes that fail the filter.

This also clarifies the difference from naïve joint-filtering, which skips non-matching nodes without expanding their neighbors and so disconnects the frontier (§2). Tunneling avoids this by preserving neighbor expansion. It eliminates only the SSD read, not the traversal step, so the search can cross arbitrarily long non-matching regions through a sequence of cheap in-memory hops.

Algorithm 1 NemoANN search loop (simplified).

```

Require: Query  $q$ , filter predicate  $f$ , search list size  $L$ , result size  $K$ , I/O width  $W$ 
1:  $C \leftarrow$  sorted candidate frontier, initialized with the entry point
2:  $Q \leftarrow \emptyset$  ▷ in-flight SSD reads
3: while  $C$  has undispatched candidates or  $Q \neq \emptyset$  do
4:   while  $|Q| < W$  and  $C$  has an undispatched candidate do
5:      $c \leftarrow$  best undispatched candidate in  $C$ 
6:     if  $\text{filter\_store}[c]$  satisfies  $f$  then
7:       Submit asynchronous read for  $c$ 's SSD record
8:       Add  $c$  to  $Q$  and mark  $c$  as dispatched
9:     else
10:       $\text{nbrs} \leftarrow \text{neighbor\_store}[c]$ 
11:      for each unvisited  $n \in \text{nbrs}$  do
12:         $d_n \leftarrow \text{PQ\_dist}(q, n)$ 
13:        if  $d_n$  improves the frontier threshold then
14:          Insert  $(n, d_n)$  into  $C$ 
15:        end if
16:      end for
17:      Mark  $c$  as visited and filter-failing
18:    end if
19:  end while
20: for each completed read  $r$  returned by the SSD do
21:   Compute the exact distance for  $r$ 
22:   Expand  $r$ 's neighbors from the returned SSD record
23:   Insert promising neighbors into  $C$ 
24:   Remove  $r$  from  $Q$  and mark  $r$  as visited
25: end for
26: end while
27: return top- $K$  filter-passing candidates from  $C$ 

```

Example. In Figure 4, the candidate list is $[C, B, A, D]$. B passes the filter and takes the SSD path, but C fails. Instead of reading C from SSD, NemoANN tunnels through it, scoring C ’s stored neighbors with PQ distances and inserting the promising ones, which updates the list to $[N_1, N_3, C, B]$. C never leaves memory, yet still connects the search to new candidates N_1 and N_3 .

Putting it together. Algorithm 1 summarizes the integrated search loop. The important structural property is that both paths, SSD retrieval and in-memory tunneling, feed candidates into the same sorted frontier. The rest of the search remains unchanged. The algorithm repeatedly selects promising frontier nodes, expands them, and returns the top- K filter-passing candidates.

3.4 Design Implications and Trade-offs

Approximate distances for tunneling. Graph tunneling uses PQ distances rather than exact distances from full vectors. This is acceptable because traversal only needs a good priority signal to decide which neighbors to explore next, not the precise ranking required for the final result set. Since filter-failing nodes are excluded from the results, the top- K output is always drawn from filter-passing nodes evaluated with exact distances, regardless of the approximation used during traversal.

Two factors may affect tunneling quality. Tunneled nodes are prioritized with PQ distances, and when $R_{\max} < R$ the in-memory adjacency contains only a subset of each node's neighbors. This truncation applies only to filter-failing nodes. A filter-passing node is read from SSD in any case, and the read returns its complete adjacency list of up to R neighbors, so the search expands every passing node at full degree (Algorithm 1). Truncation therefore never removes a node from the graph; it only narrows the out-edges used while routing through a non-matching node. Neither factor affects the final-result rule above, and PQ errors do not accumulate across hops, since each candidate is rescored independently against the original query and the frontier is globally resorted after every expansion. When needed, the system can compensate by modestly increasing the search list size L , which we evaluate in §5.

Consecutive non-matching nodes. At low selectivity, a tunneled node often exposes neighbors that also fail the filter. NemoANN handles this case naturally. If such a neighbor later becomes the best frontier candidate, it is tunneled as well. In this way, single-hop tunneling composes into multi-hop traversal without any special multi-hop operator.

This design has two advantages. First, it avoids materializing all nodes within multiple hops of a filtered region, which would grow the work exponentially. Second, it lets the frontier be re-ranked after every hop, preserving the best-first character of the search. Long non-matching stretches are a workload-dependent challenge. If filter predicates are tightly correlated with graph locality, the search may need to cross larger filtered regions before reaching matching nodes, which can require a larger frontier or a larger $R_{\max}$ to maintain recall. We explicitly investigate these cases in the evaluation rather than assuming label independence.

Memory overhead. The primary cost is the neighbor store:

$$\text{MEM}_{\text{neighbor}} = N \times (1 + R_{\max}) \times 4 \text{ bytes}, \quad (1)$$

where N is the number of nodes and $R_{\max}$ is the number of in-memory neighbors retained per node. Table 2 shows representative sizes for $N = 100\text{M}$ and 1B. Combined with the PQ vectors already kept by the baseline and a modest filter store, the total memory footprint remains practical on commodity servers. This footprint reduces the largest dataset that a server with fixed memory can hold. However, the reduction is adjustable at runtime because Eq. 1 is linear in $R_{\max}$. At $R_{\max}=8$, the neighbor store costs about as much as the PQ codes the baseline already keeps, and at the default $R_{\max}=32$ it costs about four times as much (Table 2).

NemoANN treats $R_{\max}$ as a *runtime* parameter, not an index-build parameter. The neighbor store is constructed at load time by scanning the on-disk graph and extracting the first $R_{\max}$ neighbors of each node, without modifying the index itself. Operators can

Table 2: Memory overhead for N -size datasets.

Component	Formula	Size (100M)	Size (1B)
Filter store (single-label)	$N \cdot B$	100 MB	1 GB
Filter store (multi-label)	varies	~900 MB	~9 GB
Neighbor store ($R_{\max} = 32$)	$N \cdot 33 \cdot 4 \text{ B}$	12.3 GB	123 GB
Neighbor store ($R_{\max} = 16$)	$N \cdot 17 \cdot 4 \text{ B}$	6.3 GB	63 GB
Neighbor store ($R_{\max} = 8$)	$N \cdot 9 \cdot 4 \text{ B}$	3.4 GB	34 GB
PQ vectors (baseline)	$N \cdot 32 \text{ B}$	3.2 GB	32 GB

therefore adjust $R_{\max}$ freely across deployments or restarts without rebuilding the graph. The load-time scan is a one-time cost per process start, not a per-query cost; it reads the graph file once in order, so it runs at sequential-read bandwidth and grows linearly with the index size, adding one pass to the startup loading PipeANN already performs for PQ vectors and metadata. Larger values of $R_{\max}$ improve tunneling quality by exposing more candidate routes through non-matching regions, while smaller values reduce memory footprint and per-node in-memory work. We use $R_{\max}=32$ as the default. At billion scale this keeps the neighbor store at 123 GB (Table 2), so it fits in a 256 GB server together with the PQ vectors, whereas $R_{\max}=48$ would already require about 183 GB by Eq. 1; tighter memory budgets can halve the store to 63 GB with $R_{\max}=16$. Unlike a degree-16 graph that fixes connectivity at build time, NemoANN truncates only the in-memory routing copy of a full-degree graph, allowing $R_{\max}$ to be increased and reloaded without rebuilding the index.

3.5 Discussion

Index updates. A label change rewrites only the node's filter-store entry, leaving the graph and neighbor store untouched, while a filter-aware index must alter graph edges for the same change. A vector insertion or deletion rewrites a bounded set of on-disk adjacency lists in dynamic Vamana-style indexes [6], and since the neighbor store is a truncated copy of these lists, NemoANN only re-copies the entries the update already rewrote on SSD. The maintenance cost thus scales with the update's write set rather than the index size, and in the worst case the whole store can be rebuilt by the same sequential scan used at load time. We leave a concurrent implementation and an evaluation under updates to future work.

Extremely low selectivity. Graph traversal is not the best strategy at every selectivity. When the matching set is small enough, scanning its PQ codes in memory and reranking the top candidates with a few SSD reads costs less than any traversal; prior systems similarly route highly selective queries to specialized structures [11, 15]. NemoANN already maintains the two structures this strategy needs, the filter store and the PQ codes, so a per-query router that sends extremely selective queries to an exhaustive scan is a natural extension, which we leave to future work.

4 Implementation

We implement NemoANN in C++ by extending the PipeANN code-base [5]. The implementation adds two read-only metadata structures and modifies the candidate dispatch path in PipeANN's asynchronous search loop.

Table 3: Datasets used in evaluation.

Dataset	Vectors	Dim	Type	Labels
BigANN-100M	100M	128	uint8	synth., 10 classes
DEEP-100M	100M	96	float	synth., 10 classes
YFCC-10M	10M	192	uint8	real, ~200K tags
BigANN-1B	1B	128	uint8	synth., 10 classes

Filter store. The filter store is allocated at index load time from a separate metadata file and is indexed by node ID. For single-label predicates, each entry stores a fixed-width label. For multi-label predicates, each entry stores the metadata needed for in-memory predicate evaluation, loaded from a sparse matrix representation. The structure is read-only during search and shared across all query threads.

Neighbor store. The neighbor store stores up to $R_{\max}$ neighbors per node in a contiguous array with fixed stride. It is built at load time by a sequential scan over the on-disk graph. We map it with `mmap` and `MAP_POPULATE` so that its pages are materialized before query processing begins. Without `MAP_POPULATE`, the store would page in on demand through the page cache; we materialize it up front so that performance is predictable from the first query. Like the filter store, the neighbor store is read-only and shared across query threads.

Search loop integration. NemoANN inserts the filter check before PipeANN’s SSD submission path. Candidates that satisfy the predicate follow the original path unchanged. Candidates that fail the predicate are expanded through the neighbor store using PipeANN’s existing PQ distance computation and candidate insertion routines. The I/O completion path, `io_uring` submission and completion logic, and PQ infrastructure are otherwise unchanged.

5 Evaluation

5.1 Methodology

Testbed. All experiments run on a single server with two Intel Xeon Silver 4514Y CPUs (Sapphire Rapids, 32 physical cores supporting 64 logical cores), 256 GB DDR5, and a Samsung PM9A3 2 TB NVMe SSD (Gen4). For §5.3.3, we additionally use a Samsung 9100 PRO 4 TB NVMe SSD (Gen5). The server runs Ubuntu 22.04 with Linux kernel 5.15.

Datasets. Table 3 summarizes the used datasets. BigANN-100M and DEEP-100M use synthetic, uniform 10-class labels, modeling the common category-based filtering scenario. YFCC-10M [22] uses real multi-label metadata (~200K tag vocabulary) with variable per-query selectivity. BigANN-1B validates scalability to billion-scale workloads.

Compared systems. We compare six systems:

- **DiskANN** [23]: synchronous beam search ($W=8$).
- **PipeANN** [5]: asynchronous pipelined search ($W=32$).
- **NemoANN**: asynchronous pipelined search with pre-I/O filter checking and graph tunneling ($W=32$, $R_{\max}=32$).
- **Vamana** [23]: in-memory graph search (no SSD).
- **F-DiskANN** [4]: filter-aware graph index with per-label medoids and hard candidate filtering.
- **PipeANN-Filter** [7]: speculative filtering on SSD over a densified graph. ($W=32$).

DiskANN, PipeANN, and NemoANN search the *same* standard Vamana graph index built with graph degree $R=96$ and build-time search list $L_{\text{build}}=128$, with PQ compression (32 chunks). F-DiskANN uses a *different* index built by FilteredVamana, a label-aware construction that embeds filter information into the graph structure. PipeANN-Filter uses its own densified index built with the code and default settings ($R=96$, 2-hop degree $R_d=1100$, $L_{\text{build}}=128$, 32-byte PQ), so each record occupies 8 KB sectors.

Note that W has different semantics across systems. In DiskANN it is the synchronous beam width (all W reads must complete before the next round), while in PipeANN and NemoANN it is the asynchronous pipeline depth (up to W concurrent in-flight I/Os). We follow each system’s recommended settings, as in the PipeANN paper [5]. §5.3.9 shows that NemoANN’s throughput plateaus at $W \geq 8$, so the specific choice of $W=32$ does not inflate our results.

Metrics. We report Recall@10, throughput (QPS), mean latency, and mean SSD I/Os per query. The search list size L is swept to trace Pareto curves. Latency experiments use 1 thread to isolate per-query search efficiency without inter-thread contention; throughput experiments use 32 threads to stress the I/O pipeline. Default selectivity is 10% unless stated otherwise.

5.2 Main Results

5.2.1 Overall Performance. We show recall-performance curves on BigANN-100M and DEEP-100M in Figure 5. Each point corresponds to a different search list size L .

Latency. On BigANN-100M, NemoANN achieves ~0.73 ms mean latency at 90% recall, which is 2.0× faster than PipeANN, 7.6× faster than PipeANN-Filter, and 7.8× faster than DiskANN. The improvement is most pronounced at moderate recalls (60–85%), where the majority of visited candidates fail the filter and are resolved entirely via in-memory tunneling. At high recall larger than 95%, NemoANN remains faster than all baselines, though the gap narrows as the absolute number of filter-passing I/Os grows. On DEEP-100M, the trend holds with slightly larger absolute latencies due to DEEP’s float32 vectors (384 bytes per vector vs. BigANN’s 128 bytes), which increase the CPU cost of each exact distance computation without affecting NemoANN’s I/O elimination.

Throughput. NemoANN achieves ~16.9K QPS at 90% recall on BigANN-100M, compared to ~2.1K for PipeANN, an 8.0× improvement, and ~3.4K for PipeANN-Filter, a 5.0× improvement. Throughput gains exceed latency gains because 32 threads share a fixed CPU-side I/O processing budget. Each SSD read triggers not only a disk access but also sector parsing, exact distance computation, and candidate management (§5.3.4). Reduced I/O in NemoANN frees this budget, enabling throughput to scale with the number of threads rather than per-I/O overhead. The advantage persists across the recall. For example, at 95% recall, NemoANN sustains a 7.4× throughput gap over PipeANN and 5.6× over PipeANN-Filter. PipeANN-Filter outperforms PipeANN at 32 threads (its 2-hop neighbors reach recall with fewer reads), but at 1 thread it is slower than PipeANN because each of its reads fetches a two-sector record and expands many 2-hop candidates.

5.2.2 DRAM-Performance Tradeoff. To investigate the trade-off between performance and memory, we sweep $R_{\max}$ from 4 to 64 on BigANN-100M. Figure 6 shows throughput at 90% recall as a

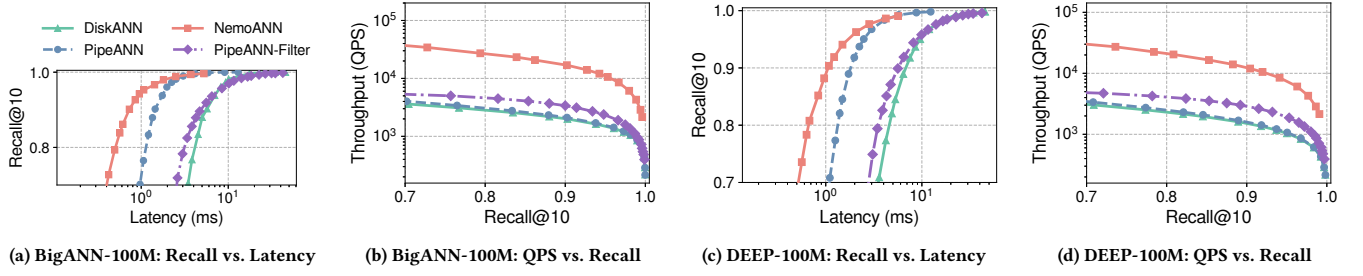


Figure 5: Recall-latency (1 thread) and throughput-recall (32 threads) tradeoff curves on 100M-scale datasets. NemoANN consistently outperforms both PipeANN and DiskANN across the entire recall range.

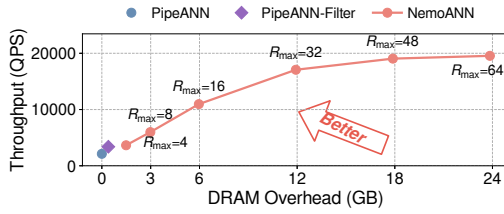


Figure 6: Throughput at 90% recall vs. DRAM overhead.

function of DRAM overhead. Even $R_{\max}=8$ at ~ 3.4 GB delivers $2.8\times$ higher throughput than PipeANN. PipeANN-Filter adds only its in-memory label Bloom filter (0.4 GB) on top of PipeANN and reaches 3.4K QPS; NemoANN matches it at $R_{\max}=4$ (1.9 GB, 3.7K QPS) and exceeds it by $1.8\text{--}5.8\times$ from $R_{\max}=8$ upward. Throughput saturates beyond $R_{\max}=48$ with 19K QPS ($9.0\times$ over PipeANN; $19.5K$ at 64), while at 1 thread $R_{\max}=48$ and 64 are slower than 32 as per-node work grows (§3.4). Every configuration in this range outperforms PipeANN at every recall level it reaches, so the memory used by the neighbor store is not a fixed overhead but a tunable knob that lets operators trade memory for performance. Even its smallest setting adds only a modest footprint over the PQ codes the baseline already keeps. This sweep also bounds the capacity cost of tunneling. Memory spent on the neighbor store could otherwise hold a larger dataset, and the knob controls how much; at $R_{\max}=8$ the store adds ~ 3.4 GB, about as much as the PQ codes the baseline itself keeps, yet throughput remains $2.8\times$ higher than PipeANN. The alternatives are less attractive, since post-filtering adds no memory but issues an SSD read for every non-matching node, and fully in-memory search uses ~ 57 GB against NemoANN's ~ 17 GB (§5.2.5).

5.2.3 I/O Reduction. We quantify the SSD I/Os NemoANN eliminates and whether the reduction matches expectations. At selectivity s , a fraction s of visited nodes pass the filter and require SSD I/O; the remaining $1-s$ are intercepted by pre-I/O filter checking. Hence, the expected I/O reduction is $1/s$ compared to a post-filter baseline.

Figure 7 (a) shows the mean I/Os per query as a function of recall on BigANN-100M. At the same L , NemoANN issues $\sim 10\times$ fewer I/Os than PipeANN (e.g., ~ 206 vs. ~ 21 at $L=200$); at 90% recall, PipeANN issues ~ 206 I/Os per query and PipeANN-Filter ~ 128 ,

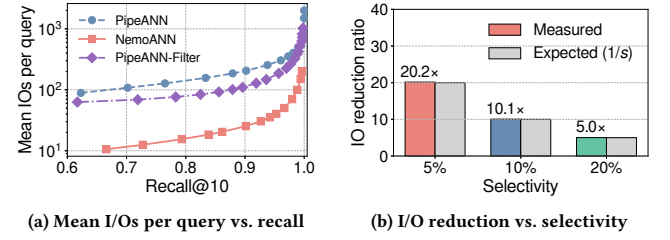


Figure 7: I/O reduction on BigANN-100M. (a) NemoANN issues far fewer I/Os than PipeANN and PipeANN-Filter at every recall (1 thread; PipeANN-Filter reads 8 KB records, the others 4 KB). (b) The measured reduction closely follows the theoretical expectation of $1/s$.

while NemoANN issues only ~ 25 . The curves stay nearly parallel across the recall range. As L grows, all systems visit proportionally more nodes, but NemoANN intercepts the same fraction ($1-s$) at each step, confirming that the I/O reduction is a structural property of the filter mechanism, not an artifact of a particular L . Figure 7 (b) compares the measured reduction ratio against the expected $1/s$ at three selectivities: $20.2\times$ at 5% (expected: $20\times$), $10.1\times$ at 10% (expected: $10\times$), and $5.0\times$ at 20% (expected: $5\times$). The close match validates that pre-I/O filter checking with graph tunneling achieves the optimal I/O count without sacrificing graph connectivity.

5.2.4 Performance at Billion-Scale. We evaluate on BigANN-1B. The 1B index is built with $R=128$, $L_{\text{build}}=200$ to accommodate the larger graph. At this scale, the default $R_{\max}=32$ requires ~ 123 GB for the neighbor store; together with PQ codes (~ 32 GB), the total fits within the server's 256 GB memory capacity.

Figure 8 shows the recall-latency and throughput-recall tradeoff curves. At 90% recall with 32 threads, NemoANN achieves 9.7K QPS, outperforming PipeANN at 1.7K QPS by $5.7\times$ and DiskANN at 1.4K QPS by $6.9\times$. This confirms that NemoANN's pre-I/O filter checking and graph tunneling scale to billion-scale datasets without degradation. PipeANN-Filter is not evaluated at this scale: its densified records occupy two 4 KB sectors per vector, so a BigANN-1B index would need ~ 8.2 TB, exceeding the capacity of every SSD in our testbed.

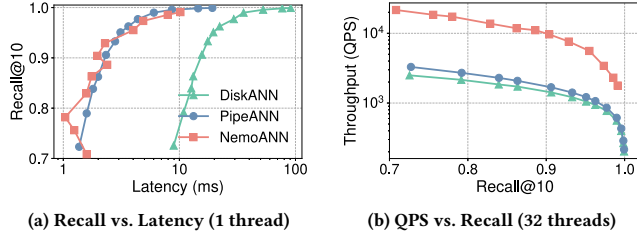


Figure 8: Billion-scale results on BigANN-1B (recall-latency and throughput-recall). NemoANN’s advantage holds at 10 $\times$ the scale of the 100M experiments.

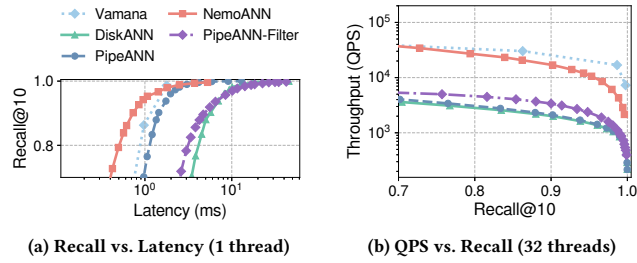


Figure 9: Comparison to Vamana (post-filtering).

5.2.5 In-Memory Search: Vamana. In this experiment, we compare against Vamana [23], which loads the full graph and all full-precision vectors into memory (~ 57 GB) and applies post-filtering. Vamana computes exact distances for every visited node. Figure 9 (a) shows that NemoANN achieves lower single-thread latency than Vamana at the same recall despite issuing SSD I/Os, because the CPU savings on non-matching nodes outweigh the I/O cost for matching ones. In Figure 9 (b) with 32 threads, Vamana’s throughput advantage becomes more apparent as in-memory computation scales freely with more threads while each SSD I/O carries fixed per-operation overhead, yet NemoANN still reaches within 1.6 $\times$ of Vamana at 90% recall while using only ~ 17 GB versus ~ 57 GB, because it keeps full-precision vectors on SSD and needs only compact routing metadata in memory.

5.2.6 Comparison to Filter-Aware Index. F-DiskANN [4] builds a label-aware graph (FilteredVamana) that routes edges preferentially through same-label nodes. We use the official DiskANN codebase on BigANN-100M with 10-class labels (we only fix a label-file parsing performance bug that otherwise stalls index loading for single-label files). Figure 10 shows that F-DiskANN achieves high recall and reduces I/Os by $\sim 25\%$ over DiskANN at 90% recall, confirming a routing benefit. However, the throughput improvement over DiskANN is modest (2.6K vs. 2.0K QPS at 32 threads), because although every read serves a matching node, discarding non-matching waypoints lengthens paths between matching nodes, so many more of them must be visited and read.

At 90% recall, NemoANN achieves 16.9K QPS, outperforming F-DiskANN’s 2.6K QPS by 6.6 $\times$ and PipeANN-Filter’s 3.4K QPS by

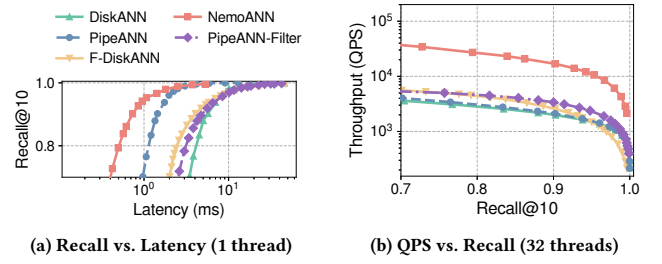


Figure 10: Comparison with F-DiskANN and PipeANN-Filter. F-DiskANN uses a FilteredVamana index with label-aware edge routing and per-label medoid entry points.

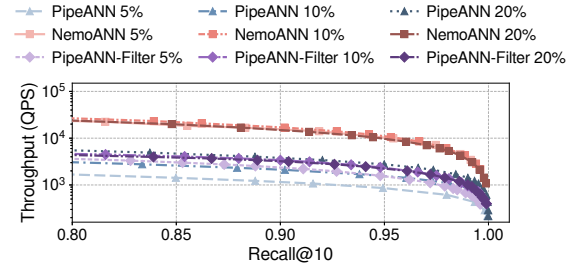


Figure 11: QPS vs. Recall at 5%/10%/20% selectivity.

5.0 $\times$ in throughput. The same trend holds on DEEP-100M, where NemoANN reaches 12.0K QPS versus F-DiskANN’s 789 QPS and PipeANN-Filter’s 2.6K QPS. Both approaches avoid I/O for non-matching nodes, but F-DiskANN pays for it by rebuilding the *graph structure* and fixing the label space at build time, whereas NemoANN pays with in-memory adjacency in the *search engine* and keeps the original graph and arbitrary predicates.

5.3 Deep Analysis

5.3.1 Selectivity Sensitivity. A key property of NemoANN is that its benefit scales with filter restrictiveness; lower selectivity yields greater I/O reduction. We evaluate this by varying selectivity across 5%, 10%, and 20% on BigANN-100M with 32 threads.

Figure 11 shows the throughput–recall tradeoff at each selectivity. PipeANN’s throughput degrades as selectivity decreases, because more nodes must be visited to find enough filter-passing results. NemoANN’s throughput, in contrast, is largely insensitive to selectivity. Additional non-matching nodes are resolved via in-memory tunneling, keeping the per-query I/O count low. At 90% recall, NemoANN outperforms PipeANN by 13.1 $\times$ at 5%, 8.0 $\times$ at 10%, and 4.0 $\times$ at 20%, and PipeANN-Filter by 6.4 $\times$, 5.0 $\times$, and 4.3 $\times$, respectively. PipeANN-Filter degrades less than PipeANN as selectivity drops, but still reads 4–8 $\times$ more records than NemoANN at 90% recall. The I/O reduction tracks 1/s (§5.2.3), while the throughput gain grows more slowly because tunneling adds CPU work for every non-matching candidate. Two further experiments probe below the 5% point. The Zipf workload (§5.3.6) includes rare-class queries at 3.4% selectivity, and most YFCC queries (§5.3.5) require rare tags,

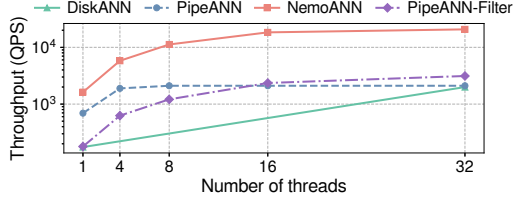


Figure 12: Throughput scaling from 1 to 32 threads on BigANN-100M, $L=200$ (PipeANN-Filter at the L that matches PipeANN’s recall).

which pulls the median per-query selectivity to 0.15%. In both cases NemoANN retains its advantage over PipeANN. On YFCC it also reaches a higher maximum recall (91.5%) than PipeANN (83%), although PipeANN expands every visited node with its full on-disk adjacency; the truncated in-memory lists therefore do not cap recall below what the full-adjacency baseline reaches.

5.3.2 Scalability. Figure 12 reports throughput for 1–32 threads at $L=200$. PipeANN plateaus at ~ 2.1 K QPS from 8 threads, and DiskANN converges to the same level (~ 2.0 K) at 32 threads despite PipeANN’s $4.0\times$ single-thread advantage, because both saturate the same aggregate I/O processing budget of ~ 430 K IOPS. PipeANN-Filter (plotted at the L that matches PipeANN’s recall) scales from 0.18K to 3.1K QPS and gains little beyond 16 threads. NemoANN does not raise this ceiling but needs far less of it: it issues ~ 21 instead of ~ 206 I/Os per query, and within the saturated budget throughput is inversely proportional to I/Os per query, so its gap over PipeANN widens from $2.3\times$ at 1 thread to $9.9\times$ at 32 threads (20.7K QPS), matching the $10\times$ I/O reduction. NemoANN itself scales sublinearly ($12.8\times$) for the same reason, as its aggregate rate at 32 threads approaches the same ~ 430 K IOPS budget. Because each thread issues its next query as soon as the previous one completes, these curves also determine mean latency; once PipeANN plateaus, each additional thread only lengthens its latency.

5.3.3 Impact of SSD Speed. A natural question is whether faster SSDs diminish the benefit of NemoANN. We compare the Gen4 PM9A3 and Gen5 Samsung 9100 PRO, which differ by approximately $2\times$ in random 4 KB read throughput.

Table 4 reveals each SSD sensitivity. At 1 thread, DiskANN benefits the most ($1.53\times$) because every query serially waits for I/O; PipeANN’s asynchronous pipeline already hides most device latency, limiting its gain to $1.10\times$; and NemoANN sees no benefit ($0.99\times$) because its ~ 20 I/Os per query leave the bottleneck entirely at CPU. At 32 threads, PipeANN shows *zero* benefit from the faster SSD. This is the key result. Were the SSD the saturated resource, doubling its IOPS capacity would improve throughput proportionally, so the zero gain shows that the binding constraint is on the host side; Table 5 attributes it to the processing chain triggered by each read. For reference, the Gen4 device is rated at over 800K random 4 KB read IOPS, roughly twice the ~ 430 K IOPS ceiling, so the device itself is never the saturated resource in our configurations. DiskANN benefits modestly ($1.06\times$) because its synchronous batching has lower per-I/O CPU overhead, leaving device latency a nontrivial share of its time, while NemoANN remains

Table 4: Performance on Gen4 vs. Gen5 SSDs (BigANN-100M, Recall@10 $\geq 90\%$). The Gen5/Gen4 column shows how much each system benefits from the faster SSD.

	Gen5/Gen4 (1 thread)	Gen5/Gen4 (32 threads)
DiskANN	1.53 $\times$	1.06 $\times$
PipeANN	1.10 $\times$	1.00 $\times$
NemoANN	0.99 $\times$	1.04 $\times$

Table 5: Per-query time breakdown at $\sim 86\text{--}90\%$ recall (1 thread, BigANN-100M).

Component	PipeANN (μ s)	%	NemoANN (μ s)	%
SSD I/O (submit + poll)	75	5.2	10	1.6
Tunneling (PQ + AdjIndex)	—	—	311	50.5
Processing (exact dist.)	991	68.6	109	17.7
Other (list mgmt., loop)	379	26.2	185	30.1
Total	1445	100	615	100

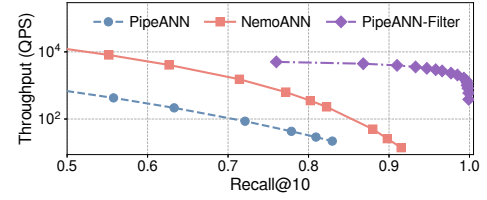


Figure 13: Throughput–recall tradeoff curves on YFCC-10M, an extreme multi-label workload (conjunctive tag predicates; median per-query selectivity 0.15%).

SSD-independent ($1.04\times$) simply because it has already eliminated the wasted reads. Faster SSDs alone thus cannot solve the filtered search problem. The effective lever is reducing the *number* of I/Os, not the *speed* of each.

5.3.4 Time Breakdown. Table 5 decomposes per-query latency at $\sim 86\text{--}90\%$ recall (1 thread). In PipeANN, exact distance computation dominates (68.6%), while SSD I/O itself is only 5.2%. Each I/O triggers an expensive processing chain that far exceeds the device access time. NemoANN replaces this path with in-memory tunneling for non-matching nodes, shrinking processing by $9.1\times$ from 991 to 109 μ s and total latency by $2.3\times$ from 1445 to 615 μ s.

At 32 threads, the gap widens further. Because each I/O carries the full processing chain above (submission, polling, sector parsing, and distance computation), 32 threads collectively exhaust their CPU budget at ~ 430 K aggregate IOPS, a CPU-side ceiling rather than an SSD limit (§5.3.3). Beyond this ceiling, throughput becomes inversely proportional to I/Os per query, which explains the $9.9\times$ throughput gap from the $10\times$ I/O reduction reported in §5.3.2.

5.3.5 An Extreme Case: Multi-Label YFCC-10M. We further evaluate NemoANN on an extreme workload, YFCC-10M from the BigANN benchmark [22], where results must contain all query tags from a ~ 200 K-term vocabulary. Such conjunctions make most queries extremely selective, and the median query matches only

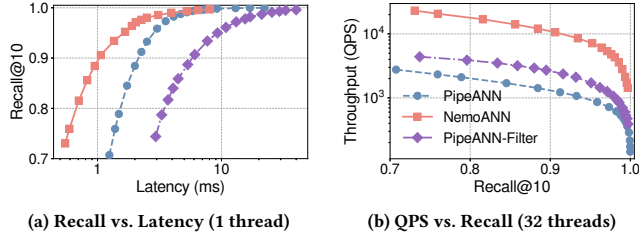


Figure 14: BigANN-100M with Zipf-distributed labels ($\alpha=1.0$, 10 classes). Selectivity ranges from 3.4% (rare) to 34.1% (common) across queries.

0.15% of vectors (mean 1.9%). Against post-filtering, NemoANN’s advantage grows further. It achieves $16.9\times$ higher throughput than PipeANN at $\sim 35\%$ recall with $18.7\times$ fewer I/Os, and reaches 91.5% recall, whereas PipeANN stalls at 83% even at $L=20K$.

PipeANN-Filter, however, is far faster here, reaching 91% recall at 4.0K QPS. Its cost model routes 42–75% of the queries to pre-filtering, which ranks the matching set from its label index with in-memory PQ codes and bypasses graph traversal entirely. When a query matches only hundreds to thousands of vectors, such a scan is cheaper than any traversal. This advantage reflects an extreme setting. Pre-filtering cost grows with the matching-set size $s \cdot N$, so the small 10M scale keeps the median matching set at $\sim 15K$ vectors, whereas its authors report pre-filtering falling below 100 QPS once a query matches over a million vectors [7]. NemoANN instead focuses solely on graph traversal and targets the 1–10% selectivities common in practice (§2.2), where matching sets reach millions of vectors already at the 100M scale. A pre-filtering path for extremely selective queries is therefore complementary to NemoANN.

5.3.6 Skewed Label Distributions. Real-world label distributions are typically skewed; a few labels are common while many are rare. We test robustness by replacing uniform labels with a Zipf distribution ($\alpha=1.0$, 10 classes), where the most common class covers 34% of vectors and the rarest covers only 3.4%. Queries are distributed uniformly across classes, creating a mix of high- and low-selectivity queries in each run.

Figure 14 shows that NemoANN maintains its advantage under skewed distributions. The mixed selectivities are favorable for NemoANN. Rare-class queries (3.4% selectivity) see even larger I/O reductions, while common-class queries (34%) still benefit because two-thirds of candidates fail the filter. The overall throughput improvement of $8.7\times$ over PipeANN and $4.5\times$ over PipeANN-Filter at 90% recall reflects the selectivity-weighted average.

5.3.7 Spatial Label Correlation. The preceding experiments assign labels uniformly at random, but real-world metadata often correlates with the vector space. To measure the impact of such correlation, we assign labels using k -means clustering ($k=10$ on BigANN-100M) with a mixing parameter α . At $\alpha=0$ labels are random; at $\alpha=1$ each node receives the label of its nearest cluster center. Selectivity remains 10% (10 classes) at all α .

Figure 15 shows the results. Recall in this experiment and in §5.3.8 is measured against the exact filtered ground truth (§5.1), so

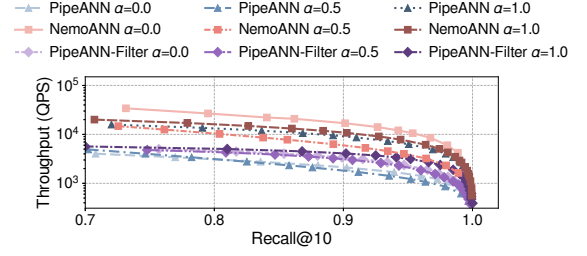


Figure 15: Effect of label-vector correlation. Labels are assigned via k -means; $\alpha=0$ is random, and $\alpha=1$ assigns each node the label of its nearest cluster center.

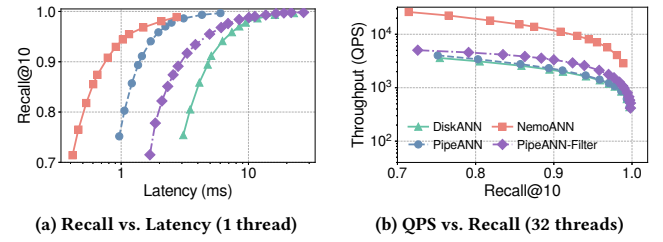


Figure 16: Range predicate (L2-norm binning, 10 bins).

no reference-induced recall ceiling arises. All systems reach at least 98.7% recall at every α , even though NemoANN routes through non-matching nodes using only its truncated in-memory lists. At 90% recall, NemoANN outperforms PipeANN at every α ($8.0\times$ at $\alpha=0$, $3.1\times$ at 0.5 , $1.2\times$ at 1), but the gap shrinks as correlation increases. Against PipeANN-Filter, the gap shrinks from $5.4\times$ at $\alpha=0$ to $1.8\times$ at $\alpha=0.5$ and near parity at $\alpha=0.75$ (not plotted), before widening again to $2.6\times$ at $\alpha=1$. At $\alpha=0$, NemoANN eliminates $\sim 90\%$ of I/Os; at $\alpha=1$, traversal naturally stays within the matching cluster, leaving fewer wasted I/Os to eliminate.

5.3.8 Range Predicates. To validate that NemoANN handles non-categorical predicates, we construct a range predicate by binning each vector’s L2 norm into 10 equal-frequency bins and selecting one bin per query ($\sim 10\%$ selectivity). This predicate is geometric and creates concentric-shell-like regions in the vector space.

Figure 16 shows latency and throughput curves. At 90% recall with 32 threads, NemoANN achieves 11.2K QPS, $5.3\times$ higher than PipeANN’s 2.1K QPS and $3.4\times$ higher than PipeANN-Filter’s 3.3K QPS; with 1 thread, NemoANN reaches 90% recall at ~ 0.78 ms vs. ~ 1.5 ms for PipeANN, ~ 3.3 ms for PipeANN-Filter, and ~ 5.1 ms for DiskANN. Measured against the exact ground truth, all systems approach full recall (NemoANN reaches 98.9%). The result confirms that NemoANN is predicate-agnostic. It applies equally to range predicates without any change to the index or search algorithm.

5.3.9 Pipeline Depth. In PipeANN, pipeline depth W controls the number of in-flight I/Os via `io_uring`. Since our solution issues $\sim 10\times$ fewer I/Os per query, one might expect a smaller W to suffice. We sweep $W \in \{1, 2, 4, 8, 16, 32\}$ on BigANN-100M.

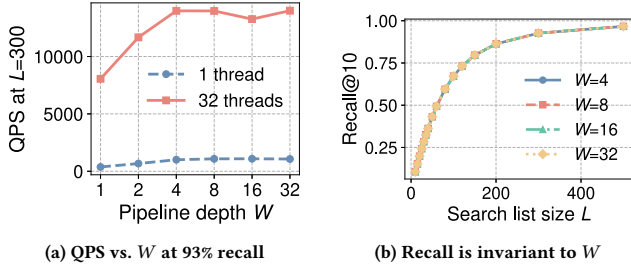


Figure 17: Pipeline depth (W) sweep for NemoANN on BigANN-100M. Recall is identical across all W ; throughput plateaus at $W \geq 8$ (32 threads) and at $W \geq 4$ (1 thread).

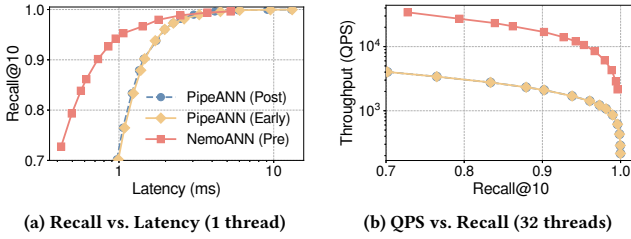


Figure 18: Ablation on BigANN-100M ($s=0.1$). PipeANN (Early) filters after each SSD read, skipping exact distance computation for non-matching nodes; NemoANN (Pre) additionally eliminates the SSD reads themselves.

Figure 17 (b) confirms that recall is invariant to W . All values produce identical curves across L , since W affects only I/O scheduling, not candidate selection. Figure 17 (a) shows throughput at $L=300$ (~93% recall). At 32 threads, throughput improves from $W=1$ (8.8K QPS) to $W=4$ (14.1K QPS), then plateaus through $W=32$ (14.0K QPS). The remaining I/Os per query still benefit from concurrency, but 4 concurrent I/Os suffice to hide per-read latency, even though NemoANN issues $\sim 10\times$ fewer of them. At 1 thread, throughput plateaus at $W \geq 8$ (~1.1K QPS), confirming that a single thread is CPU-bound even at moderate depths.

5.3.10 Ablation: I/O Elimination vs. CPU Savings. NemoANN’s performance gains come from two sources: (i) eliminating SSD I/Os for non-matching nodes, and (ii) skipping exact distance computation on them. To separate them, we add PipeANN (Early), which filters *after* the read. Non-matching nodes still incur the I/O but skip the exact distance, while neighbor expansion is unchanged to preserve connectivity.

Figure 18 compares the three designs. At 1 thread (Figure 18(a)), PipeANN (Early) and PipeANN (Post) are nearly identical. At 90% recall, latency is 1.47 ms vs. 1.45 ms, within measurement noise. Skipping exact distance saves only tens of μs per query because BigANN’s 128-D vectors make each of ~ 206 computations cheap (~ 90 ns). In contrast, NemoANN (Pre) (i.e., NemoANN (Joint)) reduces latency to 0.73 ms ($2.0\times$) and reaches ~ 16.9 K QPS ($8.0\times$) by eliminating SSD reads. Removing distance computation without

removing I/O does not address the bottleneck. As Table 5 shows, per-I/O cost is dominated by submission, polling, and sector parsing, and both PipeANN variants still issue ~ 206 I/Os per query, hitting the same ~ 430 K IOPS ceiling. Only NemoANN eliminates the SSD reads themselves, confirming that *what to read* matters far more than *what to compute*.

6 Related Work

SSD-based graph ANNS. DiskANN [23] introduced SSD-resident graph search with batched beam search. Subsequent work improves SSD efficiency through asynchronous I/O pipelines [5], locality-aware layout [28, 34], layout-runtime co-design [37], page-aligned placement [12], and direct-insert updates [6]. GPS [33] decouples adjacency from exact vectors by caching full adjacency for offline-selected nodes, but targets unfiltered, query-independent layout optimization through disk-layout changes. Neither the inverted-index SPANN [3] nor the cluster-partitioned OrchANN [2] integrates predicate-aware filtering, so metadata constraints fall back to post-filtering. NemoANN instead avoids fetching non-matching nodes and is complementary to any on-disk graph index.

Filtered ANNS. Most work is in-memory, including approaches that embed attributes into the index by fusing vector-attribute distances [27, 30], by stitching per-label-set subgraphs via label containment [1], or by densifying edges to keep predicate-induced subgraphs navigable [19]. Others maintain multiple filter-specific subindexes selected at query time [11, 15]. For disk-based systems, Filtered-DiskANN [4] builds a label-aware Vamana graph with per-label medoids. NaviX [21] uses predicate-agnostic pre-filtering on disk-based HNSW index, but evaluates the predicate over the full dataset per query and keeps adjacency on disk. PipeANN-Filter [7] is a recent work that explores a superset of valid nodes and verifies at rerank time, but storing $10\text{--}20\times$ extra 2-hop neighbors inflates its on-disk footprint, making billion-scale serving impractical. NemoANN keeps the original graph and serves non-matching nodes as in-memory routing waypoints, eliminating their SSD reads without rebuilding the index or its on-disk layout.

7 Conclusion

This paper presented NemoANN, an I/O-efficient SSD-based system for filtered vector search. A node that fails the filter contributes nothing to the results, so the search consults only its neighbor list. That list is small enough to keep in memory. NemoANN exploits this by evaluating the filter before every SSD read; a failing node is traversed through an in-memory copy of its neighbor list, so its read is avoided and graph connectivity is preserved. In our experiments, NemoANN reduces I/Os by about $10\times$ and improves throughput by $8.0\times$ on a 100M-vector benchmark at 10% selectivity and 90% recall, with larger gains at lower selectivity.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and feedback. This research was sponsored by the National Research Foundation of Korea (NRF) grants funded by the Ministry of Science and ICT (No. RS-2025-00522990). Gyuyeong Kim is the corresponding author.

References

- [1] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. 2024. Navigating Labels and Vectors: A Unified Approach to Filtered Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 6, Article 246 (2024). doi:10.1145/3698822
- [2] Lizheng Chen, Pinhuan Wang, Shaonan Ma, Jie Zhang, Qing Wang, Zhengyi Yang, Heng Zhang, Mingxing Zhang, Fan Yang, Guihai Chen, Chen Tian, and Chengying Huan. 2026. OrchANN: Hierarchical Orchestration for Skewed Out-of-Core Vector Search. *arXiv preprint arXiv:2512.22838* (2026).
- [3] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search. In *Proc. of NeurIPS*. Curran Associates, Inc., 5199–5212.
- [4] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *Proc. of the ACM Web Conference (WWW)*. ACM, 3406–3416. doi:10.1145/3543507.3583552
- [5] Hao Guo and Youyou Lu. 2025. Achieving Low-Latency Graph-Based Vector Search via Aligning Best-First Search Algorithm with SSD. In *Proc. of USENIX OSDI*. Boston, MA, USA, 171–186.
- [6] Hao Guo and Youyou Lu. 2026. OdinANN: Direct Insert for Consistently Stable Performance in Billion-Scale Graph-Based Vector Search. In *Proc. of USENIX FAST*. Santa Clara, CA, USA, 133–147.
- [7] Hao Guo, Jiwu Shu, and Youyou Lu. 2026. PipeANN-Filter: An Efficient Filtered Vector Search System on SSD. *arXiv:2605.17992 [cs.OS]* <https://arxiv.org/abs/2605.17992>
- [8] Zhengding Hu, Vibha Murthy, Zaifeng Pan, Wanlu Li, Xiaoyi Fang, Yufei Ding, and Yuke Wang. 2025. HedraRAG: Co-Optimizing Generation and Retrieval for Heterogeneous RAG Workflows. In *Proc. of ACM SOSR*. Association for Computing Machinery, 623–638. doi:10.1145/3731569.3764806
- [9] Amir Ingber and Edo Liberty. 2025. Accurate and Efficient Metadata Filtering in Pinecone's Serverless Vector Database. In *Proc. of ICML Workshop on Vector Databases*.
- [10] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *Proc. of USENIX ATC*. Boston, MA, USA, 585–600.
- [11] Yicheng Jin, Yongji Wu, Wenjun Hu, Bruce M. Maggs, Jun Yang, Xiao Zhang, and Danyang Zhuo. 2026. Curator: Efficient Vector Search with Low-Selectivity Filters. *Proc. ACM Manag. Data* 4, 1, Article 21 (2026). doi:10.1145/3786635
- [12] Dingyi Kang, Dongming Jiang, Hanshen Yang, Hang Liu, and Bingzhe Li. 2025. Scalable Disk-Based Approximate Nearest Neighbor Search with Page-Aligned Graph. *arXiv preprint arXiv:2509.25487* (2025).
- [13] Sukjin Kim, Seongyeon Park, Si Ung Noh, Junguk Hong, Taehee Kwon, Hunseong Lim, and Jinho Lee. 2025. PathWeaver: A High-Throughput Multi-GPU System for Graph-Based Approximate Nearest Neighbor Search. In *Proc. of USENIX ATC*. Boston, MA, USA, 1501–1517.
- [14] Mocheng Li, Xiao Yan, Baotong Lu, Yue Zhang, James Cheng, and Chenhao Ma. 2025. Attribute Filtering in Approximate Nearest Neighbor Search: An In-depth Experimental Study. *Proc. ACM Manag. Data* 3, 6, Article 298 (2025). doi:10.1145/3769763
- [15] Zhaoheng Li, Silu Huang, Wei Ding, Yongjoo Park, and Jianjun Chen. 2025. SIEVE: Effective Filtered Vector Search with Collection of Indexes. *Proc. VLDB Endow.* 18, 11 (2025), 4723–4736. doi:10.14778/3749646.3749725
- [16] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.
- [17] Microsoft. 2025. Azure AI Search. <https://learn.microsoft.com/en-us/azure/search/>.
- [18] Jason Mohoney, Devesh Sarda, Mengze Tang, Shihabur Rahman Chowdhury, Anil Pacaci, Ihab F. Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. 2025. Quake: Adaptive Indexing for Vector Search. In *Proc. of USENIX OSDI*. Boston, MA, USA, 153–169.
- [19] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proc. ACM Manag. Data* 2, 3, Article 120 (2024). doi:10.1145/3654923
- [20] Siddhant Ray, Rui Pan, Zhuohan Gu, Kuntai Du, Shaoting Feng, Ganesh Ananthanarayanan, Ravi Netravali, and Junchen Jiang. 2025. METIS: Fast Quality-Aware RAG Systems with Configuration Adaptation. In *Proc. of ACM SOSR*. Association for Computing Machinery, 606–622. doi:10.1145/3731569.3764855
- [21] Gaurav Sehgal and Semih Salihoglu. 2025. NaviX: A Native Vector Index Design for Graph DBMSs With Robust Predicate-Agnostic Search Performance. *Proc. VLDB Endow.* 18, 11 (2025), 4438–4450. doi:10.14778/3749646.3749704
- [22] Harsha Vardhan Simhadri, Martin Aumüller, Amir Ingber, Matthijs Douze, George Williams, Magdalen Dobson Manohar, Dmitry Baranchuk, Edo Liberty, Frank Liu, Ben Landrum, et al. 2024. Results of the Big ANN: NeurIPS'23 Competition. *arXiv preprint arXiv:2409.17424* (2024).
- [23] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Proc. of NeurIPS*. Curran Associates, Inc., 13748–13758.
- [24] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. 2024. Scalable Billion-point Approximate Nearest Neighbor Search Using SmartSSDs. In *Proc. of USENIX ATC*. Santa Clara, CA, USA, 1135–1150.
- [25] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Hai Jin, Xuecang Zhang, Junhua Zhu, and Yu Zhang. 2025. Towards High-throughput and Low-latency Billion-scale Vector Search via CPU/GPU Collaborative Filtering and Re-ranking. In *Proc. of USENIX FAST*. Santa Clara, CA, USA, 171–185.
- [26] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xianguo Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proc. of ACM SIGMOD*. Association for Computing Machinery, 2614–2627. doi:10.1145/3448016.3457550
- [27] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jionghang Ni. 2023. An Efficient and Robust Framework for Approximate Nearest Neighbor Search with Attribute Constraint. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*.
- [28] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xianguo Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proc. ACM Manag. Data* 2, 1, Article 14 (2024). doi:10.1145/3639269
- [29] Puqing Wu, Minhui Xie, Enrui Zhao, Dafang Zhang, Jing Wang, Xiao Liang, Kai Ren, and Yunpeng Chai. 2025. Turbocharge ANNS on Real Processing-in-Memory by Enabling Fine-Grained Per-PIM-Core Scheduling. In *Proc. of USENIX ATC*. Boston, MA, USA, 1223–1241.
- [30] Haikun Xu, Guy Blelloch, Laxman Dhulipala, Lars Gottesbüren, Rajesh Jayaram, and Jakub Łącki. 2026. JAG: Joint Attribute Graphs for Filtered Nearest Neighbor Search. *arXiv preprint arXiv:2602.10258* (2026).
- [31] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proc. of ACM SOSR*. Association for Computing Machinery, 545–561. doi:10.1145/3600066.3613166
- [32] Jiayi Yao, Hanchen Li, Yuhang Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proc. of ACM EuroSys*. Association for Computing Machinery, 94–109. Best Paper Award. doi:10.1145/3689031.3696098
- [33] Peiqi Yin, Xiao Yan, Qihui Zhou, Hui Li, Xiaolu Li, Meiling Wang, Lin Zhang, Xin Yao, and James Cheng. 2026. GPS: Revisiting the Data Layout for Disk-based High-Dimensional Vector Search. *Proc. ACM Manag. Data* 4, 3, Article 192 (2026). doi:10.1145/3802069
- [34] Ziyang Yue, Bolong Zheng, Ling Xu, Kanru Xu, Shuhao Zhang, Yajuan Du, Yunjun Gao, Xiaofang Zhou, and Christian S. Jensen. 2025. Select Edges Wisely: Monotonic Path Aware Graph Layout Optimization for Disk-based ANN Search. *Proc. VLDB Endow.* 18, 11 (2025), 4337–4349. doi:10.14778/3749646.3749697
- [35] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *Proc. of USENIX OSDI*. Boston, MA, USA, 377–395.
- [36] Xiang Zhang, Chao Zhang, Ju Fan, Guoliang Li, and Xiaoyong Du. 2026. VecBench: A Controllable Benchmark for Filtered Vector Search. *Proc. ACM Manag. Data* 4, 3, Article 248 (2026). doi:10.1145/3802125
- [37] Weichen Zhao, Yuncheng Lu, Yao Tian, Hao Zhang, Jiehui Li, Minghao Zhao, Yakun Li, and Weining Qian. 2026. Optimizing SSD-Resident Graph Indexing for High-Throughput Vector Search. *arXiv preprint arXiv:2602.22805* (2026).
- [38] Zilliz. 2024. VectorDBBench: A Benchmark Tool for Vector Databases. <https://github.com/zilliztech/VectorDBBench>.