# PathSeer: Adaptive Neighbor Handling for Efficient Filtered ANN Search

ZHIYUE LI, Tsinghua University, China
GUANGYAN ZHANG, Tsinghua University, China
RUOCHUN JIN, National University of Defense Technology, China
BOJUN LI, National University of Defense Technology, China
XIAOGUANG REN, Intelligent Game and Decision Lab, China
WENJING YANG, National University of Defense Technology, China

Filtered approximate nearest neighbor (ANN) search over high-dimensional vectors and associated attributes is critical for recommendation systems and RAG-based LLMs. However, existing methods face performance bottlenecks under workloads of varying characteristics due to static and single neighbor-handling strategies—either computing distances before filtering or filtering before distance computation—which fail to adapt to diverse data and workload patterns. In this paper, we propose PathSeer, an approach that enables efficient filtered ANN search through dynamic and hybrid neighbor-handling strategies. PathSeer introduces three key techniques: (1) a fusion vector indexing scheme that supports two neighbor-handling strategies within a single index, (2) a dynamic neighbor traversal strategy that allows adjusting the ratios of different neighbor-handling strategies at each step of index traversal while preserving high search performance, and (3) a heuristic parameter tuning mechanism that adapts the ratios of neighbors employing different neighbor-handling strategies to workload characteristics. Evaluations on six workloads show that PathSeer delivers 1.17×–47.4× higher throughput for filtered ANN search compared to existing methods, without sacrificing recall.

CCS Concepts: • **Information systems → Top-k retrieval in databases**; **Data management systems**.

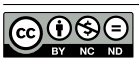
Additional Key Words and Phrases: filtered ANN search, vector database, predicate-agnostic filtering, adaptive traversal, graph-based index

**ACM Reference Format:**
Zhiyue Li, Guangyan Zhang, Ruochun Jin, Bojun Li, Xiaoguang Ren, and Wenjing Yang. 2026. PathSeer: Adaptive Neighbor Handling for Efficient Filtered ANN Search. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 221 (June 2026), 27 pages. https://doi.org/10.1145/3802098

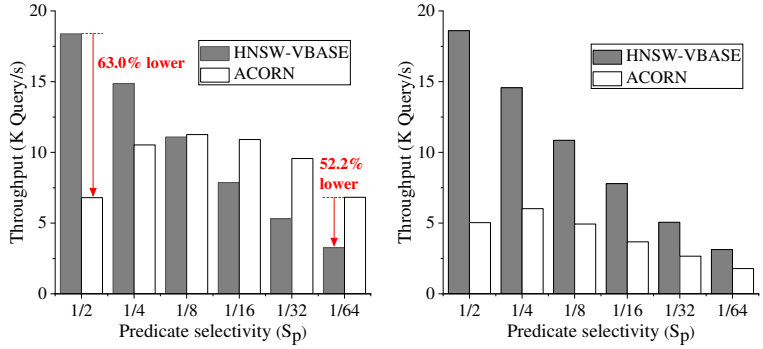
## 1 Introduction

Approximate nearest neighbor (ANN) search is widely used in various information retrieval applications such as recommendation systems [36, 55] and retrieval-augmented generation (RAG) for large language models (LLMs) [22, 39, 64]. Given high-dimensional vectors that embed multi-modal information as queries, ANN search retrieves semantically similar contents based on vector proximity [23, 32, 38]. Moreover, the search semantics can be more precisely controlled with *filtered ANN search*, where the attributes of retrieved vectors conform to the query predicate [33], which

Guangyan Zhang is the corresponding author: gyzh@tsinghua.edu.cn.
Authors' Contact Information: Zhiyue Li, lizhiyue23@mails.tsinghua.edu.cn, Tsinghua University, Beijing, China; Guangyan Zhang, gyzh@tsinghua.edu.cn, Tsinghua University, Beijing, China; Ruochun Jin, jinrc@nudt.edu.cn, National University of Defense Technology, Changsha, China; Bojun Li, libojun24@nudt.edu.cn, National University of Defense Technology, Changsha, China; Xiaoguang Ren, rxg_nudt@126.com, Intelligent Game and Decision Lab, Beijing, China; Wenjing Yang, wenjing.yang@nudt.edu.cn, National University of Defense Technology, Changsha, China.




ACM 2836-6573/2026/6-ART221
https://doi.org/10.1145/3802098




(a) Low filter overhead. ($R_p = 0.05$) (b) High filter overhead. ($R_p = 0.41$)

Fig. 1. Comparison between HNSW-VBASE and ACORN under different $R_p$ and $S_p$. Throughput is evaluated when recall@10 is 90% as suggested by the BigANN benchmark [43].

proves to enhance retrieval accuracy for both e-commerce recommendation [55, 61] and RAG-based LLMs [19, 39, 64].

Unlike the vector data that is often uniformly formatted, query predicates and vector attributes vary in form and are dynamic. First, the varying query predicates result in dramatically different filter overheads. For example, when searching for similar products by a reference image on the e-commerce platform [35, 55] with range filtering on price and categorical filtering on color and brand (implemented as numerical comparison [8, 15]), the predicates of simple numerical comparisons can each be evaluated in tens of nanoseconds. By contrast, complex predicates, such as regex matching [33] and fuzzy filtering [42] used in keyword search on image captions or documents, may take hundreds to thousands of nanoseconds for each filtering. Second, vector attributes may alter independently of the vector data. For example, Apple Inc. queries its industrial knowledge graphs for semantically similar entities with filters on entity relationships [29], where such relationships vary independently of entity vectors due to knowledge updates [18, 24, 50]. Similarly, prices on e-commerce platforms fluctuate during promotions without altering product images and their corresponding vectors.

Thus, it is challenging to perform efficient filtered ANN search due to its diverse and dynamic nature. Existing methods that can handle such workloads are **predicate-agnostic** methods [12, 28, 33, 40, 63], which make no assumption of the query predicates and vector attributes. These methods maintain vector attributes independently from the vector index and perform predicate filtering during vector search, which can support any query predicate and do not need to reconstruct the vector index when vector attributes change [33]. The most widely used vector index among these methods is the graph-based vector index [6, 7], achieving both high performance and accuracy [13, 17, 34, 65]. Searching on a graph-based index involves traversing the index graph, where each step traverses the neighbors of the nearest candidate vector to select new candidates for the following search. Specifically, there are two ways of handling each neighbor vector regarding the order of distance computation and predicate filtering. The first type performs **d**istance computations first on each neighbor and only **f**ilters on those that are fairly **n**ear the query vector (denoted as DFN neighbor-handling). The second type performs predicate filtering first and only performs **d**istance computation **o**n **s**atisfactory vectors (denoted as DOS neighbor-handling). Existing graph-based predicate-agnostic methods exploit static and single neighbor-handling strategies across

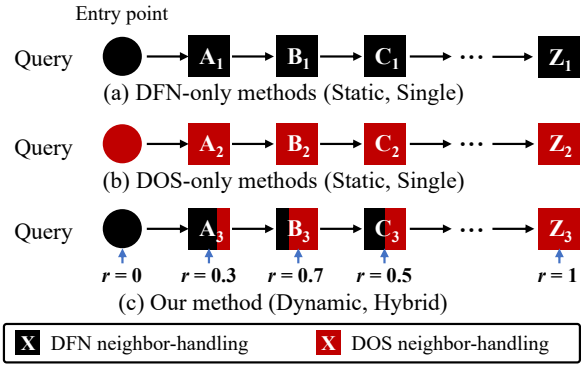



Fig. 2. Comparison of filtered ANN search methods with different ways of selecting neighbor-handling strategies.

different neighbors of each traversal, either being the DFN-only method (e.g., HNSW-VBASE[1]) or the DOS-only method (e.g., ACORN [33] and NaviX [40]).

Although researchers have concluded that DOS-only methods are more efficient with the assumption that the distance computation constitutes major overhead [33], we find it does not always hold, as the predicate filtering overhead can be non-negligible compared to the distance computation. Given $R_p$ defined as the relative overhead between the predicate filtering and distance computation, we have found that the $R_p$ of complex predicates can reach 0.42 (see Table 1). Moreover, the *predicate selectivity* $S_p$, defined as the proportion of vectors that conform to query predicates (referred to as satisfactory vectors), also significantly impacts query performance. As shown in Figure 1, we illustrate this by evaluating HNSW-VBASE and ACORN on the SIFT dataset [1] under various $S_p$ and $R_p$, where the performance of both ACORN and HNSW-VBASE varies significantly with different $R_p$ and $S_p$ and neither consistently outperforms the other. Worse yet, there exists a 52.2% performance degradation if an improper search method is chosen.

The common limitation of DFN-only and DOS-only methods is that they impose **static and single** neighbor-handling strategies, ignoring the temporal and spatial variations of the workload characteristics. We illustrate this with Figure 2, in which each square denotes traversing the neighbors of the candidate vector, and the area of each color means the proportion of vectors handled by the related neighbor-handling strategy. For one thing, the way both methods handle the neighbors is **the same** across different queries and neighbor traversals, which cannot adapt to workloads with temporal variations between different queries. Moreover, it cannot capture the spatial variation of attribute-correlated workloads [16], whose satisfactory vectors spread unevenly in the vector space. Although some DOS-only methods like NaviX [40] schedule which neighbors to traverse according to local $S_p$, they still rely on one neighbor-handling strategy and have limited space for scheduling. For another, both methods use the same neighbor-handling strategy for **all** neighbors during each neighbor traversal, either achieving few predicate filtering with more distance computation or vice versa. However, such a dichotomous selection of neighbor-handling strategies misses the chance for a better trade-off between these two operations that can optimize the aggregated overhead.

In view of these limitations, we propose the dynamic and hybrid selection of neighbor-handling strategies as shown in Figure 2(c). For each neighbor traversal, it divides the neighbors into two parts

[1]VBASE [63] proposes the relaxed monotonicity to derive the stop condition for filtered ANN search methods, like the graph-based HNSW [25]. We refer to the HNSW search with this stop condition as HNSW-VBASE.

and applies the DFN and DOS neighbor-handling, respectively. Besides, the proportion of neighbors handled by the DOS neighbor-handling is tunable with a filtering ratio $r$, which is carefully chosen according to workload characteristics. The dynamic and hybrid selection of neighbor-handling strategies can adapt to the temporal and spatial variations of the workloads, achieving consistent high search performance and accuracy. Furthermore, we devise three techniques to implement this idea. First, we design a fusion vector indexing scheme to support both neighbor-handling strategies within one index simultaneously. Second, we propose a dynamic neighbor traversal strategy that enables tuning of the filtering ratio $r$ while ensuring high search performance. Third, we devise a heuristic parameter tuning method that tunes the filtering ratio $r$ of each neighbor traversal based on the temporal and spatial characteristics of the workload. By integrating these techniques, we propose an efficient filtered ANN search approach called PathSeer, implement it in Faiss [27], and compare it with existing approaches, including the graph-based HNSW-VBASE, ACORN, and NaviX, as well as the cluster-based IVFFlat [12]. Evaluations on six vector datasets demonstrate that PathSeer can achieve 1.17×–47.4× higher throughput of filtered ANN search over existing methods without compromising the search recall.

Our contributions are summarized as follows.

- We reveal that real-world filtered ANN search workloads can have non-negligible filtering overhead and varying predicate selectivity, which existing predicate-agnostic methods fail to handle efficiently due to their static and single neighbor-handling strategies.
- We design a predicate-agnostic filtered ANN search method that exploits the dynamic and hybrid neighbor-handling strategies, which can automatically adapt to workloads of temporal and spatial variations.
- We systematically compare our method to existing filtered ANN search methods on six real-world workloads that cover different query predicates, demonstrating that our method can achieve a better trade-off between the search speed and accuracy, while enabling flexible attribute variations.

## 2 Background and Motivation

We first introduce the filtered ANN search and existing methods. Then, we exploit real-world datasets to analyze the characteristics of the filtered ANN search and the problems of existing methods, both in Faiss [27] and a popular commercial vector database, Weaviate [53].

### 2.1 Filtered ANN Search and Related Methods

Let $D = \{(v_1, m_1), (v_2, m_2), ..., (v_n, m_n)\}$ be a vector dataset that contains $n$ pairs of a $d$-dimensional vector $v_i$ and its related attribute $m_i$ ($i \in [1, n]$). A filtered $k$ nearest neighbor query $q = (v_q, p_q)$ extracts $k$ nearest vectors to $v_q$ from $D$, whose attributes conform to the query predicate $p_q$. Typical distance metrics between two vectors include cosine distance [56] and Euclidean distance [57]. Considering the diverse and dynamic natures of filtered ANN search, we make no assumptions on the query predicate $p_q$ and vector attributes $m_i$. Moreover, $m_i$ may change without altering $v_i$. As extracting the exact $k$ nearest neighbors scales poorly in large datasets [14, 21, 59], we particularly focus on the filtered approximate nearest neighbor (ANN) search for fast queries with acceptable accuracy losses. The query accuracy is measured with $recall@k = \frac{|G \cap R|}{k}$, in which $G$ is the ground truth set of $k$ nearest vectors to $v_q$ that conform to $p_q$, and $R$ is the set of retrieved vectors [33, 45].

Although researchers have proposed vector indexes that couple the vector attributes [8, 15, 49, 58], such as designing new distance metrics that incorporate the similarity of attributes between vectors [49, 58] or grouping the vectors based on their attributes [8, 15], these indexes are limited to label matching on categorical attributes [33], which restricts their applicability to general scenarios.

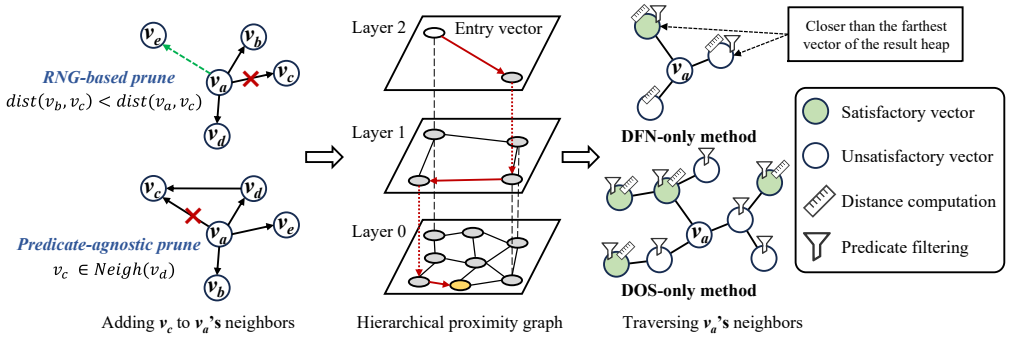



Fig. 3. Index construction and vector search processes of the DFN-only and DOS-only methods.

Moreover, these methods have to reconstruct their attribute-coupled vector indexes when the attributes change independently of the vectors.

On the contrary, the predicate-agnostic search methods [12, 28, 33, 40, 63] maintain attributes separately from the vector indexes, naturally fitting the diverse and dynamic filtered ANN search workloads. We primarily focus on approaches with the graph-based vector index [28, 33, 63], which is preferable for high performance under high accuracy [13, 17, 34, 65]. Comparison with the cluster-based index can be found in our experiments. As shown in the middle part of Figure 3, the vector indexes of these methods are hierarchical proximity graphs on dataset vectors. Each new vector will build directed edges towards its neighbors, the nearby vectors within the same layer of graph index. Besides, to enable fast location of dataset vectors near $v_q$, these methods build proximity subgraphs by sampling vectors from the lower layers [25]. Searching on a graph index involves recursively popping the nearest candidate vector from a candidate heap, traversing its neighbors for unvisited vectors, and supplementing them into the candidate heap for the following search. Meanwhile, a result heap is maintained to record $k$ nearest satisfactory vectors for the final result. As demonstrated in the right part of Figure 3, regarding the way of selecting neighbor-handling strategies, existing works can be categorized into two types:

***DFN-only method.*** The DFN-only method performs distance computations on all neighbors of the current candidate vector $v_a$ and supplements them into the candidate heap. Besides, only the vectors that are close enough to be added to the result heap participate in predicate filtering. The DFN-only method trades less predicate filtering with more distance computations, suitable for higher $R_p$. During index construction, the DFN-based method exploits the Relative Neighborhood Graph (RNG) based pruning [20] to enhance the spatial sparsity of the approximate graph for better search performance [25, 33]. As shown on the left of Figure 3, $v_c$ is pruned since $dist(v_b, v_c) < dist(v_a, v_c)$, making room for a potential new neighbor $v_e$. Such pruning expands $v_a$'s neighbors towards a more spatially uniform and sparse distribution. This helps to find satisfactory vectors near the query vector with fewer distance computations during the search, especially under a high $S_p$.

***DOS-only method.*** The DOS-only method performs predicate filtering on all neighbors, and only the satisfactory ones participate in distance computation and are added to the candidate heap. This avoids distance computations on unsatisfactory vectors but increases the number of predicate filtering, making it suitable for workloads with low $R_p$ and low $S_p$. As the predicate filtering reduces the number of neighbors eligible for the candidate heap and hence impedes the connectivity of the search subgraph [19], existing DOS-only methods [33, 37, 40] exploit a "two-hop" search method that traverses both $v_a$'s neighbors and the neighbors of $v_a$'s neighbors, as illustrated in the right part of Figure 3. Moreover, since the satisfactory vectors are unknown during index construction,

Table 1. Measurements of metrics that reflect the characteristics of six filtered ANN search workloads.

| Workloads | SIFT | H&M | LAION | Paper | arXiv | Amazon |
|---|---|---|---|---|---|---|
| Vector dimension | 128 | 2048 | 512 | 200 | 768 | 768 |
| Distance computation overhead | 418ns | 1901ns | 662ns | 508ns | 890ns | 1006ns |
| Predicate filtering overhead | 38ns | 136ns | 280ns | 41ns | 146ns | 52ns |
| Average $R_p$ | 0.05 | 0.07 | 0.42 | 0.08 | 0.16 | 0.05 |
| Average $S_p$ | 0.083 | 0.050 | 0.013 | 0.026 | 0.005 | 0.402 |
| Max($S_p$) / Min ($S_p$) | 10 | 4049 | 17978 | 10 | 12195 | 3257 |

most DOS-only methods preserve complete neighbor information either by not pruning [37] or compressing neighbor information with predicate-agnostic pruning [33], as shown in the left part of Figure 3. However, the DOS-only method introduces more distance computation than the DFN-only method under a high $S_p$ since the vectors traversed by the "two-hop" search can be far from the query, as shown in Section 4.2.

## 2.2 Characteristics of Filtered ANN Search

In this section, we reveal the temporal and spatial variations of the filtered ANN search by analyzing six workloads (detailed in § 5.1) in Faiss [27].

***Temporal Variations.*** The temporal variations denote that the workload characteristics vary over time due to the change of the proportion of different query predicates [29]. We demonstrate this by analyzing the $S_p$ and $R_p$ of our evaluated workloads in Table 1. First, $R_p$ can vary substantially across workloads. This is because different workloads have different vector dimensions and query predicates, which result in varying distance computation and predicate filtering overheads. For example, the SIFT, Paper, and Amazon workloads filter by integer matching or comparison, which exemplify low-filtering-overhead scenarios such as the price and categorical filters in e-commerce recommendation [55]. By contrast, the H&M, LAION, and arXiv workloads filter with composite predicates or string matching, which are typical in high-filtering-overhead scenarios such as relationship filtering in knowledge graphs [29], image outlier detection by descriptions [33], or retrieval refinement with fuzzy filter [42]. Similarly, the average $S_p$ also varies across different workloads due to the difference in query predicates and vector attributes.

***Spatial Variations.*** The spatial variations denote the skewed distribution of satisfactory vectors in the vector space, which is observed in real-world cases [16]. We show this by counting the local $S_p$ around each query and calculating the ratio between the maximum local $S_p$ and the minimum local $S_p$ of each workload. A lower value of this metric means that the distribution of satisfactory vectors is less skewed. As shown in Table 1, the SIFT and Paper workloads have the least difference between the $Max(S_p)$ and $Min(S_p)$ since the attribute for each vector is randomly generated without considering the spatial location. However, other workloads with real-world attributes demonstrate huge differences between $Max(S_p)$ and $Min(S_p)$, reflecting strong spatial correlation between the vectors and their attributes.

## 2.3 Drawbacks of Existing Methods

We evaluate HNSW-VBASE and ACORN on the SIFT dataset with varying $S_p$ and count the number of distance computations and predicate filtering. Two drawbacks of existing methods can be derived from the results, and we have found similar trends in other datasets.

***Poor workload adaptivity.*** The total overhead of distance computations (dist) and predicate filtering (filter) can be modeled as $n_{dist} \cdot O_{dist} + n_{filter} \cdot O_{filter}$, in which $n$ is the operation number

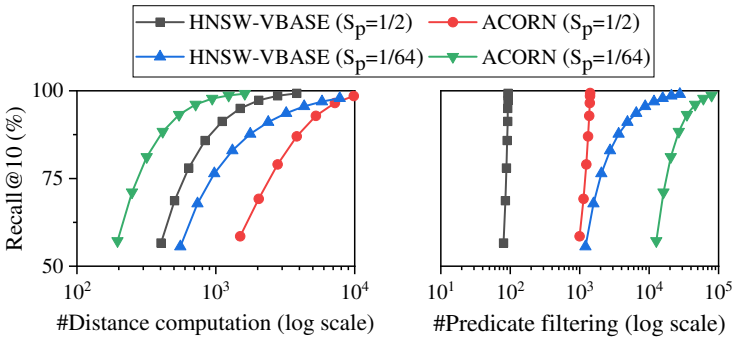



Fig. 4. Number of distance computations and predicate filtering of HNSW-VBASE and ACORN among different recall rates.

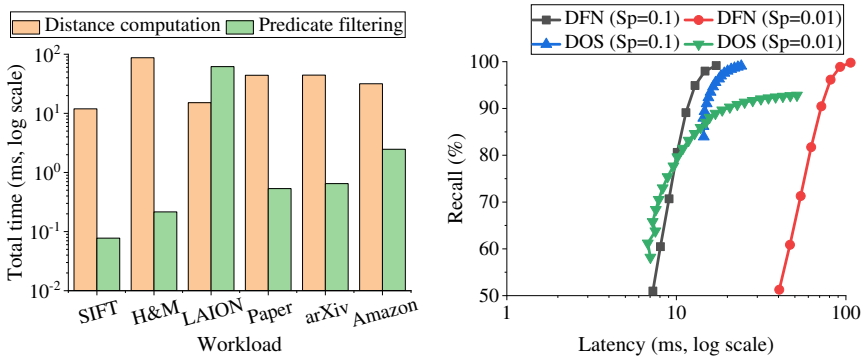
and $O$ is the average operation overhead. Although it is challenging to reduce the number of both operations while keeping high recalls for a filtered ANN search, reducing the total overhead is possible by keeping a good trade-off between these two operations according to their overheads. However, both HNSW-VBASE and ACORN push this trade-off towards the extreme, either achieving fewer distance computations with more predicate filtering or vice versa, as shown in Figure 4. For example, when $S_p = \frac{1}{64}$, ACORN reduces the number of distance computations by 89.9% to achieve the recall@10 of 90% compared to HNSW-VBASE, but at the expense of 20.5× more predicate filtering. The static and single neighbor-handling strategies in existing methods make them unable to adapt to the temporal and spatial variations of real-world workloads.

***Inefficient candidate supplementation.*** The search efficiency of graph-based indexes degrades if there is an insufficient supplementation of candidate vectors during neighbor traversal, which results in poor connectivity of the search subgraph [19, 33]. This is problematic when only filtering on "one-hop" neighbors since the number of new candidate vectors can be limited after predicate filtering. ACORN solves this by querying both "one-hop" and "two-hop" neighbors during each neighbor traversal. However, the "two-hop" neighbors may be far from the query vector since the spatial proximity is not transitive. As shown in Figure 4, compared to the HNSW-VBASE, the "two-hop" search strategy expands ACORN's predicate filtering by orders of magnitude, severely degrading the performance if the predicate filtering cost is non-negligible compared to the distance computation cost. Besides, it even enlarges the number of distance computations when the predicate selectivity $S_p$ is very high (like when $S_p = \frac{1}{2}$ in this experiment). Methods such as NaviX [40] partially alleviate this problem by not traversing the "two-hop" neighbors when local $S_p$ is high, but the problem of inefficient candidate supplementation remains at a low $S_p$.

## 2.4 Measurement on Vector Database Systems

Since the distance computation and predicate filtering overheads can be affected by data organization, we also evaluate the above findings on a popular commercial vector database, Weaviate [53].

***Non-negligible filtering overhead.*** For each workload, we sample ten thousand vectors to build a Weaviate flat index [52] and extract the per-query summation of the distance computation and predicate filtering overheads with Weaviate's metric monitor [54]. As shown in Figure 5(a), although there remain non-negligible filtering overheads for workloads LAION and Amazon, workloads such as SIFT and H&M demonstrate much lower filtering overheads. This is because the Weaviate pre-filters on all vectors in the dataset to generate an allow-list before querying, which can be optimized with offline-built inverted indexes [51]. Although pre-filtering amortizes the individual




(a) Overhead measurement. (b) Adaptivity to different $S_p$.

Fig. 5. Measurement of the workload characteristics and their effects on query performance on Weaviate.

filtering overhead, it increases the total proportion of filtering overhead (e.g., occupying 70% [40] of a query) since an ANN search only needs distance computations of a portion of the dataset vectors. On the contrary, this paper focuses on "filter-during-search" [8] that only performs predicate filtering when needed, enabling more flexible scheduling of distance computation and predicate filtering overhead, which yields lower total filtering overhead.

***Poor workload adaptivity.*** We also evaluate DFN-only and DOS-only methods on Weaviate with different selectivity levels. As shown in Figure 5(b), it reconfirms that neither of these methods can consistently outperform the other with varying $S_p$. This trend is also reported in Weaviate's official experiments [37].

## 3 PathSeer Overview

We propose PathSeer, a graph-based filtered ANN search system for efficient and adaptive retrieval.

### 3.1 Challenges and Solutions

The key idea of PathSeer is the dynamic and hybrid selection of neighbor-handling strategies for each neighbor traversal, which simultaneously exploits the DFN and DOS neighbor-handling strategies and carefully tunes their proportions according to workload characteristics. However, there are mainly three challenges when realizing this idea. PathSeer exploits three techniques to tackle these challenges.

First, it is challenging to simultaneously exploit the DFN and DOS neighbor-handling strategies within one index, as they prefer different index structures. Specifically, the DFN neighbor-handling prefers a sparse index structure with neighbor pruning [20], which existing works have shown to be efficient for finding the nearest neighbors [25, 33]. On the contrary, the DOS neighbor-handling prefers a dense index structure that records the complete neighbor information to avoid missing satisfactory neighbors [33]. Regarding this challenge, we design a fusion vector indexing scheme that integrates the index features of both neighbor-handling strategies. Vector neighbors are divided into two zones, named the sparse zone and the expanding zone. Nearby neighbors are put in the sparse zone and pruned with RNG-based pruning [20] to keep spatial sparsity, fitting the DFN neighbor-handling. The remaining neighbors, including those pruned out from the sparse zone, are put in the expanding zone for complete neighbor information, fitting the DOS neighbor-handling.

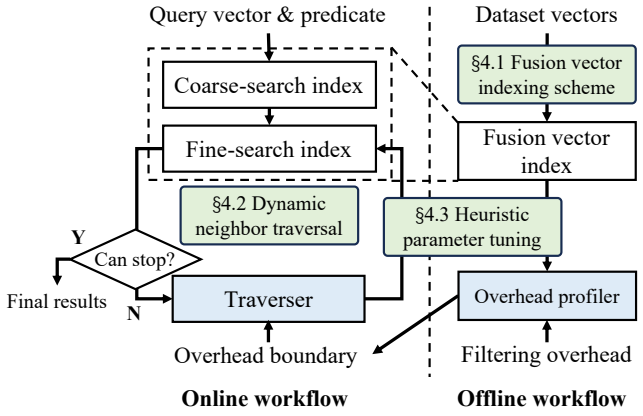



Fig. 6. The offline and online workflows of PathSeer.

Second, simply mixing DFN and DOS neighbor-handling strategies during each neighbor traversal can lead to poor search performance. Specifically, the DOS neighbor-handling strategy only performs distance computation on satisfactory neighbors and will have insufficient numbers of candidate vectors for graph traversal under a low $S_p$, impeding the search performance [19, 33]. Although the DOS-only methods solve this problem by the "two-hop" search, this approach leads to inefficient candidate supplementation as we have shown in Section 2.3. Regarding this challenge, we devise a dynamic neighbor traversal strategy to enable tuning the proportion of neighbor-handling strategies while keeping high search performance. It leverages the observation that the number of candidate vectors provided by the DFN neighbor-handling is not influenced by $S_p$ since it performs distance computation on both satisfactory and unsatisfactory vectors. Regarding this, the dynamic neighbor traversal fully traverses the sparse zone with DFN neighbor-handling and partially traverses the expanding zone with DOS neighbor-handling according to the filtering ratio $r$, guaranteeing both the connectivity of the search subgraph and workload adaptivity.

Third, it is challenging to decide the best filtering ratio $r$ for each neighbor traversal since it is hard to derive a clear relationship between $r$ and the query efficiency. Specifically, the query efficiency includes both the search speed and recall, which are typically contradictory goals [60]. Tuning $r$ will not only influence the search speed by changing the number of distance computations and predicate filtering, but also alter the search recall since the number of traversed neighbors changes accordingly. Regarding this challenge, we design a heuristic parameter tuning mechanism. It is inspired by previous works [25, 33], which suggest that a bounded number of distance computations for each neighbor traversal yield a good balance of search speed and recall for the graph-based methods. We step further by taking both the distance computation and the predicate filtering overheads into account, indicated with a new metric called *virtual overhead*. By profiling the best virtual overhead boundary offline, we tune the $r$ for each neighbor traversal online according to this boundary, achieving high search efficiency even for dynamic workloads.

### 3.2 PathSeer Workflow

Figure 6 presents the main workflow of PathSeer, which includes both offline and online workflows.

For the offline phase, PathSeer exploits the fusion vector indexing scheme (§ 4.1) to construct a predicate-agnostic index on the dataset vectors. The fusion vector index is a hierarchical graph-based index that comprises both coarse-search and fine-search indexes. The coarse-search index consists of multiple proximity graphs on sampled dataset vectors, and the fine-search index is a

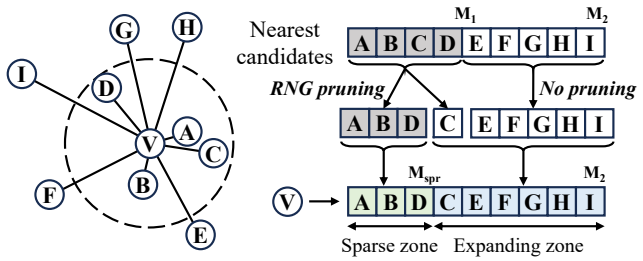



Fig. 7. The zone division and rearranging pruning of the fusion vector indexing scheme.

proximity graph on all dataset vectors. Moreover, to achieve workload adaptivity, an overhead profiler is responsible for deriving the best virtual overhead boundary for each combination of the index configuration and predicate filtering overhead.

For the online phase, PathSeer first searches on the coarse-search index with the query vector to locate a dataset vector near the query vector as the fine-search entry point [25]. Then, when traversing the neighbors of each candidate vector from the fine-search index, PathSeer exploits the dynamic neighbor traversal (§ 4.2) to divide the neighbors into two parts and apply two neighbor-handling strategies respectively. Moreover, with the heuristic parameter tuning (§ 4.3), the filtering ratio $r$ of each neighbor traversal is dynamically decided according to workload characteristics for good adaptivity. Traversing the fine-search index is recursively performed until the stop condition is met.

## 4 Key Techniques

### 4.1 Fusion Vector Indexing Scheme

Similar to HNSW [25], the fusion vector index is hierarchical, with the bottom layer serving as the fine-search index and the upper layers serving as the coarse-search index. It exploits *zone division* to maintain an index that is suitable for different neighbor-handling strategies. As shown in Figure 7, each neighbor of a vector $V$ either belongs to the *sparse zone* or the *expanding zone*. The size of the sparse zone does not exceed $M_1$, and the total size of both zones does not exceed $M_2$. The selection of $M_1$ and $M_2$ will be discussed in Section 4.3.4.

PathSeer builds the fusion vector index without involving vector attributes. Algorithm 1 presents the main procedure of inserting a new vector $v_{pt}$ into the fusion vector index. First, PathSeer generates a random vector layer number $l_{pt}$ according to an exponential distribution [25]. Then, it retrieves an entry vector $v_n$ from the layer $(l_{pt} + 1)$ of the coarse-search index as HNSW does (see Algorithm 1 of [25]) (line 1), and $v_{pt}$ is inserted into each layer numbered from $l_{pt}$ to 0 (line 2). Within each layer, PathSeer searches no more than $M_2$ nearest candidate vectors as HNSW does (see Algorithm 2 of [25]), splitting them into the $M_1$ nearest ones and the remaining ones, denoted by $V_{spr}$ and $V_{exp}$, respectively (line 4). Finally, PathSeer updates the index for $v_{pt}$ (lines 5-13), in which PathSeer specially exploits the following pruning and victim selection methods:

***Rearranging pruning.*** As shown in Figure 7, PathSeer exploits the *rearranging pruning* to combine the index features of two neighbor-handling strategies within one index. On the one hand, candidates for the sparse zone (i.e., $V_{spr}$) are pruned with the RNG-based pruning [20] to enhance the spatial sparsity and boost the search efficiency of the DFN neighbor-handling (line 5). On the other hand, the remaining candidates (i.e., $V_{exp}$), together with those that are pruned from the sparse zone (line 10), are placed in the expanding zone to provide more complete neighbor information for efficient DOS neighbor-handling. Unlike ACORN that exploits predicate-agnostic

**Algorithm 1** Fusion vector indexing algorithm.

**Input:** Vector $v_{pt}$, vector layer number $l_{pt}$
**Output:** none

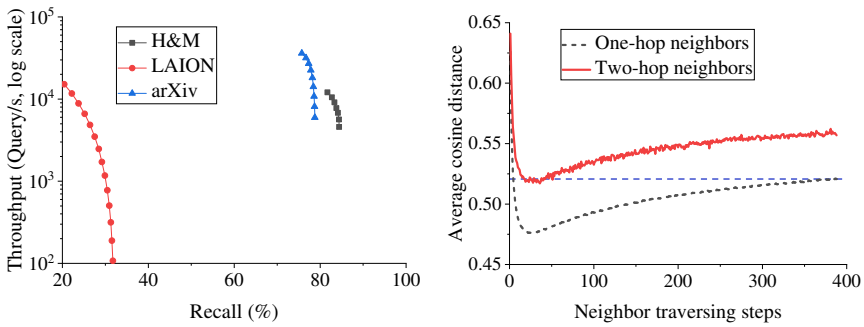
```
1:  v_n ← coarse_search(v_pt, l_pt + 1)
2:  for l = l_pt, ..., 0 do
3:      (M_1, M_2) ← get_layer_size_limit(l)
4:      (V_spr, V_exp) ← search_candidates(v_n, v_pt, l, M_1, M_2)
5:      (V_spr, V_pruned) ← RNG_pruning(v_pt, V_spr, M_1)
6:      add_spr_zone_batch(v_pt, V_spr, l)
7:      for v in V_spr do
8:          add_spr_zone(v, v_pt, l)
9:      if l == 0 then
10:         V_exp ← V_exp ∪ V_pruned
11:         add_exp_zone_batch(v_pt, V_exp, l)
12:         for v in V_exp do
13:             add_exp_zone_fast(v, v_pt, l)
```

pruning for DOS-only neighbor-handling [33], PathSeer does not prune the expanding zone since it does not exploit the "two-hop" search, as we will show in Section 4.2.

***Binary-search-based victim selection.*** Inserting a new vector $v_{pt}$ involves connecting $v_{pt}$ and its neighbors (lines 6-13). When the number of neighbors exceeds the size limit, a victim neighbor is selected for eviction. However, we have found that directly applying the heap-based victim selection [5, 27] to select the farthest neighbor incurs significant index construction overhead, especially when inserting $v_{pt}$ to the expanding zones of its neighbors (line 13). Regarding this, we optimize it with a binary-search-based victim selection algorithm. The main opportunity is that, as PathSeer does not prune the expanding zone neighbors, the only metric for victim selection is the spatial similarity. Specifically, for any vector $v_i$, PathSeer keeps its expanding zone neighbors in a distance ascending order regarding $v_i$. When inserting a new candidate $v_{exp}$ to the expanding zone, PathSeer quickly locates a position $p$ where $v_{exp}$ should be inserted with the binary search. If $p$ is within the expanding zone's size limit $M_2$, PathSeer uses *memmove* [10] to move forward the neighbors within $[p, M_2 - 1)$ by one unit, making room for $v_{exp}$. According to experiments in Section 5.5, the binary-search-based victim selection reduces the index construction time by 95% without degrading query throughput–recall performance.

PathSeer only maintains the expanding zone for the fine-search index (lines 9-13). Since vectors in the coarse-search index are sampled vectors from the entire dataset, neighbors from the coarse-search index may be spatially far from the query's local region. Therefore, searching the expanding zones of the coarse-search index has limited benefit for retrieving the *top-k* nearest satisfactory vectors to the query. For the fine-search index, PathSeer maintains both the sparse zones and the expanding zones to retrieve the *top-k* nearest satisfactory vectors from the entire dataset.

Finally, we introduce the data structure of the fusion vector index and how it supports vector updates. PathSeer employs the appendable lists to store vector data, adjacency lists of the graph index, and other metadata (e.g., the sparse zone size $M_{spr}$) for each vector. Each inserted vector is assigned a monotonically increasing internal ID, which is used to quickly locate vector data and other metadata in the appendable lists. Vector insertion is naturally supported since the fusion vector index can be built incrementally, and the related data of new vectors can be directly added to the appendable lists. There is no specifically designed algorithm for deletion in PathSeer, since




(a) Performance of DOS-only method when only traversing the "one-hop" neighbors.

(b) The distance between the query vector and different kinds of neighbors across the traversal steps.

Fig. 8. Problems of applying "one-hop" or "two-hop" search in DOS-only neighbor traversal.

PathSeer is compatible with typical deletion methods on graph-based indexes, such as marking the deleted vectors or graph reconstruction [62].

### 4.2 Dynamic Neighbor Traversal Strategy

PathSeer exploits a dynamic neighbor traversal strategy to enable adjusting the proportion of different neighbor-handling strategies for workload adaptivity. Adjusting the proportion of different neighbor-handling strategies within a graph-based index requires a careful design, since applying the filter during traversal may harm the connectivity of the search subgraph [19], resulting in poor search recall rates. As shown in Figure 8(a), we evaluate the six workloads from Section 5 with the DOS-only search method on the "one-hop" neighbors, where three of them fail to reach a high recall (e.g., 90% [43]) even under exhaustive retrieval.

Although previous works [33, 40] solve the above problem with the "two-hop" search, we find that traversing the "two-hop" neighbors is not efficient, as most of them are far from the query vector. We illustrate this by measuring the average distance between the query vector and the traversed neighbors of the fine-search layer under the Amazon workload. As shown in Figure 8(b), most "two-hop" neighbors are even farther than the farthest "one-hop" neighbor throughout the retrieval. This results in excessive numbers of predicate filtering and distance computations on vectors far from the query and severely degrades the query performance.

Based on the above analysis, we conclude two guidelines when designing PathSeer's search algorithm. First, the search subgraph should keep good connectivity regardless of the workload $S_p$, which is realized by traversing all sparse-zone neighbors with the DFN neighbor-handling strategy and adding both the satisfactory and unsatisfactory vectors to the candidate heap. Second, we avoid inefficient candidate supplementation by not exploiting the "two-hop" neighbors for the DOS neighbor-handling strategy.

Figure 9 illustrates the main search procedure of PathSeer. For each query, PathSeer maintains a candidate heap and a result heap of the vectors, whose maximum sizes are a configurable parameter ($ef$) and the number of nearest neighbors the user requires ($k$), respectively. If a heap exceeds its size limit after adding a new vector, it will pop out the vector that is farthest from the query vector. For each query, PathSeer exploits the coarse search algorithm to find an entry vector from the fine-search layer (step 1). Then, PathSeer initializes the candidate heap with the entry vector and repeatedly pops the vector that is nearest to the query vector for dynamic neighbor traversal. With

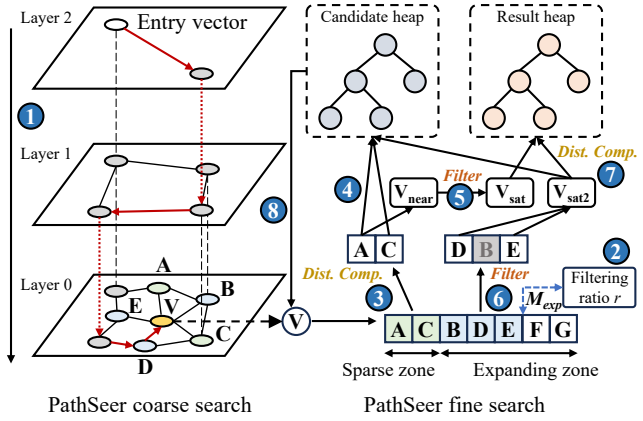



Fig. 9. Illustration of the search process and dynamic neighbor traversal.

a dedicated $M_{exp}$ for each neighbor traversal, PathSeer only traverses neighbors within $M_{exp}$. This implicitly decides the filtering ratio $r$ since $r = 1 - M_{spr}/M_{exp}$ (step 2). To enhance the connectivity of the search subgraph, all sparse zone vectors are handled by the DFN neighbor-handling (step 3). Specifically, PathSeer performs distance computation on all sparse zone vectors and supplements them into the candidate heap (step 4). Besides, only vectors that are eligible to be added to the current result heap ($V_{near}$) are filtered, and the satisfactory ones ($V_{sat}$) are added to the result heap (step 5). Expanding zone vectors within $M_{exp}$ are handled by the DOS neighbor-handling, each filtered with the query predicate (step 6). Only the satisfactory ones ($V_{sat2}$) participate in distance computation and are added to both the candidate heap and result heap (step 7). The above traversal is repeatedly performed until the candidate heap is empty (step 8).

Algorithm 2 presents the details of dynamic neighbor traversal. After initialization and the coarse search (lines 1-8), for each step of the traversal, the dynamic neighbor traversal handles all sparse zone vectors with the DFN neighbor-handling (lines 11-19). Afterwards, PathSeer traverses the expanding zone vectors with the DOS neighbor-handling (lines 20-29). Expanding zone traversal may terminate early (lines 21-22) according to $M_{exp}$ derived from Section 4.3.

### 4.3 Heuristic Parameter Tuning Method

The heuristic parameter tuning is mainly used to tune each $M_{exp}$ (and $r$ accordingly) of the dynamic neighbor traversal. Empirically, increasing the filtering ratio will handle more vectors with the DOS neighbor-handling, which reduces the number of distance computations, but at the expense of more predicate filtering. However, the search recall rate will also change as the filtering ratio grows since more vectors are traversed. Therefore, the query throughput at a certain recall can change in a non-monotonic way as the filtering ratio grows, which makes it challenging to derive the best filtering ratio for each neighbor traversal theoretically.

Regarding this challenge, we propose a hypothesis that an efficient filtered ANN query should keep each neighbor traversal overhead within a certain boundary, keeping a good balance between the search overhead and completeness. This hypothesis is consistent with existing graph-based search methods [33, 63] that restrict the maximum number of distance computations for each neighbor traversal, under the assumption that the predicate filtering overhead is negligible. However, we consider a more general case by taking the filtering overhead into account.

**Algorithm 2** Dynamic neighbor traversal algorithm.

**Input:** query vector $v_q$, query predicate $p_q$, number of required vectors $k$, size of dynamic list $ef$
**Output:** $heap_r$ of $k$ approximately nearest satisfactory vectors

```
vt ← Empty visited table
heap_c ← Empty candidate heap of maximum size ef
heap_r ← Empty result heap of maximum size k
v_n ← coarse_search(v_q, 1)
heap_c.push(v_n, dist(v_q, v_n))
vt.set(v_n)
if p_q(v_n) then
    heap_r.push(v_n, dist(v_q, v_n))
while heap_c.size() > 0 do
    (v_0, d_0) ← heap_c.pop_min()
    for v_1 in sparse_zone_neighbors(v_0, 0) do
        if vt.get(v_1) then
            continue
        vt.set(v_1)
        d ← dist(v_1, v_q)
        heap_c.push(v_1, d)
        if heap_r.size() < k or d < heap_r.farthest_dist() then
            if p_q(v_1) then
                heap_r.push(v_1, d)
    for v_1 in expanding_zone_neighbors(v_0, 0) do
        if traverse_should_stop() then
            break
        if vt.get(v_1) then
            continue
        vt.set(v_1)
        if p_q(v_1) then
            d ← dist(v_1, v_q)
            heap_c.push(v_1, d)
            heap_r.push(v_1, d)
```

*4.3.1 Virtual overhead and its characteristics.* Based on the above analysis, we propose a metric to depict the neighbor traversal overhead called *virtual overhead*. When traversing the neighbors of a candidate vector, assuming that the local predicate selectivity is $S_p$, we first define the traversal overhead of an individual vector as $O_i = O_{filter} + O_{dist} \cdot S_p$, in which the $O_{filter}$ and $O_{dist}$ are the average overhead of predicate filtering and distance computation, respectively. Then the virtual overhead of traversing the first $M_{exp}$ neighbors is defined as $M_{exp} \cdot O_i$ accordingly. Note that this definition includes some approximations for the traversal overhead of the sparse zone. The main reason is that the sparse zone sizes $M_{spr}$ vary among different vectors due to the RNG-based pruning, and accounting for its overhead results in significant variations of the virtual overhead even when the satisfactory vectors are uniformly distributed. This will make it hard for PathSeer to decide a fixed metric offline that can guide efficient parameter tuning online.

By defining the virtual overhead, we verify our hypothesis with the six datasets by profiling the relationship between the virtual overhead and the query performance. We start with the workload with uniformly distributed satisfactory vectors by generating synthetic attributes for each dataset

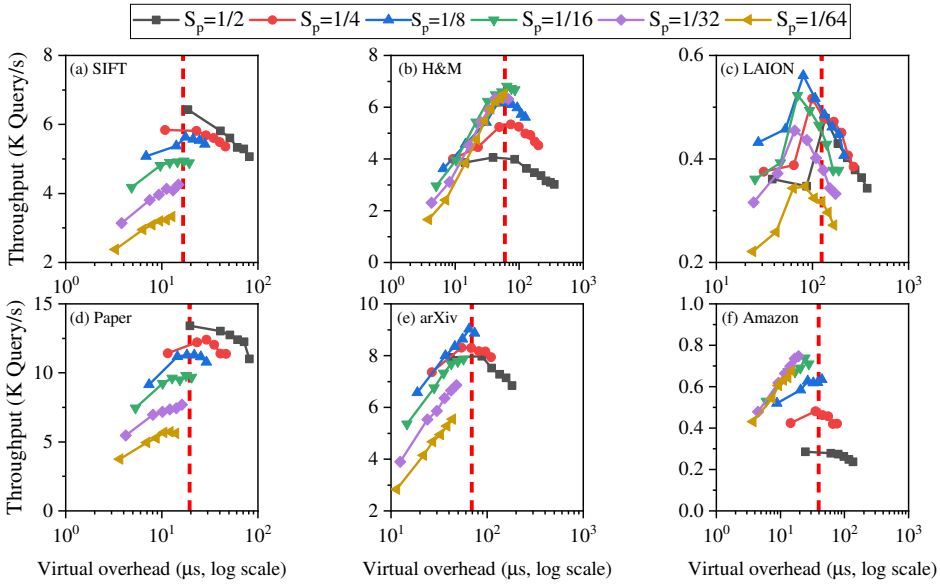



Fig. 10. Relationship between query throughput and average virtual overhead under different vector datasets.

to alter the $S_p$, while retaining the actual distance computation and predicate filtering overheads. As shown in Figure 10, the dots within each line are derived by altering the fixed $M_{exp}$ for all neighbor traversals (which results in different virtual overheads) and measuring their query throughput when the recall@10 equals 0.9. It can be observed that for each workload, **the query throughput tends to peak around a similar $VO_B$ across different $S_p$**, which is in accord with our hypothesis. We have the following intuitive explanation for this observation. Assuming that $M_{exp}$ neighbors are traversed for a neighbor traversal, the virtual overhead can be estimated as $M_{exp} \cdot (O_{filter} + O_{dist} \cdot S_p)$. Decreasing $S_p$ reduces the latter term, but the $M_{exp}$ (i.e., the former term) that leads to the best search performance will increase since a lower $S_p$ prefers more neighbors handled by the DOS strategy, as shown in Figure 1. This makes the overall virtual overhead change only slightly.

*4.3.2 Adaptive search based on virtual overhead.* The above observation motivates us to design a heuristic parameter tuning mechanism based on the virtual overhead. The main idea is to terminate the neighbor traversal early if the current virtual overhead reaches the boundary $VO_B$, which is irrelevant to the workload's $S_p$. Such an overhead-bounded search method implicitly takes the workload characteristics (i.e., $R_p$ and local $S_p$) into consideration, which automatically decides the $M_{exp}$ for each neighbor traversal without user involvement. Figure 11 illustrates this with three cases. In case 1, the workload has both a small $R_p$ and $S_p$, and all $M_2$ neighbors are traversed as the accumulated virtual overhead is less than $VO_B$. In both cases 2 and 3, the dynamic neighbor traversal terminates early since the actual virtual overhead reaches the $VO_B$. Specifically, the high $S_p$ of case 2 workload brings more distance computation overhead since it has more satisfactory vectors, and the high $R_p$ of case 3 workload makes the overhead of each predicate filtering larger. Since the $M_{exp}$ and $r$ conform with $r = 1 - M_{spr}/M_{exp}$, the above search process also decides the filtering ratio $r$ for each neighbor traversal, which implicitly decides the proportion of DFN and DOS strategies as Figure 2 suggests.

When handling real-world filtered ANN search workloads with spatially variant $S_p$, PathSeer heuristically decides the $M_{exp}$ (and $r$) for each neighbor traversal by early terminating each neighbor

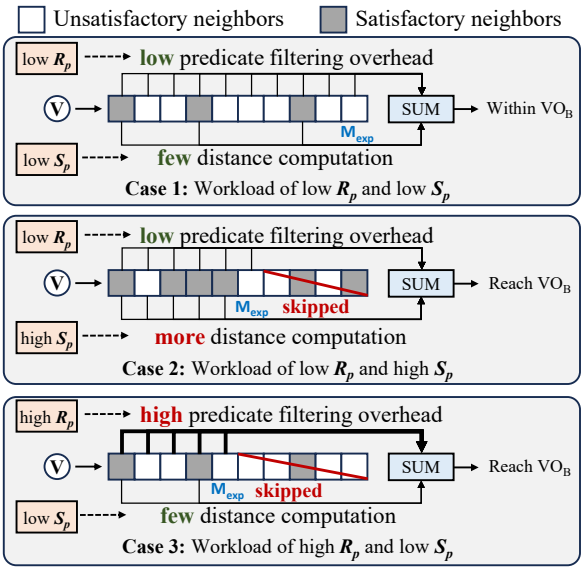



Fig. 11. Heuristic parameter tuning for different workload characteristics with virtual overhead boundary.

traversal according to local $S_p$ and the $VO_B$ derived from synthetic and uniformly distributed $S_p$. This can capture the attribute correlation since a smaller $S_p$ (i.e., low-correlation areas) will produce a larger $M_{exp}$ (and $r$) and vice versa. Such a heuristic parameter tuning method is reasonable considering that $VO_B$ weakly correlates with $S_p$ (see Section 4.3.1). This design enables PathSeer to tune the search based on workload characteristics online without knowing the distribution of $S_p$ before querying. We evaluate the effectiveness of this heuristic parameter tuning method in Section 5.3.

*4.3.3 Profiling the virtual overhead boundary $VO_B$.* PathSeer profiles the virtual overhead boundary $VO_B$ offline without knowing the online query in advance. For a fusion vector index on a vector dataset, PathSeer generates synthetic query vectors by averaging sampled dataset vectors. Then, PathSeer evaluates workloads of different $S_p$ levels by generating synthetic attributes for both the dataset and the query vectors and exploits the attribute matching as the query predicate. The predicate filtering overhead $O_{filter}$ is simulated by busy-waiting during each filter. Specifically, for each $S_p$ level, PathSeer evaluates and fits, with quadratic functions, the relationship between virtual overhead and performance under a fixed recall rate. Finally, PathSeer derives the best virtual overhead $VO_B$ that results in the lowest average throughput degradation compared to the peak throughput among all evaluated $S_p$. In Figure 10, we use the red lines to depict the profiled $VO_B$ among the six workloads.

The above profiling only assumes knowing the distance computation and predicate filtering overheads of the online query, and the distribution of $S_p$ is not required before online querying. On the one hand, although the distance computation overhead can vary with hardware setups, these factors are usually stable after the vector database is deployed, so it is not costly to profile for each new setup. On the other hand, the predicate filtering overhead may vary among different predicates. One tentative solution is to profile the $VO_B$ of multiple $O_{filter}$ offline and measure the actual $O_{filter}$ online to select a matched $VO_B$ for querying.

Table 2. Vector datasets used for evaluation.

| Workload | #Vectors | #Queries | Dim | Query predicate |
|---|---|---|---|---|
| SIFT | 1000000 | 10000 | 128 | Integer matching |
| H&M | 104100 | 1000 | 2048 | Composite predicate |
| LAION | 11637603 | 10000 | 512 | String matching |
| Paper | 2000000 | 10000 | 200 | Integer comparison |
| arXiv | 122687 | 10000 | 768 | Composite predicate |
| Amazon | 5980397 | 10000 | 768 | Integer comparison |

*4.3.4 Selection of $M_1$ and $M_2$.* As we have shown in Section 4.1, the fusion vector index has two user-defined parameters $M_1$ and $M_2$. Although the heuristic parameter tuning does not decide them directly, suitable values for them can be estimated as follows. First, a larger $M_2$ enables more scheduling space for heuristic parameter tuning, which, however, expands the index size. Thus, the $M_2$ can be estimated as the largest possible $M_{exp}$, which can be derived from $S_p$, $O_{filter}$, and $O_{dist}$ of typical workloads. Second, $M_1$ influences the trade-off between the search-subgraph's connectivity and the number of distance computations. We choose $M_1$ as follows: Assuming that for a candidate $M_c$ and its profiled virtual overhead boundary $VO_c$, $P$ is the average throughput of all synthetic profiling workloads when the virtual overhead boundary is set to $VO_c$, then $M_1$ should be the $M_c$ with the largest $P$.

## 5 Performance Evaluation

Our evaluation tries to answer the following questions:

- How does PathSeer perform compared to other filtered ANN search approaches? (§ 5.2)
- What are the effects of individual techniques used by PathSeer? (§ 5.3)
- Can PathSeer adapt to workloads with different characteristics? (§ 5.4)
- What is the overhead of PathSeer's fusion vector index? (§ 5.5)
- To what extent is the performance of PathSeer sensitive to its hyperparameters? (§ 5.6)

### 5.1 Experimental Setup

***Platform.*** We conduct the experiments on a server that consists of two 10-core Intel Xeon Silver 4210R CPUs and 128GB DDR4 memory. Each core supports two concurrent threads through hyper-threading. The operating system is Ubuntu 18.04 with Linux kernel version 5.4. The evaluated programs are compiled with g++ 7.5.0. When evaluating the query throughput, we use OpenMP [30] to execute different queries in parallel. To obtain stable performance, we let the number of OpenMP threads equal the number of CPU physical cores and bind the threads to cores by setting *OMP_PROC_BIND* to true.

***Evaluated workloads.*** The evaluation is conducted on six vector datasets as summarized in Table 2. Details of the workload construction are listed below. Query vectors are randomly extracted from the dataset vectors if the dataset itself does not have them.

- **SIFT** is built from the SIFT1M dataset [1]. Following [15, 33, 47], when evaluating the overall performance, we assign random integers to both the dataset and query vectors according to a uniform distribution over the range [0, 6). Moreover, to evaluate workloads of different characteristics, we change the range of generated integers and add a busy sleep function for each predicate filtering, resulting in different $S_p$ and $R_p$, respectively. The query predicate is that the vector attribute equals the query attribute.

Table 3. Parameters used for the compared filtered ANN search methods.

| | IVFFlat ($N_{cluster}$) | HNSW-VBASE ($M$) | NaviX ($M$) | ACORN ($M$, $M_\beta$, $\gamma$) | PathSeer ($M_1$, $M_2$) |
|---|---|---|---|---|---|
| SIFT | 512 | 32 | 32 | (64, 128, 12) | (64, 128) |
| H&M | 256 | 64 | 32 | (32, 64, 156) | (16, 128) |
| LAION | 512 | 64 | 64 | (64, 128, 38) | (32, 128) |
| Paper | 512 | 32 | 32 | (32, 64, 40) | (64, 128) |
| arXiv | 256 | 64 | 64 | (32, 64, 189) | (64, 128) |
| Amazon | 2048 | 64 | 64 | (32, 64, 3) | (64, 128) |

- **H&M** is built with the H&M dataset [3]. We generate embeddings of the images with EfficientNet-B5 [44] and extract three meaningful columns (product type, perceived color, and department name) from its attribute table as the vector attributes, randomly dropping some columns for each query to simulate partial-attribute filtering. These categorical attributes are converted to integers before evaluation. The query predicate is that the vector's attributes conform to the constraints on the attributes.
- **LAION** is built from the image-text dataset LAION-400M [2] by extracting over 10M items from it. Similar to [33], we use the image embeddings as the dataset vectors, while the corresponding textual captions are employed as attributes. The query predicate is string matching, where the matched words are selected from frequent and meaningful words extracted from all captions.
- **Paper** is built by assigning a citation count for each vector from the Paper dataset [48]. The citation count is initially sampled from a log-normal distribution and then increases over iterations following a Poisson-based preferential attachment process with temporal aging [46]. The query predicate is that the paper citation is at least 50.
- **arXiv** is built with the task embeddings from the dataset of arXiv papers [31]. Following [8], attributes of each vector are the publication time extracted from the arXiv ID, as well as the categorical labels from the "tasks", "labels", and "dataset" columns. The query predicate is that the publication time of the dataset vector is later than that of the query, and the labels of the dataset vector conform to the label constraints in the query. When extracting queries from the dataset, we omit those with no label.
- **Amazon** is built with the Amazon Product Graph [26]. We use the 2014 version and retain only items with valid product images and non-null price values. Each product is represented by a CLIP-based image embedding, and its price is used as the attribute. The query predicate is that the price of the dataset vector is lower than the price of the query vector.

***Baseline methods.*** We compare PathSeer with the graph-based HNSW-VBASE [63], ACORN [33], and NaviX [40], as well as the cluster-based IVFFlat [12]. The evaluation of ACORN is based on its source code [5], adopted from the Faiss codebase [27] version 1.7.3. For fair comparison, we implement PathSeer in the Faiss codebase of the same version with about 4K LOC. The implementations of HNSW-VBASE and IVFFlat are also adopted from this Faiss codebase. Besides, we integrate NaviX [40] in our benchmark following its source code [41]. All methods run in the "filter-during-search" [8] mode that only performs predicate filtering when needed. Attributes are stored in hash tables for fast individual accesses, and the query predicate is customized with IDSelector [4] for each workload. The distance metric in all experiments is the cosine distance [56]. All methods have tunable index construction parameters that affect the query performance. For the compared methods, we perform an offline grid search [11] on valid parameters and choose the best

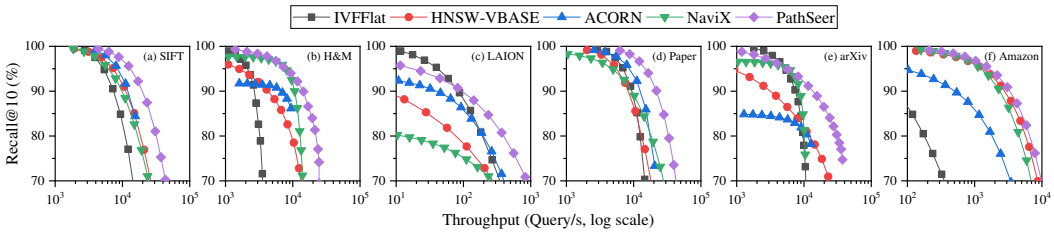



Fig. 12. Query throughput of different filtered ANN search methods under varying recall rates.

one for each workload, as detailed in Table 3. For PathSeer, we set $M_1$ according to Section 4.3.4 and $M_2$ to 128 for all workloads, and discuss the parameter sensitivity in Section 5.6.

## 5.2 Overall Performance

In this section, we compare the overall performance of different filtered ANN search methods by measuring the throughput they can achieve for the same recall rates. As all evaluated methods have search parameters to balance the search performance and recall rates, such as the number of searched clusters (*nprobe*) for IVFFlat and the size of dynamic lists ($ef$) for the other graph-based methods, we demonstrate the performance of different methods by plotting the performance-recall curve under different search parameters. The results are shown in Figure 12.

***Comparison with graph-based methods.*** Compared to HNSW-VBASE and ACORN, PathSeer consistently achieves the best performance across different workloads and recall rates. For example, at recall@10 of 85% across six workloads, PathSeer achieves throughput speedups over HNSW-VBASE, ACORN and NaviX ranging from 1.2× to 7.5×, 1.7× to 17.5× and 1.5× to 2.3× (3.2×, 4.9× and 1.9× on average), respectively. When the recall target is 90%, PathSeer achieves throughput speedups over HNSW-VBASE, ACORN and NaviX ranging from 1.2× to 13.5×, 1.6× to 10.6× and 1.3× to 2.3× (4.4×, 4.2× and 1.7× on average), respectively. This is because both HNSW-VBASE and ACORN exploit static selections of neighbor-handling strategies, either introducing excessive distance computation overhead or predicate filtering overhead. While NaviX adjusts its neighbor traversal strategy according to local $S_p$, it only relies on the DOS neighbor handling strategy with limited adaptivity, which even fails to reach target recall rates under the LAION workload. By contrast, the dynamic and hybrid selection of neighbor-handling strategies in PathSeer finds a suitable search path for high recall rates while keeping a good balance between the number of distance computations and predicate filtering, thus achieving the best query performance.

***Comparison with cluster-based method.*** Compared to IVFFlat, across all six workloads with recall@10 of 85% and 90%, PathSeer achieves throughput speedups ranging from 1.7× to 43.5× and 1.2× to 47.4× (9.9× and 10.2× on average), respectively. This performance gain is not only attributed to the dynamic and hybrid selection of neighbor-handling strategies in PathSeer, but also to PathSeer's graph-based index, which enables fine-grained modeling on the vector space to locate target vectors with less unnecessary traversal of vectors far from the query vector. However, when the target recall rate is near 100%, IVFFlat does show comparable or even higher performance than PathSeer, since a relatively large proportion of dataset vectors needs to be traversed to achieve high recalls. This requires the graph-based method to perform an excessive number of distance computations to navigate the proximity graph, resulting in worse performance than the cluster-based method.

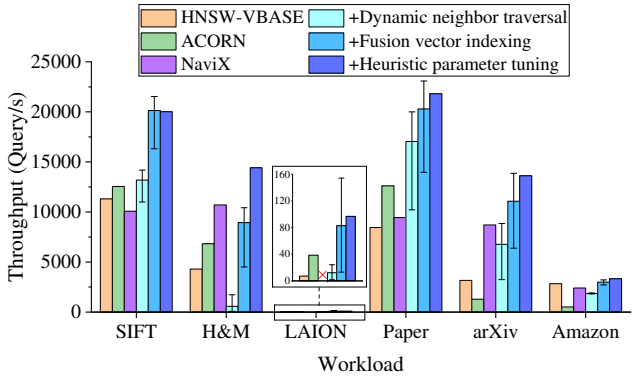



Fig. 13. Effects of individual techniques in PathSeer. The error bars denote the minimum and maximum throughput among different search parameters.

### 5.3 Effects of Individual Techniques

To isolate the improvement brought by PathSeer's key techniques, we evaluate the throughput of PathSeer with the six workloads and progressively add the three techniques. When evaluating PathSeer without heuristic parameter tuning, we use a series of $M_{exp}$ values that are fixed for all neighbor traversals, evaluate the performance of each $M_{exp}$, and present their mean, minimum, and maximum throughput, respectively. HNSW-VBASE, ACORN, and NaviX are employed as baselines to demonstrate the performance of DFN-only and DOS-only methods. We demonstrate the throughput of different methods when the recall@10 equals 90% in Figure 13.

First, we evaluate PathSeer's dynamic neighbor traversal without the fusion vector index scheme by simply setting the first $M_1$ neighbors as the sparse zone neighbors and the remaining neighbors as the expanding zone neighbors for each dataset vector. Under this configuration, PathSeer improves the throughput from 5.1% to 101.6% compared to the baselines for the SIFT and Paper workloads. This is because the dynamic neighbor traversal exploits the dynamic and hybrid selection of the DFN and DOS neighbor-handling, which keeps a good balance between the number of distance computations and predicate filtering. However, for other workloads, PathSeer has up to 94.6% performance degradation compared to the baselines, since these vector datasets introduce much more distance computations during the DFN neighbor-handling with RNG-based pruning.

Second, the fusion vector indexing scheme can efficiently solve the above problem and boost PathSeer's performance. Compared to only using the dynamic neighbor traversal, integrating the fusion vector indexing scheme improves mean throughput by 19.1% to 1446.5%. This is because the rearranging pruning not only keeps a group of sparse neighbors for efficient DFN neighbor-handling, but also maintains enough close neighbors for efficient DOS neighbor-handling. However, such performance improvements require manual parameter tuning. Selecting an improper search parameter may result in severe performance degradation, such as the 42.4% degradation of the arXiv workload.

Finally, adding the heuristic parameter tuning enables PathSeer to tune the $M_{exp}$ for each neighbor traversal based on workload characteristics, which improves the throughput by 19.9% on average. Specifically, for H&M and Amazon, the heuristic parameter tuning delivers better performance than statically assigning the same $M_{exp}$ for each neighbor traversal. This demonstrates that the dynamic selection of neighbor-handling strategies enhances PathSeer's adaptivity to workload characteristics, thus achieving higher overall performance. For the other workloads, the heuristic

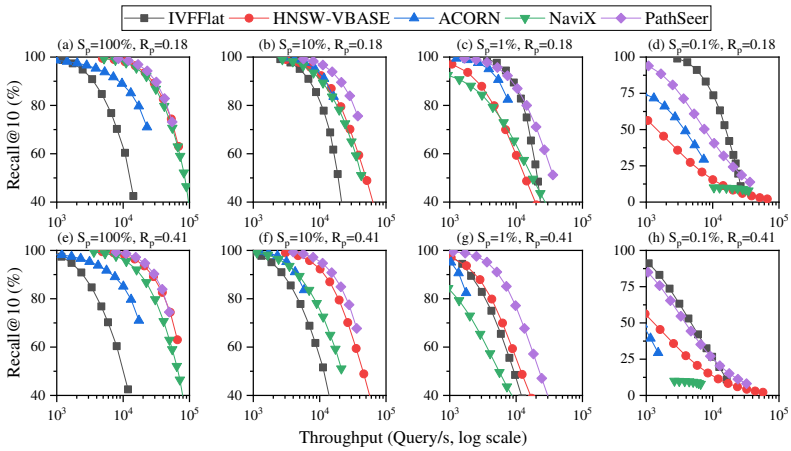



Fig. 14. Performance-recall curves of different methods with workloads of varying characteristics.

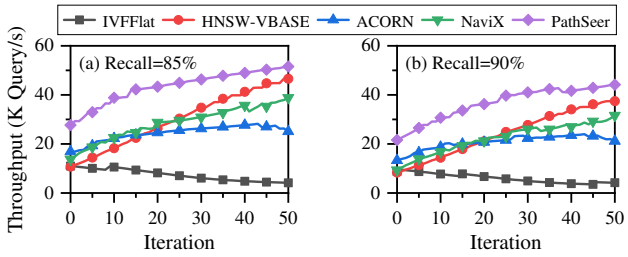
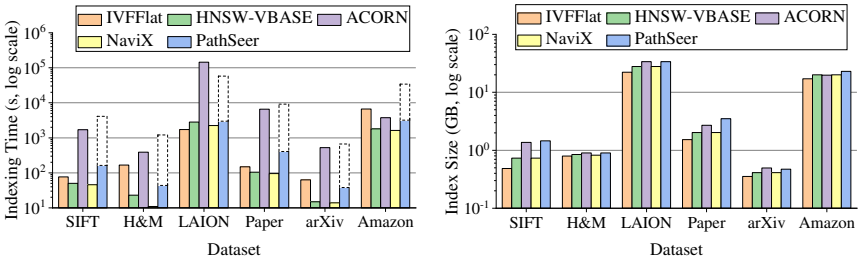



Fig. 15. The throughput of different methods on the Paper dataset with temporally varying attributes.

parameter tuning does not find the configuration that consistently outperforms the static parameter selection. This is because the heuristic parameter tuning is a greedy algorithm that only considers the local selectivity for each neighbor traversal, missing the chance of the globally optimal choice. However, the heuristic parameter tuning enables PathSeer to quickly adapt to any variance of workload characteristics without manual involvement or costly parameter tuning.

### 5.4 Adaptivity to Workload Characteristics

To evaluate whether the dynamic and hybrid selection of neighbor-handling strategies in PathSeer can adapt to workloads with different characteristics, we generate synthetic workloads of different $S_p$ and $R_p$ with the SIFT dataset. For each method, the index construction parameters are fixed for all workloads as in Table 3, and PathSeer's $VO_B$ is specifically profiled for each $R_p$ but is kept the same for different $S_p$ according to Section 4.3.3. Figure 14 presents the performance-recall curves of different methods under varying $S_p$ and $R_p$. It can be observed that for workloads of different characteristics, neither the DFN-only method (HNSW-VBASE) nor the DOS-only methods (ACORN and NaviX) consistently outperform the others, since they rely on static and single neighbor-handling strategies. By contrast, thanks to the hybrid and dynamic selection of neighbor-handling strategies, PathSeer adapts to different $S_p$ and $R_p$ and achieves consistent high performance among different workload characteristics. PathSeer may perform worse than the cluster-based IVFFlat when $S_p$ is 0.1%. This comes from the high distance computation overheads that the graph-based methods pay to navigate through the proximity graph.




(a) Construction time of different methods. (b) Index size of different methods.

Fig. 16. Index overheads of different methods. The white blocks in Figure 16(a) denote time reduction brought by PathSeer's binary-search-based victim selection.

We further evaluate different methods with a workload whose vector attributes are dynamically evolving. We use the Paper dataset, in which the citation of each paper increases over iterations without altering the paper embedding. The query predicate is that the paper citation is higher than 50, and the predicate selectivity $S_p$ increases over iterations as the citations of all papers increase. For each method, the index construction parameter is fixed across iterations, but we may tune the $ef$ or *nprobe* to derive the performance under target recall rates. As shown in Figure 15, all existing methods struggle to handle workloads with dynamically evolving attributes. Specifically, for about the first 20 iterations, both ACORN and NaviX consistently outperform the other two baselines since their DOS neighbor-handling strategy can efficiently avoid distance computation on the unsatisfactory vectors for higher performance. However, they begin falling short of HNSW-VBASE afterwards since the DOS neighbor-handling introduces more distance computations than the DFN neighbor-handling under high $S_p$, even though NaviX has an adaptive search algorithm based on local $S_p$. By contrast, PathSeer consistently achieves the best performance across all iterations since its dynamic selection of neighbor-handling strategies adapts to the varying workload by consistently tuning the search parameter for high performance.

### 5.5 Index Overheads

In this section, we evaluate the index construction time and index size of the fusion vector index in PathSeer. Figure 16(a) presents the index construction time of different methods. The "binary-search-based" victim selection in PathSeer effectively reduces the index construction time by 95% on average, as shown in the white portion of Figure 16(a) (note that the y-axis is on a logarithmic scale). Compared to IVFFlat, HNSW-VBASE, and NaviX, PathSeer increases the construction time by 32.4%, 178.4%, and 247.4% on average, since it needs to traverse more neighbors for the expanding zone when adding a new vector into the index. However, PathSeer can reduce the construction time by 76.8% compared to ACORN, as there is no pruning operation on neighbors from the expanding zone in PathSeer. Moreover, neighbor eviction is accelerated with the binary search optimization during the index construction.

Figure 16(b) presents the index sizes of different methods. It can be observed that the cluster-based method IVFFlat has the smallest index size since it does not need to record fine-grained neighbor information, unlike the graph-based methods. Within the graph-based methods, PathSeer increases the index size by 38.1%, 7.9%, and 38.6% on average compared to HNSW-VBASE, ACORN, and NaviX, mainly as the expanding zone of PathSeer enlarges the number of neighbors that need to be saved for each vector.

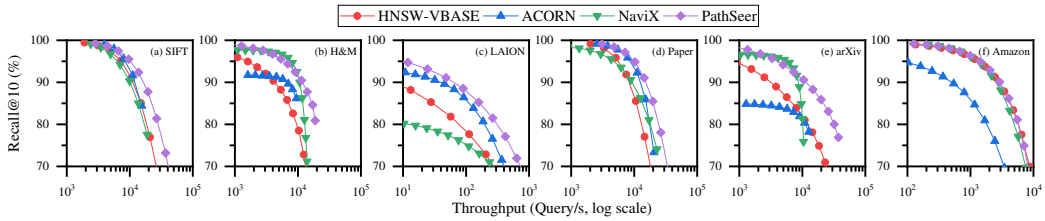



Fig. 17. Performance comparison between PathSeer and the other graph-based methods without expanding the index size.

Table 4. PathSeer's query throughput of different workloads with different parameters $M_1$ and $M_2$. Each cell contains the throughput when the recall target is 85%/90%.

| Workload | $M_1$=32 | | | $M_2$=128 | | |
|---|---|---|---|---|---|---|
| | $M_2$=64 | $M_2$=96 | $M_2$=128 | $M_1$=16 | $M_1$=32 | $M_1$=64 |
| SIFT | 24K/18K | 25K/19K | 22K/17K | 18K/13K | 22K/17K | **26K/20K** |
| H&M | 15K/11K | 17K/12K | 16K/13K | **20K/16K** | 16K/13K | 16K/12K |
| LAION | 165/63 | 231/95 | 224/97 | 175/55 | **224/97** | 197/66 |
| Paper | 26K/20K | 28K/21K | 28K/22K | 28K/21K | 28K/22K | **28K/22K** |
| arXiv | 22K/12K | 25K/15K | 27K/17K | 29K/18K | 27K/17K | **23K/14K** |
| Amazon | 4K/2K | 4K/2K | 4K/3K | 2K/700 | 4K/3K | **5K/3K** |

Regarding PathSeer's expanded index size, there is a concern about whether PathSeer can maintain the high search performance of graph-based methods while keeping the index size low. Regarding this, we adjust PathSeer's index construction parameters to ensure that its index size does not exceed the index sizes of the other graph-based methods. Figure 17 presents the throughput-recall curves of this new setup. When PathSeer's index size is restricted by shrinking the expanding zone, PathSeer shows certain performance degradation since the heuristic parameter tuning has less scheduling opportunity. However, PathSeer can still achieve performance improvements for most recall targets. For example, compared to HNSW-VBASE, ACORN, and NaviX, PathSeer improves throughput by an average of 2.6×, 4.5×, and 1.6× at a recall of 85%, and by 3.3×, 3.5×, and 1.4× at a recall of 90%, respectively.

### 5.6 Sensitivity Analysis

In this section, we evaluate how sensitive PathSeer's performance is to two index construction parameters $M_1$ and $M_2$. For each workload, we build multiple indexes with varying combinations of $M_1$ and $M_2$ and evaluate PathSeer's performance. Table 4 presents the throughput when recall@10 is 85% and 90%, respectively.

First, by fixing the $M_1$ to 32 and varying the $M_2$ from 64 to 128, we demonstrate how the size limit of the expanding zone influences the query performance. It can be observed that the performance of most workloads improves with a larger $M_2$, since a larger $M_2$ increases the maximum virtual overhead PathSeer can achieve and enables efficient search of queries with low $S_p$. Workloads such as SIFT and Amazon are less sensitive to increasing $M_2$ since their workload characteristics allow their virtual overhead boundary to be reached by traversing fewer neighbors. Consequently, selecting an optimal $M_2$ based on the workload-correlated virtual overhead boundary in practical deployment can effectively reduce index size with little performance loss.

Second, we demonstrate the effect of the size limit on the sparse zone by evaluating the performance of different $M_1$ while keeping $M_2$ as 128. It can be observed that the performance variation regarding $M_1$ is more complex since increasing $M_1$ not only enhances the connectivity of the search subgraph but also introduces more distance computation overheads. However, selecting $M_1$ based on Section 4.3.4 (highlighted in Table 4) can achieve the best performance for most workloads, manifesting the effectiveness of this parameter selection method without sacrificing recall.

## 6 Related Work

***Filtering separately.*** Some vector databases like Milvus [47], Chroma [9], Pinecone [19], and AnalyticDB-V [55] handle predicate filtering separately from the vector search, either by pre-filtering or post-filtering. The pre-filtering first finds satisfactory vectors from the whole dataset and then finds the $k$ nearest vectors within, but suffers from large predicate selectivity [33]. The post-filtering first extracts $k'$ (larger than the target $k$) nearest vectors with pure vector search and filters them to obtain the final result, but a suitable $k'$ is hard to derive for both high search performance and accuracy [63]. PathSeer schedules the distance computation and predicate filtering in the fine-grained neighbor traversal level, manifesting high search performance and workload adaptivity.

**Filtering during search.** Some methods perform predicate filtering during the search [8]. On the one hand, methods like VBASE [63], NHQ [49], and HQANN [58] perform distance computation first and then filter on them for the final result, though the latter two systems design specialized distance metrics that integrate the categorical and low-cardinality vector attributes. On the other hand, methods like Filtered-DiskANN [15], ACORN [33], UNG [8], and NaviX [40] check the vector attribute first and only perform distance computation on satisfactory vectors. However, existing methods exploit single and static selection of neighbor-handling strategies. PathSeer differs from them by exploiting the dynamic and hybrid selection of neighbor-handling strategies for workload adaptivity.

**Filtering on specialized index.** Some works like NHQ [49], HQANN [58], Filtered-DiskANN [15], and UNG [8] build specialized vector indexes for filtered ANN search. Specifically, they mainly focus on ANN search with label matching on categorical attributes. This simple filtering pattern enables these works to build specialized indexes that gather satisfactory vectors, achieving high search accuracy with fewer steps of traversal. PathSeer differs from these works as PathSeer is a predicate-agnostic search method, which is not limited to a particular type of predicate and supports flexible attribute altering without index reconstruction.

## 7 Conclusion

In this paper, we present PathSeer, an efficient system for filtered ANN search. Guided by the key idea of dynamic and hybrid selection of neighbor-handling strategies, PathSeer introduces three techniques to adapt to both temporal and spatial variations of the workloads: a fusion vector indexing scheme, dynamic neighbor traversal, and heuristic parameter tuning. Experimental results show that PathSeer achieves 1.17×–47.4× higher throughput than existing methods.

## Acknowledgments

We thank all reviewers for their insightful comments and helpful suggestions. This work was supported by the National Natural Science Foundation of China (No. 62302503, No. 62025203, and No. 62402499), the Innovation Research Foundation of NUDT (No. ZK23-15), the Open Research Fund from State Key Laboratory of High Performance Computing of China (No. 202401-09), and the Young Elite Scientists Sponsorship Program by CAST (No. YESS20240767).

## References


[1] 2010. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/.

[2] 2021. LAION-400-MILLION open dataset. https://laion.ai/blog/laion-400-open-dataset/.

[3] 2022. H&M personalized fashion recommendations. https://www.kaggle.com/competitions/h-and-m-personalized-fashion-recommendations.

[4] 2026. Struct faiss::IDSelector. https://faiss.ai/cpp_api/struct/structfaiss_1_1IDSelector.html.

[5] Guestrin Lab at Stanford University. 2024. ACORN. https://github.com/guestrin-lab/ACORN.

[6] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2025. Graph-based vector search: An experimental evaluation of the state-of-the-art. *Proceedings of the ACM on Management of Data* 3, 1, Article 43 (Feb. 2025), 31 pages. doi: 10.1145/3709693

[7] Zheng Bian, Man Lung Yiu, and Bo Tang. 2025. IGP: Efficient multi-vector retrieval via proximity graph index. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Padua, Italy). Association for Computing Machinery, New York, NY, USA, 2524–2533. doi: 10.1145/3726302.3730004

[8] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. 2024. Navigating labels and vectors: A unified approach to filtered approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–27.

[9] Chroma. 2025. Metadata filtering. https://docs.trychroma.com/docs/querying-collections/metadata-filtering.

[10] Cppreference. 2025. std::memmove. https://cppreference.com/w/cpp/string/byte/memmove.

[11] Theresa Eimer, Marius Lindauer, and Roberta Raileanu. 2023. Hyperparameters in reinforcement learning and how to tune them. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 366, 46 pages.

[12] Faiss. 2025. IndexIVFFlat. https://github.com/facebookresearch/faiss/blob/main/faiss/IndexIVFFlat.cpp.

[13] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 461–474. doi: 10.14778/3303753.3303754

[14] Jianyang Gao and Cheng Long. 2023. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[15] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. 2023. Filtered-DiskANN: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*. 3406–3416.

[16] Mihajlo Grbovic and Haibin Cheng. 2018. Real-time personalization using embeddings for search ranking at Airbnb. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 311–320.

[17] Hao Guo and Youyou Lu. 2025. Achieving low-latency graph-based vector search via aligning best-first search algorithm with SSD. In *19th USENIX Symposium on Operating Systems Design and Implementation*. 171–186.

[18] Ihab F Ilyas, Theodoros Rekatsinas, Vishnu Konda, Jeffrey Pound, Xiaoguang Qi, and Mohamed Soliman. 2022. Saga: A platform for continuous construction and serving of knowledge at scale. In *Proceedings of the 2022 international conference on management of data*. 2259–2272.

[19] Amir Ingber and Edo Liberty. 2025. Accurate and efficient metadata filtering in Pinecone's serverless vector database. In *The 1st Workshop on Vector Databases*.

[20] Jerzy W Jaromczyk and Godfried T Toussaint. 1992. Relative neighborhood graphs and their relatives. *Proc. IEEE* 80, 9 (1992), 1502–1517.

[21] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing systems* 32 (2019).

[22] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[23] Mingjie Li, Yuan-Gen Wang, Peng Zhang, Hanpin Wang, Lisheng Fan, Enxia Li, and Wei Wang. 2022. Deep learning for approximate nearest neighbour search: A survey and future directions. *IEEE Transactions on Knowledge and Data Engineering* 35, 9 (2022), 8997–9018.

[24] Xiangyu Liu, Yang Liu, and Wei Hu. 2024. Knowledge graph error detection with contrastive confidence adaption. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 38. 8824–8831.

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

[26] Julian McAuley. 2014. Amazon product images dataset (2014 version). https://cseweb.ucsd.edu/~jmcauley/datasets/amazon/links.html.

[27] Meta. 2025. Faiss. https://github.com/facebookresearch/faiss.

[28] Non metric space library. 2025. HNSWlib. https://github.com/nmslib/hnswlib.

[29] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-throughput vector similarity search in knowledge graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.
[30] OpenMP. 2024. OpenMP home page. https://www.openmp.org/.
[31] Malte Ostendorff. 2022. arXiv paper embeddings dataset. https://huggingface.co/datasets/malteos/aspect-paper-embeddings/tree/main.
[32] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector database management techniques and systems. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile). Association for Computing Machinery, New York, NY, USA, 597–604. doi: 10.1145/3626246.3654691
[33] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and predicate-agnostic search over vector embeddings and structured data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.
[34] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient approximate nearest neighbor search in multi-dimensional databases. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.
[35] Qdrant. 2024. A complete guide to filtering in vector search. https://qdrant.tech/articles/vector-search-filtering/#conclusion-real-life-use-cases-of-filtering.
[36] Navid Rekabsaz, Oleg Lesota, Markus Schedl, Jon Brassey, and Carsten Eickhoff. 2021. TripClick: the log files of a large health web search engine. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2507–2513.
[37] Abdel Rodriguez, John Trengrove, and Joon-Pil Hwang. 2024. How we speed up filtered vector search with ACORN. https://weaviate.io/blog/speed-up-filtered-vector-search.
[38] Viktor Sanca and Anastasia Ailamaki. 2023. E-Scan: Consuming contextual data with model plugins. In *VLDB Workshops*.
[39] Bhaskarjit Sarmah, Dhagash Mehta, Benika Hall, Rohan Rao, Sunil Patel, and Stefano Pasquali. 2024. HybridRAG: Integrating knowledge graphs and vector retrieval augmented generation for efficient information extraction. In *Proceedings of the 5th ACM International Conference on AI in Finance*. 608–616.
[40] Gaurav Sehgal and Semih Salihoğlu. 2025. NaviX: A native vector index design for graph DBMSs with robust predicate-agnostic search performance. *Proc. VLDB Endow.* 18, 11 (July 2025), 4438–4450. doi: 10.14778/3749646.3749704
[41] Gaurav Sehgal and Semih Salihoğlu. 2025. Faiss-navix. https://github.com/gaurav8297/faiss-navix.
[42] Ryan Siegler. 2024. Optimizing vector search with metadata filtering and fuzzy filtering. https://medium.com/kx-systems/optimizing-vector-search-with-metadata-filtering-41276e1a7370.
[43] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. 2022. Results of the NeurIPS'21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track*. PMLR, 177–189.
[44] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6105–6114.
[45] 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 *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 171–185.
[46] Dashun Wang, Chaoming Song, and Albert-László Barabási. 2013. Quantifying long-term scientific impact. *Science* 342, 6154 (2013), 127–132. doi: 10.1126/science.1237825
[47] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 international conference on management of data*. 2614–2627.
[48] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2022. Navigable proximity graph-driven native hybrid queries with structured and unstructured constraints. *CoRR* abs/2203.13601 (2022).
[49] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2023. An efficient and robust framework for approximate nearest neighbor search with attribute constraint. *Advances in Neural Information Processing Systems* 36 (2023), 15738–15751.
[50] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. 2017. Knowledge graph embedding: A survey of approaches and applications. *IEEE transactions on knowledge and data engineering* 29, 12 (2017), 2724–2743.
[51] Weaviate. 2026. Inverted indexes. https://docs.weaviate.io/weaviate/concepts/indexing/inverted-index.
[52] Weaviate. 2026. Weaviate flat index. https://docs.weaviate.io/weaviate/concepts/vector-index/#flat-index.
[53] Weaviate. 2026. Weaviate homepage. https://weaviate.io.
[54] Weaviate. 2026. Weaviate monitoring. https://docs.weaviate.io/deploy/configuration/monitoring.

[55] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3152–3165.
[56] Wikipedia. 2025. Cosine distance. https://en.wikipedia.org/wiki/Cosine_similarity#Cosine_distance.
[57] Wikipedia. 2025. Euclidean distance. https://en.wikipedia.org/wiki/Euclidean_distance.
[58] Wei Wu, Junlin He, Yu Qiao, Guoheng Fu, Li Liu, and Jin Yu. 2022. HQANN: Efficient and robust similarity search for hybrid queries with structured and unstructured constraints. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 4580–4584.
[59] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. 2023. SPFRESH: Incremental in-place update for billion-scale vector search. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 545–561.
[60] Tiannuo Yang, Wen Hu, Wangqi Peng, Yusen Li, Jianguo Li, Gang Wang, and Xiaoguang Liu. 2024. VDTuner: Automated performance tuning for vector data management systems. In *2024 IEEE 40th International Conference on Data Engineering*. IEEE, 4357–4369.
[61] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2241–2253.
[62] Song Yu, Shengyuan Lin, Shufeng Gong, Yongqing Xie, Ruicheng Liu, Yijie Zhou, Ji Sun, Yanfeng Zhang, Guoliang Li, and Ge Yu. 2025. A topology-aware localized update strategy for graph-based ANN index. *arXiv preprint arXiv:2503.00402* (2025).
[63] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, et al. 2023. VBASE: Unifying online vector similarity search and relational queries via relaxed monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation*. 377–395.
[64] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2024. Chat2data: An interactive data analysis system with RAG, vector databases and LLMs. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4481–4484.
[65] Zhenhua Zhu, Jun Liu, Guohao Dai, Shulin Zeng, Bing Li, Huazhong Yang, and Yu Wang. 2023. Processing-in-hierarchical-memory architecture for billion-scale approximate nearest neighbor search. In *2023 60th ACM/IEEE Design Automation Conference*. IEEE, 1–6.

Received October 2025; revised January 2026; accepted February 2026