SIEVE: Effective Filtered Vector Search with Collection of Indexes

Zhaoheng Li[1], Silu Huang[2], Wei Ding[2], Yongjoo Park[1], Jianjun Chen[2]
University of Illinois Urbana-Champaign[1] Bytedance Inc.[2]
{zl20,yongjoo}@illinois.edu,{silu.huang,wei.ding,jianjun.chen}@bytedance.com

Abstract

Many real-world tasks such as recommending videos with the *kids* tag can be reduced to finding most *similar* vectors associated with *hard* predicates. This task, *filtered vector search*, is challenging as prior state-of-the-art graph-based (unfiltered) similarity search techniques quickly degenerate when hard constraints are considered. That is, effective graph-based filtered similarity search relies on sufficient connectivity for reaching the most similar items within just a few hops. To consider predicates, recent works propose modifying graph traversal to visit only the items that may satisfy predicates. However, they fail to offer the just-a-few-hops property for a wide range of predicates: they must restrict predicates significantly or lose efficiency if only a small fraction of items satisfy predicates.

We propose an opposite approach: instead of constraining traversal, we build many indexes each serving different predicate forms. For effective construction, we devise a three-dimensional analytical model capturing relationships among index size, search time, and recall, with which we follow a workload-aware approach to pack as many useful indexes as possible into a collection. At query time, the analytical model is employed yet again to discern the one that offers the fastest search at a given recall. We show superior performance and support on datasets with varying selectivities and forms: our approach achieves up to 8.06× speedup while having as low as 1% build time versus other indexes, with less than 2.15× memory of a standard HNSW graph and modest knowledge of past workloads.

1 Introduction

Finding *semantically similar* items satisfying *hard constraints* is a common task. Tired moms may search for eye-catching videos (semantic) tagged "safe-for-kids" (hard). Online shoppers may search for scary costumes (semantic) with a specific price range and ratings (hard) [72]. This task is called *filtered vector search*: we query similar[1] *vectors*—encoding semantics—associated with *hard* precicates. The problem has been increasingly studied recently [13, 14, 19, 20, 28, 36, 42, 45, 48, 58, 63, 72] as high-quality vector embeddings become available via modern machine-learning models [32, 33, 40].

Some latest works tackle filtered vector search by altering graph-based approximate nearest neighbor search (ANNS) indexes—prior state-of-the-art for *unfiltered vector search* [15, 24, 30], aiming to *constrain* their search: FilteredVamana interleaves graph traversal and filter evaluations; ACORN [42] induces a query-time subgraph by visiting only the nodes that satisfy predicates. These approaches are more effective than naïve methods like *pre-filtering*, which requires a (slow) linear scan for similarity computations. That is, graph-based methods navigate items through edges, reaching targets within just a few hops. Effective filtered vector search aims to serve filtered queries using such *compact* graphs; if the property—*small world*—falls apart, graph traversal will lose efficiency.

Unfortunately, **existing graph-based methods fail to offer the small-world property for low-selectivity predicates,** thus delivering significantly poorer performance (i.e., slow search, low recall). Moreover, we cannot simply resort to pre-filtering as its linear scan is still too costly unless the selectivity is *too low*. This selectivity band (e.g., 1%–10%)[2] is called "unhappy middle" [20]. FilteredVamana aims to mitigate the issue by linking attribute-sharing vectors to create local, dense, per-filter subgraphs, but requires restricting filter forms (e.g., categorical attribute matches) [19]. ACORN [42]'s ability to support general predicates comes at a cost: induced subgraph becomes too sparse, losing the small-world property. This degenerative behavior has been theoretically proven [4, 26, 30]. We conjecture that having a single graph is not expressive enough to handle *all* predicates, whose support[3] may overlap with one another in a complex way. We may need to employ multiple graphs, each specialized for a different set of (overlapping) predicates.

Our Goal. We aim to offer *compact* graphs for nearly all filtered queries, regardless of filter selectivity or form, by building a *collection* of indexes. A collection is more expressive than a single index. By leveraging *filter stability* observed in real-world filtered vector search workloads [36, 50], we can tailor indexes to observed past workloads to maximize expected search quality—speed and recall. R1D4 Each index can be carefully chosen to serve multiple predicates: a RI2 graph, e.g., built for `stars=1-3`, can also serve `stars=1` if it is *dense enough* for that sub-predicate. An index collection requires more memory than one index; yet, indexes are relatively small versus raw data, i.e., high-dimensional vectors. In our experiments, hundreds of (small) additional indexes took only as much memory as one base index (over the entire data), while the additional indexes boost performance significantly compared to relying on the base index alone. Overall, our proposed index collection can succeed if it offers high-quality filtered search to nearly all queries, each using a compact graph, while consuming a small amount of memory.

Challenge. Building an effective index collection is challenging due to conflicting goals. During index construction, graphs can trade recall for smaller size (Fig 1a), allowing more indexes and higher coverage. Yet, graphs must be dense enough for high recall. Likewise, we can trade search speed for higher recall at query time by over-searching in graphs (Fig 1b). This relationship must be quantified to find which index to use for a given query. These unique properties make our task significantly different from existing problems [46, 49]. For example, materialized view selection targets *exact* querying, whereas ANN is *approximate* with speed/recall trade-offs.

Our Approach. Our framework (called SIEVE[4]) builds an index collection tailored to an observed past query workload to maximize expected throughput with a memory budget (e.g., 10 GB) and specified recall (e.g., 95%). Conceptually, every candidate index is

[1]Vector similarity is commonly measured using Euclidean distance, inner product, or cosine similarity. Our work is orthogonal to the distance metric.

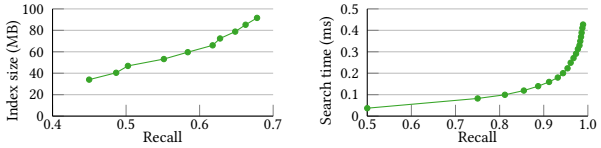
[2]This range is from our experiments; their actual values depend on datasets.

[3]The support of a predicate is a subset of data that satisfies the predicate.

[4]SIEVE is an acronym for Set of Indexes for Efficient Vector Exploration.

Table 1: Comparison between SIEVE (ours) and other indexing methods for filtered vector search.

Approach	Query Selectivity			Complex Filters	Potential Weaknesses
	high	medium	low		
Partition-Based [20, 36]	×	✓	✓	×	Restrictive filter format
Data Attr.-aware Graphs [19, 58, 63]	✓	✓	×	×	Requires limiting data attribute cardinality (<200)
Intra-Search Filtering [42]	✓	✓	×	✓	Potentially excessive TTI (up to 219× of regular HNSW)
Exhaustive Indexing [28, 72]	✓	✓	✓	×	High memory cost, restrictive filter format
Index Collection (Ours, SIEVE)	✓	✓	✓	✓	Needs bounded extra memory and past workload (up to 2.15× of HNSW)



(a) Mem. vs. recall, 100K random vecs. (b) Time vs. recall, 100k random vecs.

Figure 1: ANN vector search trade-offs: (Left) indexes can be built with varying recall/memory trade-offs. (Right) indexes can be *over-searched* to trade slower search for higher recall.

assigned *benefit*, capturing the marginal performance gain it offers when added to a collection; Adding the index incurs memory *cost*. Using the benefit and cost, we incrementally grow a collection. This iterative approach isn't new; what's new are (1) how to estimate the benefit/cost and (2) how to serve queries with the index collection.

First, we design an analytical, predicate form-agnostic benefit/cost model capturing three-dimensional relationships among index size, search time, and recall, allowing us to find the minimal (i.e., most sparse) graph satisfying a specific recall. Since each index is smaller, more indexes are allowed within a memory budget, thus accelerating search for more predicates. Our model is based on empirical observations and existing small-world network theories [4].

Second, query serving dynamically chooses the index to achieve the fastest search given a specific recall. This query-time selection is needed since our indexes may overlap, meaning a query may be served by more than one index. For optimal selection, we yet again employ the three-dimensional model to determine (1) which index to use, and (2) what values to use for search-related parameters.

Difference from Existing Work. SIEVE significantly differs from existing vector search works (Table 1). Versus works in MV selection [2, 69], partitioning [50, 65] and query rewriting [17, 18] for exact queries, SIEVE's optimizations notably considers recall, and performs theory-driven index tuning (§4.2) and dynamic, recall-aware serving (§5.2) for desired memory/speed/recall tradeoffs.

Contributions. We propose SIEVE, an indexing framework for filtered vector search (§3) with the following contributions:

- **Index Selection:** We introduce a three-dimensional cost model for evaluating the search speed, recall, and memory cost of vector search strategies, which we use to jointly perform index selection and parameterization under bounded memory. (§4)
- **Query Serving:** We utilize our derived cost model to derive a dynamic search strategy that selects the most efficient serving method and parameterization at any target recall. (§5)
- **Effective Filtered Vector Search**: We show via experimentation that SIEVE achieves up to 8.06× speedup over existing indexes with <2.15× memory of a standard HNSW and modest past workload knowledge on diverse query filter formats. (§7)

Algorithm 1: HNSW_SEARCH_LAYER

Input: *query vector* q, *enter points* ep, *exploration factor* ef, *layer* l_c
Output: ef *closest vectors to* q
Initialize visited set V, candidate set C, top-ef set W to ep;
while $|C| > 0$ **do**
  $c \leftarrow$ extract nearest vector in C to q
  $f \leftarrow$ furthest vector in W to q
  if $dist(c,q) > dist(f,q)$: **break**
  $nbrs \leftarrow neighborhood(c)$ at layer l_c //ACORN filters here
  for *each* $e \in nbrs$ **do**
    if $e \in V$: **continue**
    $V \leftarrow V \cup e$
    $f \leftarrow$ furthest vector from W to q
    if $dist(e,q) < dist(f,q)$ *or* $|W| < ef$: **then**
      $C \leftarrow C \cup e$
      $W \leftarrow W \cup e$ //hnswlib filters here
      If $|W| > ef$: remove furthest vector from W to q
Return W.

2 Motivation

SIEVE builds on HNSW[30], a performant unfiltered vector index[10, 71]. We describe HNSW (§2.1), how works have (ineffectively) extended it to filtered search (§2.2), and our ideas for building and using an HNSW index collection for effective filtered search (§2.3).

2.1 HNSW Graph

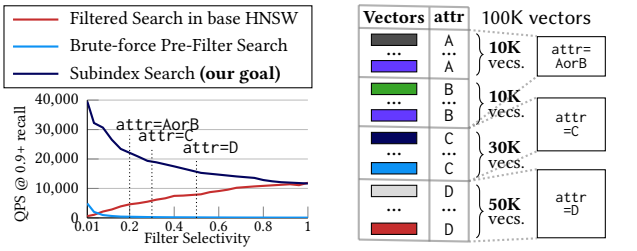
HNSW is a graph-based vector index [30] which combines the idea of *small-world graphs* [4] and skip-lists [62] to create a multi-layer graph structure for effective similarity search on vector datasets.

HNSW Graph Structure. An HNSW graph consists of multiple layered small-world graphs [4]. The topmost (entry) layer contains the fewest vectors and features long edge lengths, facilitating long-range vector space travel; the bottom (base) layer contains all vetors and features short edge lengths, representing local neighborhoods. HNSW graphs are built by incrementally inserting vectors: each vector is linked to a number of neighbors in each layer, controlled by a construction parameter M, which acts as an outdegree limit and ensures vectors connect to other similar vectors in each layer.

HNSW Graph Search. Given a query vector, HNSW graph layers are traversed from top to bottom, using the long higher-layer links to go to relevant neighborhoods containing similar data vectors, then using the short lower-layer links to find top-k choices. Alg. 1 presents the per-layer search algorithm.

2.2 Existing Filtering Methods Underperform

While the original HNSW graph proposal did not consider performing filtered vector search, a number of HNSW-based filtered search methods have been proposed, which this section will overview.



(a) QPS vs. sel. on 100K-vector dataset (b) Example Candidate Subindexes

Figure 2: SIEVE aims to build subindexes with high relative speedup and applicability to many queries (e.g., `attr=AorB`)

Post-Search Filtering. Trivially, for filtered top-k queries, the graph can be over-searched for top-k/sel vectors where sel is the filter selectivity. Then, results that don't match the filter are dropped, expecting that k out of top-k/sel results would remain on average: if not, another search with top-$2k/sel$ is performed, and so on.

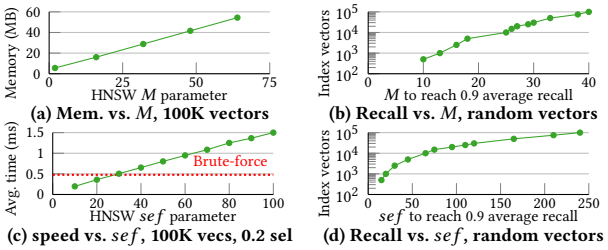
Result-Set-Filtering. hnswlib [37] passes the filtered query to the HNSW graph, where the filter is evaluated before adding candidate vectors into the top-k result set (line 13, Alg. 1). While hnswlib improves over post-search filtering by returning k satisfying results in one search round, it still has $(1-sel)$ chance that a candidate cannot be added into the top-k set due to the filter. Hence, while recall is negligibly impacted as search continues until finding all k results, search time scales inversely with selectivity (Fig 2a), effectively still falling to the 'unhappy middle' when sel is low, but there still too many points for pre-filter search (e.g., large datasets [7]). [5]

Other Filter Application Methods. ACORN [42] applies filtering at neighbor expansion (line 6, Alg. 1), effectively searching in an induced subgraph of satisfying vectors in the HNSW graph. However, as subgraph induction is equivalent to edge and node removal, the subgraph can lose small-world properties, notably connectivity [64], required for effective search if it is too sparse [38]; searching as is with Alg. 1 can result in early stops and low recall. Hence, ACORN modifies both HNSW construction and search, notably expanding into 2-hop neighbors to avoid subgraph sparsity. However, ACORN can still underperform when even the 2-hop subgraph is sparse. As we will show via experimentation (§7.2), result-set filtering sometimes outperforms ACORN and vice versa. SIEVE uses result-set-filtering in its HNSW indexes as it is applicable without specialized graph construction, which may incur excessive time-to-index (TTI, §7.3) and limit discussion to result-set-filtering in the following sections. However, SIEVE can also use ACORN's filtering instead given minor adjustments.

2.3 SIEVE's Intuition for Faster Search

Existing HNSW-based filtering methods underperform on "unhappy-middle" selectivities. SIEVE aims to mitigate this by workload-driven fitting of a HNSW (sub)index collection over data subsets in which these queries can be effectively served from their matching points being dense in the subindexes. As mentioned in §1, building and using HNSW graphs involves speed/recall/memory trade-offs (Fig 1); hence, SIEVE should decide both *which* and *how* to build and use the index collection; this section describes our intuitions.

[5]This feature is commonly implemented via assigning IDs to inserted vectors, then computing a binary ID→{0,1} mapping from the IDs of vectors that pass the filter (§6).



(a) Mem. vs. M, 100K vectors (b) Recall vs. M, random vectors
(c) speed vs. sef, 100K vecs, 0.2 sel (d) Recall vs. sef, random vectors

Figure 3: M and sef respectively increase the memory size and search time (Left), but smaller graphs require smaller M and sef values to reach the same recall (Right).

Three-Dimensional Modeling. Without loss of generality, SIEVE treats recall and memory as constraints and optimizes for speed, as users (1) often know how much (bounded) memory to use for indexing [29, 50, 51] and (2) have target query recalls (e.g., SLOs [46, 49]). This is different versus MV Selection for (exact) querying which is typically constrained only by memory; SIEVE's intuition is that with theory-driven modeling, it can first *reduce* the recall dimension by reasoning *how* an index should be built (explained shortly) to reach different target recalls. Then, SIEVE can use established selection methods to choose *which* indexes to build with bounded memory to maximize serving speed. Finally, during serving, SIEVE can determine with similar modeling *which* and *how* to use built indexes for fastest search under a (potentially new) target recall.

How to Build Indexes? Each subindex's memory size scales linearly with the (1) indexed vector count and (2) density-controlling construction parameter M (§2.1). M can be tuned for different memory/recall tradeoffs: higher M increases both memory size and recall (from increased density) and vice versa (Fig 3a). A target recall effectively dictates the *lowest M value* each index can be built with;[6] Intuitively, smaller indexes need lower M values to reach the same target recall (e.g., Fig 2's `attr=C` requires lower M to serve queries at average x recall vs. `attr=D`, Fig 3b), which we describe in §4.2.

What Indexes to Build? SIEVE aims to build subindexes that efficiently serve (observed) queries with which alternative methods (e.g., brute-force KNN) are inefficient (i.e., *marginal benefits*). Suppose we have the base HNSW index in Fig 2b: While building subindex (`attr=D`) benefits its respective filtered query, `attr=D` has high selectivity (50%) that the base index serves it *fast enough* via result-set-filtering. In comparison, subindex (`attr=AorB`) is high marginal benefit: It serves (`attr=AorB`) significantly faster than the base index (Fig 2). Subindexes can also serve non-exact matching filtered queries: For example, (`attr=AorB`) can also serve (`attr=A`) effectively, which has high-enough (50%) selectivity in the subindex. This expands utility of subindexes like (`attr=AorB`) from applicability to other filters. We describe SIEVE's index selection in §4.3.

How to Serve Queries? SIEVE decides between indexed search or brute-force KNN when serving queries with a built index collection. A key parameter controlling HNSW indexes' search speed/recall tradeoff is the search exploration factor sef (Alg. 1): higher sef (*over-searching* the graph) trades lower speed for higher recall (Fig 3c). SIEVE will need to tune sef if the serving target recall is

[6]SIEVE optimizes for target average/expected recall as to the best of our knowledge, there exists no indexing scheme that guarantees *absolute*, per-query recall, as query hardness can vary [59]. Hence, we limit discussions to average/expected recall.

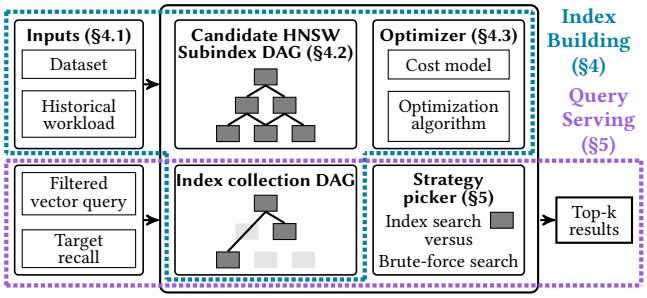


Figure 4: SIEVE framework. SIEVE fits an index collection from an observed historical workload, then serves queries strategically with the collection and other metrics.

higher than that assumed at construction; Similar to M, SIEVE aims to use the *lowest* sef for indexed searches, and smaller subindexes also require smaller sef for the same target (Fig 3d). Then, given the best found index and sef, SIEVE evaluates whether falling back to brute-force KNN is faster (e.g., $sef > 30$, Fig 3c), which also always has perfect recall. We describe SIEVE's serving strategy in §5.2.

3 SIEVE Framework Overview

SIEVE (Fig 4) effectively serves filtered vector queries by building and using an index collection. §3.1 describes SIEVE's index construction; §3.2 describes how SIEVE serves filtered vector queries.

3.1 SIEVE Construction

During construction, SIEVE aims to build a collection of the most beneficial HNSW subindexes given a memory budget and target recall based on the data distribution and a historical query workload.

Inputs. SIEVE takes as input (1) an attributed vector dataset—a set of vectors and their scalar attributes, (2) a historical query workload—a set of query filters with probability/frequency counts, (3) a target recall, and (4) a memory budget. Unlike some specialized indexes (e.g., CAPS [20], HQANN [63]), SIEVE does not restrict attribute or filter forms, only requiring filters to be evaluable on attributes, e.g., `A in attr` evaluates to `True` for `attr={A,B}`. (§4.1)

Cost Modeling. SIEVE models candidate indexes' memory size and serving speed given their construction with sufficient density/M to serve queries at the target recall (§2.3). It then accordingly sets up the candidates' unit (marginal) benefits for optimization. (§4.2)

Optimization. SIEVE selects the subindexes to build under the memory budget with greedy submodular optimization, prioritizing high-unit marginal benefit and/or high (re)use-probability subindexes in a manner akin to *Materialized View Selection* [5, 47, 69]. (§4.3)

Indexing. SIEVE builds the chosen subindexes over the dataset. SIEVE always includes the *base index* over the entire vector dataset in the collection, which acts as a fallback for queries that any other subindex in the collection cannot effectively handle. This design choice allows SIEVE to handle arbitrary (un-)filtered queries (§7).

3.2 Serving Queries with SIEVE

For serving, SIEVE aims to choose the optimal search method for filtered queries based on the subindexes in the built collection and (a potentially different from construction-time) target recall.

Table 2: Table of Symbols

Symbols	Definition
$\{\mathcal{V}, \mathcal{A}\}$	Attributed vector dataset of N vectors
$\mathcal{H} = \{(h_1, c_1)...\}$	Set of weighted observed historical query filters
B	SIEVE indexing memory budget
$card(f) \to [0, N]$	Cardinality of filter f in the dataset
M_∞	M of the root index I_∞ representing build-time target recall
$\mathcal{M}_\downarrow(I_h) \to \mathbb{Z}^+$	Subindex M downscaling function
$C(I_h, f) \to \mathbb{R}^+$	Indexed search cost for query with filter f in subindex I_h
$C_{bf}(f) \to \mathbb{R}^+$	Brute-force search cost for query with filter f over dataset
$S(I_h) \to \mathbb{R}^+$	In-memory size of subindex I_h
I_∞	Root index constructed over entire dataset
γ	Brute-force scaling constant
$cor(w, f, h)$	Query correlation of w given filter f and subindex I_h
$\mathcal{I} := \{I_{h_1}, ..., I_{h_i}\}$	Index collection of i subindexes
$C(\mathcal{I}, f) \to \mathbb{R}^+$	Cost of best possible search for query filter f given $\mathcal{I}$
sef_∞	sef of the root index I_∞ representing serving-time target recall
$\mathcal{S}_\downarrow(I_h) \to \mathbb{Z}^+$	Subindex sef downscaling function

Identifying the Optimal Indexed Search. The first, straightforward approach SIEVE uses for serving a query is with a built subindex. SIEVE finds the best subindex/sef combination for serving the query at target recall—following intuition in §2.3, preferably a small subindex in which the query is dense, using low sef. (§5.1)

Choosing Search Method. SIEVE chooses between serving the query with the best-found subindex/sef combination (with result-set-filtering if needed) or brute-force KNN. SIEVE estimates serving speed of both methods with its cost model, then chooses the faster one, analogous to *MV-aware query rewriting* [1, 54, 66]. (§5.2)

4 Optimized Index Collection Construction

This section covers how SIEVE builds its index collection. We describe preliminaries in §4.1, SIEVE's cost model and optimization problem (SIEVE-Opt) in §4.2, and solution to SIEVE-Opt in §4.3.

4.1 Preliminary and Definitions

Definition 4.1. An **Attributed Dataset** is a set of n vectors $\mathcal{V} = \{v_1, ..., v_n\}$ and a set of n attribute sets $\mathcal{A} = \{a_1, ..., a_n\}$, where each a_i is an attribute value set associated with each vector $v_i \in \mathbb{R}^d$.

Fig 5 depicts an example where each a_i is a set of strings. Notably, SIEVE's definition imposes no restriction on the form of a_i.

Definition 4.2. An **Filtered Query Workload** is pair of sets of m vectors and m filters $\mathcal{M} = \{w_1, ..., w_m\}$ and $\mathcal{F} = \{f_1, ..., f_m\}$, where f_i is the filter of query vector $w_j \in \mathbb{R}^d$ and each $f_i : \mathcal{A} \to \{0, 1\}$ is a function that maps attribute values $a_j \in \mathcal{A}$ to a binary indicator.

Each query filter f_i can be evaluated on the attributes a_j of each vector u_j: a_j *satisfies* f_i if $f_i(a_j) = 1$. For example, in Fig 5, f_1 = (`A in attrs`) (shortened to `A` for brevity) evaluates to 1 on a_1 = `{A,E}`. We define the *cardinality* of each filter f_i as the number of dataset rows that satisfy the filter, i.e., $card(f_i) = |\{a_j \in \mathcal{A} | f_i(a_j) = 1\}|$.

Definition 4.3. A **Filtered Vector Search Problem** takes an attributed dataset $(\mathcal{V}, \mathcal{A})$, a filtered query workload $(\mathcal{M}, \mathcal{F})$, and a similarity metric $\mathcal{S}$. The output is a $|\mathcal{M}| \times k$ matrix $\mathcal{R}^*$ of top-k results, where each row $R_i = \{v_{i_1}, ..., v_{i_k}\}$ is the top-k closest vectors in $\mathcal{V}$ based on $\mathcal{S}$ that satisfy filter f_i, i.e., $f_i(a_{i_l}) = 1, \forall 1 \le l \le k$.

A solution $\mathcal{R}$'s quality is commonly evaluated via (1) *recall*, measuring the average proportion of results that are actual top-k results

Attributed vec. dataset

Vecs. ($\mathcal{V}$)	Attrs. ($\mathcal{A}$)
	{A, E}
	{A, E}
	{A, E}
	{B, D}
	{C, F}
	{D, E}
	{D, E}
	{D, E}

Historical Query Filters

Filter ($\mathcal{F}$)	Count (C)
A	2
A∨B	1
A∨B∨C	3
D	2
E	3
D∧E	1

Candidate Subindex DAG

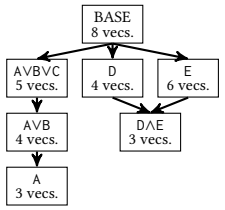


Figure 5: Example attributed vector dataset, historical workload, and SIEVE's corresponding candidate subindex DAG.

$\mathcal{R}^*$, given by $\frac{|\mathcal{R} \cap \mathcal{R}^*|}{m \cdot k}$, and (2) *latency*, referring to the average per-query result generation time, expressed as $\frac{t}{m}$ where t is the total search time and m is the query count. *Queries-per-second* (QPS, $\frac{m}{t}$) is commonly reported as latency's inverse. Effective filtered vector search can be achieved by serving queries with *(sub)indexes* (§2.3):

Definition 4.4. A **Subindex** I_{f_i} is an index constructed over a subset of data points that satisfy filter f_i, i.e., $\mathcal{V}_{f_i} := \{v_j | f_i(a_j) = 1\}$.

For example, the subindex I_{A} only indexes the first three rows in Fig 5. The *base index* indexing all rows can be expressed as $\mathcal{I}_\infty$, where ∞ is a 'dummy filter' that always evaluates to 1. Given a subindex I_{f_i}, it (1) can be used to evaluate a filtered query (w_j, f_j) with serving cost $C(I_{f_i}, w_j, f_j) \rightarrow \mathbb{R}+$, and (2) has a (in-memory) size $S(I_{f_i}) \rightarrow \mathbb{R}+$, for which we perform cost modeling in §4.2.

Definition 4.5. A **Historical Query Workload** is a query filter tally $\mathcal{H} = \{(h_1, c_1), ..., (h_l, c_l)\}$ where filter h_i has occurred c_i times.

Fig 5 depicts a workload with 6 unique filters. SIEVE assumes *filter stability* [36, 50] for anticipated future workloads: the future workload's filter distributions $\mathcal{F}$ follow those observed in $\mathcal{H}$.

R1D1 R1D2 R11 RI2

Remark. SIEVE's definitions only require all query filters to be evaluable on all dataset attributes, hence can inherently handle arbitrarily complex predicates and attributes. However, the specific predicate and attribute forms may affect certain optimization nuances of SIEVE, such as adaptability to workload shifts (§7.7.2).

4.2 SIEVE-Opt: Problem Setup

This section defines SIEVE's problem: candidate subindexes for construction and their benefits/costs (i.e., index speed/size when serving queries at target recall), then formalizes SIEVE-Opt.

Candidate Subindex DAG. There are exponentially many possible subindexes for an attributed dataset $(\mathcal{V}, \mathcal{A})$. Besides the base index, SIEVE limits its problem space by only considering subindexes corresponding to observed past filters in $\mathcal{H}$. For example, in Fig 5, I_{A} is a candidate while I_{B} is not. The YFCC dataset, with 100K historical queries, produces 23,920 candidates [7]. For optimization, SIEVE organizes candidates in a directed acyclic graph (DAG) where edges represent subsumption (e.g., $(I_{\mathsf{A}\vee\mathsf{B}}, I_{\mathsf{A}})$ in Fig 5), which enables computing of subindex unit marginal benefits (described in §4.3).[7]

Defining Target Recall. SIEVE aims to build its index collection such that queries can be served at a target recall. Without loss of generality, SIEVE takes in a base M_∞ value for *calibrating* the target recall, defined as the average recall of searching in the base index I_∞ built with $M = M_\infty$ and $sef = k$, where k is the number of results to return and the lower bound of sef (i.e., no over-searching).

[7] SIEVE's DAG contains all subsumption relationships, e.g., if A subsumes B and B subsumes C, all 3 edges $(A \rightarrow B, B \rightarrow C, A \rightarrow C)$ will exist. The DAG is drawn as a Hasse diagram [61] instead (i.e., no $A \rightarrow C$) in subsequent figures for brevity.

Table 3: SIEVE's Cost Model for Filtered HNSW Search

Operation	Cost
Brute-force search	$C_{bf}(f) = \gamma card(f)$
Indexed Search	$C(I_h, sef, w, f) = log(card(h)) \cdot sef \cdot (\frac{card(h)}{card(f)})^{cor(w,f,h)}$ if h subsumes f else ∞
Index Size	$S(I_h) = M \cdot card(h) = \frac{M_\infty log(card(h))}{log(N)} card(h)$

Indexing Parameters at Target Recall. SIEVE aims to evaluate and build candidate subindexes with sufficient parameters to serve queries at the target recall (§2.3). SIEVE can tune either M (for construction) or sef (for serving) to achieve this; however, for construction, SIEVE assumes that all subindexes will use a uniform minimum $sef = k$, and aims to build subindexes to serve queries at target recall by only tuning M (without increasing sef): this is because $sef = k$ is the lowest-recall, highest-speed search parameterization; if SIEVE's subindexes (with sufficient M) serves queries at target recall with $sef = k$, SIEVE's entire index collection can too; hence, SIEVE can then evaluate subindexes based on highest potential speedups. Versus M_∞ used to build I_∞, candidate subindexes $S(I_h)$ are evaluated and built with downscaled M values (Fig 3b):

Definition 4.6. The *Subindex M downscaling function* $\mathcal{M}_\downarrow$ takes in a subindex I_h, and returns the M value required to build I_h with to achieve at least the same average query serving recall as the base index I_∞ built with M_∞: $\mathcal{M}_\downarrow(I_h) := \frac{M_\infty log(card(h))}{log(N)}$.

SIEVE's intuition for $\mathcal{M}_\downarrow$ is that layers of the HNSW graph (§2.1) are based on Delaunay graphs [12], which requires nodes to have suitable degree ($\Theta(logN)$) for search effectiveness. As M controls node degree, each subindex I_h's M, $\mathcal{M}_\downarrow(I_h)$ should be proportional to its size's logarithm :$\mathcal{M}_\downarrow(I_h) \propto (log(card(h)))$. For example, if the root index I_∞ in Fig 5 is built with $M_\infty = 32$, subindex I_{D} would be evaluated with $M_{\mathsf{D}} = \frac{32log(4)}{log(8)} \approx 21$. Hence, for following optimization, SIEVE will evaluate the memory size (explained shortly) of each candidate subindex I_h assuming construction with downscaled $M = \mathcal{M}_\downarrow(I_h)$ and $sef = k (= 1$, for discussion).

Subindex Memory Size. Each HNSW subindex I_h has a memory size scaling linearly with indexed points $card(h)$ and M: $S(I_h) = M \cdot card(h)$ (Fig 1a). For example, I_{D} indexing 4 points built with $\mathcal{M}_\downarrow(I_h) = 21$ incurs a memory size of $21 \times 4 = 84$. Notably, due to M's effect on memory, SIEVE's M downscaling ($\mathcal{M}_\downarrow$) saves memory for smaller subindexes, potentially allowing more subindexes to be built under the same memory constraint versus a naive method that builds all subindexes with a uniform M_∞ (§7.6).

Subindex Search Cost. Without loss of generality, SIEVE define subindexes' search costs as their serving latency:

Definition 4.7. The *Indexed Search Cost Function* (with result-set-filtering, §2.2) C takes a subindex I_h, sef, and a filtered query (w, f), and returns the expected latency of using I_h with sef to serve (w, f): $C(I_h, sef, w, f) := log(card(h)) \cdot sef \cdot (\frac{card(h)}{card(f)})^{cor(w,f,h)}$.

SIEVE bases C on three observations: (1) HNSW's search time scales logarithmically [30] with graph size, (2) scales linearly with

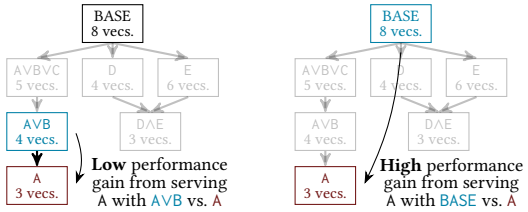


Figure 6: *Diminishing returns* during subindex selection: constructing A when A∨B exists (left) brings lower marginal benefits compared to doing so when A∨B doesn't exist (right).

sef [30], and (3) there is $\frac{card(h)}{card(f)}$ probability that a data vector similar to the query vector w passes the filter with result-set-filtering (§2.2), scaled by *query correlation*—$cor(w, f, h)$—the ratio of average distance from w to points that satisfy f in I_h versus non-satisfying points [42].[8] Positive correlation (i.e., w is similar to satisfying vectors) improves query performance ($cor(w, f, h) < 1$), mitigating low selectivity's effects as satisfying vectors are reached faster. Conversely, negative correlation amplifies low selectivity's impact and increases query cost ($cor(w, f, h) > 1$). SIEVE assumes constant correlation across all subindexes and filters, i.e., $cor(w, f, h) == c$, and for discussion, set $c = 1$ and simplify $C(I_h, sef, w, f)$ as $C(I_h, f)$ (as sef is also assumed to be fixed at 1) in this section. For example, in Fig 5, serving a query with filter A with $I_{A\vee B}$ incurs $\frac{4log(4)}{3}$ cost. SIEVE constrains for simplicity that a subindex I_h can only serve a query with filter f if h subsumes f; otherwise, $C(I_h, f) = \infty$. [9]

Brute-force Search Cost. Any query (w, f) can be served via brute-force KNN, performing distance computations between w and all vectors in $\{\mathcal{V}, \mathcal{A}\}$ that satisfy f, i.e., $\mathcal{V}_f := \{v_i | f(a_i) = 1\}$. This trivially incurs cost $C_{bf}(f) = card(f)$ linear to the cardinality.

Aligning Search Costs. The alignment between indexed and brute-force search costs is influenced by factors such as distance function implementation (e.g., varying SIMD methods[9]) and index memory access patterns [16]. Hence, SIEVE requires *aligning* the indexed and brute-force search costs by scaling C_{bf} with a constant $\gamma \in \mathbb{R}+$: SIEVE compares C with $\gamma \cdot C_{bf}$ when evaluating indexed versus brute-force search. For illustration purposes, however, we assume $\gamma = 1$. The aligned costs allow us to define the cost of the *best serving method* for a query (w, f) given an index collection $\mathcal{I}$:

Definition 4.8. The collection query serving cost function C takes in a subindex collection $\mathcal{I} := I_{h_1}, ..., I_{h_x}$ and a filtered vector query (w, f), and returns the cost of the *best possible serving strategy* given $\mathcal{I}$: $C(\mathcal{I}, f) := min(C_{bf}(f), min(\{C(I_h, f) | I_h \in \mathcal{I}\})$.

$C(\mathcal{I}, f)$ represents the lower cost of (1) brute-force KNN and (2) searching with the smallest subindex subsuming (w, f) in $\mathcal{I}$: if $\mathcal{I}$ is the entire DAG in §4.2, $C(\mathcal{I}, \mathsf{A}) = log(3)$ as it is best served by its corresponding subindex I_A, while $C(\mathcal{I}, \mathsf{F}) = 1$, as its best indexed search (with I_∞) costs $8log(8) \div 1 \approx 16.6$, more than brute-force KNN (1). With the collection serving cost $C(\mathcal{I}, f)$ and index size $S(I_h)$, SIEVE's optimization problem, SIEVE-Opt, can be defined:

Problem 1. SIEVE-Opt

Input: *(1) Attributed Vector Dataset* $\{\mathcal{V}, \mathcal{A}\}$

[8] M also potentially affects latency; however, there is no definite analytical nor empirical trend [30, 43] (§7.6), hence we omit it for simplicity.

[9] We study unconstrained cases, e.g., for multi-subindex search in appendix A.1.

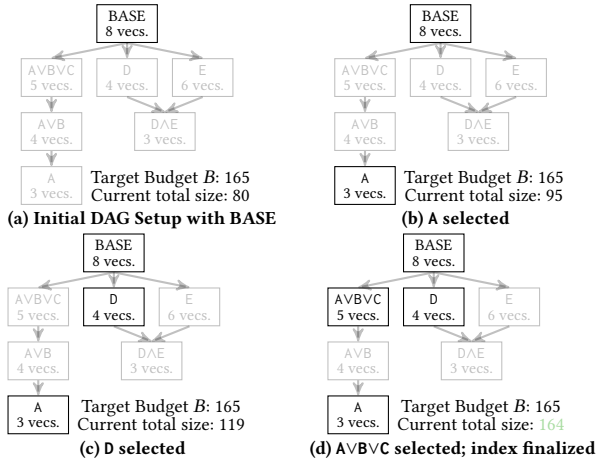


Figure 7: Solving SIEVE-Opt for inputs in Fig 5. Subindexes with the highest unit marginal benefit are iteratively added until the total index collection size reaches the budget B.

(2) Historical Workload Distribution $\mathcal{H} = \{(h_i, c_i)\}$
(3) M_∞ *representing target recall*
(4) memory budget for subindex collection B

Output: *(1) A subindex collection to construct* $\mathcal{I} := \{I_{h_{i_1}}, ..., I_{h_{i_x}}\}$

Objective function: *Minimize collection query serving cost over historical workload* $C(\mathcal{I}, \mathcal{H}) = \sum_{i=1}^{|\mathcal{H}|} c_i \cdot C(\mathcal{I}, h_i)$.

Constraints:

Base index must exist: $I_\infty \in \mathcal{I}$

Each subindex I_h *is built with sufficient* $M = \mathcal{M}_\downarrow(I_h)$ *value*

Total subindex size is less than the memory budget $\sum_{l=1}^{x} S(I_{h_{i_l}}) \leq B$

Presence of the Base Index. SIEVE enforces that I_∞ must exist for handling arbitrary (e.g., unseen) filtered queries. Worst case, SIEVE can serve queries with the better of I_∞ or brute-force KNN, which lower-bounds SIEVE's serving performance (§7.1).

4.3 SIEVE-Opt: Solution

This section presents our solution to SIEVE-Opt, whose formulation naturally gives rise to a greedy solution for subindex selection [3].

Marginal Benefits. Adding a new index I_h into $\mathcal{I}$ will only decrease the collection query serving cost: $C(\mathcal{I}, \mathcal{H}) - C(\mathcal{I} \cup \{I_h\}, \mathcal{H}) \geq 0 \; \forall \mathcal{I}, \mathcal{H}, I_h$. This is the *marginal benefit* of adding I_h into $\mathcal{I}$ w.r.t. $\mathcal{H}$. For example, in Fig 6, if $\mathcal{I} = \{I_{A\vee B}, I_\infty\}$, $\mathcal{H} = \{(\mathsf{A}, 1)\}$ (left), adding I_A into $\mathcal{I}$ incurs marginal benefit of $\frac{4log(4)}{3} - log(3) \approx 0.75$. This is less versus the right setup (4.45), where I_A is added into a collection $\mathcal{I} = \{I_\infty\}$ with only a base index—there is *diminishing returns* with adding I_A when $I_{A\vee B}$ already exists. This property is generalizable:

$$\underbrace{C(\mathcal{I} \cup \{I_h\}, \mathcal{H}) - C(\mathcal{I}, \mathcal{H})}_{\text{large marginal benefit}} \leq \underbrace{C(\mathcal{J} \cup \{I_h\}, \mathcal{H}) - C(\mathcal{J}, \mathcal{H})}_{\text{small marginal benefit}} \; \forall \mathcal{I} \subseteq \mathcal{J}$$

That is, the query serving cost $C(\mathcal{I}, \mathcal{H})$ is a *supermodular set function* w.r.t $\mathcal{I}$ [55], and SIEVE-Opt is a *supermodular minimization problem* with the knapsack memory constraint B [8]. This problem class gives rise to an empirically effective greedy algorithm—GreedyRatio [3, 34]:[10] It starts with $\mathcal{I} = \{I_\infty\}$, then iteratively

[10] While theoretically bounded solutions exist [52], their high overhead (e.g., $O(|\mathcal{H}|^5)$ computations) makes them impractical [34].

adds the highest marginal benefit/index size ratio subindex (*unit marginal benefit* $\frac{C(\mathcal{I}\cup\{I_h\},\mathcal{H})-C(\mathcal{I},\mathcal{H})}{S(I_h)}$) until reaching the constraint.

Example (Fig 7). Using Fig 5's problem setting, $M_\infty = 10$ for I_∞ and $\sum_{I_h\in\mathcal{I}} S(I_h) < 165 = B$, GreedyRatio proceeds as follows:

(1) **Step 1:** I_A is selected (top right). Its unit benefit is high (0.253): serving A with I_A is much better than via brute-force search, and I_A is space efficient, requiring only $\mathcal{M}_\downarrow(I_\mathsf{A}) = \frac{10log_{10}(3)}{log_{10}(10)} = 5$.
(2) **Step 2:** I_D is selected (bottom left). Its unit benefit is high (0.217): serving D with I_D is much better than via the root index I_∞.
(3) **Step 3:** $I_{\mathsf{A}\vee\mathsf{B}\vee\mathsf{C}}$ is selected (bottom right). While its unit marginal benefit (0.209) is decreased by the already constructed I_A, it still has high marginal benefits for serving both A∨B∨C and A∨B.

No index can be further added to $\mathcal{I}$ without exceeding B, hence, $\mathcal{I} = \{I_\infty, I_\mathsf{A}, I_{\mathsf{A}\vee\mathsf{B}\vee\mathsf{C}}, I_\mathsf{D}\}$ is the index collection that SIEVE constructs.

Analysis. GreedyRatio has time complexity $O(E+|\mathcal{H}|log(|\mathcal{H}|))$ where E is the candidate subindex DAG's edge count (Fig 5), using a priority queue for sorting unit marginal benefits and after adding each subindex, updating its parents' and children's benefits. Optimization time is negligible versus SIEVE's construction time: For example, on the YFCC dataset [7] with 6,006 candidates to optimize over, SIEVE solves SIEVE-Opt in (only) 18ms, versus the 136 seconds for building the index collection post-optimization (§7.3).

5 SIEVE: Query Serving

This section describes SIEVE's dynamic query serving strategy with the built index collection and a (potentially different from construction-time) target recall. SIEVE first finds the optimal subindex for an incoming query (§5.1), then determines the optimal search method—parameterized index search or brute-force KNN (§5.2).

5.1 Identifying the Optimal Subindex

This section describes how SIEVE efficiently finds optimal subindexes for query serving. SIEVE's cost model (§4.2) dictates that a query (w, f) is best served with the smallest subindex I_h (i.e., minimum $card(h)$) in $\mathcal{I}$ where the subindex filter h subsumes the query filter f, following uniform query correlation assumptions in §4.2.

Index Collection DAG. Similar to the candidate DAG in §4.1, SIEVE maintains a DAG over the built index collection simplified into a *Hasse diagram [61]* by applying a transitive reduction: given two subindexes $I_h, I_q \in \mathcal{I}$, a directed edge (I_h, I_q) exists only if h subsumes q, and there is no other $I_u \in \mathcal{I}$ such that h subsumes u, and u subsumes q. For example, Fig 8 (center) depicts the DAG built over the index collection from solving SIEVE-Opt in Fig 7.

Hasse Diagram Traversal. For any given filtered query (w, f), the Index Collection DAG can be efficiently traversed via running BFS starting from the root I_∞ to find the best subindex I_h: at each step, if the current subindex I_q's filter does not subsume the query filter f, none of its descendants can either. In other words, for any descendant I_p of I_q in the DAG, p cannot subsume f, allowing the entire subgraph rooted at I_q to be pruned from the search process. For example, in Fig 8, the subindex $I_{\mathsf{A}\vee\mathsf{B}\vee\mathsf{C}}$ does not subsume the query filter D∧(C∨E), hence its child I_A can be skipped, efficiently leading to the best subindex I_D for the query to be found. In practice, for the YFCC workload with 100,000 filtered queries and an index collection with 658 subindexes, finding the optimal subindex for all queries took (only) 297 ms, which is a low percentage of the total search time (e.g., minimum 20.76 seconds, Fig 9).

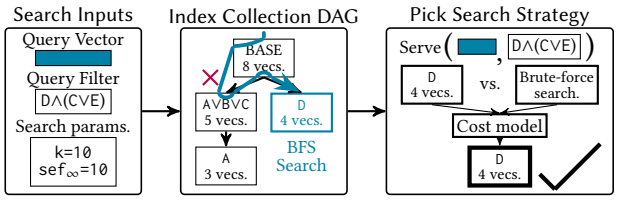


Figure 8: Choosing an optimal search strategy for a query with the constructed index collection from Fig 7. SIEVE finds the best applicable subindex, then chooses indexed or brute-force search based on estimated search costs.

5.2 Determining Optimal Search Strategy

This section outlines how SIEVE determines the serving method based on the best-found subindex. It first determines the search parameter (sef) required for the (new) target recall, then chooses between indexed search with the found sef or brute-force KNN.

Search Parameterization. Users provide a *global* sef_∞ to SIEVE (potentially different from the assumed build-time $sef = k$) for each query representing the serving-time target recall, defined as the expected recall of (over-)searching the base index I_∞ with sef_∞. Following §2.3, SIEVE aims to serve queries with sef values to match the target recall; hence, versus sef_∞, lower sef (increments) can be used when serving queries with subindexes:

Definition 5.1. The *Subindex sef downscaling function* $\mathcal{S}_\downarrow$ takes in a subindex I_h, and returns the sef value required to search I_h with to achieve at least the same average query serving recall as searching the base index I_∞ with sef_∞: $\mathcal{S}_\downarrow(I_h) := max(k, \frac{sef_\infty log(card(h))}{log(N)})$, where k is the neighbors to query (and minimum value of sef, §4.2) and assuming I_h was built with proportional $M = \mathcal{M}_\downarrow(I_h)$.

SIEVE's intuition for $\mathcal{S}_\downarrow$ is that each HNSW search visits $O(logn)$ points [30] in its hierarchical structure (§2.1): to maintain recall, the proportion between sef—the dynamic closest neighbors list size—and $logn$ must be maintained, i.e., lists must cover a consistent proportion of the visited $logn$ points, hence $\mathcal{S}_\downarrow(I_h) \propto log(card(h))$. For example, if $sef_\infty = 50$ is specified for the base index in Fig 8 as the serving-time recall, the same recall can be achieved with $\mathcal{S}_\downarrow(I_\mathsf{D}) = \frac{50\times log(4)}{log(8)} = 33$ when searching in the subindex I_D. As HNSW's search time scales linearly with sef [30] (Fig 3c), SIEVE's sef downscaling saves search time versus a static strategy that uses uniform sef_∞ for indexed searches while maintaining recall (§5.2).

Indexed vs. Brute-force Search. Given a query (w, f), its best subindex I_h in §5.1, and downscaled $sef_h = \mathcal{S}_\downarrow(I_h)$, SIEVE chooses between serving the query with indexed or brute-force KNN with its cost model from §4.2: it chooses the lower-cost method out indexed search ($C(I_h, sef_h, f)$) and brute-force search ($\gamma C_{bf}(f)$). For example, in Fig 8, assuming $k = 1$, serving D∧(C∨E) with I_D at $sef_\infty = 1$ has a lower cost $max(1, \frac{1log(4)}{log(8)}) \times \frac{4log(4)}{3} \approx 1.84$ vs. brute-force KNN (3), but at $sef_\infty = 3$, brute-force KNN is faster as the indexed search cost becomes $max(1, \frac{3log(4)}{log(8)}) \times \frac{4log(4)}{3} \approx 3.670 \geq 3$.

6 Discussion

Size of Optimization Space. One potential problem for SIEVE is an exploding optimization space when the historical workload contains many distinct filter templates. To address this, SIEVE currently prunes small-cardinality candidates with no marginal benefit over brute-force KNN prior to solving SIEVE-Opt in §4.2; if still insufficient, SIEVE can also only use top-k-common filters as candidates, which would often sufficiently approximate the full problem due to filter commonality [50]. Large optimization spaces may also affect SIEVE during workload shifts, which we study in §7.7.2. R1D2 RI2

Availability of Filter Cardinalities. SIEVE assumes availability of accurate filter cardinality info ($card(h)$). This is because many recent vector search frameworks [36, 39, 42, 57, 60] separately manage scalar attributes using methods such as inverted indexes, B-trees, or partitioning. For search, filters will first be applied on scalars to compute a *bitmap* of passing vectors' IDs (implying cardinality via nonzero count), which is then passed to the vector index. However, SIEVE is also compatible with cardinality estimation techniques [41, 67] if accurate cardinalities are unavailable.

Filter Evaluation Costs. While reported as part of total query serving time in experiments (§7), SIEVE omits modeling of filter evaluation costs from optimization (§4). This is because SIEVE currently follows the aforementioned bitmap-based filtering[11]: for each query, SIEVE computes the bitmap before choosing the serving strategy (i.e., brute-force KNN or indexed search), hence its computation cost is orthogonal to SIEVE's optimizations. Moreover, we find that bitmap computation time is negligible in our experiments; for example, on the UQV dataset, evaluating the complex, up to 10-attribute disjunction filters for 10K queries took (only) 16ms—0.2% of total query serving time at 0.95 recall (Fig 9). R2D3 RI7

Workload Shifts. We design SIEVE for production workloads, where prior works [22, 36, 50, 51] have shown query filter stability, enabling future queries (to be served with SIEVE) to be predicted from past filters. Regardless, if filter distribution shifts from $\mathcal{H}$ to a new $\mathcal{H}'$, SIEVE can be incrementally updated by re-solving SIEVE-Opt over $\mathcal{H}'$ to find a new collection $\mathcal{I}'$, build new indexes in $\mathcal{I}'-\mathcal{I}$, then delete indexes in $\mathcal{I}-\mathcal{I}'$, (which we study in §7.7). Notably, the base index I_∞, which forms a significant part of SIEVE's build time and memory size, does not need updating. Furthermore, SIEVE is robust to moderate shifts (§7.5), and even for complete shifts (serving queries from unrelated $\mathcal{H}'$ when fit on $\mathcal{H}$), SIEVE's performance will be lower bounded by SIEVE-NoExtraBudget (§7.7.2). R2C3 RI4

Multi-Subindex Search. SIEVE currently only considers serving queries with a single subindex that subsumes the query filter for indexed search (§4.2). One potential alternative is to use multiple subindexes, e.g., re-ranking results from subindexes I_p and I_q to answer $p \vee q$. This can be useful for queries that SIEVE otherwise finds no good serving strategy (e.g., those with 'unhappy middle' selectivities, but the best subindex found is the base index $\mathcal{I}_\infty$); However, finding optimal subindex sets for multi-index search is computationally hard.[12] We evaluate the feasibility and potential gains of multi-subindex search in detail in appendix A.1. R3C2 R3C4 RI6

[11] In particular, SIEVE builds inverted indexes for the set inclusion filtered-datasets and B-trees for range-filtered datasets. SIEVE currently does not partition its datasets.
[12] Equivalent to weighted set cover [68].

7 Experiments

This section studies SIEVE's performance on various real and synthetic filtered vector search workloads. We describe our experiment setup in §7.1, study end-to-end query serving performance (§7.2), index build time and size (§7.3), effect of memory budget (§7.4) and historical workload (§7.5) on index quality, our dynamic index construction and serving parameterization (§7.6), and finally, how SIEVE can adapt to cold starts and large workload shifts (§7.7). R1D3

7.1 Experiment Setup

Datasets. We use 6 experiment datasets (Table 4): YFCC [7] is from the Neurips'23 BigANN benchmark [21], Paper, SIFT are used by ACORN [42]; UQV, GIST, MSONG are used by NHQ [58]:[13]

(1) **YFCC-10M [7]** contains 10M media object embeddings. Each vector has a subset of 200K possible attributes. Dataset queries have filters of form (`A in attr`) or (`A∧B in attr`).
(2) Paper **[58]** contains 2M paper embeddings. We generate data attributes where each vector has the $i^{th}/20$ attribute with $1/i$ probability as in NHQ [58] and Milvus [57]. Query filters are generated following a zipf distribution (i.e., *filter commonality* [50]) of the form $\bigwedge_{i=1}^{k} A_i$ `in attr`, as described in HQI [36].
(3) UQV **[56]** contains 1M video embeddings. We generate data/query filters following methodology described for the Paper dataset, with $1 \le i \le 200K$ and disjunctive OR filters ($\bigcup_{i=1}^{k} A_i$ `in attr`). R1D1
(4) GIST **[6]** contains 1M scene embeddings. We generate 2 normally-distributed numerical attributes X and Y for each vector, and zipf-distributed disjunctive range filters $x_i < X < x_j \vee y_i < Y < y_j$. RI1
(5) SIFT **[53]** contains 1M image embeddings. We generate data/query filters following methodology described for the GIST dataset, with conjunctive range filters: $x_i < X < x_j \wedge y_i < Y < y_j$.
(6) MSONG **[31]** contains 1M song embeddings and 200 queries. We generate query filters uniformly of the form a_i `in attr` for 80% of queries, with the remaining 20% being unfiltered.

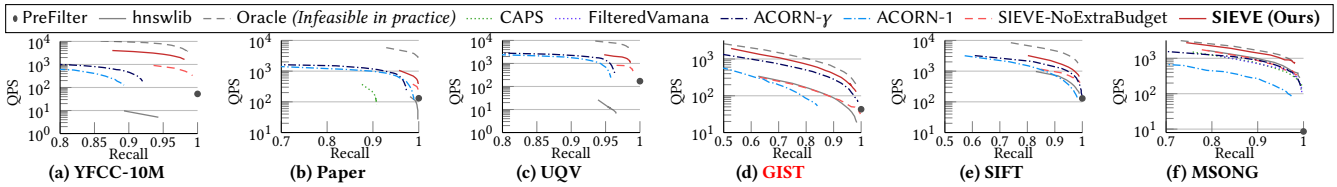
Methods. We evaluate SIEVE against these existing methods:

(1) ACORN-γ **[42]:** We use $M=32$, $M_\beta=64$, and $\gamma = max(80, 1/s_{min})$ where s_{min} is the minimum filter selectivity (e.g., 1/13 on SIFT) as suggested. For each dataset, we sweep the selectivity threshold for falling back to brute-force KNN from 0.0005 to 0.05.
(2) ACORN-1 **[42]:** Ablated ACORN-γ with $\gamma = 1$ and $M_\beta = 32$.
(3) hnswlib **[39]:** An HNSW implementation with result-set-filtering. We use the more performant out of $M = \{16, 32\}$ and $efc = 40$.
(4) SIEVE-**NoExtraBudget:** Ablated SIEVE with memory budget $B = S(I_\infty)$ that only builds the dataset-wide base HNSW index (and no subindexes). Equivalent to hnswlib that falls back to brute-force KNN based on SIEVE's seving strategy (§5.2).
(5) PreFilter**:** We first use the predicate filter, then perform brute-force KNN with hnswlib's SIMD-enabled distance function.
(6) Oracle**:** Exhaustive indexing method which ACORN-γ aims to mimic [42]; it builds a subindex for every observed filter. Oracle is expected to outperform SIEVE but incur higher TTI and memory cost; we present it as a bound for SIEVE's performance.
(7) FilteredVamana **[19]** only supports filters of form `A in attr` (or no filter); we compare against it on MSONG only. We build in-memory using DiskANNPy's recommended parameters [11].

[13] NHQ is not applicable on the filter formats we use for experimentation.

Table 4: Summary of Datasets for Evaluation.

Dataset	# Vectors	Dim	Data Type	# Queries	# Attrs.	Predicates	Pred. Type	# Unique Preds.	Avg. Selectivity
YFCC-10M [21]	10000000	192	Images	100000	200000	$\bigwedge_{i=1}^{k} A_i$ in attr, $1 \le i \le 2$	attr. match+AND	23930	0.018
Paper [58]	2029997	200	Text	10000	20	$\bigwedge_{i=1}^{k} A_i$ in attr, $2 \le i \le 5$	attr. match+AND	2500	0.019
UQV [56]	1000000	256	Video	10000	200000	$\bigcup_{i=1}^{k} A_i$ in attr, $3 \le i \le 10$	attr. match+OR	2500	0.037
GIST [6]	1000000	960	Scenes	1000	2	$x_i < X < x_j \vee y_i < y < y_j$	range filter+OR	200	0.097
Sift [53]	1000000	128	Image	10000	2	$x_i < X < x_j \wedge y_i < y < y_j$	range filter+AND	200	0.196
Msong [31]	992272	420	Audio	200	20	A in attr, No filter	attr. match	20	0.616



(a) YFCC-10M (b) Paper (c) UQV (d) GIST (e) SIFT (f) MSONG

Figure 9: SIEVE's Recall@10-QPS curves vs. baselines on various datasets. SIEVE achieves up to 8.06× QPS increase at 0.9 recall@10 vs. the next best non-Oracle method (ACORN-γ).

(8) CAPS **[20]** only supports conjunctive attribute matching on under 256 data attributes; we compare against it on Paper/MSONG only. We sweep cluster count from 10-1000 on each dataset.

For SIEVE, we use hnswlib [39] for our index collection. For each dataset, we sweep $M_\infty = \{16, 32\}$ and $efc = 40$ for the base index, with downscaled M and same efc for subindexes (§4.2). Budget B is set to **3**× hnswlib's index size on the same dataset.[14] The brute-force scaling constant γ is empirically chosen where $\gamma \cdot c_{bf}(f) = c(I_h, f)$ when $card(f) = card(h) = 1000$, i.e., brute-force KNN and perfect-selectivity indexed search cost the same for a 1K-cardinality filtered query with $sef = k$ (§4.2). The query correlation factor $q(w, f, h)$ is set to 0.5 (i.e., average positive correlation, §4.2) for all w, f, h.

R2D2 RI5 ***Fitting and Workload.*** For each dataset, we use the first 25% slice of queries (unless otherwise stated, e.g., in §7.5 and §7.7.1) as SIEVE's observed workload $\mathcal{H}$ for fitting, then serve all queries (including the slice used for fitting) with the built index, following methodology in prior workload-aware indexing works [36].

Measurement. For Oracle, hnswlib, SIEVE-NoExtraBudget, we generate QPS-recall@10 curves with $sef \in [10, 110]$ in steps of 10. For SIEVE, we use $sef_\infty \in [10, 110]$ for the base index and downscale sef for subindexes (§5.2). For ACORN-γ, ACORN-1, we vary $sef \in [10, 510]$ in steps of 50. For CAPS, we vary $np \in [3K, 30K]$ in steps of 3K. For FilteredVamana, we vary $L \in [10, 510]$ in steps of 50.

Environment. Experiments are run on an Ubuntu server with 2 AMD EPYC 7552 48-core Processors and 1TB RAM. We store datasets on local disk, build indexes in-memory with 96 threads, and run queries sequentially with 1 thread in Neurips'23 BigANN challenge's environment [21], which reports best-of-5 QPS. Our Github repository [70] contains our SIEVE implementation, dataset attribute, query filter generation, and experiment scripts.

7.2 High and Generalized Search Performance

This section studies SIEVE's overall filtered vector search performance vs. existing baselines: we run each method on applicable datasets and compare their generated QPS-recall@10 curves.

Fig 9 reports results. SIEVE performs best out of all non-Oracle approaches on all 5 datasets; versus ACORN-γ, the existing state-of-the-art, SIEVE achieves up to 8.06× speedup (YFCC) at 0.9 recall. Notably, SIEVE also achieves higher recalls (>0.99 in SIEVE vs. peaking at ~0.95 in ACORN-γ) on the low-selectivity datasets (YFCC, Paper, UQV): while ACORN-γ's induced subgraphs fail to mimic HNSW graphs for low selectivity filters (§2.2), SIEVE actually builds the (sub)indexes in which the filters are dense for effective search.

Bounded Performance. SIEVE-NoExtraBudget is the *lower bound* of SIEVE's performance (§5.2) in cases where the best subindex SIEVE finds for any query is the base index I_∞ (e.g., a complete workload shift, §7.7.2), and can only decide between searching with I_∞ or brute-force KNN with its cost model and target recall. While the performance gap between SIEVE and SIEVE-NoExtraBudget is significant (4.01× QPS on YFCC at 0.95 recall), the latter is still effective in its own right—outperforming ACORN-γ on YFCC, and MSONG while underperforming on Paper, UQV, GIST, and SIFT.

High Generalizability. SIEVE's filter and attribute format-agnostic formulation (§4.1) allows it to handle (1) any number of data attributes and (2) a wide range of predicate forms—conjunctions on YFCC and Paper, disjunctions on UQV, range filters on GIST and SIFT, and unfiltered queries on MSONG, unlike FilteredVamana and CAPS, which struggle with high-cardinality data attribute sets, disjunctions, and range queries. In addition, SIEVE still significantly outperforms CAPS on Paper (10.61× QPS @ 0.9 recall) and both CAPS and FilteredVamana on MSONG (2.29× QPS @ 0.9 recall).

Handles Diverse Selectivities. We additionally study SIEVE's per-query selectivity band performance in appendix A.2.

7.3 Low Construction Overhead

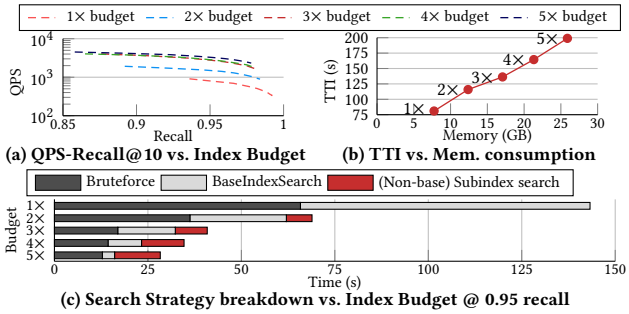
This section studies SIEVE's construction overhead. We fix B as 3× hnswlib's index size on the same dataset (§7.1), and compare SIEVE's TTI and memory consumption vs. existing methods.

Table 5 reports results. SIEVE effectively adheres to its memory budget as seen in its memory size versus hnswlib: Notably, while SIEVE's uses 3× hnswlib's *index size* as budget, its actual memory consumption including datasets is less than 3×, being as low as 1.29× on GIST as the high-dimensional (960) vectors contribute significantly to memory size (3.84GB). The TTI increase is also less than 3× (1.68× on YFCC to 2.78× on GIST), as subindexes's build time scales *superlinearly* with vector count [30], hence larger indexes (e.g., base index) take more indexing time per vector. Versus

[14] Excluding dataset size, hence total size (dataset included) is less than 3× of hnswlib.

Table 5: SIEVE's TTI (seconds) and memory consumption (GB) vs. baselines. Numbers from indexing configurations used to generate QPS-recall@10 curves in Fig 9. N/A indicates a method not applicable to that dataset.

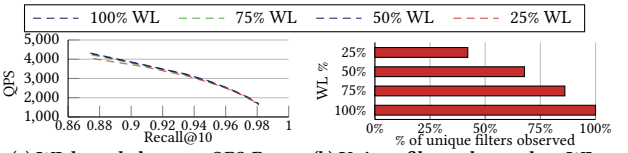
Index	hnswlib		Oracle		CAPS		FilteredVamana		ACORN-γ		ACORN-1		**SIEVE (Ours)**	
Dataset	TTI	Mem.	TTI	Mem.	TTI	Mem.	TTI	Mem.	TTI	Mem.	TTI	Mem.	TTI	Mem.
YFCC	80.968	7.759	479.849	84.275	N/A	N/A	N/A	N/A	17782.532	11.445	116.378	6.266	136.132	17.066
Paper	30.435	2.874	1811.820	76.014	8.893	4.419	N/A	N/A	2529.372	3.485	25.061	2.570	59.581	5.260
UQV	10.140	1.797	1588.513	68.573	N/A	N/A	N/A	N/A	1179.420	2.090	13.565	1.635	19.312	3.194
GIST	74.828	4.317	1477.429	14.204	N/A	N/A	N/A	N/A	5791.505	4.479	50.247	4.104	208.393	5.552
SIFT	10.335	0.943	539.409	19.520	N/A	N/A	N/A	N/A	265.534	1.142	10.133	0.965	25.377	2.241
MSONG	24.467	2.068	427.15	9.226	22.501	3.383	323.639	2.384	638.60	2.222	27.707	2.017	63.857	3.271



(a) QPS-Recall@10 vs. Index Budget (b) TTI vs. Mem. consumption

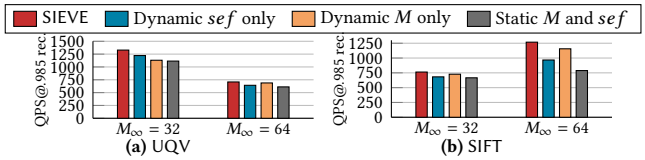
(c) Search Strategy breakdown vs. Index Budget @ 0.95 recall

Figure 10: SIEVE's budget vs. QPS-recall curves; and search strategy breakdowns (YFCC). It prioritizes subindexes that can bring higher benefits to queries (§4.2).



(a) WL knowledge % vs. QPS-Rec. (b) Unique filters observed vs. WL %

Figure 11: SIEVE's QPS-recall vs. workload knowledge (YFCC). After seeing only 42% of unique filters in a 25% workload slice, it achieves ~96% QPS of a collection fit on the full workload.



(a) UQV (b) SIFT

Figure 12: SIEVE's recall-aware index construction and serving parameterization achieves up to 1.60× QPS increase at high recalls (~0.985) vs. static, non-recall-aware ablations.

Table 6: SIEVE's recall-aware index tuning allows for building more indexes under the same memory constraint.

Dataset ($M_\infty = 32$)	UQV		SIFT	
Dynamic index (M) tuning	**Yes**	No	**Yes**	No
# Indexed vectors	**2286197**	2005930	**2298746**	2074841
# Subindexes	**200**	169	**22**	16

Table 7: SIEVE's recall-aware search parameterization increases search efficiency at high (0.985) recalls.

Dataset ($M_\infty = 32$)	UQV		SIFT	
Dynamic search (sef) parameterization	**Yes**	No	Yes	No
Avg. HNSW distance computations	**5917**	6111	**6448**	6665
Avg. brute-force distance computations	**1136**	1195	**583**	629

ACORN-γ, while SIEVE uses more memory (up to 1.96×, SIFT), its TTI is a fraction of ACORN-γ's (0.8% on YFCC to 9.5% on SIFT) as it avoids ACORN-γ's specialized graph building (§2.2). This makes SIEVE desirable when TTI is the main constraint instead of memory (e.g. on-disk indexing). Versus Oracle, which has potentially prohibitive size (84.2GB, YFCC), SIEVE performs competitively (Fig 9) using as little as 3.0% and 4.6% of Oracle's TTI and memory (UQV).

7.4 Efficient Usage of Memory Budget

This section studies SIEVE's performance vs. its indexing budget B, which we vary from 1× size of hnswlib (equivalent to SIEVE-NoExtraBudget) to 5× and report the QPS-Recall@10 curves, resource consumption, and search strategy breakdowns on YFCC.

Fig 10 reports results. SIEVE's QPS-Recall@10 trade-off expectedly increases with more budget. However, contrasting the linear increase in TTI and memory, the performance increase is diminishing: each 1× budget brings less QPS-recall benefits—the total serving time decrease for the 100K queries at 0.95 recall from 1× to 2× is 2.63×, but only 1.22× from 4× to 5×. This is because SIEVE prioritizes adding high-benefit subindexes to its collection (§4.3, also verified in appendix A.3), which is reflected in its search strategy breakdown in Fig 10c: the first budget increase from 1× to 2× focuses on building subindexes for queries with highest gains from being served by a subindex versus brute-force or base index search, decreasing the spent time of the 2 methods by 25.5s and 51.6s, respectively. Further budget increases yield smaller reductions in brute-force (<19.3s) or base index search (<10.4s) time.

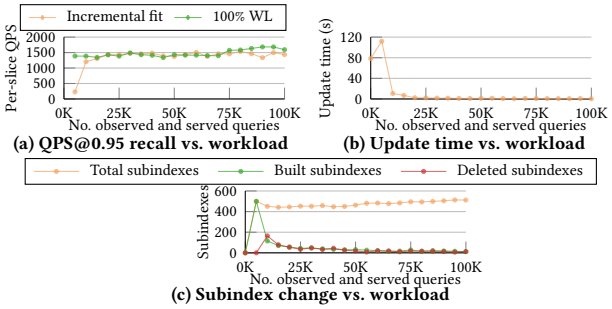
7.5 Effective Fitting from Historical Workload

This section studies the impact of discrepancies between the historical workload $\mathcal{H}$ used to build SIEVE and the actual workload. We vary the query slice size we use as the historical workload for SIEVE from 25% (our default for other experiments) to 100% on YFCC, and report the QPS-recall@10 of each fitted index collection.

Fig 11 reports results. SIEVE fit with 25% workload performs comparably (96% QPS at 0.9 recall) versus theoretically optimized SIEVE fit with 100% workload despite (1) the 25% slice only containing 42% of unique filter templates and (2) the two fits being significantly different—170/711 indexes in SIEVE (25% WL) are absent in SIEVE (100% WL) and 141/682 vice versa, showcasing SIEVE's robustness to moderate workload shifts (we study larger shifts in §7.7.2). At intermediate values, while each 25% slice increases SIEVE's choices from seeing more unique filters, performance increase is negligible.

7.6 Dynamic Construction & Serving

This section studies the effectiveness of SIEVE's dynamic, recall-aware index construction (§4.2) and serving parameterization (§5.2). We compare SIEVE's QPS-recall@10 curves with ablated versions of



(a) QPS@0.95 recall vs. workload (b) Update time vs. workload (c) Subindex change vs. workload

Figure 13: Cold starting with SIEVE on YFCC: SIEVE can be initialized with only the base index, then incrementally updated during serving; SIEVE's performance quickly approaches an optimal fit after it observes and serves 3 5K workload slices.

SIEVE—using static $M = M_\infty$ for all subindexes and/or static $sef = sef_\infty$ for all searches—on UQV and Paper with $M_\infty = \{32, 64\}$.

We report the results in Fig 12. At high (0.985) recall, SIEVE achieves up to 1.60× QPS increase versus SIEVE with no optimization and up to 1.09× QPS increase with only one of dynamic subindex construction (M) or query serving (sef) (UQV, $M_\infty = 64$). Interestingly, while the $M_\infty = 64$ setting is more performant than $M_\infty = 32$ on SIFT, the opposite holds for UQV; we hypothesize that this is due to intrinsic hardness difference of the datasets, i.e., $M_\infty = 32$ suffices for the base index in UQV,[15] and further increasing M_∞ results in increased latency with negligible recall gains.

More Indexes Under Same Memory Constraint. SIEVE's recall-aware subindex construction downscales the M parameter of smaller indexes in the collection (§4.2): this decreases the memory consumption of these small indexes (Fig 3b), which results in more built indexes under the same memory constraint (Table 6).

Fewer Distance Computations. SIEVE's dynamic search parameterization downscales sef when searching with smaller indexes (§5.2). This increases smaller indexes's search efficiency from incurring fewer distance computations (Fig 3c), and reduces searches SIEVE falls back to serving via brute-force KNN on (Table 7).

7.7 Handling Cold Starts and Workload Shifts

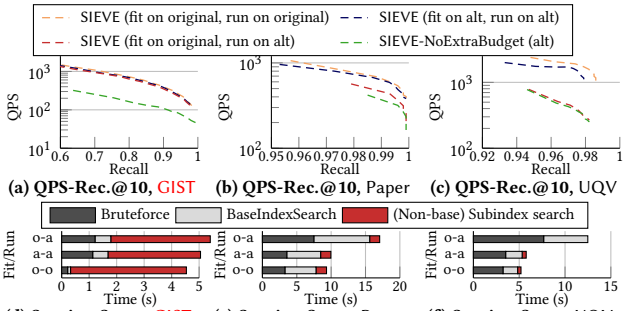
This section studies SIEVE's robustness to cold starting with no historical workload (§7.7.1) and sudden workload shifts (§7.7.2).

R2D1 R3C2 R2C3 RI4

7.7.1 SIEVE *can Effectively Cold Start.* We choose the YFCC dataset, temporally slice the 100K queries into 20 5K workload slices, then sequentially serve slices to SIEVE initialized with no historical workload (i.e., $\mathcal{H} = \emptyset$) and an index collection with only the base index $\mathcal{I}_\infty$. Each slice H', after serving, is added to the historical workload (i.e., $\mathcal{H} \leftarrow \mathcal{H} \cup H'$) and SIEVE's index collection is incrementally updated following procedures described in §6. We study per-slice query performance and SIEVE's update time between slices.

We report results in Fig 13. As observed in Fig 13a, while SIEVE's per-slice QPS is significantly lower than the theoretically optimized SIEVE (100% WL) (§7.5) on the first 2 slices—SIEVE only has the base index for the first and an (suboptimal) index collection fit to the first 5K queries on the second, SIEVE effectively cold starts as it observes more slices, approaching 97% QPS of 100% WL by the 3rd slice. This is also seen in SIEVE's update time (Fig 13b) and number of built/deleted subindexes per update (Fig 13c): while SIEVE's first update takes significant time (111s)—it uses all its budget to build 499 subindexes fit to the first slice on top of the lone base index, subsequent updates become increasingly faster due to SIEVE's observed workload $\mathcal{H}$ quickly approaching the true workload distribution: it builds and deletes fewer subindexes per update, with update time becoming sub-second after the 15th slice.

[15] Recall that M_∞ is user-specified (§4.2).



(a) QPS-Rec.@10, GIST (b) QPS-Rec.@10, Paper (c) QPS-Rec.@10, UQV

(d) Serving Strat., GIST (e) Serving Strat., Paper (f) Serving Strat., UQV

Dataset	Update time (s) (% of init. time)	Subindex # change	New subindexes size (GB) (% of original size)
GIST	149.325 (66.7%)	15→14 (-15, +14)	1.264 (22.7%)
Paper	47.648 (79.9%)	76→90 (-75, +89)	2.298 (43.6%)
UQV	15.110 (78.2%)	200→163 (-200, +163)	1.397 (43.7%)

(g) Updating SIEVE's fitted index from original to alternative workload

Figure 14: SIEVE can degrade when serving a new workload when fit on a prior workload (top); yet, it still potentially finds optimization opportunities (middle). SIEVE can be re-fit to the new workload, which is faster than the initial index building as the base index remains unchanged (bottom).

7.7.2 SIEVE *can Adapt to Complete Workload Shifts.* We choose the GIST, Paper and UQV datasets, on which we generate alternative query workloads with different filter templates[16] that follow similar distribution characteristics (i.e., zipf and average selectivity, §7.1). We study the query performance of serving queries from the alternative workload on SIEVE fit from the alternative workload versus SIEVE fit on the original workload (i.e., to simulate a complete workload shift), and overhead for re-fitting SIEVE's index collection from the original to the alternative workload.

We report results in Fig 14. As expected, serving a workload with SIEVE that significantly differs from the workload that SIEVE was fit on expectedly causes degradation, achieving only 92%, 71% and 49% QPS of an index collection fit with the corresponding workload on GIST, Paper and UQV, respectively (Fig 14a, Fig 14b, Fig 14c).

R1D2 RI2

Quantifying Degradation. While query filter templates in the original and alternative workloads almost completely differ for all datasets—the optimal index collections of the workloads share only 1 subindex on Paper and 0 on GIST and UQV (Fig 14g), the degree of degradation depends on the *sparsity* of the filter space: two random filters on the GIST dataset are most likely to have a subsumption relation, followed by UQV, and finally, Paper, as GIST only has 2 attributes to apply range filters on versus Paper and UQV's 20 and 200K for attribute matching, respectively. Hence, SIEVE still finds significant opportunities for query serving with subindexes despite

[16] We accomplish this via setting alternative seeds for randomized generation.

the workload shift on GIST (Fig 14d) to achieve a 2.77× speedup versus SIEVE-NoExtraBudget, finds fewer opportunities on Paper (Fig 14e) with 1.35× speedup, and finds almost no opportunities on UQV (Fig 14f) and degrades to only 1.03× speedup. Degradation can also occur if the predicates themselves are sparse by complexity (e.g., `A < 5 AND B in attr AND C like \w+`). Hence, a current limitation of SIEVE (and other general workload-driven approaches [36, 50, 51]) is that sparse predicate spaces (e.g., UQV) are inherently more difficult for SIEVE to robustly optimize for w.r.t. workload shifts. However, SIEVE can be updated upon detecting such degradation/shifts either by incremental adaptation as in §7.7.1 or completely refitting to the new workload—notably, even if no subindexes are kept on refit (Fig 14g), refitting is still faster than complete rebuild as the base index does not need updating (§6).

7.8 Experimentation Summary

This section summarizes our experiment findings. We accordingly claim the following following our experimental evaluation on SIEVE:

R1D1 **End-to-end Comparison vs Other Indexes**

RI1
(1) *Effective and generalizable search:* SIEVE handles arbitrary data attribute and query filter formats, achieving up to 8.06× higher QPS at 0.9 recall@10 versus the next best alternative on low and high selectivity query workloads alike (§7.2).
(2) *Low construction overhead:* SIEVE operates within its memory budget, requiring only up to 2.15× memory of hnswlib and just 1% of time-to-index (TTI) versus ACORN-γ [42] (§7.3).

Deeper Performance Analysis of SIEVE (Ours)

(1) *Effective usage of memory budget:* SIEVE achieves large performance gains even with small budget (e.g., 2× of hnswlib) as its modeling effectively prioritizes high-benefit subindexes (§7.4).
(2) *Effective fitting from historical workload:* SIEVE's construction requires only modest knowledge of the workload distribution—an index collection fit from a 25% workload slice performs within 96% of a collection fit on the true distribution (§7.5).
(3) *Effective recall-aware construction and serving:* SIEVE's recall-aware index (M) tuning and dynamic serving (sef) achieves up to 1.19× higher QPS at 0.985 recall versus ablated, recall-agnostic SIEVE versions with static parameterization (§7.6).

R2D1 R3C2 R2C3 RI4
(4) *Handling cold starts and workload shifts*: We show that SIEVE can handle cold start scenarios with no workload knowledge and complete workload shifts via incremental refitting (§7.7).

8 Related Work

Filtered Vector Search Indexing. There exists a variety of filtered vector search indexing works [13, 14, 19, 20, 28, 36, 42, 45, 58, 63, 72]. Filtered-Diskann [19] proposes the Vamana-based [24] FilteredVamana and StitchedVamana, which only apply to single-attribute filters.[17] CAPS [20] and HQI [36] partitions data based on data attributes/vector centroids to maximize query-time partition skipping; the latter is also workload-aware. However, CAPS only applies to low-cardinality conjunctions, while HQI is for a significantly different batched query setting. NHQ [58] and HQANN [63] are graph-based indexes that jointly index vectors and attributes, but support only soft filters. SeRF [72] and IVF2 [28] are specialized indexes for single-attribute range queries [18] and the YFCC dataset,[19] respectively. ACORN [42] supports filtered vector search on arbitrary predicates via subgraph traversal. In comparison, SIEVE supports arbitrary predicates similar to ACORN, but achieves higher QPS/recall with a workload-aware index collection (§4.2) magnitudes faster to build (§7.3). Versus SeRF, which also conceptually builds multiple overlapping subindexes (exhaustively over the attribute space then compresses them in SeRF's case), SIEVE provides control over index size with its budget (§4.1), while SeRF has a data-dependent, super-linear size which is worst case quadratic w.r.t. data size.[20] Additionally, SIEVE still outperforms other specialized methods (e.g., CAPS) that only support limited predicates (§7.2). RI3

Vector Search Plan Selection. Cost-based query plan selection has been studied in vector indexing systems [57, 60]. Milvus [57] derives a cost model to choose between partitioned and pre-filter search. AnalyticDB-V [60]'s cost model incorporates query selectivity. In comparison, SIEVE's modeling dynamically picks the best strategy *considering recall*, unlike these systems (and ACORN's proposal of resorting to brute-force KNN at low selectivity [42]) which treat indexes as already tuned to serve with sufficient recall, and uses recall-agnostic models/thresholds at serving time. SIEVE's dynamic serving *bounds* its performance: it will always be faster than the best of brute-force/indexed search at any given recall (§7.2).

Materialized View Selection. There is extensive work on selecting Materialized Views (MVs) for speeding up (exact) queries [23, 25, 27, 29, 35, 44], which typically assume availability of a historical workload that accurately predicts the future workload (*filter stability* [36, 50, 51]). One common issue addressed in these works is the large candidate MVs optimization space; BigSubs [25] mitigates this via randomized approximation, while SparkCruise [44] performs optimization on a reduced problem space according to subsumptions. While SIEVE shares similarities these works such as fitting from historical workload (§4.1) and using a greedy approximation algorithm (§4.3), it orthogonally derives a method to incorporate recall (§4.2), a key and unique dimension present in vector indexing for filtered search but lacking in MV selection for (exact) queries.

9 Conclusion

This paper presents SIEVE, an indexing framework enabling efficient filtered vector search via an index collection. SIEVE proposes a three-dimensional cost model for memory size, search speed, and recall, which it uses to accurately determine benefits and costs of individual indexes at a target recall to build an effective set of indexes with bounded memory via workload-driven optimization. During query serving, SIEVE determines the fastest search strategy with the index collection at a potentially new target recall—whether to use a parameterized index search, or fallback to brute-force KNN. SIEVE achieves up to 8.06× QPS increase over existing indexing methods at 0.9 recall on a variety of datasets while requiring as little as 1% TTI versus other specialized indexes.

[17] The paper claims that both indexes also support disjunctions, but this functionality is absent from the open-source implementation SIEVE uses for experimentation.

[18] Hence, we omit SeRF from our experiments as it is not applicable.

[19] IVF2 was tuned specifically for YFCC, handling only filtered queries of form `a (AND b) in attr`. We omit it from experimentation due to its lack of generality.

[20] The SeRF paper claims that index size and build time are both $O(nlogn)$ under uniform attribute distribution assumptions, where n is the dataset size.

References

[1] [n. d.]. Query Rewrite With A Nested Materialized View. https://patents.google.com/patent/US20090228432A1/en.

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. 2000. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, Vol. 2000. 496–505.

[3] Georgios Amanatidis, Federico Fusco, Philip Lazos, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Rebecca Reiffenhäuser. 2021. Submodular maximization subject to a knapsack constraint: Combinatorial algorithms with near-optimal adaptive complexity. In *International Conference on Machine Learning*. PMLR, 231–242.

[4] Luıs A Nunes Amaral, Antonio Scala, Marc Barthelemy, and H Eugene Stanley. 2000. Classes of small-world networks. *Proceedings of the national academy of sciences* 97, 21 (2000), 11149–11152.

[5] Kamel Aouiche, Pierre-Emmanuel Jouve, and Jérôme Darmont. 2006. Clustering-based materialized view selection in data warehouses. In *East European conference on advances in databases and information systems*. Springer, 81–95.

[6] Antonio Torralba Aude Oliva. [n. d.]. Modeling the shape of the scene: a holistic representation of the spatial envelope. http://people.csail.mit.edu/torralba/code/spatialenvelope/.

[7] big-ann benchmarks. 2024. YFCC10M - Neurips23 BigANN Challenge Filter Track. https://github.com/harsha-simhadri/big-ann-benchmarks/blob/main/benchmark/datasets.py.

[8] Chandra Chekuri. [n. d.]. Combinatorial Optimization. https://courses.grainger.illinois.edu/cs586/sp2022/main.pdf.

[9] CloudFlare. [n. d.]. Computing Euclidean distance on 144 dimensions. https://blog.cloudflare.com/computing-euclidean-distance-on-144-dimensions/.

[10] cmuparlay. 2024. Benchmarking nearest neighbors.

[11] DiskANN. [n. d.]. DiskAnnPy - API. https://microsoft.github.io/DiskANN/docs/python/latest/diskannpy.html.

[12] David P Dobkin, Steven J Friedman, and Kenneth J Supowit. 1990. Delaunay graphs are almost as good as complete graphs. *Discrete & Computational Geometry* 5 (1990), 399–407.

[13] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).

[14] Joshua Engels, Ben Landrum, Shangdi Yu, Laxman Dhulipala, and Julian Shun. [n. d.]. Approximate Nearest Neighbor Search with Window Filters. In *Forty-first International Conference on Machine Learning*.

[15] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143* (2017).

[16] 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.

[17] Parke Godfrey, Jarek Gryz, Andrzej Hoppe, Wenbin Ma, and Calisto Zuzarte. 2009. Query rewrites with views for XML in DB2. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 1339–1350.

[18] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing queries using materialized views: a practical, scalable solution. *SIGMOD Rec.* 30, 2 (May 2001), 331–342. https://doi.org/10.1145/376284.375706

[19] 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.

[20] Gaurav Gupta, Jonah Yi, Benjamin Coleman, Chen Luo, Vihan Lakshman, and Anshumali Shrivastava. 2023. CAPS: A Practical Partition Index for Filtered Similarity Search. *arXiv preprint arXiv:2308.15014* (2023).

[21] harsha simhadri. 2024. Big ANN Benchmarks. https://github.com/harsha-simhadri/big-ann-benchmarks.

[22] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database Cracking.. In *CIDR*, Vol. 7. 68–78.

[23] Milena G Ivanova, Martin L Kersten, Niels J Nes, and Romulo AP Gonçalves. 2010. An architecture for recycling intermediates in a column-store. *ACM Transactions on Database Systems (TODS)* 35, 4 (2010), 1–43.

[24] 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).

[25] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 800–812.

[26] G Kashyap and G Ambika. 2019. Link deletion in directed complex networks. *Physica A: Statistical Mechanics and its Applications* 514 (2019), 631–643.

[27] Asterios Katsifodimos, Ioana Manolescu, and Vasilis Vassalos. 2012. Materialized view selection for XQuery workloads. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 565–576. https://doi.org/10.1145/2213836.2213900

[28] Ben Landrum. 2024. IVF2 Index: Fusing Classic and Spatial Inverted Indices for Fast Filtered ANNS. https://big-ann-benchmarks.com/neurips23_slides/IVF_2_filter_Ben.pdf.

[29] Zhaoheng Li, Xinyu Pi, and Yongjoo Park. 2023. S/C: Speeding up Data Materialization with Bounded Memory. In *2023 IEEE 39th international conference on data engineering (ICDE)*. IEEE, 1981–1994.

[30] 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.

[31] Information Management and Preservation. 2024. Million Song Dataset Benchmarks. https://www.ifs.tuwien.ac.at/mir/msd/.

[32] Meta. [n. d.]. Llama Models. https://www.llama.com/.

[33] Microsoft. [n. d.]. Introducing Phi-3: Redefining what's possible with SLMs. https://azure.microsoft.com/en-us/blog/introducing-phi-3-redefining-whats-possible-with-slms/.

[34] Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, and Amin Karbasi. 2016. Fast constrained submodular maximization: Personalized data summarization. In *International Conference on Machine Learning*. PMLR, 1358–1367.

[35] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. 2001. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data* (Santa Barbara, California, USA) *(SIGMOD '01)*. Association for Computing Machinery, New York, NY, USA, 307–318. https://doi.org/10.1145/375663.375703

[36] 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.

[37] Bilegsaikhan Naidan, Leonid Boytsov, Yury Malkov, and David Novak. 2015. Non-metric space library manual. *arXiv preprint arXiv:1508.05470* (2015).

[38] Mark EJ Newman, Cristopher Moore, and Duncan J Watts. 2000. Mean-field solution of the small-world network model. *Physical Review Letters* 84, 14 (2000), 3201.

[39] nmslib. 2024. Hnswlib - fast approximate nearest neighbor search. https://github.com/nmslib/hnswlib.

[40] OpenAI. [n. d.]. OpenAI. https://openai.com/.

[41] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. Quicksel: Quick selectivity learning with mixture models. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1017–1033.

[42] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proc. ACM Manag. Data* 2, 3, Article 120 (May 2024), 27 pages. https://doi.org/10.1145/3654923

[43] Pinecone. [n. d.]. Hierarchical Navigable Small Worlds (HNSW). https://www.pinecone.io/learn/series/faiss/hnsw/.

[44] Abhishek Roy, Alekh Jindal, Hiren Patel, Ashit Gosalia, Subru Krishnan, and Carlo Curino. 2019. Sparkcruise: Handsfree computation reuse in spark. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1850–1853.

[45] Viktor Sanca and Anastasia Ailamaki. 2024. Efficient Data Access Paths for Mixed Vector-Relational Search. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*. 1–9.

[46] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-time content recommendations at Twitter. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1281–1292.

[47] Amit Shukla, Prasad Deshpande, Jeffrey F Naughton, et al. 1998. Materialized view selection for multidimensional datasets. In *VLDB*, Vol. 98. 488–499.

[48] Harsha Vardhan Simhadri, Martin Aumüller, Amir Ingber, Matthijs Douze, George Williams, Magdalen Dobson Manohar, Dmitry Baranchuk, Edo Liberty, Frank Liu, Ben Landrum, et al. 2024. Results of the Big ANN: NeurIPS'23 competition. *arXiv preprint arXiv:2409.17424* (2024).

[49] Justin JongSu Song, Wookey Lee, and Jafar Afshar. 2019. An effective high recall retrieval method. *Data & Knowledge Engineering* 123 (2019), 101603.

[50] Liwen Sun, Michael J Franklin, Sanjay Krishnan, and Reynold S Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1115–1126.

[51] Liwen Sun, Michael J Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented partitioning for columnar layouts. *Proceedings of the VLDB Endowment* 10, 4 (2016), 421–432.

[52] Maxim Sviridenko. 2004. A note on maximizing a submodular set function subject to a knapsack constraint. *Operations Research Letters* 32, 1 (2004), 41–43.

[53] Texmex. 2024. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/.

[54] Dimitri Theodoratos and Wugang Xu. 2004. Constructing search spaces for materialized view selection. In *Proceedings of the 7th ACM international workshop on Data warehousing and OLAP*. 112–121.

[55] Donald M Topkis. 1998. *Supermodularity and complementarity.* Princeton university press.
[56] UQV. 2024. UQV. http://staff.itee.uq.edu.au/shenht/UQVIDEO/.
[57] 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.
[58] 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. *arXiv preprint arXiv:2203.13601* (2022).
[59] Zeyu Wang, Qitong Wang, Xiaoxing Cheng, Peng Wang, Themis Palpanas, and Wei Wang. 2024. Steiner-hardness: A query hardness measure for graph-based ann indexes. *Proceedings of the VLDB Endowment* 17, 13 (2024), 4668–4682.
[60] 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.
[61] Wikipedia. [n. d.]. Hasse Diagram - Wikipedia. https://en.wikipedia.org/wiki/Hasse_diagram.
[62] Wikipedia. 2024. Skip List Data Structure - Wikipedia. https://en.wikipedia.org/wiki/Skip_list.
[63] 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.
[64] Wentao Xiao, Yueyang Zhan, Rui Xi, Mengshu Hou, and Jianming Liao. 2024. Enhancing HNSW Index for Real-Time Updates: Addressing Unreachable Points and Performance Degradation. *arXiv preprint arXiv:2407.07871* (2024).
[65] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 193–208. https://doi.org/10.1145/3318464.3389770
[66] Zhengyu Yang, Danlin Jia, Stratis Ioannidis, Ningfang Mi, and Bo Sheng. 2018. Intermediate data caching optimization for multi-stage and parallel big data frameworks. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 277–284.
[67] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *arXiv preprint arXiv:1905.04278* (2019).
[68] Neal E Young. 2008. Greedy set-cover algorithms (1974-1979, chvátal, johnson, lovász, stein). *Encyclopedia of algorithms* (2008), 379–381.
[69] Chuan Zhang, Xin Yao, and Jian Yang. 2001. An evolutionary approach to materialized views selection in a data warehouse environment. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 31, 3 (2001), 282–294.
[70] Li Zhaoheng, Huang Silu, Ding Wei, Park Yongjoo, and Chen Jianjun. 2025. SIEVE - Github. https://github.com/BillyZhaohengLi/SIEVE-vldb25/.
[71] Zilliz. 2024. Faiss vs. HNSWlib: Choosing the Right Vector Search Tool for Your Application.
[72] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

A Appendix

A.1 Multi-index Search

R3C2 R3C4 RI6

This section studies the feasibility of serving filtered queries with multiple subindexes which when unioned, cover the query filter, as discussed in §6. While SIEVE's current serving strategy considers only (single) subindex search vs. brute-force KNN (§5), it can be extended to multi-subindex search—given a filtered query w, f and index collection $\mathcal{I}$, SIEVE aims to choose between these options:

(1) Single-index search: $argmin_{I_h \in \mathcal{I}} C(I_h, sel_{\downarrow}(I_h), w, f)$
(2) Brute-force KNN: $C_{bf}(f) = card(f)$
(3) Multi-index search: $argmin_{I' \subseteq \mathcal{I}} \sum_{I_h \in I'} C(I_h, sel_{\downarrow}(I_h), w, f)$ such that $f \subseteq \bigcup h_{I_h \in I'}$[21]

[21] We omit re-ranking time as we find it to be negligible (e.g., contributes to ~ 0.1% of total query time when re-ranking results from up to 10 subindexes).

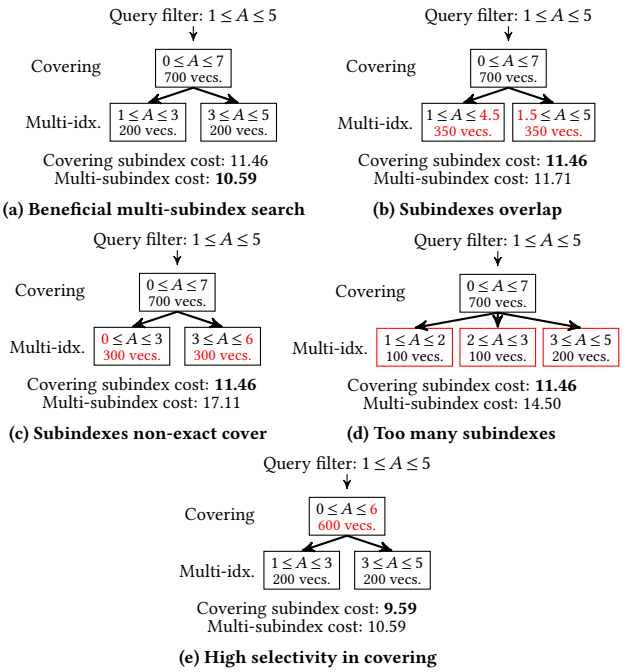


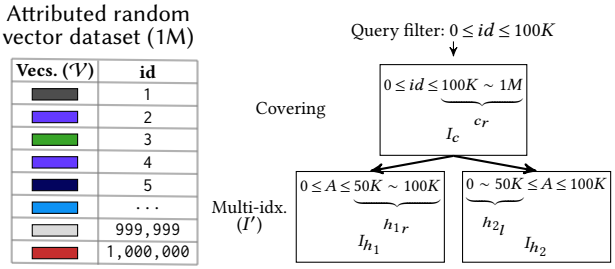
Figure 15: Multi-index search can be beneficial when the query predicate can be disjointly and exactly covered by few subindexes and have low selectivity in the smallest single covering subindex (top-left). Subindexes in the cover overlapping (top-right), covering non-passing vectors (middle-left), there being too many subindexes in the cover (middle-right), and the query having high selectivity in a single covering subindex (bottom) can all affect potential gains of multi-index search (empirically studied in Fig 16).

Where I' is the subset of indexes for multi-index search. As (only) the union of subindexes needs to cover the query filter, the constraint that each individual subindex I_h must cover f in the cost function C is lifted (i.e., as opposed to the original definition in §4.2). Another notable difference is that to evaluate $C(I_h, f)$ for each I_h in I', SIEVE must estimate the *conditional selectivity* of f in I_h (e.g., $sel(1 \le A \le 5 | 0 \le A \le 3)$, discussed in more detail shortly).

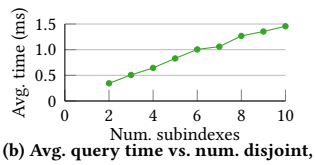
Potential Benefits. Based on SIEVE's cost model (§4.2), multi-index search can be beneficial when the query filter f is almost disjointly and exactly covered by a few subindexes $I' \subseteq \mathcal{I}$, that is:

(1) $\forall I_h, I_k \in I', |h \cap k|$ is minimized.
(2) $|\bigcup_{h, I_h \in I'} - f|$ is minimized.
(3) $|I'|$ is minimized.

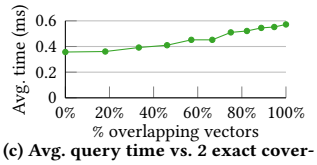
This is because (1) we want to avoid duplicately searching satisfying vectors in covering subindexes, (2) maximize conditional selectivity of the query filter in covering subindexes, and (3) as HNSW graphs's search time is logarithmic (i.e., sub-linear) w.r.t. vector count, we aim to use few, large indexes as opposed many, small indexes. The query must also be difficult to serve with alternative methods, i.e., it having low selectivity in the smallest single subindex that covers it (potentially the base index $\mathcal{I}_\infty$), but still having high enough cardinality such that brute-force KNN is also expensive.



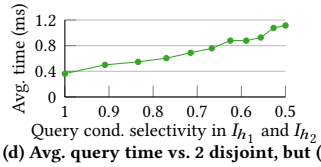
(a) Experiment setup; I_c covers the query filter; I_{h_1} and I_{h_2} jointly cover the query filter. Parameter sweeps are performed on c_r, h_{1_r}, h_{2_l}, and number of indexes in I', e.g., $|I'| = 4$ would contain 4 subindexes each with 25K vectors.



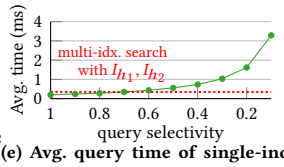
(b) Avg. query time vs. num. disjoint, exact covering subindexes.



(c) Avg. query time vs. 2 exact covering, but overlapping subindexes.



(d) Avg. query time vs. 2 disjoint, but non-exact covering subindexes.

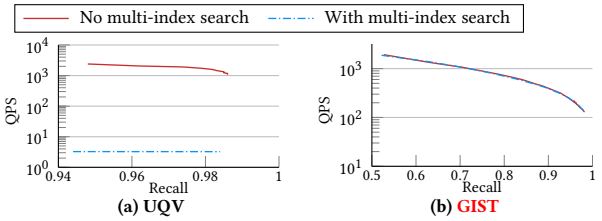


(e) Avg. query time of single-index search vs. query selectivity.

Figure 16: Quantitative evaluation of multi-subindex search benefits. All average query times reported are for serving at 0.9 recall. Plots are generated by adjusting $|I'|$ (top-left), adjusting h_{1_r} and h_{2_l} (top-right), adding random non-matching points (i.e., $100K < id < 1M$) into I_{h_1} and I_{h_2} (bottom-left), and adjusting c_r (bottom-right) without altering the query.

Motivating Example (Fig 15). Given the query filter $1 \le A \le 5$, a near-optimal scenario for multi-index serving is using the two subindexes $1 \le A \le 3$ and $3 \le A \le 5$ which exactly and disjointly cover $1 \le A \le 5$. At the same time, the smallest (single) subindex that covers $1 \le A \le 5$ is $1 \le A \le 7$, for which a (moderately selective) filtered search would have higher cost than the multi-index search following our cost model in §4.2 (Fig 15a). However, if the two covering subindexes overlap (Fig 15b), contain non-satisfying vectors (Fig 15c), or if we instead had to use three exactly covering and disjoint subindexes (Fig 15d), the cost of the best multi-index search would become higher than that of single index search. At the same time, the existence of a smaller single index that covers the query filter with high selectivity (e.g., $1 \le A \le 6$, Fig 15e) can also potentially render multi-index search (relatively) sub-optimal.

Quantitative Evaluation (Fig 16). We evaluate the scenarios discussed in Fig 15 on a test dataset with 1M 16-dimension random vectors and a query that matches 100K vectors (Fig 16a). As hypothesized in Fig 15 according to SIEVE's cost model, adjusting each factor—increasing subindex count (Fig 16b), increasing overlap between subindexes (Fig 16c), and decreasing (conditional) selectivity of the query in the subindexes (Fig 16d) all increase the latency of multi-index search. We also observe in Fig 16e that multi-index search with 2 disjoint, exactly covering subindexes (I_{h_1} and I_{h_2}) is only beneficial if the query's selectivity in the (single) covering subindex I_c is less than 0.7. Furthermore, if the query's selectivity in I_c is 1 (i.e., $c_r = 100K$ and I_c exactly matches the filter), serving with I_c becomes more optimal versus any possible multi-index search.[22]



(a) UQV **(b) GIST**

Dataset	Bruteforce #	Non-base subidx. #	base #	**multi-idx. #**	**multi-idx. opt. time %**
UQV	4721	3383	1896	0	99.56%
GIST	152	844	2	4	0.13%

(c) Search strategy and multi-index opt. overhead breakdowns @ 0.98 recall

Figure 17: SIEVE's Recall@10-QPS curves vs. ablated SIEVE with multi-index search enabled on UQV and GIST (top). Opportunities for multi-index search are rare, while finding multi-index covers can also incur high overheads (bottom).

Difficulty of Multi-Index Search. Notably, the problem of finding the best union of subindexes I' for a multi-subindex search is NP-hard (equivalent to weighted set cover where each candidate subindex I_h is weighted by query serving cost $C(I_h, f)$), necessitating efficient greedy algorithms in cases where SIEVE has non trivially-sized index collections [68]. Furthermore, SIEVE must also make repeated evaluations of query filter f's conditional selectivity in candidate subindexes I_h (e.g., the aforementioned $sel(1 \le A \le 5 | 0 \le A \le 3)$) for cost estimation. Versus cases where I_h subsumes f (i.e., for single-index search) where f's selectivity in I_h can be trivially computed as $\frac{card(f)}{card(h)}$, conditional selectivities can potentially involve more expensive, data dependent computations.

Testing Multi-Index Search . We study the potential benefits versus costs of multi-index search in more detail by augmenting SIEVE to allow it to choose multi-index search for query serving (when more optimal versus both single-index search and brute-force KNN) following multi-index covers found with the greedy algorithm for weighted set cover [68]. We evaluate the augmented SIEVE's performance versus SIEVE with no such augmentation on the disjunction-filtered datasets UQV and GIST.[23]

We report results in Fig 17. The query time between SIEVE with and without multi-index search enabled is negligible on GIST at <1% difference at 0.98 recall on GIST (Fig 17b): this is because due to the various limiting factors discussed in Fig 15e, we find that multi-index is rarely the optimal choice versus single-index search and brute-force KNN, being optimal for only 4/1000 queries (Fig 17c), all of which union only 2 subindexes. Additionally, while the overhead for finding near-optimal multi-index search strategies is negligible on GIST, it incurs prohibitive overhead on UQV, reducing the QPS @ 0.98 recall by more than 200× (Fig 17a); this is because the greedy algorithm for weighted set cover has time complexity $O(mn)$ [68]

[22] According to empirical results and SIEVE's cost model; we omit the proof for brevity.
[23] YFCC, Paper, SIFT, and MSONG contain (almost) no opportunities for multi-index search as their filters are conjunction-based: using attribute matching as an example, to cover `A`, one must union `A and B`, `A and C`,..., i.e., (many) subindexes with conjunctions between A and all other possible attributes in the attribute/filter space.

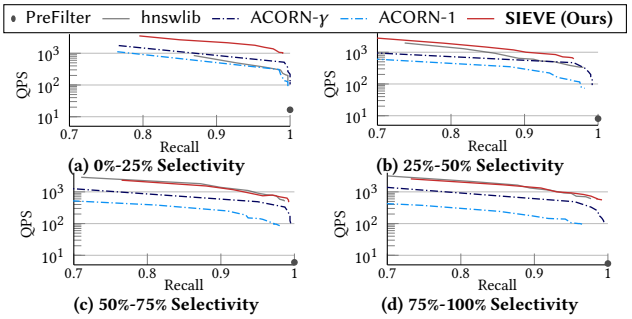


Figure 18: SIEVE's Recall@10-QPS curves vs. baselines on various selectivity bands on MSONG.

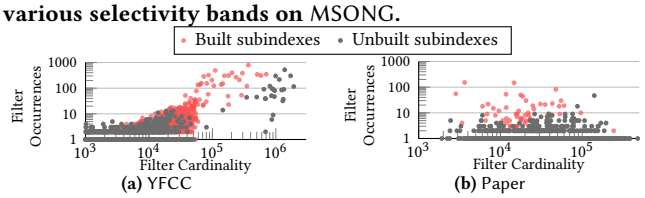


Figure 19: SIEVE's candidate vs. built subindexes according to their observed historical filter properties at 3× budget.

where m is the candidate count (i.e, cardinality of SIEVE's index collection $|\mathcal{I}|$, which is 14 on GIST and 200 on UQV) and n is the size of the attribute/filter space, of which UQV's is significantly larger than GIST's (Table 4). Hence, while multi-index search may bring potential benefits on workloads with sparse attribute/filter spaces such as GIST and can be a valuable extension for SIEVE, we defer exploration of low-overhead implementations of this technique and its extensions (e.g., additionally considering multi-index search for cost modeling during construction time, §4.2) to future work.

A.2 SIEVE's Performance vs. Selectivity Band

Fig 18 reports SIEVE's performance vs. query selectivity bands on MSONG. As theorized in §2.3, SIEVE's building of subindexes for "unhappy-middle" queries (verified in Fig 19) provide large performance gains—2.00× and 4.48× speedup vs. ACORN-γ and hnswlib at 0.99 recall—on the lowest band (Fig 18a). SIEVE also deprioritizes optimizing for high-selectivity queries as it has limited budget; it simply routes them to the base index, performing the same as hnswlib (Fig 18c, Fig 18d). Interestingly, ACORN-γ's neighbor expansion is *detrimental* at high selectivity, incurring unnecessary overhead; SIEVE (and hnswlib) outperforms it by 2.36× at 0.99 recall on the highest band (Fig 18d).

A.3 Distribution of SIEVE's Built Subindexes

Fig 19 presents the distribution of SIEVE's built vs. candidate (i.e., not built) subindexes for the YFCC and Paper datasets at 3× budget: following SIEVE's intuition in §2.3 and cost model in §4.2, SIEVE prioritizes subindexes with medium (*unhappy middle*) selectivity filters and/or high historical occurrences: Serving applicable queries with smaller subindexes bring limited benefits versus with brute-force KNN, while larger (non-base) subindexes provide only marginal improvements over serving with the base index.