Curator: Efficient Vector Search with Low-Selectivity Filters

YICHENG JIN, Duke University, USA
YONGJI WU, University of California, Berkeley[1], USA
WENJUN HU, Yale University, USA
BRUCE M. MAGGS, Duke University, USA
JUN YANG, Duke University, USA
XIAO ZHANG, Cisco ThousandEyes[1], USA
DANYANG ZHUO, Duke University, USA

Embedding-based dense retrieval has become the cornerstone of many critical applications, where approximate nearest neighbor search (ANNS) queries are often combined with filters on labels such as dates and price ranges. Graph-based indexes achieve state-of-the-art performance on unfiltered ANNS but encounter connectivity breakdown on low-selectivity filtered queries, where qualifying vectors become sparse and the graph structure among them fragments. Recent research proposes specialized graph indexes that address this issue by expanding graph degree, which incurs prohibitively high construction costs. Given these inherent limitations of graph-based methods, we argue for a dual-index architecture and present Curator, a partition-based index that complements existing graph-based approaches for low-selectivity filtered ANNS. Curator builds specialized indexes for different labels within a shared clustering tree, where each index adapts to the distribution of its qualifying vectors to ensure efficient search while sharing structure to minimize memory overhead. The system also supports incremental updates and handles arbitrary complex predicates beyond single-label filters by efficiently constructing temporary indexes on the fly. Our evaluation demonstrates that integrating Curator with state-of-the-art graph indexes reduces low-selectivity query latency by up to 20.9× compared to pre-filtering fallback, while increasing construction time and memory footprint by only 5.5% and 4.3%, respectively.

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

Additional Key Words and Phrases: vector search, approximate nearest neighbor, filtered search, indexing

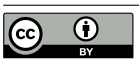
ACM Reference Format:
Yicheng Jin, Yongji Wu, Wenjun Hu, Bruce M. Maggs, Jun Yang, Xiao Zhang, and Danyang Zhuo. 2026. Curator: Efficient Vector Search with Low-Selectivity Filters. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 21 (February 2026), 27 pages. https://doi.org/10.1145/3786635

1 Introduction

The rise of deep learning has transformed information retrieval by enabling the representation of unstructured data as high-dimensional vectors that capture semantic relationships [22, 28, 36]. Approximate nearest neighbor search (ANNS) over these vector embeddings has become a

[1]Yongji Wu and Xiao Zhang were with Duke University at the time this work was conducted.

Authors' Contact Information: Yicheng Jin, Duke University, Durham, NC, USA, yicheng.jin@duke.edu; Yongji Wu, University of California, Berkeley[1], Berkeley, CA, USA, wuyongji317@gmail.com; Wenjun Hu, Yale University, New Haven, CT, USA, wenjun.hu@cantab.net; Bruce M. Maggs, Duke University, Durham, NC, USA, bmm@cs.duke.edu; Jun Yang, Duke University, Durham, NC, USA, junyang@cs.duke.edu; Xiao Zhang, Cisco ThousandEyes[1], Durham, NC, USA, xz234@alumni.duke.edu; Danyang Zhuo, Duke University, Durham, NC, USA, danyang@cs.duke.edu.



ACM 2836-6573/2026/2-ART21
https://doi.org/10.1145/3786635

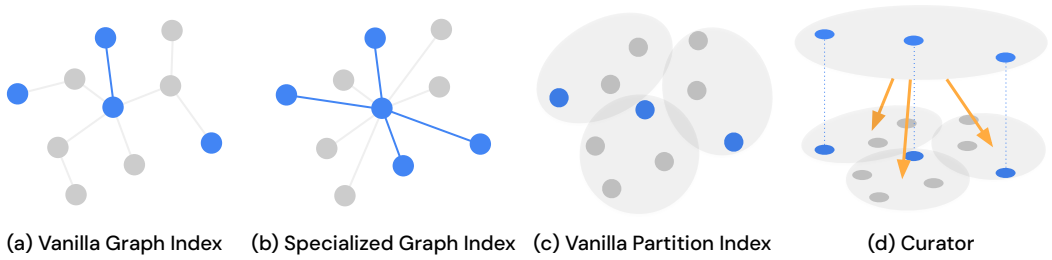


Fig. 1. Comparison of different approaches to filtered ANNS. Blue points represent qualifying vectors, gray points represent non-qualifying vectors. (a) Vanilla graph-based approach suffers from connectivity breakdown at low selectivity. (b) Specialized graph-based approach requires increasing graph degree significantly to maintain connectivity. (c) Vanilla partition-based approach suffers from granularity mismatch at low selectivity. (d) Curator adopts a hierarchical partitioning approach that buffers qualifying vectors in sparse regions at higher levels of the tree, adapting clustering granularity to filter selectivity.

fundamental primitive powering modern applications from web search [30, 50] to recommendation systems [8, 29, 53] and retrieval-augmented generation [23, 32].

Pure semantic similarity, however, often proves insufficient for real-world applications. Modern systems increasingly require *filtered ANNS*, where vector similarity search is constrained by structured predicates over metadata attributes. For instance, e-commerce platforms must find semantically similar products within specific price ranges and categories, while enterprise search systems need to respect user access permissions and document types. This combination of semantic similarity and structured filtering has driven significant interest from both industry [34, 35, 42, 44, 45] and research communities [5, 12, 33, 43, 46, 52, 55, 56].

Among the various challenges in filtered ANNS, handling queries with diverse *selectivity*—the fraction of vectors qualifying the filter conditions—is particularly difficult. The difficulty varies dramatically across the selectivity spectrum, with each regime requiring different algorithmic strategies and presenting distinct performance bottlenecks.

1.1 Why Existing Approaches Fail at Low Selectivity

Existing approaches to filtered ANNS can be broadly categorized by their underlying index structures: flat indexes (brute-force search), graph-based indexes, and partition-based indexes. While each index type has found success in specific scenarios, they all encounter limitations when handling low-selectivity queries. Fig. 1 illustrates these challenges across different approaches.

Graph connectivity breakdown. As shown in Fig. 1(a), filtered search on graph-based indexes exposes a critical trade-off: as selectivity decreases, the sub-graph of qualifying vectors becomes increasingly sparse and disconnected. This connectivity breakdown forces algorithms to choose between two unsatisfactory options: visit all neighbors of each node, which is expensive since most neighbors are unqualified, or skip unqualified neighbors entirely, which is both incomplete—as some qualifying vectors become unreachable—and inefficient, since graph search requires sufficient connectivity for optimal performance.

Densification becomes prohibitive. Recent approaches attempt to address connectivity issues through graph densification, as illustrated in Fig. 1(b). Both ACORN [33] and Filtered DiskANN [12] build dense graphs to ensure connectivity among qualifying vectors. However, the required graph degree grows rapidly as selectivity decreases. If each node needs m qualified neighbors and selectivity is s, then the total degree must be approximately m/s. This relationship makes densification

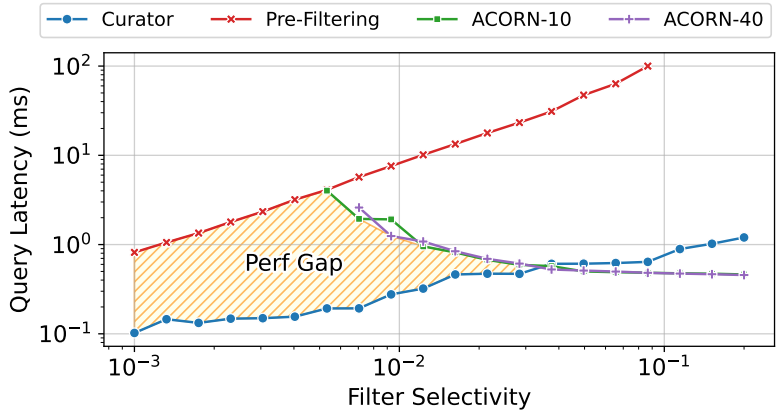


Fig. 2. Performance comparison between Curator and the state-of-the-art solution (ACORN + pre-filtering) across varied selectivity levels in the low-selectivity regime. ACORN-γ ($\gamma = 10, 40$) denotes ACORN with different graph degrees, with larger γ indicating denser index graphs. Curator closes the performance gap of ACORN at low selectivity with minimal overhead. Detailed experiment setup is provided in §6.4.

prohibitively expensive for low-selectivity queries, as our empirical evaluation demonstrates significant increases in construction time and memory usage. Alternative approaches like UNG [5] attempt different strategies but face significant limitations, as detailed in §2.

Pre-filtering scales poorly. When densification becomes impractical, many systems fall back to pre-filtering: first identifying all qualifying vectors, then performing brute-force search over this subset. While this approach guarantees completeness, it scales poorly with the absolute number of qualifying vectors. On large-scale datasets, low-selectivity filters can still match tens of thousands of vectors, making brute-force search computationally expensive.

Partition granularity mismatch. Given these limitations of graph-based approaches, one might consider partition-based alternatives. Indexes like IVF (Inverted File) cluster vectors and reduce search scope by examining only nearby clusters, and they are not vulnerable to connectivity breakdown. However, naively applying them incurs another issue: partitions built for all indexed vectors are too fine-grained for filtered search. As shown in Fig. 1(c), when selectivity is low, each cluster contains only a few qualifying vectors, thus the search must visit significantly more clusters compared to unfiltered search, leading to substantial overhead for distance computations against cluster centroids.

1.2 Our Approach: Adaptive Hierarchical Partitioning

The limitations across existing approaches suggest the need for a specialized index dedicated to low-selectivity filters. This raises a natural question: should we design a dedicated index or improve existing specialized indexes to handle a wider range of selectivity? Given the fundamental connectivity challenges that graph-based indexes face at low selectivity, we design Curator to complement existing graph-based approaches through a dual-index architecture rather than attempting to extend their performance into the low-selectivity regime. Graph-based methods excel at high-selectivity scenarios, while Curator specializes in the low-selectivity regime where traditional approaches struggle. This complementary design enables the two indexes to work together, achieving better overall performance across the entire selectivity spectrum. The key requirement for this approach to be cost-effective is that Curator must maintain low memory and construction overhead.

Following this complementary design philosophy, this paper introduces Curator, a partition-based index that employs adaptive hierarchical clustering to achieve efficient low-selectivity search. As a partition-based approach, Curator deliberately accepts slower search performance than graph-based indexes at high selectivity, where graph connectivity remains strong, in exchange for robust performance at low selectivity where graph-based approaches suffer from connectivity breakdown. As illustrated in Fig. 1(d), Curator uses hierarchical partitioning where qualifying vectors in sparse regions are buffered at higher levels of the tree, adapting clustering granularity to the local density of qualifying vectors. The system constructs specialized indexes for different filter predicates, all embedded within a shared hierarchical clustering tree that maintains low overhead through structural sharing.

Curator supports both label-based filtering—where vectors are associated with categorical labels known in advance and queries return the nearest vectors matching a given label—and arbitrary complex predicates involving multiple attribute types and logical operators. The core challenge with complex predicates is that they are typically not indexed directly due to the vast number of possible combinations. Prior works resort to inefficient multi-stage filtering or inflexible special handling of a small subset of predicate types in index design. Curator addresses this by dynamically constructing temporary indexes that mirror the shared tree structure with minimal overhead, enabling efficient evaluation of complex predicates without requiring pre-built specialized indexes for every combination.

To validate this complementary design, we evaluate Curator against ACORN (the state-of-the-art graph-based index) using a semi-synthetic dataset constructed from YFCC-10M with 20 logarithmically distributed selectivity levels in $[0.001, 0.2]$, enabling systematic coverage of the low-selectivity regime. As shown in Fig. 2, brute-force search (pre-filtering fallback) exhibits linear scaling, while ACORN shows sharp performance degradation at low selectivity. Curator fills this gap with minimal construction and memory overhead. The results show that increasing graph degree (from ACORN-10 to ACORN-40) offers limited performance improvement at low selectivity, confirming that densifying graphs is not an effective solution and making Curator an ideal complementary approach.

1.3 Contributions

This paper makes the following key contributions:

- **Adaptive partitioning for low-selectivity ANNS**: We design Curator, a hierarchical partition-based index that adapts clustering granularity to the distribution of qualified vectors, achieving high performance for low-selectivity queries while maintaining minimal memory overhead.
- **Efficient complex predicate support**: We present a novel algorithm for handling arbitrary predicates by dynamically constructing temporary indexes with minimal overhead.
- **Comprehensive system design**: We develop efficient algorithms including batch construction, incremental updates, and specialized search procedures for both single-label filters and complex predicates, ensuring search efficiency and completeness while adapting to dynamic changes in data distributions.
- **Extensive experimental validation**: We conduct comprehensive experiments on real-world datasets demonstrating that Curator reduces low-selectivity query latency by up to 20.9× compared to the state-of-the-art solution (ACORN + pre-filtering), while increasing construction time and memory footprint by only 5.5% and 4.3%, respectively, confirming the effectiveness of our complementary approach.

2 Background

2.1 Problem Formulation

We formulate the problem of filtered ANNS as follows. We start with single-label search and then generalize to complex predicate search:

Single-label search. Let $\mathcal{L} = \{l_1, l_2, \ldots, l_{|\mathcal{L}|}\}$ represent a finite universe of *labels*, and consider a dataset $S = \{(x, L_x) \mid L_x \subseteq \mathcal{L}\}$ where each object consists of a vector x and an associated set of labels L_x. Single-label search involves queries that check for the existence of a single label in the label set: a query (x_q, l, k) requests the k nearest neighbors of the query vector x_q within the subset of vectors where $l \in L_x$.

Complex predicate search. Consider a more general dataset $S = \{(x, A_x)\}$ where each object consists of a vector x and associated attributes A_x including categorical labels, numerical values, timestamps, and other structured data. Complex predicate search involves arbitrary filter predicates σ combining range queries, composite conditions across multiple attributes, and complex logical operators. A query (x_q, σ, k) requests the k nearest neighbors of the query vector x_q within the subset $\mathcal{P}_\sigma$ of vectors qualifying predicate σ. Complex predicate search subsumes single-label search and additionally supports AND queries (x_q, L_q, k) where $L_q \subseteq L_x$, and OR queries (x_q, L_q, k) where $L_q \cap L_x \neq \emptyset$.

For both search types, the *selectivity* of filter σ is defined as $|\mathcal{P}_\sigma|/|S|$, which fundamentally determines query complexity. We measure search quality using $\texttt{Recall@K} = \frac{|R \cap R^*|}{k}$, where R and R^* denote the retrieved results and ground truth respectively.

2.2 Filtered ANNS Approaches

Below, we review existing methods of filtered ANNS organized by how filtering is integrated. This taxonomy is orthogonal to the underlying index type, focusing instead on how adaptations are applied to search algorithms and index structures. For each approach, we first discuss its application to single-label search scenarios and then examine its support for complex predicate search.

Metadata filtering. This approach integrates filtering with no or minimal modifications to search algorithms and index structures, utilizing a single shared index. It includes three subtypes: (1) *Pre-filtering* [42, 44, 45] retrieves all qualified vectors from a scalar index and performs brute-force search, since most vector indexes cannot efficiently support searching subsets of vectors. (2) *Post-filtering* [42, 45] filters intermediate results from unfiltered ANNS, though determining sufficiently large k to ensure enough results remain after filtering is challenging and overshoot leads to inefficiency. (3) *Inline filtering* [20, 24, 34, 52] excludes unqualified vectors during search, eliminating intermediate k selection but requiring explicit algorithmic support. This approach can support arbitrary complex predicates.

Per-label indexing. This approach constructs dedicated vector indexes for each label and directs queries to the corresponding index without changing search algorithms [21]. While it delivers optimal search performance for single-label queries by eliminating query-time filtering, it incurs significant construction and memory overhead since each vector may be indexed multiple times across different per-label indexes. Extending this to per-predicate indexing for complex predicate search is generally impossible due to the combinatorial explosion of possible predicates.

Specialized indexes. Recent research has developed specialized indexes for filtered ANNS modifying both index construction and search algorithms, with most work focusing on graph-based approaches. These methods can be categorized into three main strategies:

(1) *Fused distance.* These methods [43, 46] employ hybrid distance metrics that integrate both attribute and vector similarity throughout index construction and query processing. They are limited to equality-based attribute matching and require queries to specify values of all attributes for attribute similarity computation. This severely restricts query expressiveness, making them unsuitable for the problem settings we are considering.

(2) *Graph densification.* Filtered DiskANN [12] and ACORN [33] follow a two-phase approach: dense graph construction then compression. Filtered DiskANN builds dense graphs by merging per-label indexes, while ACORN directly builds large-degree graphs. For compression, Filtered DiskANN prunes edges while preserving connectivity, whereas ACORN removes edges between nodes reachable as two-hop neighbors, compensating by visiting both direct and two-hop neighbors during search. Both face the fundamental connectivity challenge that the required graph degree scales with 1/selectivity. When this scaling becomes prohibitive, ACORN falls back to pre-filtering.

(3) *Subgraph stitching.* UNG [5] constructs separate graph indexes for each unique label set and connects them using cross-group edges. During query processing, it traverses subgraphs corresponding to matching label sets, with connectivity among label sets guaranteed by cross-group edge construction to ensure completeness. However, this approach often becomes inefficient when label set groups become highly fragmented with few nodes per group, which frequently occurs in real datasets. In such cases, the graph structure degrades as most edges become cross-group connections that prioritize completeness over search efficiency. This leads to significant regression in search performance and long construction times for the stitching process in our evaluation.

These specialized approaches exhibit varying support for complex predicates due to their tight integration of filtering with index structures. Filtered DiskANN supports only OR queries, and UNG supports only AND queries, while ACORN extends beyond these limitations to handle general complex predicates.

3 Curator Overview

This section presents Curator's architecture and design principles. Fig. 3 illustrates Curator's core design: a single shared *base index* and a set of *per-label indexes* embedded within it. This strategy allows per-label indexes to share the base index's spatial partitioning while adapting their granularity to their unique label distributions, addressing the fundamental limitations of existing approaches for low-selectivity queries outlined in §2.

3.1 Base Index

The base index T is a standard *hierarchical k-means clustering tree* [27] that partitions all vectors in the dataset. The tree recursively partitions the vector space using k-means clustering until reaching sufficiently fine granularity, i.e., when a node contains fewer vectors than the predefined *leaf capacity*. The structure of the base index thus adapts to the distribution of all indexed vectors, which we refer to as the *vector distribution*. Within the base index, each node stores a centroid that represents the corresponding cluster during search, while leaf nodes store the actual vector data within the cluster. This base index serves as the shared partitioning structure for embedded per-label indexes.

We choose hierarchical k-means as the base index for two key reasons. First, as a partition-based index, it does not suffer from the inherent connectivity issues that plague graph-based indexes at low selectivity, as partitions remain well-defined regardless of label sparsity. Second, the hierarchical structure naturally embeds varied partitioning granularities for different filters within a single tree, allowing per-label indexes to adapt their clustering depth to label-specific distributions. Regarding search performance, hierarchical k-means is typically slower than graph-based indexes

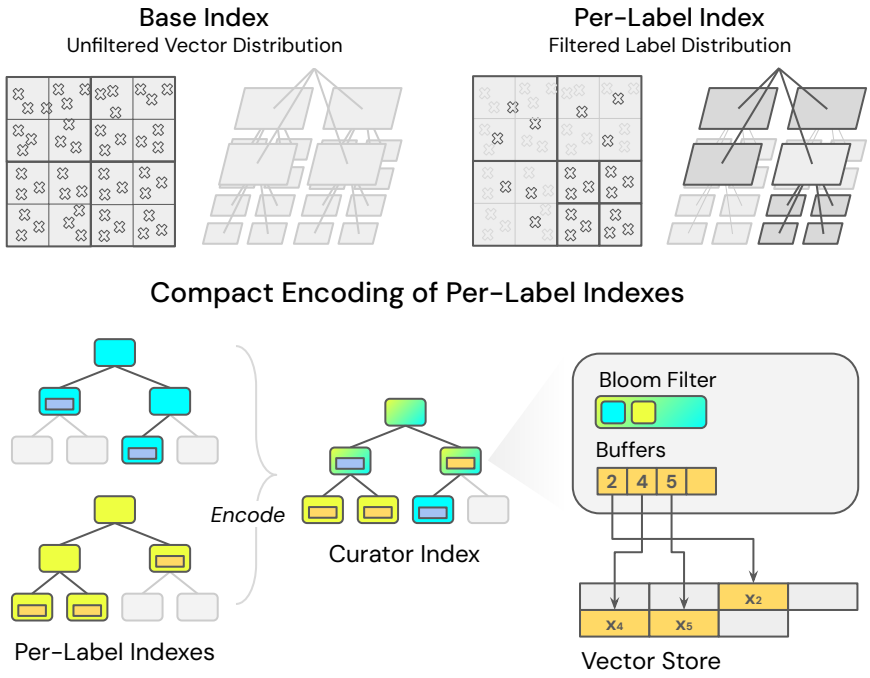


Fig. 3. Overview of Curator. The base index represents the finest-grained partitioning on unfiltered vector distribution, while the per-label indexes are embedded within the base index, adapting to the unique distribution of their respective labels. Per-label indexes are compactly encoded using buffers and Bloom filters.

like HNSW but comparable to IVF for unfiltered search. This performance positioning is confirmed in our evaluation (Fig. 8), where Curator traversing per-label clustering trees achieves performance comparable to per-label IVF.

3.2 Per-Label Indexes

Per-label indexes are embedded within the base index to adapt to the unique distribution of their respective labels. Unlike traditional approaches that build separate indexes for each label, Curator's per-label indexes reuse the base index's spatial partitioning while adjusting granularity to label-specific vector densities. This hierarchical storage strategy ensures that all vector data resides exclusively in the base index, avoiding the costly vector duplication that would occur if per-label indexes stored their own copies of vector data.

Per-label indexes are compactly encoded using two key data structures: *buffers* that store vector identifiers at leaf nodes, and *Bloom filters* that define the structure of per-label indexes.

Vector Identifiers and Buffers. To avoid vector duplication, per-label indexes store buffers—sorted lists of *vector identifiers*—at leaf nodes rather than actual vectors. The *buffer capacity* B_{max} controls leaf node granularity. These identifiers are constructed hierarchically to encode each vector's tree location: each node is assigned a *node identifier* as a sequence of branch indices from the root, and each vector's identifier combines its containing leaf's node identifier with its index within that leaf. These two components are bit-packed into a single integer with zero padding for uniform length. This construction ensures each node identifier represents a *vector ID subspace* containing all vectors with that prefix, so sorting vector identifiers groups vectors from the same sub-tree contiguously

Table 1. Notations used in the paper

Notation	Description
$\mathcal{P}$	Set of indexed vectors
$\mathcal{L}$	Finite universe of labels
$\mathcal{P}_l$	Vectors associated with label l
$\mathcal{L}_x$	Labels associated with vector x
T, T_l	Index and per-label index for label l
$\mathcal{P}_n$	Cluster represented by tree node n
μ_n	Centroid of node n
$\mathcal{P}_{n,l}$	Buffer for label l at node n
$\mathcal{B}_n$	Bloom filter at node n
C_n	Child nodes of node n
B_{max}	Buffer capacity

(Fig. 5). This property enables efficient construction of temporary per-predicate indexes (§4.2), batch construction of per-label indexes (§5.2), and buffer split/merge operations during updates (§5.1).

Bloom Filters. The base index contains three distinct node types with respect to each per-label index: *internal nodes* requiring further partitioning, *leaf nodes* containing buffers with qualified vectors, and *external nodes* not included in the per-label index. During search, Curator must efficiently determine the boundaries of each per-label index—consisting of leaf nodes and external nodes—to know when to backtrack. While leaf nodes are easily identified by the presence of buffers, distinguishing between internal and external nodes is challenging since both lack buffers.

A straightforward approach would maintain an exact set at each node n representing the per-label indexes containing it, denoted as $\mathcal{T}_n = \{l \mid n \in T_l\}$. During search, checking membership $l \in \mathcal{T}_n$ would determine whether the current node is inside T_l. However, storing $\mathcal{T}_n$ exactly becomes costly when the number of unique labels in the dataset is large. Therefore, Curator approximates $\mathcal{T}_n$ using Bloom filters $\mathcal{B}_n$ at each node in the base index. Bloom filters are space-efficient probabilistic data structures that support membership queries with bounded false positive rate p. Importantly, Bloom filters have no false negatives, ensuring that when $l \notin \mathcal{B}_n$, the search algorithm has definitively reached an external node and can safely backtrack. While false positives may cause occasional unnecessary exploration of nodes outside the per-label index, they do not affect search correctness since external nodes do not contain buffers and thus never contribute to the result set.

4 Search Algorithms

This section presents the core search algorithms that enable Curator to deliver efficient search when the filter selectivity is low. We distinguish two query types: single-label search and complex predicates that involve more attribute types and logical operators. The embedded per-label index structure ensures both *efficiency* by avoiding visits to unqualified vectors and adapting structure to label distribution, and *completeness* by covering all qualifying vectors within a connected clustering tree. For single-label filters, Curator traverses the corresponding pre-built per-label index; for complex predicates, it first constructs a temporary per-predicate index with minimal overhead and then applies the exact same search algorithm.

4.1 Single-Label Search

Single-label search finds qualifying vectors by scanning clusters closest to the query vector within the space partitioning represented by the per-label index (Fig. 4). Algorithm 1 implements this in

Algorithm 1: Single-Label Search

Input : Index T, query vector x_q, label l, beam width b, result set size ef
Output: k approximate nearest neighbors of x_q

```
Initialize Q ← BeamSearch(T.root, b)                          // Frontier
Initialize R ← ∅                                              // Result set
while Q ≠ ∅ do
    n ← arg min_{n' ∈ Q} S(n', x_q)
    Q ← Q \ {n}
    if l ∉ B_n then                                           // ❶ Outside T_l
        continue
    else if P_{n,l} ≠ ∅ then                                  // ❷ Leaf node
        R' ← R ∪ P_{n,l}
        if |R'| > ef then
            R' ← top-ef closest vectors in R'
        if R' = R then
            break
        R ← R'
    else                                                      // ❸ Internal node
        Q ← Q ∪ C_n
return top-k closest vectors in R
```

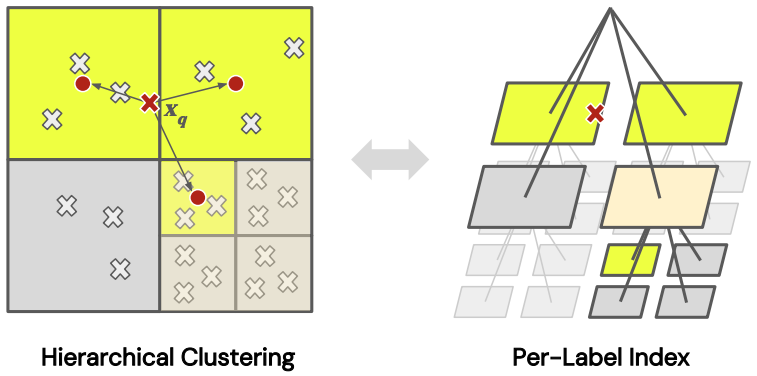


Fig. 4. Single-label search on Curator. Highlighted clusters and tree nodes are visited by the search algorithm.

two phases: beam search for robust initialization that traverses to the neighborhood of the query vector, followed by best-first search for efficient exploration of that neighborhood.

Phase 1: Beam Search. Algorithm 1 begins with beam search to initialize the search frontier Q with tree nodes representing nearest clusters. This phase traverses from the root until reaching leaf nodes of the per-label index, maintaining the top-b closest nodes at each level. Without beam search, best-first search starting from the root would be misguided by the imprecision of high-level nodes, causing the algorithm to become trapped in suboptimal subtrees without exploring subtrees whose centroids are relatively far from the query vector but still contain nearby vectors within their

boundaries. Exploring multiple promising paths simultaneously mitigates this centroid imprecision by increasing the likelihood that the actual nearest leaf nodes are covered by the beam search.

Phase 2: Best-First Search. Using the frontier initialized by beam search, the second phase performs best-first search, systematically exploring the neighborhood guided by heuristic scores. The algorithm processes nodes based on their type with regard to the per-label index T_l: *leaf nodes* containing buffers update the result set with their qualified vectors. For nodes without buffers, the Bloom filter $\mathcal{B}_n$ determines whether the node is an *internal node* requiring expansion or an *external node* outside T_l where the search should backtrack. During exploration, the search maintains a dynamic result set $\mathcal{R}$ containing the *ef* closest vectors discovered so far. Each time a buffer is visited, the algorithm updates $\mathcal{R}$ and checks for convergence—if the result set remains unchanged (i.e., no vector in the buffer is closer than the worst member of $\mathcal{R}$), the search terminates. Larger *ef* values widen the exploration radius, trading more work for higher recall. In the limit, when *ef* is sufficiently large, the traversal visits the entire per-label index encompassing all qualifying vectors, thus guaranteeing completeness. We found this adaptive termination criterion accommodates varying query difficulties more effectively than approaches that use a fixed computational budget.

Heuristic Function. Both search phases employ the same heuristic function to score candidate nodes: $S(n, x_q) = \left\|\mu_n - x_q\right\| - \alpha \cdot \frac{1}{|\mathcal{P}_n|} \sum_{x \in \mathcal{P}_n} \|x - \mu_n\|$. This heuristic estimates the minimum query-to-cluster distance by combining centroid distance with cluster radius. The α coefficient encourages exploring large, distant clusters to avoid local optima.

4.2 Complex Predicate Search

The search algorithm described in the previous section supports only single-label categorical filters, which limits query expressiveness in two key ways: (1) it only supports categorical attributes, preventing range queries on numerical attributes, and (2) it does not support logical operators like AND, OR, or NOT to compose multiple filtering conditions.

Approach Overview. To address these limitations, we extend the system to support arbitrary complex predicates while maintaining the performance characteristics of single-label search. The key insight is that any complex predicate ultimately identifies a subset of vectors that qualify the given conditions. Instead of modifying the existing search infrastructure, we treat this subset as if it were associated with a virtual label, allowing us to reuse the same hierarchical partitioning and search algorithms. By constructing a temporary index containing only these qualifying vectors, we can apply the same search algorithm used for single-label filters.

The challenge lies in efficiently constructing this temporary index without expensive operations. A naive approach would require traversing the index for each qualified vector to find its corresponding position in the temporary index, which involves distance computations and tree traversals.

Our approach exploits a key property of the vector identifier construction described in §3: given a list of qualified vectors sorted by their IDs, vectors from the same sub-tree are grouped contiguously. This property enables efficient temporary index construction through binary search operations. Given a sorted list of qualifying vector identifiers, we can recursively partition them into sub-trees by finding the boundaries between different child nodes using binary search, following the same hierarchical structure as the main index.

The sorted list of vector IDs, which we expect as the input for temporary index construction, is typically produced by an external relational database. This list could be represented as a posting list from an inverted index or a compressed bitmap. Evaluating attribute predicates and composing bitmaps inside the database is far cheaper than vector distance computation, so we treat this sorted list as the algorithm input. For further optimization, the costs of filter evaluation and temporary

Algorithm 2: Temporary Index Construction

Input: Sorted qualified vector IDs $\mathcal{P}_\sigma$, main index T
Output: Temporary hierarchical index T_σ

$T_\sigma \leftarrow$ InitializeIndex()
$T_\sigma.root \leftarrow$ BuildSubtree($\mathcal{P}_\sigma, T.root, 0, |\mathcal{P}_\sigma|$)
return T_σ

Function BuildSubtree($\mathcal{P}_\sigma, n, l, r$)
  /* Build sub-tree rooted at n for $\mathcal{P}_\sigma[l:r]$ */
  $n_\sigma \leftarrow$ InitializeNode()
  if $r - l \leq B_{max}$ **then** // ❶ Leaf node of T_σ
    return n_σ
  foreach $c \in C_n$ **do** // ❷ Split $\mathcal{P}_\sigma[l:r]$
    /* c covers vector ID range $[c.min, c.max)$ */
    $l_c \leftarrow$ BinarySearch($\mathcal{P}_\sigma, c.min, l, r$)
    $r_c \leftarrow$ BinarySearch($\mathcal{P}_\sigma, c.max, l, r$)
    if $l_c \leq r_c$ **then**
      $c_\sigma \leftarrow$ BuildSubtree($\mathcal{P}_\sigma, c, l_c, r_c$)
      n_σ.AddChild(c_σ)
  return n_σ

index construction can be reduced by only performing these operations in the sub-trees projected to be searched by the algorithm, rather than evaluating the predicate over all vectors and constructing the complete temporary index upfront.

Temporary Index Construction. Algorithm 2 presents the temporary index construction procedure. The algorithm exploits the sorted vector ID property through a recursive tree building process that mirrors the structure of the main index T. The BuildSubtree function constructs each node n_σ in the temporary index T_σ by processing a corresponding range $\mathcal{P}_\sigma[l:r]$ of sorted qualified vector IDs. The algorithm operates as follows for each node: (1) Leaf condition: If the number of qualified vectors in the current range does not exceed the buffer capacity B_{max}, the algorithm creates a leaf node and terminates recursion for this sub-tree. (2) Hierarchical partitioning: Otherwise, it iterates through each child c of the corresponding main index node n. For each child, binary search operations locate the subset of qualified vectors whose IDs fall within the child's vector ID range $[c.min, c.max)$. This partitioning step leverages the contiguous property of sorted vector IDs to efficiently split the qualified vector list. After partitioning, the construction algorithm is then applied recursively to build the sub-tree for each child.

Fig. 5 illustrates this process using a simplified scenario where each node has two children and both leaf capacity and buffer capacity are 2, allowing vector IDs to be represented as 3-digit binary numbers. The left panel shows the base index with vector ID ranges annotated for each node, such as [000-111] for the root and [000-011], [100-111] for its children, corresponding to the $[c.min, c.max)$ ranges referenced in the algorithm. The middle panel demonstrates the recursive partitioning process: starting with sorted qualified vector IDs $\{000, 010, 100, 101, 110\}$, binary search finds the partition boundary at vector ID 100 (the start of the right sub-tree's range [100-111]), splitting the list into groups $\{000, 010\}$ and $\{100, 101, 110\}$. The partitioning continues recursively until the leaf condition is met—for instance, the right sub-tree is further split into $\{100, 101\}$ and

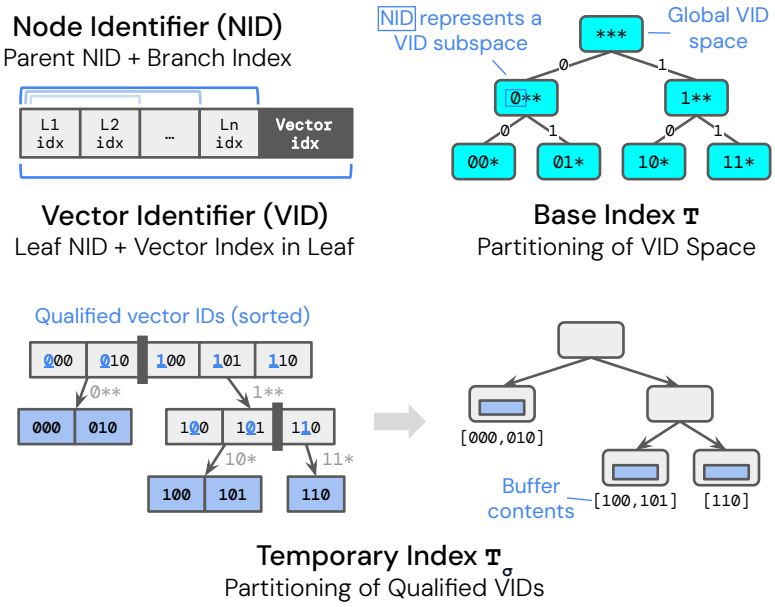


Fig. 5. Temporary index construction for complex predicates. The sorted qualified vector IDs are recursively partitioned following the base index structure, enabling efficient construction of a temporary index.

{110}, with the latter qualifying the leaf condition (≤ 2 vectors). The right panel shows the resulting temporary index where the qualified vector IDs are distributed into buffers in leaf nodes.

The temporary index maintains a lightweight structure where each node stores only: (1) a pointer to the corresponding node in the base index, (2) the range offsets $[l, r)$ in the qualified vector ID list, and (3) pointers to child nodes. This design minimizes memory overhead while preserving the hierarchical structure necessary for efficient search.

Search Process. Once the temporary index is constructed, the search algorithm proceeds identically to the single-label search case. While there are minor layout differences between the per-label indexes embedded in the main index and the constructed temporary index, they are conceptually equivalent, thus the difference does not affect the correctness or efficiency of the search algorithm.

Pre-indexing Filters. Despite finding that the overhead of temporary index construction is minimal in our evaluation, in certain scenarios it may be desirable to further eliminate this overhead by pre-constructing indexes for filter predicates, particularly when some filter predicates are frequently encountered. The simplest approach is caching the constructed temporary indexes as a side product of complex predicate search, which requires minimal additional implementation effort. Alternatively, we can permanently integrate the temporary indexes into the main index structure to save memory and reuse update routines. This integration process involves assigning a virtual label to the predicate to reference the temporary index in future queries, distributing the qualifying vectors as buffers across the appropriate nodes in the hierarchy, and updating the Bloom filters accordingly. The integration is straightforward due to the one-to-one mapping between nodes of the temporary index and the main index. Both approaches achieve nearly identical search performance in our evaluation, suggesting that the choice between them can be made based on application-specific considerations.

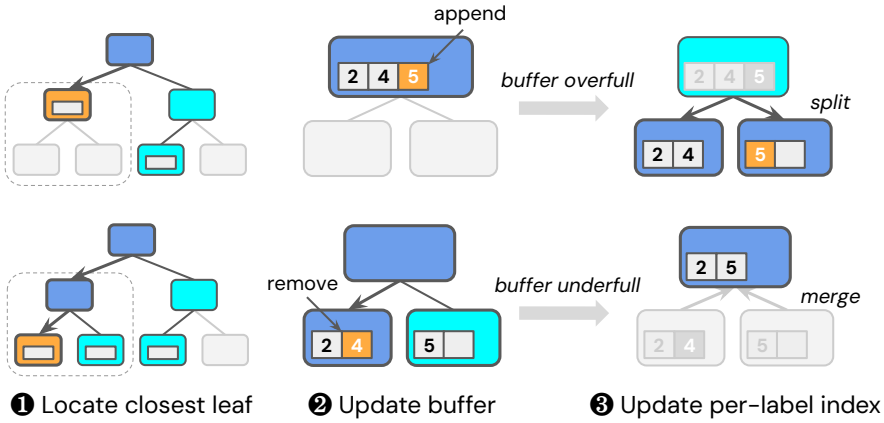


Fig. 6. Label update operations in Curator. Top: label insertion triggers buffer overflow, causing vectors to be flushed to child nodes. Bottom: label deletion creates underfull buffers in child nodes that merge into the parent node.

Curator treats pre-construction or caching of temporary indexes as a pluggable policy. Deployments may supply custom logic or rely on standard workload analysis tools and query optimizers to identify frequent, high-payoff predicates within resource budgets, preserving compatibility with existing index management pipelines.

5 Index Maintenance

Curator maintains both a base index fitted to the overall vector distribution and multiple per-label indexes adapted to their respective label distributions. As data evolves, the system must efficiently update these structures while preserving search performance. This section describes the incremental update mechanisms and rebuilding strategies that maintain index quality with minimal overhead.

5.1 Incremental Updates

Curator supports efficient incremental updates to both vectors and labels, each requiring different maintenance strategies.

Label Updates. Label updates modify per-label indexes through buffer operations and Bloom filter maintenance, as illustrated in Fig. 6. Throughout this process, Curator maintains two index invariants that ensure both search performance and correctness: buffer capacity limits and Bloom filter correctness.

Curator maintains buffer capacity limits using a strategy similar to buffer trees [2]. When a buffer overflows due to label insertion, vectors are flushed to child nodes at the next level (Fig. 6 top). Conversely, when sibling buffers' combined size falls below capacity due to deletions, they merge into the parent node (Fig. 6 bottom). These operations occur recursively and avoid distance computations by leveraging the root-to-leaf path encoded in vector identifiers. Efficiency is further enhanced by the contiguous grouping property (§3), where vectors belonging to the same child are always grouped contiguously within buffers, enabling efficient partitioning via array slicing.

Curator maintains Bloom filter correctness when buffer operations cause structural changes to per-label indexes. For label insertions that expand the per-label index to new nodes, the label is simply added to the Bloom filter at those nodes. For buffer merging that removes nodes from per-label indexes, the Bloom filter of parent node p must be recomputed recursively based on its

children C_p, since Bloom filters cannot remove elements:

$$\mathcal{B}_p = \{l \mid l \in \mathcal{L}, \mathcal{P}_{p,l} \neq \emptyset\} \cup \bigcup_{c \in C_p} \mathcal{B}_c \tag{1}$$

Here, the first term represents labels with buffers stored at p (per-label trees where p is a leaf node), while the second aggregates child Bloom filters (per-label trees where p is an internal node). Updates propagate recursively from modified nodes upward until no further changes occur.

Vector Updates. Unlike label updates, vector updates modify only the base index by finding the nearest leaf node through greedy search and adding or removing vector identifiers there. For deletions, the root-to-leaf path encoded in vector identifiers eliminates the need for greedy search, as the target leaf node can be directly identified from the vector's encoded path. Vector updates could use a similar split/merge strategy as label updates, but this would require updating multiple per-label indexes affected by the structural changes. This impact is typically minimal since few per-label indexes reach the leaf nodes of the base index due to label sparsity, but Curator instead enforces leaf capacity through periodic rebuilding (§5.2) for simplicity.

5.2 Index Rebuilding

While incremental updates handle individual changes efficiently, accumulated updates can cause bucket imbalances that degrade search performance [49]. To address this, index rebuilding involves reconstructing either the entire base index or a sub-tree, followed by batch construction of per-label indexes within the affected region.

Curator offers two rebuilding strategies that provide different trade-offs between overhead and freshness. *Global rebuilding* reconstructs the entire base index using hierarchical k-means clustering, followed by batch construction of all per-label indexes. The batch construction of per-label indexes follows the same recursive partitioning procedure as temporary index construction for complex predicates (§4.2). This provides optimal index quality but incurs higher overhead to maintain the same level of freshness. In contrast, *local rebuilding* targets only sub-trees with significant updates, reducing overhead while maintaining freshness where needed. To determine when local rebuilding is necessary, Curator tracks update statistics at each node: total vectors $|\mathcal{P}_n|$ and updates since last rebuild U_n. When the update ratio $U_n/|\mathcal{P}_n|$ exceeds a predefined threshold, the sub-tree enters a rebuilding queue for batch processing at defined intervals or on manual request, with new sub-trees atomically swapped in and counters reset.

6 Evaluation

This section empirically evaluates our method across various dimensions. Overall, our results demonstrate:

- Curator achieves competitive search latency with minimal build time and index overhead compared to other approaches.
- Curator can be efficiently integrated into existing graph-based indexes with minimal overhead to address their performance issues at low selectivity. Supplementing ACORN with Curator improves search performance by up to 20.9× with merely 5.5% and 4.3% overhead in construction time and memory footprint, respectively.
- Curator efficiently supports incremental updates of both vectors and labels, as well as low-selectivity queries with arbitrary complex predicates.

Table 2. Characteristics of Evaluation Datasets

	YFCC-10M	arXiv
Source Data	Image	Text
# Vectors	10M	2M
Vector Dimension	192	384
Unique Labels	1000	100
Labels per Vector	5.53	9.93

6.1 Experimental Setup

Environment. Curator and the baselines are implemented in C++ and compiled using GCC 9.4.0 with `-O3` and `AVX512` optimization. Our implementation builds upon the FAISS library [20] infrastructure. All evaluations run on a server equipped with an Intel Xeon Gold 5215 @2.5GHz processor and 256GB of RAM, running Linux Ubuntu 20.04 LTS. Unless otherwise specified, all experiments are conducted in a single-threaded environment.

Datasets. We use the following real-world datasets for evaluation and provide their statistics in Table 2. Each dataset is randomly divided into training and test sets, with the test set containing 10,000 vectors. (1) The YFCC-10M dataset is a 10M vector subset of the original YFCC100M dataset [41], which serves as the standard benchmark in the NeurIPS'23 Big-ANN competition [39]. This dataset features a hybrid of vector and scalar data, with images embedded using the CLIP model [36] and annotated with labels representing objects, locations, etc. We use the 1000 most popular labels to ensure all baselines consume a manageable amount of memory. (2) The arXiv dataset [7] contains metadata from 2.3M arXiv papers, including titles, authors, abstracts, and paper categories (e.g., `cs.DB` denotes the field of database systems). We embed paper abstracts using the `all-MiniLM-L6-v2` [14] model and use the 100 most popular categories as labels. While Curator's memory efficiency supports larger datasets, we limit evaluation to these sizes to enable comparison with memory-intensive baselines (see §6.3).

Baselines. We evaluate the performance of Curator against the following indexes: IVF [20], HNSW [25], Parlay-IVF [21], Filtered DiskANN [12], and ACORN [33]. IVF and HNSW represent state-of-the-art partition-based and graph-based indexes, respectively. For both IVF and HNSW, we develop variants of per-label indexing and inline filtering to facilitate filtered search. These variants are referred to as Per-label IVF/HNSW and Shared IVF/HNSW, abbreviated as P-IVF/HNSW and S-IVF/HNSW, respectively, throughout the paper. Parlay-IVF is the top open source entry in the filtered search track of the Big-ANN competition, which constructs per-label graph-based indexes with shared vector storage. Due to excessive memory requirements, we omit evaluation of Per-label IVF and Per-label HNSW baselines on the YFCC-10M dataset. We also exclude NHQ [43] from our evaluation as it requires specifying the presence or absence of all labels in filter predicates, and UNG [5] due to its poor performance on our datasets (as discussed in §2).[2]

In all experiments, we store vectors and compute distances in `float32` precision without scalar or product quantization [17], and conduct parameter sweeps to determine the Pareto-optimal configurations for each index. For experiments displaying the performance of only a single configuration per index, we choose the configuration that achieves 90% recall with the lowest

[2]We present an extended evaluation including `pgvector`, a PostgreSQL extension that supports vector search with SQL-based filtering, in our technical report [19]. `pgvector` achieves the lowest query throughput among the evaluated systems, largely due to executor overhead and disk I/O in PostgreSQL. We therefore exclude these results from the main paper, as the comparison is not directly comparable to our in-memory indexing approaches.

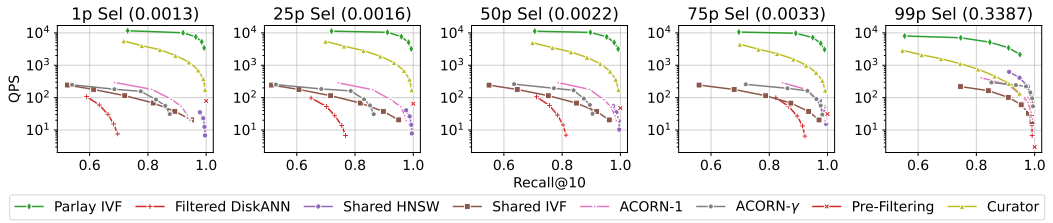


Fig. 7. Trade-offs between recall and query throughput for single-label filters with varied selectivity on YFCC-10M. Each line represents an algorithm's performance across various search parameter configurations. Parenthetical numbers in subplot titles indicate the actual selectivity value for each filter.

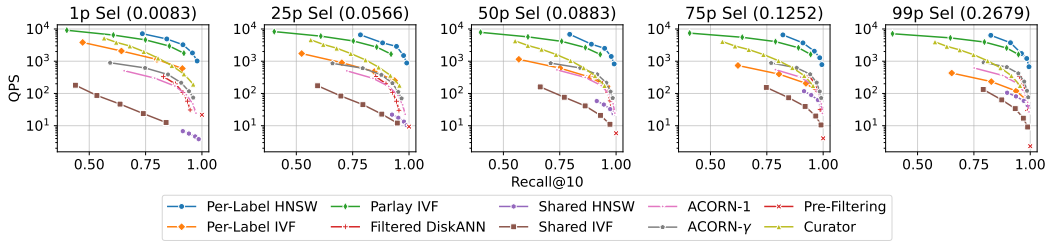


Fig. 8. Trade-offs between recall and query throughput for single-label filters with varied selectivity on arXiv. Each line represents an algorithm's performance across various search parameter configurations. Parenthetical numbers in subplot titles indicate the actual selectivity value for each filter.

search latency. We list the parameter space of each baseline below: (1) Per-label IVF: $nlist \in \{10, 20, 40\}$, $nprobe \in \{1, 2, 4, 8\}$; (2) Per-label HNSW: $efc \in \{16, 32, 64, 128\}$, $M \in \{8, 16, 32\}$, $ef \in \{8, 16, 32, 64, 128\}$; (3) Shared IVF: $nlist \in \{1000, 2000, 4000, 8000, 16000\}$, $nprobe \in \{8, 16, 32, 64, 128\}$; (4) Shared HNSW: $efc \in \{16, 32, 64, 128\}$, $M \in \{16, 32, 64\}$, $ef \in \{8, 16, 32, 64, 128\}$; (5) Parlay-IVF: $M \in \{8, 16, 32\}$, $ef \in \{8, 16, 32, 64, 128\}$; (6) Filtered DiskANN: $efc \in \{64, 128, 256, 512\}$, $M \in \{64, 128, 256, 512\}$, $ef \in \{64, 128, 256, 512, 1024\}$, $\alpha = 1.2$; (7) ACORN-γ: $M \in \{16, 32, 64\}$, $\gamma \in \{10, 20, 40\}$, $M_\beta \in \{M, 2M, 4M\}$, $ef \in \{8, 16, 32, 64, 128\}$; (8) ACORN-1: $M \in \{16, 32, 64\}$, $\gamma = 1$, $M_\beta = 2M$, $ef \in \{8, 16, 32, 64, 128\}$; (9) Curator: $nlist \in \{16, 32\}$, $B_{max} \in \{64, 128, 256\}$ (buffer and leaf capacity), $ef \in \{64, 128, 256, 512, 1024\}$, $b = 4$ (beam size).

6.2 Query Performance

Single-Label Search. Fig. 7-8 show the recall-throughput trade-offs across varied filter selectivity levels. Overall, per-label indexing approaches achieve the best search performance when feasible: Parlay-IVF consistently delivers optimal results by sharing vector data among per-label indexes, while Per-Label HNSW and IVF's performance on arXiv do not scale to the 10M-scale YFCC-10M dataset due to excessive memory overhead. However, these approaches come with prohibitive memory and construction costs, as demonstrated in our analysis in the subsequent sections, making them impractical for many real-world deployments.

For specialized indexes, performance degrades significantly as selectivity decreases due to graph connectivity issues. Filtered DiskANN's performance drops most severely, suffering from its less efficient compression strategy, while ACORN-1 slightly outperforms ACORN-γ at very low selectivity levels. Metadata filtering baselines generally exhibit lower query throughput but show interesting behavior at high selectivity levels: Shared HNSW achieves better performance than specialized indexes at the 99th percentile selectivity (0.34) on YFCC-10M, demonstrating that the

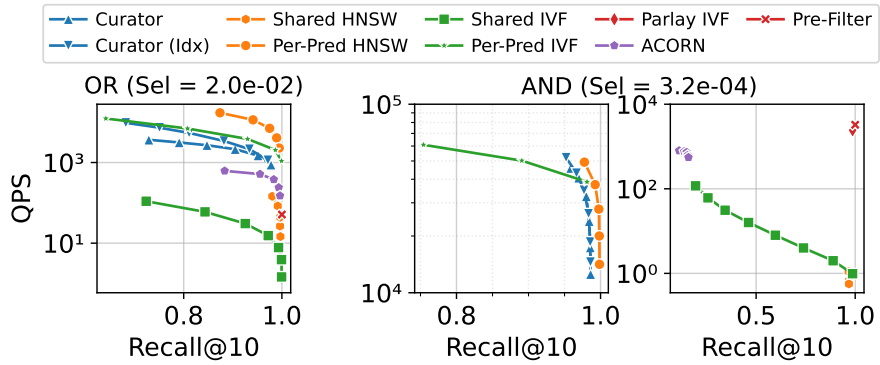


Fig. 9. Recall vs query throughput trade-offs for complex predicate queries. Each line represents an algorithm's performance across various search parameter configurations. Average selectivity values are annotated in subplot titles. AND query results are split into two subplots due to the wide throughput range. Parlay-IVF only supports AND queries.

query-time overhead of specialized indexes can offset the benefits of fewer unnecessary distance computations.

Curator consistently achieves competitive performance across all selectivity levels, particularly excelling at low selectivity where specialized indexes and metadata filtering baselines struggle. For example, Shared HNSW is dominated by pre-filtering at low selectivity, while Curator achieves higher throughput than pre-filtering at near-perfect recall and offers a flexible recall-throughput trade-off. Additionally, Curator matches Per-Label IVF's search performance on arXiv while requiring significantly lower memory usage and construction costs, demonstrating the efficiency of its embedded per-label indexes.

This performance profile reflects Curator's deliberate design trade-offs as discussed in §1. Per-label indexing approaches (Parlay-IVF, Per-Label HNSW/IVF) achieve the best search performance by eliminating query-time filtering but require excessive memory overhead and construction costs due to vector duplication, as demonstrated in §6.3. Similarly, Curator exhibits slower search than specialized graph-based indexes (ACORN) at high selectivity, which is expected since Curator's partition-based approach prioritizes robustness to sparsity over the precision of graph connectivity. However, Curator's performance advantage emerges precisely where it was designed to excel: at low selectivity where graph connectivity breaks down, Curator maintains efficient search while other approaches either fail entirely or fall back to expensive pre-filtering.

Complex Predicate Queries. We construct queries with complex predicates based on the YFCC-10M dataset, evaluating two types of filter predicates: OR queries ($l_1 \vee l_2$, requiring existence of either label in the vector's label list) and AND queries ($l_1 \wedge l_2$, requiring existence of both labels). Given the average label selectivity of approximately 1% in YFCC-10M, OR queries have selectivity around 2% while AND queries have selectivity around 0.01%. For each predicate type, we randomly generate 100 label combinations and select 100 query vectors from the test set, resulting in 10,000 total queries.

We evaluate Curator's performance against Shared HNSW, Shared IVF, Parlay-IVF (AND queries only), and ACORN, which are the only baselines supporting complex predicates. Two variants of Curator are evaluated: (1) predicate not indexed, where Curator constructs a temporary index structure at query time for qualified vectors and traverses it, and (2) predicate indexed, where qualified vectors for each predicate are pre-indexed as virtual labels, eliminating the need for

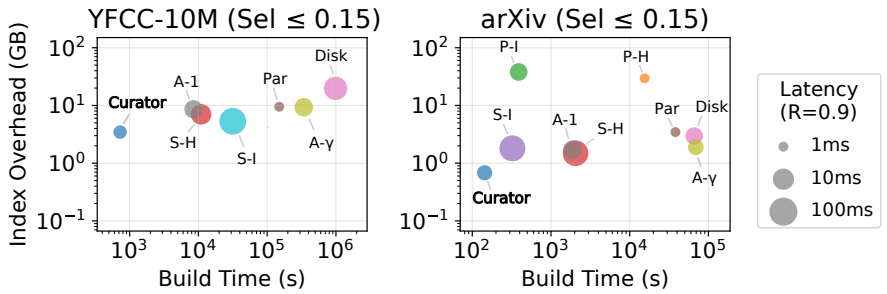


Fig. 10. Trade-off between index overhead, build time and search latency for evaluated indexes. We report search latency at 0.9 recall on low selectivity queries (≤ 0.15). Marker size indicates search latency.

temporary index construction during search. Per-predicate HNSW and IVF approaches, which build separate indexes for all possible predicates, serve as references for optimal performance but are generally impractical due to prohibitive memory and construction costs.

As shown in Fig. 9, Curator with predicate indexed achieves near-optimal performance compared to the per-predicate approaches, demonstrating the index quality of Curator's temporary indexes. Furthermore, Curator without indexing achieves similar performance to the indexed version, significantly outperforming ACORN on both filter types and showcasing the efficiency of temporary index construction at query time. ACORN fails to achieve >0.2 recall on AND queries due to limited connectivity in the graph structure at low selectivity.

The results demonstrate that Curator provides a unified solution for both single-label and complex predicate search: while it delivers competitive performance on single-label filters across all selectivity levels, it uniquely excels at complex predicates where most existing approaches either fail entirely or require prohibitive memory overhead for per-predicate indexing.

6.3 Resource Efficiency

Performance vs Resource Trade-offs. Fig. 10 provides a comprehensive view of the fundamental trade-offs between search performance, memory efficiency, and construction cost across all evaluated approaches. This analysis focuses on low-selectivity scenarios (≤ 0.15) where the differences between approaches are most pronounced.

The results demonstrate that Curator achieves competitive search performance on low-selectivity queries while maintaining the lowest in-memory overhead and build time among all evaluated approaches. Only two baselines, Parlay-IVF and Per-Label HNSW, achieve lower search latency than Curator, but at significantly higher resource costs. Specifically, Parlay-IVF requires 206× and 266× longer build times on YFCC-10M and arXiv respectively, with 2.8× and 5.1× higher memory overhead. Per-Label HNSW (on arXiv) incurs 108× longer build time and 43× higher memory overhead for only marginal latency improvements.

This analysis reveals why Curator represents an attractive practical solution: it delivers performance competitive with the best approaches while requiring minimal additional resources, making it suitable for deployment scenarios where memory and construction time constraints are critical considerations.

Construction Time. Fig. 11 compares the index construction time of Curator against all baselines. The index construction time includes the time to optionally train the index structure (required for Curator and IVF baselines) and to insert all vectors and labels. Filtered DiskANN, ACORN, and Parlay-IVF are constructed in batch mode, whereas all other indexes are built incrementally.

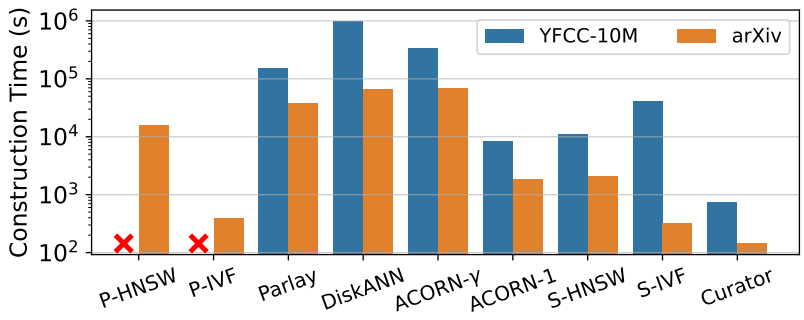


Fig. 11. Index construction time for the evaluated indexes.

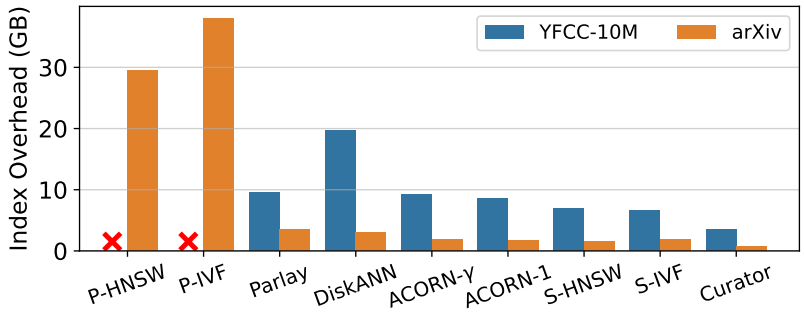


Fig. 12. Index overhead for evaluated indexes. Index overhead is computed by subtracting the raw vector storage size (excluding label lists) from the total memory footprint.

Overall, Curator achieves the lowest construction time across all evaluated approaches, requiring only 727s on YFCC-10M and 144s on arXiv. Specialized indexes like ACORN-γ and Filtered DiskANN incur dramatically higher construction costs on both datasets, primarily due to the overhead of constructing and compressing the intermediate dense graph. Specifically, ACORN-γ requires 468× and 482× longer construction time than Curator on YFCC-10M and arXiv respectively, while Filtered DiskANN requires 1,360× and 459× longer time. These specialized approaches require significantly more construction time than even building separate per-label indexes: both ACORN-γ and Filtered DiskANN take around 4× longer time to build than Per-Label HNSW on arXiv, highlighting the substantial overhead of these sophisticated indexing strategies.

Memory Overhead. Index overhead represents the additional memory required by indexes beyond storing the original vectors (assuming no compression), calculated by subtracting raw vector storage size from the total memory footprint. We subtract vector storage because all indexes maintain at least one copy of vector data, making this a fair comparison baseline. However, we do not subtract label/metadata storage since indexes preserve metadata in vastly different formats: some store plain label lists or inverted indexes, while per-label indexing baselines require no metadata storage at all.

As shown in Fig. 12, Curator achieves the lowest index overhead among all evaluated approaches, requiring only 3.45GB on YFCC-10M and 0.68GB on arXiv. Per-label indexing approaches incur dramatically higher memory costs due to vector data duplication, with Per-Label HNSW and Per-Label IVF requiring 43× and 56× more overhead than Curator on arXiv, respectively. Specialized

indexes show more moderate but still significant overhead increases due to their large graph degrees: Filtered DiskANN requires 5.7× and 4.4× more memory than Curator, while ACORN variants require 2.5–2.8× more overhead across both datasets.

6.4 Integration with Graph-Based Indexes

In this section, we evaluate how integrating Curator with graph-based indexes improves search performance at low selectivity without significantly increasing construction costs.

Dataset. For our evaluation, we construct a semi-synthetic dataset from a randomly sampled 1M subset of the YFCC-10M dataset to achieve systematic coverage of the low-selectivity regime with sufficient data points at each selectivity level for reliable evaluation. Specifically, we define 20 selectivity levels distributed on a logarithmic scale within the range $[0.001, 0.2]$. The logarithmic distribution ensures dense sampling at the lower end of the selectivity spectrum where Curator is designed to excel. For each selectivity level, we generate 10 labels by randomly sampling the corresponding fraction of vectors from the dataset. We determine the optimal build configuration for each index through grid search and evaluate queries under varying search configurations. For each selectivity level, we report the minimum query latency required to achieve an average recall of 90%.

Baselines and Integration Strategy. We select ACORN as the graph-based index as it achieves best performance on our datasets. To illustrate the trade-offs of densifying graphs, we evaluate two ACORN-γ constructions with $\gamma = 10$ and $\gamma = 40$, denoted as ACORN-10 and ACORN-40, respectively. For low-selectivity queries, we combine ACORN with either pre-filtering or Curator, resulting in four hybrid indexes in total. In our integration experiments, we assume users perform selectivity estimation and choose indexes based on fixed selectivity thresholds determined through offline profiling, similar to ACORN's approach. In practice, this selection logic can also be handled by external systems such as database query optimizers, allowing flexible integration strategies tailored to specific workload patterns.

Results. As shown in Fig. 2, densifying the graph-based index yields negligible improvements at low selectivity while significantly raising construction costs, increasing construction time by approximately 5× in our experiments. Integrating ACORN-10 with Curator increases construction time and memory footprint by only 5.5% and 4.3%, respectively, while improving search performance by up to 20.9× compared to ACORN with pre-filtering at low selectivity. These results demonstrate that Curator provides an efficient complementary solution for low-selectivity queries without imposing significant resource overhead.

6.5 Update Performance

Fig. 13 compares update performance of Curator against baselines supporting incremental updates. Parlay-IVF, Filtered DiskANN, and ACORN are excluded due to their lack of incremental update support. Per-label indexing baselines do not support vector insertion, as vectors are directly inserted into corresponding per-label indexes during label insertion operations. For deletion operations, only Curator supports both vector and label deletion, so performance is shown for Curator only.

While per-label indexing baselines avoid vector insertion costs, they incur significant overhead when assigning new labels to vectors, as each assignment involves an update to the corresponding index. In contrast, metadata filtering baselines involve costly distance computations during vector insertion but achieve fast label insertion by simply appending labels to metadata. Therefore, for scenarios where each vector is associated with many labels, the total update costs for per-label indexing are substantially higher than those for metadata filtering.

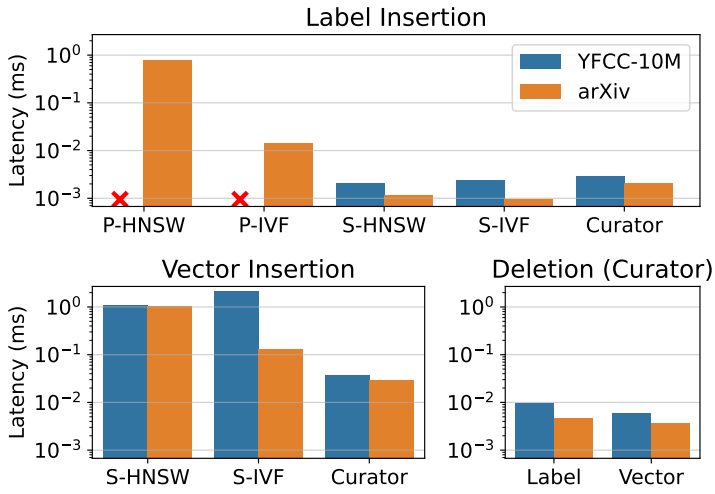


Fig. 13. Update performance for the indexes supporting incremental updates. Per-label indexing baselines do not support vector insertion. Deletion latency for vectors and labels is only shown for Curator, as other indexes do not support deletion.

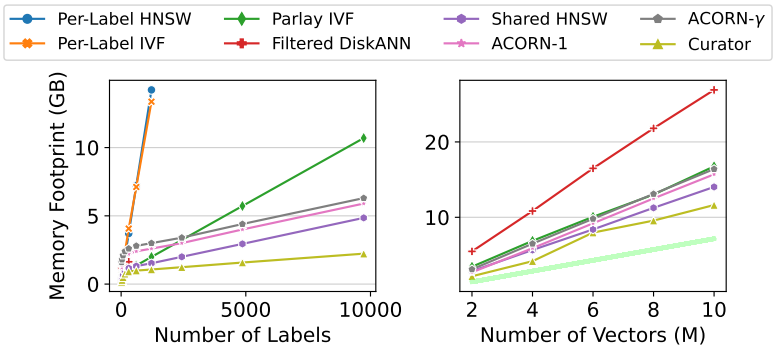


Fig. 14. Curator exhibits superior memory efficiency as the number of labels and vectors scales. The light green line shows the size of vector storage alone.

Compared to all baselines, Curator achieves consistently low latency across all update operations. Vector insertion in Curator is 4.4–58.7× faster than shared index approaches, while label insertion maintains minimal overhead of 0.002–0.003 ms across both datasets. Curator's deletion performance is also efficient, with vector deletion requiring 0.004–0.006 ms and label deletion requiring 0.005–0.010 ms. This low latency stems from the efficient navigation afforded by Curator's tree structure and fast buffer operations.

6.6 Scalability

In this section, we assess Curator's memory efficiency across varying numbers of labels and vectors.

Scaling with Number of Labels. For this experiment, we use a 1M subset of YFCC-10M, generating a series of datasets with varying numbers of labels where each label is randomly assigned to 1% of vectors. As shown in Fig. 14 (left), per-label approaches (HNSW and IVF) exhibit the steepest growth

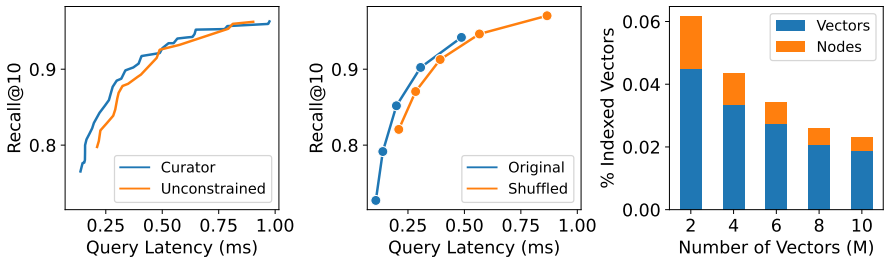


Fig. 15. Ablation studies validating Curator's design: (a) Performance impact of structural constraints on per-label indexes. (b) Performance impact of vector-label correlation. (c) Tree traversal overhead in Curator's search algorithm.

due to vector duplication across multiple indexes. Parlay-IVF grows second fastest—while it shares vector storage, each vector is indexed multiple times as the label count increases, increasing the total number of graph edges. Shared HNSW and ACORN show moderate growth since their index structures remain unchanged as labels increase, requiring only additional label storage. Curator achieves the smallest growth rate through compact encoding of embedded per-label indexes. Filtered DiskANN fails beyond 600 labels because the merged graph becomes increasingly dense, requiring more aggressive pruning to maintain a fixed maximum degree, which significantly degrades recall.

Scaling with Number of Vectors. For this experiment, we use sampled subsets of the YFCC-10M dataset ranging from 2M to 10M vectors, with average labels per vector held constant. Per-label HNSW and IVF are excluded from this comparison due to excessive memory requirements. As shown in Fig. 14 (right), the differences between graph-based approaches primarily stem from varied graph degrees: Filtered DiskANN requires significantly more edges to maintain connectivity after pruning, while Parlay-IVF, ACORN, and Shared HNSW exhibit similar memory growth rates since each vector maintains comparable total edge counts (aggregated across all per-label indexes for Parlay-IVF). Curator maintains the lowest memory footprint due to its compact encoding and partition-based design.

6.7 Ablation Study

We conduct three ablation studies to validate Curator's design and evaluate its robustness across different dataset characteristics.

Structural Constraints. Curator enforces structural sharing by constraining per-label indexes to follow the base index's hierarchical partitioning, which potentially impacts performance since per-label indexes cannot freely adapt to label distributions. To evaluate this trade-off, we compare Curator against a variant where per-label indexes are constructed independently with the same hyper-parameters. Fig. 15(a) shows nearly identical performance between both versions, demonstrating that structural constraints have minimal impact on the quality of embedded per-label indexes. This finding is further supported by the similar performance between Curator with predicate indexed versus per-predicate indexing in §6.2.

Vector-Label Correlation. Real-world datasets exhibit varying degrees of vector-label correlation—the tendency for vectors sharing the same label to cluster together in the embedding space due to semantic similarity. To assess Curator's robustness across datasets with different correlation patterns, we test on a shuffled variant of YFCC-10M where labels are randomly reassigned, effectively eliminating vector-label correlation. Fig. 15(b) shows nearly identical recall-throughput trade-offs on both original and shuffled datasets, confirming Curator's robust performance across varied

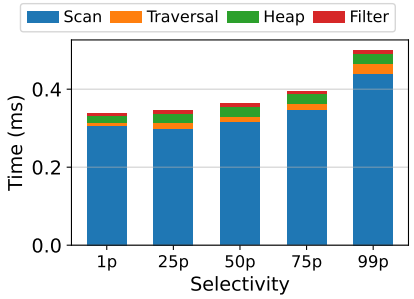


Fig. 16. Latency breakdown for Curator's single-label search on YFCC-10M across selectivity levels. Scanning (fetching vectors and computing distances) dominates at 82–86%, with minimal overhead from filtering, traversal, and heap operations.

data distributions. We additionally quantify correlation using the "Query Correlation" metric from ACORN [33], which compares a query's distance to the set of qualified vectors against a random set of equal size; larger values indicate stronger clustering by label. On YFCC-10M, the average query correlation is 0.161, whereas on the label-shuffled dataset it is 0.028, showing the effectiveness of label shuffling and confirming that the original dataset exhibits meaningful correlation.

Cost Breakdown. To understand where Curator spends time during search, we profile the internal operations of the single-label search algorithm across selectivity percentiles on YFCC-10M. Fig. 16 breaks down search latency into four categories: filtering (bloom filter checks and buffer existence lookups that confine search to the embedded sub-tree), traversal (centroid distance computations during tree navigation), scanning (fetching vectors from buffers and computing distances), and heap operations (priority queue maintenance for the search frontier). The results reveal that scanning dominates total latency, accounting for 82–86% of time across all selectivity levels. Filtering operations that restrict search to qualified regions contribute only 1–2% of total time, demonstrating that Curator's embedded sub-tree structure efficiently confines the search space. Traversal overhead remains modest at 2–5%, confirming that the tree structure prunes effectively. Heap operations for frontier management add 5–7%. This breakdown shows that Curator has low filtering and traversal overhead, with most time spent on the core work of fetching and comparing vectors in the candidate buffers.

Tree Traversal Overhead. Unlike graph-based indexes where all distance computations target individual vectors, Curator performs distance computations against both cluster centroids (during tree traversal) and vectors (during buffer scanning). To quantify the overhead of tree traversal, we measure the proportion of distance computations used for tree traversal versus buffer scanning. Fig. 15(c) shows that traversal overhead decreases from 26.9% to 18.9% as dataset size grows from 2M to 10M vectors, demonstrating that the overhead remains moderate and diminishes at larger scales.

7 Related Work

Approximate Nearest Neighbor Search. As vector similarity search becomes increasingly important across various modern applications, many efficient vector indexing techniques have been proposed. Graph-based indexes [9, 10, 13, 15, 16, 25] construct proximity graphs, where vectors are connected to their nearest neighbors. At query time, a best-first beam search begins from a

set of seed nodes and navigates through the graph to locate vectors near the query. Partition-based indexes, which can be further categorized into clustering-based [3, 4, 6, 20, 54], hashing-based [1, 11, 31, 37, 38, 47] and tree-based [26, 27, 40] approaches, partition the vector space into sub-spaces and represent vectors by the sub-space to which they belong. During search, the algorithm navigates the index to identify the sub-spaces closest to the query vector and examines all vectors within them.

Filtered ANNS. Existing approaches to filtered ANNS follow three main paradigms. *Metadata filtering* strategies, widely adopted in production vector databases [42, 45], use cost models to dynamically select optimal filtering strategies based on estimated selectivity. *Per-label indexing* constructs dedicated vector indexes for each label to eliminate query-time filtering overhead. While applying this approach naively incurs prohibitive memory overhead due to storing multiple copies of vectors, Parlay-IVF [21] addresses this limitation through shared vector storage across per-label indexes.

To further improve performance beyond these straightforward approaches, *specialized indexes* tailor both index construction and search algorithms for filtered ANNS. Fused distance approaches [43, 46] incorporate attribute similarity into vector distance to bias search toward qualified vectors, though they limit query expressiveness by requiring all attributes to be specified. Graph densification methods [12, 33] build high-degree graphs to ensure connectivity among qualifying vectors, then compress them to reduce memory usage, but face prohibitive construction costs at low selectivity. Subgraph stitching [5] constructs separate graphs for each unique label set and adds interconnecting edges, yet suffers from degraded index quality when these graphs are highly fragmented with few vectors in each.

Beyond categorical filtering, several works address *range filtering* on numeric attributes, where queries specify value ranges rather than discrete labels. ARKGraph [55] and SeRF [56] merge and compress per-range indexes to reduce storage overhead, while iRangeGraph [48] constructs graph indexes for fewer ranges and dynamically combines them at query time. DIGRA [18] and RangePQ [51] extend these approaches with dynamic update capabilities. Unlike Curator's focus on categorical label filtering, range filtering leverages the total order of numeric attributes to optimize for contiguous value ranges, representing a complementary but distinct problem domain.

8 Conclusion

This paper introduced Curator, a partition-based index that complements existing graph-based approaches for low-selectivity filtered ANNS. Curator employs hierarchical clustering to build specialized per-label indexes within a shared clustering tree, adapting clustering granularity to label distributions while maintaining low overhead through structural sharing. Curator supports both single-label search and complex predicate search by dynamically constructing temporary indexes with minimal overhead. Our evaluation demonstrates that integrating Curator with state-of-the-art graph indexes reduces low-selectivity query latency by up to 20.9× compared to pre-filtering fallback, while increasing construction time and memory footprint by only 5.5% and 4.3%, respectively. Curator's source code is publicly available at https://github.com/hatsu3/curator-v2.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This research was partially supported by NSF AI Institute on Edge Computing (Athena, grant No. 2112562) and NSF grants (2503010, 2402696, 2238665, 2402823), and by gifts from Adobe, Amazon, Meta, and IBM.

References

[1] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. *Advances in Neural Information Processing Systems* 28 (2015).

[2] Lars Arge. 1995. The buffer tree: A new technique for optimal I/O-algorithms. In *Workshop on Algorithms and Data structures*. Springer, 334–345.

[3] Artem Babenko and Victor Lempitsky. 2015. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence* 37, 6 (2015), 1247–1260.

[4] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 202–216.

[5] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. 2024. Navigating Labels and Vectors: A Unified Approach to Filtered Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–27.

[6] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[7] Colin B Clement, Matthew Bierbaum, Kevin P O'Keeffe, and Alexander A Alemi. 2019. On the use of arxiv as a dataset. *arXiv preprint arXiv:1905.00075* (2019).

[8] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[9] Cong Fu and Deng Cai. 2016. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv preprint arXiv:1609.07228* (2016).

[10] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment* 12, 5 (2019), 461–474.

[11] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*. 518–529.

[12] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *Proceedings of the ACM Web Conference 2023*. 3406–3416.

[13] Ben Harwood and Tom Drummond. 2016. FANNG: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5713–5722.

[14] Hugging Face. 2021. sentence-transformers/all-MiniLM-L6-v2 - Hugging Face. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2. Accessed: 2023-11-26.

[15] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355* (2018).

[16] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems* 32 (2019).

[17] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2011. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128.

[18] Mengxu Jiang, Zhi Yang, Fangyuan Zhang, Guanhao Hou, Jieming Shi, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[19] Yicheng Jin, Yongji Wu, Wenjun Hu, Bruce M. Maggs, Jun Yang, Xiao Zhang, and Danyang Zhuo. 2026. Curator: Efficient Vector Search with Low-Selectivity Filters. *arXiv preprint arXiv:2601.01291* (2026).

[20] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[21] Ben Landrum, Magdalen Manohar, Mazin Karjikar, and Laxman Dhulipala. 2023. ParlayANN's IVF. https://github.com/harsha-simhadri/big-ann-benchmarks/tree/main/neurips23/filter/parlayivf.

[22] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the International Conference on Machine Learning*. PMLR, 1188–1196.

[23] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[24] Yury Malkov and Dmitry Yashunin. 2018. hnswlib. https://github.com/nmslib/hnswlib. Accessed: 2023-11-26.

[25] Yu A Malkov and Dmitry A Yashunin. 2020. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2020), 824–836.

[26] Marius Muja and David G Lowe. 2009. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)* 2, 331-340 (2009), 2.

[27] Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence* 36, 11 (2014), 2227–2240.

[28] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005* (2017).

[29] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Marat Smelyanskiy. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[30] Pandu Nayak. 2019. Understanding searches better than ever before. https://blog.google/products/search/search-language-understanding-bert/ Accessed: 2023-11-26.

[31] Behnam Neyshabur and Nathan Srebro. 2015. On symmetric and asymmetric lshs for inner product search. In *International Conference on Machine Learning*. PMLR, 1926–1934.

[32] OpenAI. 2023. ChatGPT Retrieval Plugin. https://github.com/openai/chatgpt-retrieval-plugin. Accessed: 2023-11-26.

[33] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[34] Pinecone. 2023. Pinecone. https://www.pinecone.io/. Accessed: 2023-11-26.

[35] Qdrant. 2023. Qdrant. https://github.com/qdrant/qdrant. Accessed: 2023-11-26.

[36] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning transferable visual models from natural language supervision. In *Proceedings of the International Conference on Machine Learning*. PMLR, 8748–8763.

[37] Anshumali Shrivastava and Ping Li. 2014. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). *Advances in Neural Information Processing Systems* 27 (2014).

[38] Anshumali Shrivastava and Ping Li. 2015. Improved asymmetric locality sensitive hashing (ALSH) for maximum inner product search (MIPS). In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

[39] Harsha Simhadri, Martin Aumuller, Dmitry Baranchuk, Matthijs Douze, Amir Ingber, Edo Liberty, Frank Liu, and George Williams. 2023. NeurIPS'23 Competition Track: Big-ANN. https://big-ann-benchmarks.com/neurips23.html.

[40] Spotify. 2013. Annoy (Approximate Nearest Neighbors Oh Yeah). https://github.com/spotify/annoy. Accessed: 2023-11-26.

[41] Bart Thomee, David A Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. 2016. YFCC100M: The new data in multimedia research. *Commun. ACM* 59, 2 (2016), 64–73.

[42] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2614–2627.

[43] 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).

[44] Weaviate. 2023. Weaviate. https://github.com/weaviate/weaviate. Accessed: 2023-11-26.

[45] 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.

[46] 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.

[47] Hao Xu, Jingdong Wang, Zhu Li, Gang Zeng, Shipeng Li, and Nenghai Yu. 2011. Complementary hashing for approximate nearest neighbor search. In *2011 International Conference on Computer Vision*. IEEE, 1631–1638.

[48] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S Jensen. 2024. irangegraph: Improvising range-dedicated graphs for range-filtering nearest neighbor search. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–26.

[49] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 545–561.

[50] Belinda Zeng. 2022. Go beyond the search box: Introducing multisearch. https://blog.google/products/search/multisearch/

[51] Fangyuan Zhang, Mengxu Jiang, Guanhao Hou, Jieming Shi, Hua Fan, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[52] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*. 377–395.

[53] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep learning based recommender system: A survey and new perspectives. *ACM computing surveys (CSUR)* 52, 1 (2019), 1–38.

[54] Ting Zhang, Chao Du, and Jingdong Wang. 2014. Composite quantization for approximate nearest neighbor search. In *International Conference on Machine Learning*. PMLR, 838–846.

[55] Chaoji Zuo and Dong Deng. 2023. ARKGraph: All-Range Approximate K-Nearest-Neighbor Graph. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2645–2658.

[56] 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.

Received July 2025; revised October 2025; accepted November 2025