Elastic Index Selection for Label-Hybrid AKNN Search

Mingyu Yang
HKUST (GZ) & HKUST
myang250@connect.hkust-gz.edu.cn

Wenxuan Xia
HKUST (GZ)
wxia248@connect.hkust-gz.edu.cn

Wentao Li
University of Leicester
wl226@leicester.ac.uk

Raymond Chi-Wing Wong
HKUST
raywong@cse.ust.hk

Wei Wang*
HKUST (GZ) & HKUST
weiwcs@ust.hk

ABSTRACT

Real-world vector embeddings often carry additional label attributes, such as keywords and tags. In this context, **label-hybrid approximate k-nearest neighbor (AKNN)** search retrieves the top-k approximate nearest vectors to a query, subject to the constraint that their labels fully contain the query-label set. A naive solution builds a separate index for every query-label set, but the exponential growth of such sets makes this approach storage-prohibitive. To overcome this, we propose selectively indexing only a subset of query-label sets while still ensuring efficient processing for all queries. This is made possible by a key insight into label containment: an index built for a label set L can also serve any query whose label set L' is a superset of L, with query cost bounded by the elastic factor—the ratio between the number of vectors matching L and those matching L'. We formalize the index-selection task as a constrained optimization problem that chooses which label sets to index to satisfy space and query efficiency constraints. We prove the problem is NP-complete and propose efficient greedy algorithms for its efficiency- and space-constrained variants. Extensive experiments on real-world datasets show that our method achieves 10×−800× speedups over state-of-the-art techniques. Moreover, our approach is index-agnostic and can be seamlessly integrated into existing vector database systems.
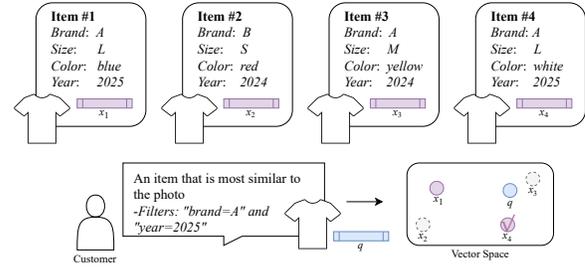
1 INTRODUCTION

The k-**nearest neighbor (KNN)** search over high-dimensional vectors is a core operation in modern data systems, underpinning a wide range of applications such as recommendation systems [42], data mining [6], face recognition [48], product search [48], and

*Corresponding author.

Figure 1: Example of label-hybrid AKNN search in an online shopping scenario ($k = 1$). Each item is represented by an embedding vector x_i ($i \in [1, 4]$) and associated with label attributes such as brand and year. A customer submits a reference photo q together with a query-label set L_q. The task is to find the item most similar to q that also satisfies the label constraint. Thus, x_2 and x_3 are discarded for label mismatch, and x_4 is returned as the nearest neighbor.

retrieval-augmented generation (RAG) for large language models [32]. In real-world deployments, vector embeddings are often associated with **label attributes**, such as product brands, keywords, and geolocations. For example, in an e-commerce setting, a user may search for items similar to a given photo while specifying label constraints like brand name and release year.

In this context, we introduce the problem of **label-hybrid k-nearest neighbor (KNN)** search, illustrated in Fig. 1. Unlike traditional KNN search, this variant considers both the vector embeddings and their associated label attributes. Formally, each entry in the database S consists of a vector embedding and a label set L. Given a query vector q and the label constraint (expressed as a query-label set L_q), the goal is to retrieve the top-k nearest neighbors of q in S, where each returned vector with label set L must contain all labels in the query-label set L_q, i.e., $L_q \subseteq L$. However, due to the curse of dimensionality [27], computing exact label-hybrid KNN in high-dimensional spaces is computationally expensive. Recent work therefore studies the **label-hybrid approximate KNN (AKNN)** problem, which efficiently returns the top-k approximate neighbors satisfying the label constraint, trading a small amount of accuracy for substantial performance gains [5, 40].

Existing Solutions. To support label-hybrid AKNN search, existing approaches often employ graph-based indexes [10, 11, 18, 19, 25, 29, 33, 34, 37, 41, 44, 46, 55] combined with filter-based search strategies, due to their good search efficiency. Specifically, the graph-based index is constructed by adding base vectors in database S as nodes and connecting each node to its nearby neighbors to form edges, which are carefully selected to support efficient navigation. At query time, on top of the graph-based index, filter-based
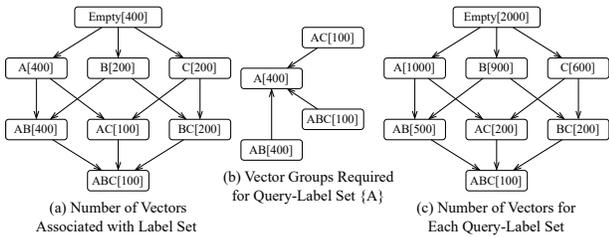
Figure 2: Motivating example where database S and queries draw labels from the alphabet $\{A, B, C\}$. Base vectors in S are grouped by their associated label sets L, each group denoted as $\{L\}[count]$. Arrows connect a group L to groups with its minimal supersets. For instance, Fig. 2(a) shows 400 vectors labeled only $\{A\}$, written as $A[400]$. Given an AKNN query with $L_q = \{A\}$, Fig. 2(b) highlights all relevant groups—those whose label sets are supersets of $\{A\}$—namely $\{A, AB, AC, ABC\}$, totaling $1,000$ matching vectors that must be indexed under $L_q = \{A\}$. Fig. 2(c) summarizes the index sizes required for each query-label set. Supporting all possible query-label sets in the workload requires indexing $5,400$ vectors—about $2.75\times$ the original dataset size.

search strategies [18] are applied to enforce label constraints and retrieve the final results. In particular, two strategies—PreFiltering and PostFiltering—are integrated into the graph-based index. The PreFiltering strategy prunes nodes (i.e., base vectors) that do not satisfy the query label constraints, removing both the nodes and their connections to neighbors during the search. In contrast, the PostFiltering strategy retains all nodes and connections, but excludes any label-mismatched nodes from being added to the results.

A base vector **matches** the query if its label set contains the query-label set, and the selectivity of a query is the number of its label-matched vectors. The performance of both strategies drops sharply when selectivity is low. (1) PreFiltering suffers from reduced accuracy as removing many label-mismatched nodes sparsifies the graph, making it difficult to reach the target vectors. (2) While PostFiltering retains all nodes, it still requires computing distances between the query and many mismatched vectors, causing inefficiency. To address these challenges, existing solutions such as Milvus [43], ADB [48], VBASE [57], and CHASE [35] dynamically select between PreFiltering and PostFiltering based on the query workload and cost estimation. Yet, the limitations of these search strategies persist. Some heuristic approaches, including NHQ [44] and HQANN [49], propose fusion distances that combine both vector similarity and label matching between the query and base vectors. These methods require manual tuning of vector–label weight parameters and still perform well below the state-of-the-art.

State-of-the-Art. To support label-hybrid AKNN search, ACORN [40] and UNG [5] represent the state-of-the-art. ACORN extends the PreFiltering strategy to address connectivity issues under low selectivity by introducing a parameter γ that increases the graph density—each node has γ times more outgoing edges than in a standard index. This denser graph improves robustness when filtering out label-mismatched nodes during query processing. However, ACORN ignores label information during index construction, leading to potential result incompleteness.

To address this, UNG exploits label containment during graph construction. It partitions base vectors into groups by their label

Table 1: We compare our method with existing solutions along three dimensions. Search Performance measures accuracy and efficiency, validated experimentally. Index Flexibility captures the ability to support label-hybrid AKNN search without dependence on a specific index structure. Space–Performance Trade-offs assess how well the method balances indexing cost and storage efficiency.

Feature	ELI (Our)	NHQ	UNG	ACORN	Filtered
Search Performance	★★★	★	★$^{\uparrow}$	★$^{\uparrow}$	★
Index Flexibility	✓	✗	✗	✗	✓
Space–Performance Trade-offs	✓	✗	✗	✗	✗

sets and builds a subgraph for each group. Subgraphs for a label set L_q are connected to those of its minimal supersets via cross-group edges, ensuring that any vector with a superset of L_q is reachable from the group of L_q, thereby guaranteeing completeness. However, both ACORN and UNG are limited to graph-based indexes and thus lack *index flexibility*. Their performance also degrades when query selectivity is low. Experiments show significant deterioration as the size of the alphabet grows, and neither method can *flexibly adjust the index structure* to meet tight storage constraints.

Motivation. Given an alphabet Σ from which both the database S and queries draw labels, we may face an exponential explosion of possible label sets. Specifically, let $\mathcal{L}$ be the set of query-label sets in the workload (a subset of the power set of Σ). A naive strategy for AKNN queries is to build a separate index for each query-label set $L_q \in \mathcal{L}$, where the index for L_q stores all vectors whose label sets contain L_q. Under this scheme, a vector with label set L must be inserted into every index whose query-label set is a subset of L. As illustrated in Fig. 2, a vector labeled $\{ABC\}$ must appear in eight indexes: $\{\emptyset, A, B, C, AB, AC, BC, ABC\}$, where $\emptyset$ denotes the top index with no label constraint ($L_q = \emptyset$). This replication creates substantial overhead: empirical studies show that for $|\Sigma| = 6$–10, the total index entries can be $64\times$–$1024\times$ larger than the original database, leading to prohibitive indexing and storage costs.

Our Solution. Indexing every label set in $\mathcal{L}$ is impractical due to the sheer number of indexes and the resulting storage cost. A natural question arises: can we *index only a subset of* $\mathcal{L}$ while still enabling efficient label-hybrid AKNN search? This is non-trivial—if a query-label set $L \in \mathcal{L}$ is not indexed, queries with label set L may be inefficient to process. Fortunately, label containment provides a key insight: an index built for L can also serve any query whose label set L' is a superset of L. We further show that the query latency is bounded by an *elastic factor*—the ratio of vectors matching L' to those matching L. Note that a higher elastic factor implies better search efficiency. For example, in Fig. 2(c), all vectors matching $\{AB\}$ are contained in those matching $\{A\}$, with an overlap ratio (or elastic factor) of at least 0.5. Thus, the index on $\{A\}$ can efficiently answer queries for $\{AB\}$. The index on $\{B\}$ has an even higher overlap with $\{AB\}$, yielding greater efficiency. We therefore formalize the problem of **index selection** as choosing a subset of $\mathcal{L}$ to index while bounding both space usage and the elastic factor.

We first show that the decision version of the index-selection problem is NP-complete. We then formulate two optimization variants: (1) an **efficiency-constrained** version that minimizes index space while guaranteeing a lower bound on the elastic factor (search

efficiency), and (2) a **space-constrained** version that maximizes the elastic factor under a fixed space budget. We propose a greedy algorithm for the first variant and extend it to the second, and we further describe how to update indexes in dynamic settings. Overall, our methods selectively index label sets to control space while maintaining high elastic factors for efficient label-hybrid AKNN search. Compared with existing approaches (see Table 1), our solution provides greater flexibility, higher search efficiency, and superior space–performance trade-offs.

Contribution. We summarize our main contributions as follows:

Problem Analysis (§ 3). We identify the limitations of existing label-hybrid AKNN solutions: limited flexibility in the underlying indexes and suboptimal query performance. To address these issues, we measure search performance using the *elastic factor*, which quantifies how an index built on a query-label set L can support queries over its supersets, motivating our novel index-selection problem.

Index Selection Problem Formulation (§ 3). We formally define the decision version of the index-selection problem: given a query workload with label sets $\mathcal{L}$, can we choose a subset of $\mathcal{L}$ so that the total index space stays within a specified bound and the elastic factor exceeds a given threshold? We prove this problem is NP-complete, showing that exact solutions are intractable.

Index Selection Problem Solutions (§ 4). Building on the decision version, we define two practical optimization variants of the index selection problem: (1) the efficiency-constrained variant, which minimizes space given a lower bound on the elastic factor; and (2) the space-constrained variant, which maximizes the elastic factor under a space limit. These formulations enable users to balance space and query efficiency. We design a greedy algorithm for the first variant and extend it to solve the second.

Extensive Experiments (§ 5). We evaluate our algorithm on multiple datasets with diverse label distributions associated with the base vectors. Experiments show that our approach attains near-optimal search efficiency with only a 100% space overhead and outperforms competitors on large-scale datasets with large alphabets, achieving $10\times$–$800\times$ speedups over state-of-the-art baselines.

Due to space constraints, some proofs and experiments are omitted, which can be found in our technical report [54].

2 PRELIMINARY

We formally define the label-hybrid approximate k-nearest neighbor search problem in § 2.1. We then review existing solutions in § 2.2. We list the commonly used notations in the Table 2.

2.1 Problem Statement

Each entry (x_i, L_i) in the dataset S consists of a d-dimensional base vector $x_i \in \mathbb{R}^d$ and an associated label set L_i, which may be empty, where all possible labels are from the alphabet Σ. A label-hybrid search is denoted by a query (q, L_q), where q is the query vector and L_q is the query-label set. When L_q is specified, we first filter the dataset S to obtain a candidate subset $S(L_q) \subseteq S$, defined as: $S(L_q) = \{(x_i, L_i) \in S \mid L_q \subseteq L_i\}$, i.e., the label-matched vectors whose label sets contain all labels in L_q. The label-hybrid k-nearest neighbor search is performed over this candidate subset. Formally,

Table 2: Summary of Notations

Notation	Description
S	Set of vectors, each with an associated label set
L_i, L_q	Base and query label sets
$S(L_q)$	Subset of S whose label sets contain L_q
$|L|$	Cardinality of label set L
$\mathcal{L}$	Collection of all label sets
$|\mathcal{L}|$	Number of label sets in $\mathcal{L}$
$\mathcal{I}$	Collection of indexes
$\mathbb{L}$	The selected indexes from $\mathcal{L}$
Σ	Label alphabet (all possible labels)
N	Cardinality of S
q	Query vector
e	Elastic factor
d	Dimensionality of vectors in S
$\delta(u, v)$	Distance (or similarity) between u and v
I	AKNN search index (e.g., HNSW)
τ	Space budget
c	Elastic-factor threshold

Definition 2.1 (Label-Hybrid KNN Search). Given a dataset S and a query (q, L_q), label-hybrid KNN search returns a set $S' \subseteq S(L_q)$ of k entries such that for every $(x_i, L_i) \in S'$ and every $(x_j, L_j) \in S(L_q)$, $\delta(x_i, q) \leq \delta(x_j, q)$, where $\delta(\cdot, \cdot)$ denotes the vector distance (We use the Euclidean distance as an example).

Exact k-nearest neighbor search often suffers from the *curse of dimensionality* [27], causing traditional methods [3, 4, 39] that perform well in low-dimensional spaces to degrade greatly in high-dimensional settings. As a result, approximate k-nearest neighbor (AKNN) search has been extensively studied [1, 7, 10, 11, 13–17, 29, 30, 33, 37, 45, 47, 51, 53, 58] for its ability to improve efficiency at the cost of some accuracy. This challenge also arises in the context of label-hybrid search. Therefore, we focus on the label-hybrid approximate k-nearest neighbor (AKNN) problem. To evaluate the result quality, we use **recall** as the metric, defined as recall = $|\hat{S} \cap S'|/|S'|$, where S' is the exact result set and $\hat{S}$ is the approximate result set with $|\hat{S}| = k$. An effective label-hybrid AKNN solution should balance high search efficiency with high recall.

Remark. Label-hybrid AKNN search can be viewed as a special case of filtered nearest-neighbor search [18], where our filtering requires the label set L_i of a base vector to contain the query-label set L_q. Other variants—such as set intersection ($L_q \cap L_i \neq \emptyset$) or set equality ($L_q = L_i$)—can be addressed using the solutions developed for our studied AKNN search problem. A detailed analysis appears in our technical report [54].

2.2 Existing Solutions

To address the label-hybrid AKNN search problem, existing approaches typically combine different types of AKNN indexes with filter-based search strategies. Among these, graph-based indexes [11, 25, 29, 34, 37, 41] are widely adopted due to their state-of-the-art search efficiency. In such indexes, base vectors in the dataset S are treated as nodes in a graph, where each node is carefully connected to its nearby vectors, enabling efficient navigation.
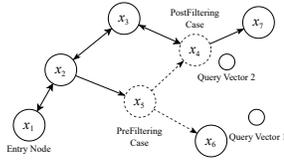
Figure 3: The Example of Filter-Based Search

Graph Index. When a graph is used as an index, search starts from a designated entry node and iteratively explores neighbors that are progressively closer to the query until the top-1 result is found. A key index design component is the edge-occlusion strategy, which guides the search toward the query while keeping the graph sparse and the node degree bounded by a constant [11, 28]. For AKNN search—where more than one result is required—beam search is typically employed. It maintains the top-m closest candidates during traversal, where m is the beam width. Under ideal indexing conditions (i.e., high index quality and the query present in the dataset) [11, 28], only one extra expansion step on the graph is needed to retrieve the next neighbor, yielding an overall search complexity logarithmic in the dataset size.

Filter-Based Search. On top of the graph index, we introduce two filter-based search strategies—PreFiltering and PostFiltering—to support label-hybrid AKNN search. Both strategies maintain the original index structure and require only label-based filtering during graph traversal. (1) PreFiltering strategy: Given a query (q, L_q), PreFiltering prunes unmatched nodes and their neighbors during traversal. As shown in Fig. 3, suppose nodes x_4 and x_5 do not satisfy the label constraint in query 1. In this case, PreFiltering removes the outgoing edges of x_5, thereby making the true nearest neighbor x_6 unreachable from the entry node x_1. (2) PostFiltering strategy: This method allows traversal through unmatched nodes but excludes them from the result set. For instance, when processing query 2 in Fig. 3, PostFiltering visits nodes x_1, x_2, x_3, x_4, and x_7. Although x_4 is label-unmatched, its outgoing edge is still used for routing, enabling the algorithm to eventually reach and return x_7.

For top-k queries, PostFiltering traverses incrementally, visiting successive nearest neighbors until k matching results are found. However, if most nodes are label-mismatched, the algorithm may still need to examine up to $O(N) = O(|S|)$ nodes to obtain k results, because these filtered-out nodes must still be visited for PostFiltering. PreFiltering faces a similar challenge—under low selectivity, it may fail to reach any valid neighbors from the entry node, making its correctness and efficiency difficult to guarantee.

3 ELASTIC INDEX SELECTION PROBLEM

This section formally defines the index selection problem, aiming to overcome the exhaustive cost in indexing space and time caused by exponential label combinations. In particular, we first introduce the elastic factor, which captures how an index built on a query-label set can be reused for its supersets. We then formally define the index selection problem as a decision problem: determining whether it is possible to select a subset of label sets for indexing such that both space usage and query efficiency remain within given bounds. Finally, we prove that the problem is computationally hard.

3.1 Elastic Factor for Index-Sharing

Given an alphabet Σ, one could in principle build an index for every label set in the power set of Σ, but the number of such sets is exponential. Fortunately, index sharing can dramatically reduce the required indexes. Specifically, an index built on a query-label set L_2 can serve as a substitute for another query-label set L_1 if it contains all base vectors associated with L_1. This holds when $L_2 \subseteq L_1$, because every vector matching L_1 also matches its subset L_2. For example, the group labeled $\{A\}$ includes all entries from its superset group $\{AB\}$, since any vector labeled $\{AB\}$ also contains label $\{A\}$. Thus, a query with label set $\{AB\}$ can be answered using an index built on $\{A\}$, allowing indexes for some query-label sets to be omitted and reducing storage costs using label containment.

Elastic Factor. We next analyze the query efficiency provided by shared indexes. For simplicity, assume we already have a subset of indexes $\mathbb{I} = \{I_1, \ldots, I_m\}$ built on a selected subset of possible query-label sets (we discuss how to choose this subset later). Each index $I_i \in \mathbb{I}$ contains the base vectors in S that match some query-label set L_i; we sometimes use I_i to refer directly to these label-matched vectors. An index can answer a query with label set L_q only if it contains all base vectors matching L_q, following the label-containment rule described earlier. Intuitively, if the set $S(L_q)$ of vectors matching the query-label set L_q is fully contained in some index $I \in \mathbb{I}$ and overlaps heavily with I, query efficiency is high because few label-mismatched vectors remain. This intuition motivates the *elastic factor*: the maximum overlap ratio, across all indexes in $\mathbb{I}$ that can answer query q, between $S(L_q)$ and the index used.

Definition 3.1 (Elastic Factor). Given a label-hybrid dataset S, a query (q, L_q), and an index set $\mathbb{I} = \{I_1, \ldots, I_m\}$ where each $I_i \subseteq S$, the *elastic factor* of $\mathbb{I}$ with respect to L_q is defined as:

$$e(S(L_q), \mathbb{I}) = \max_{S(L_q) \subseteq I_i} \frac{|S(L_q)|}{|I_i|}.$$

We only use indexes I_i satisfying $S(L_q) \subseteq I_i$, as they contain all vectors matching L_q and can thus answer query q without omission. The *elastic factor* is the maximum overlap ratio $|S(L_q)|/|I_i|$ between $S(L_q)$ and any such index $I_i \in \mathbb{I}$. When $\mathbb{I}$ contains a single index I_i, we also denote the elastic factor between L_q and I_i as $e(S(L_q), I_i)$. To guarantee a non-zero elastic factor, we include a top index $I_\top$ built over all base vectors in S without label filtering, ensuring $S(L_q) \subseteq I_\top$. The elastic factor lies in $(0, 1]$: it is 1 when an index is built exactly on the query-label set L_q and decreases toward 0 when the chosen index contains more label-mismatched vectors.

EXAMPLE 1. *Figure 4(a) lists the vector groups with their associated label sets in the database S. Three vectors have no labels (Empty[3]) and three have the label set $\{ABC\}$[3]. Figure 4(b) shows the number of vectors matched by each possible query–label set. For example, a query with label set $L_q = \{A\}$ matches all vectors labeled $\{A\}$, $\{AB\}$, $\{AC\}$, or $\{ABC\}$, totaling 10 vectors. Building index I_2 on these 10 vectors (label set $\{A\}$) supports not only $L_q = \{A\}$ but also queries such as $L_q = \{AB\}$, since it contains all required base vectors. Figure 4(c) illustrates index sharing with an elastic factor of 0.3. Index I_2 can answer the query $L_q = \{ABC\}$ because the overlap ratio is $3/10 = 0.3$, whereas index I_1 (label set $\emptyset$) cannot, as its overlap ratio $3/17$ is below the threshold.*
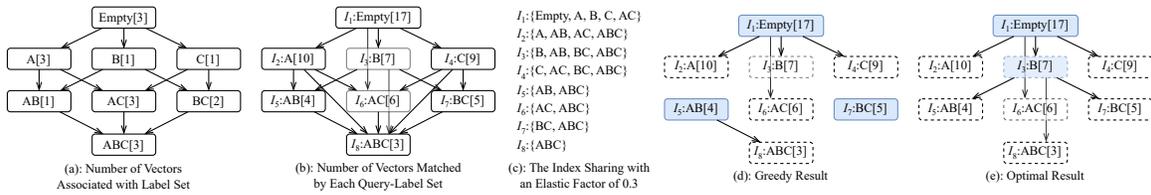
(a): Number of Vectors Associated with Label Set

(b): Number of Vectors Matched by Each Query-Label Set

(c): The Index Sharing with an Elastic Factor of 0.3

(d): Greedy Result

(e): Optimal Result

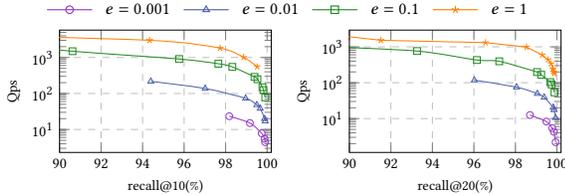**Figure 4: The Running Example of Elastic Factor and Greedy Methods**

Figure 5: The effect of the elastic factor on query efficiency (k = 10: left, k = 20: right). We randomly generate label sets for both base and query vectors and build HNSW indexes on the SIFT100M dataset. Queries are grouped by elastic factor—0.001, 0.01, 0.1, and 1—with e = 1 serving as the optimal baseline.

Connection to Query Efficiency. In the extreme, one could perform filter-based search (e.g., PostFiltering) on the top index $I_\top$, which contains all entries and whose label set (∅) is a subset of every query-label set. However, this is inefficient for queries with non-empty L_q because the search must traverse many label-mismatched nodes in the graph index. The inefficiency arises because the elastic factor of $I_\top$ is typically very low for such queries. To raise the elastic factor, additional indexes must be added to $\mathbb{I}$. The key question then becomes: when the elastic factor of $\mathbb{I}$ for a query q is at least c, what is the resulting query cost?

To answer this, we first analyze the expected cost of KNN search (ignoring label constraints). To retrieve the k nearest neighbors, we first use the index to locate the top-1 result and then incrementally continue the search from each previously discovered neighbor until all k results are obtained (e.g., extending from top-1 to top-2, and so on). Thus, the efficiency of KNN search depends on the expected number of steps required to obtain k valid results. When using an index built over the entire dataset S (as in PostFiltering), the expected number of steps $\mathbb{E}(r)$ can grow to $O(N)$ under low query selectivity, where $N = |S|$, explaining its inefficiency. In contrast, if the query is answered using an index with a constant elastic factor c, the expected number of steps to obtain k results is bounded by $O(k/c)$ [40, 52]. This bound holds because, in graph-based indexes, retrieving the next nearest-neighbor/result typically requires only one additional expansion step after locating the current result, under ideal conditions (e.g., slow growth or when the query is contained in the dataset). Extending this reasoning to our label-hybrid AKNN search, we still first locate the top-1 nearest neighbor and then continue the search until all k results *satisfying the label constraint* are retrieved. We show that, apart from the cost of finding the first neighbor, the additional cost of retrieving the remaining results remains $O(k/c)$, as formalized in Lemma 3.2.

LEMMA 3.2. *Given a dataset S, a query (q, L_q), and a selected index set $\mathbb{I}$, let $O(C)$ denote the expected time to retrieve the top-1 neighbor from a graph index. If the elastic factor satisfies $e\left(S(L_q), \mathbb{I}\right) \geq c$ for some constant $c \in (0, 1]$, then the expected time to obtain the top-k label-hybrid AKNN results with PostFiltering is $O\left(C + \frac{k}{c}\right)$.*

Proof sketch. We make two assumptions. (i) After incurring a cost of $O(C)$ to locate the top-1 nearest neighbor, each successive nearest neighbor can be retrieved in amortized constant time. Note that we do not require the visited neighbors to satisfy the query's label constraints, and the property holds for most graph indexes. (ii) Conditioned on distance ordering, at least a fraction c of the visited nearest neighbors satisfy the query's label constraints, guaranteed by $e(\cdot, \cdot) \geq c$. Since PostFiltering outputs a vector only if it matches the query label set, more than k neighbors may need to be examined. Let T denote the number of (steps to examine) neighbors until k matches are found. T follows a negative-hypergeometric (or geometric) distribution with success probability at least c, which yields $\mathbb{E}[T] \leq k/c$. Including the cost $O(C)$ for locating the first neighbor, the total expected cost is bounded by $O(C + k/c)$.

Remark on Lemma 3.2. The above result provides an *expected* bound under two standard assumptions: (1) locating the top-1 nearest neighbor costs $O(C)$, and enumerating additional neighbors requires amortized constant work; (2) Label-matched and mismatched vectors are assumed to be uniformly distributed in the graph and independent of the labels, ensuring that at least a c-fraction of visited vectors satisfy the label constraint (as guaranteed by the elastic factor). This bound is not a worst-case guarantee, as these assumptions may not always hold for graph indexes such as HNSW. Nevertheless, the elastic factor c remains a useful control knob for expected performance, because the additional time cost scales directly with k/c. Our empirical studies (Fig. 5) confirm this monotonic relationship: larger elastic factors yield higher query throughput (Qps) at fixed recall, and the observed performance drop as c decreases.

Empirical studies. To further validate Lemma 3.2, we measure the practical efficiency of PostFiltering search under varying elastic factors on the 100M subset of the SIFT1B dataset [2], denoted SIFT100M. In the classical PostFiltering strategy, the elastic factor is fixed as it uses only the top index containing all base vectors. Here, we add more indexes to $\mathbb{I}$ to vary the elastic factor and report results in Fig. 5. As shown, higher elastic factors yield better efficiency (measured in Qps), consistent with the lower time complexity predicted by theory. When the factor reaches 1, search achieves optimal efficiency—equivalent to indexing only the label-matched data. Efficiency decreases sublinearly as the elastic factor drops: for example, reducing the factor to $\frac{1}{10}$ lowers throughput by only 2× at 98% recall with k=10. This occurs because the added cost scales
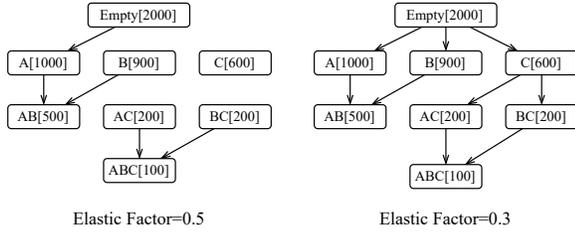
Figure 6: The Example of the EIS Problem

with k, while the top-1 search still runs in $O(C)$, usually $O(\log N)$. The gap widens for larger k (e.g., a 3× slowdown at k=20 and 98% recall), but in most applications k is small relative to N. Thus, as long as the elastic factor remains constant, the overall time cost stays bounded.

3.2 Problem Definition

We have analyzed the impact of the elastic factor on search efficiency for a *given* index set $\mathbb{I}$. When using $\mathbb{I}$, the expected query cost is directly tied to the elastic factor of $\mathbb{I}$ for each query. If the elastic factor for every query in the workload is guaranteed to exceed a constant lower bound c, then the AKNN search cost is bounded by an additional $O(k/c)$. This establishes a clear relationship between index cost (the total size of indexes in $\mathbb{I}$) and query efficiency (captured by the elastic factor). Building on this, among *all subsets of indexes* that can correctly answer the query workload, we seek a subset $\mathbb{I}$ that is both small and yields a large elastic factor, ensuring strong query performance. This motivates the formal definition of the **Elastic Index Selection** (EIS) problem.

Definition 3.3 (Fixed Efficiency Index Selection (EIS)).

Input A label-hybrid dataset S, a query workload with label sets $\mathcal{L} = \{L_1, \ldots, L_n\}$, and a universal index set $\mathcal{I} = \{I_1, \ldots, I_n\}$, where each index $I_i = S(L_i)$ is built over the base vectors matching the query-label set L_i with cost $|I_i|$. Let τ and c be non-negative real numbers.

Output Decide whether there exists a subset $\mathbb{I} \subseteq \mathcal{I}$ such that $e(S(L_i), \mathbb{I}) \geq c$ for every $L_i \in \mathcal{L}$ and the total cost satisfies $\sum_{I \in \mathbb{I}} |I| \leq \tau$.

The EIS problem is a *decision* problem: determine whether a subset of indexes from the universal set $\mathcal{I}$ can be chosen such that (1) the total space usage is within a user-specified bound τ, and (2) the elastic factor—which governs query efficiency—is at least a given threshold c. In Section 4, we present the *optimization* version, where the goal is either to minimize space cost while satisfying a required elastic-factor threshold or to maximize the elastic factor subject to a space budget.

EXAMPLE 2. *The EIS problem under different elastic-factor thresholds, $c = 0.5$ and $c = 0.3$ is in Fig. 6. When $c = 0.5$, the top index $I_\top$ (built on the empty label set) can serve queries with $L_q = \{A\}$ and $L_q = \emptyset$, since their overlaps exceed 0.5. It cannot, however, support $L_q = \{B\}$, whose overlap is only 0.45. With a more relaxed threshold of $c = 0.3$, $I_\top$ also supports $L_q = \{B\}$ and $L_q = \{C\}$, but it still cannot answer $L_q = \{AB\}$, whose overlap is 0.25, unless $c < 0.25$. Finally, when $c = 0$, the top index can serve all possible queries.*

Remark on Query Workload. In the problem definition, we assume a query workload with query-label sets $\mathcal{L}_q = \{L_1, \ldots, L_n\}$. This assumption is justified for two reasons: (1) It generalizes the traditional setting where $\mathcal{L}_q$ includes all subsets of the alphabet Σ (i.e., the power set). (2) It enables a workload-driven approach: $\mathcal{L}_q$ can reflect historical query patterns, allowing index selection and optimization to match realistic query behavior.

Although the total number of possible query-label sets can reach $2^{|\Sigma|}$ in the worst case, the effective number is typically far smaller due to mutual exclusivity among certain labels. For example, a product has only one brand name, making brand labels mutually exclusive. Our formulation explicitly captures such cases, ensuring that incompatible label combinations—like different brand names—do not co-occur in $\mathcal{L}_q$. We also assume the workload always includes the empty set $\emptyset$, representing queries with no label constraints and thus matching all base vectors. For simplicity, we treat the cost of this top index as zero, i.e., $|I_{\text{top}}| = 0$.

Remark on Index Size/Cost. The cost of each index I_i (built for query-label set L_i) reflects the space it occupies. For graph-based indexes, this cost can be approximated by the number of vectors it contains, since node degree is typically bounded by a constant [11, 28]. In practice, each node maintains M edges to support efficient memory access, where M is a user-defined parameter [36]. Thus, the overall space usage of an index can be estimated as the product of its vector count and M. In our study, however, we measure index size/cost $|I|$ simply by the number of vectors it contains.

Index Selection at Query Time. At query time, a label set L_q may be served by multiple indexes in the selected set $\mathbb{I}$. For example, a query with $L_q = \{AB\}$ can be answered by the index for $\{A\}$ or for $\{B\}$ if both are present in $\mathbb{I}$. In such cases, our method selects the index that yields the maximum elastic factor with $S(L_q)$. This choice improves query efficiency because greater overlap means fewer label-mismatched vectors in the index, reducing the cost of scanning irrelevant entries.

3.3 Problem Hardness

We prove that the EIS problem is NP-complete in Theorem 3.4, underscoring the computational challenges inherent in solving it.

THEOREM 3.4. *The EIS problem is NP-complete.*

Proof sketch. We show EIS is NP-complete by reducing from the 3-Set Cover (3-SC) problem [8, 31]. EIS is in NP as validity is verifiable in polynomial time. Given a 3-SC instance with elements $\mathcal{U}$, subsets $\mathcal{S}$, and budget k, we construct an EIS instance with a universal index set $\mathcal{I}$ containing: (1) an element index I_{u_i} for each $u_i \in \mathcal{U}$; (2) a subset index I_{s_j} for each $s_j \in \mathcal{S}$; (3) a top index $I_\top$ covering all base vectors; and (4) a bottom index I_{all}.

Let $b > 3$. We set the index sizes to $|I_{u_i}| = A = b+1$, $|I_{s_j}| = B = 2b$, $|I_\top| = N > 4b$, and $|I_{all}| = b$. We set the elastic factor $c = 2b/N$ and the cost budget $\tau = k \cdot B$. The construction ensures the following overlap properties: (1) $I_\top$ covers every subset index I_{s_j} with ratio $B/N = c$, but fails to cover element indexes I_{u_i} sufficiently (ratio $A/N < c$). (2) A subset index I_{s_j} covers an element index I_{u_i} with ratio $A/B > c$ if and only if $u_i \in s_j$. (3) To enforce cost constraints, we introduce duplicate elements u_i' such that covering a specific element u_i directly via element indexes requires selecting both I_{u_i}

and $I_{u'_i}$ (total cost $2A = 2b + 2$). However, covering u_i via a relevant subset index I_{s_j} costs only $B = 2b$. Since $2A > B$, it is strictly cheaper to cover elements using subset indexes. The detailed proof is available in our technical report.

The EIS problem is a decision problem with a yes/no answer. In practice, however, one often seeks to optimize either space usage or query time, as these are the key factors in deploying AKNN search. We therefore study two optimization variants of EIS: (1) **Efficiency-constrained variant** that minimizes total space while enforcing a lower bound on the elastic factor and (2) **Space-constrained variant** that maximizes the elastic factor subject to a given space budget. Because EIS is NP-complete, both variants are NP-hard. We next show how to address these variants.

4 PROBLEM SOLUTION

We first solve the efficiency-constrained EIS problem, then extend it to the space-constrained variant, and conclude with a discussion.

4.1 Method for Efficiency-constrained Variant

This optimization version of the EIS problem resembles Definition 3.3, but aims to compute the following output:

Output A subset $\mathbb{I} \subseteq \mathcal{I}$ such that $e(S(L_i), \mathbb{I}) \geq c$ for all $L_i \in \mathcal{L}$, and the total cost $\sum_{I \in \mathbb{I}} |I|$ is minimized.

Since this variant of the EIS problem is NP-hard, we employ a greedy algorithm to obtain an approximate solution. The basic idea is as follows. Given a collection of indexes $\mathcal{I}$ (corresponding to a query workload with label sets $\mathcal{L}$), each index $I \in \mathcal{I}$ has an associated space cost $|I|$, measured by the number of matched vectors it contains. We define a benefit function for each index and iteratively select the index with the highest benefit. By definition of the elastic factor, adding more indexes improves it and eventually covers all label sets in $\mathcal{L}$ (if all are selected). Yet, to minimize total cost, selection must be made carefully to ensure solution quality. A key preprocessing is that we always include the top index $I_\top$ in the solution. As noted in § 3.2, this guarantees that every query has at least one index covering all of its matched vectors, ensuring correctness. Thus, the answer set $\mathbb{I}$ is initialized with $I_\top$. The benefit of adding a candidate index $I \in \mathcal{I} \setminus \mathbb{I}$ to the current solution $\mathbb{I}$ is defined as $B(I, \mathbb{I})$, which is defined below.

Definition 4.1. Let $\mathbb{C} \subset \mathcal{I}$ be the set of indexes already covered by the selected set $\mathbb{I}$ with overlap ratio at least c, where c is the elastic–factor threshold: $\mathbb{C} = \{ I_i \in \mathcal{I} \mid \exists I_j \in \mathbb{I} : (I_i \subseteq I_j) \wedge (|I_i|/|I_j| \geq c) \}$. Let $\mathbb{C}'$ be the set of indexes newly covered by adding I, also with overlap ratio at least c: $\mathbb{C}' = \{ I_i \in \mathcal{I} \setminus \mathbb{C} : (I_i \subseteq I) \wedge (|I_i|/|I| \geq c) \}$. The benefit of adding I to $\mathbb{I}$ is $B(I, \mathbb{I}) = \sum_{I_i \in \mathbb{C}'} \frac{|I_i|}{|I|}$.

The benefit captures two aspects: (i) the number of previously uncovered indexes that the candidate index I can cover (with overlap ratio at least c, i.e., the set $\mathbb{C}'$), and (ii) the actual overlap ratio between I and each newly covered index $I_i \in \mathbb{C}'$. In other words, the benefit is the total overlap ratio of I with all indexes in $\mathbb{C}'$.

Algorithm. We implement the greedy method in Algorithm 1. First, we add the top index $I_\top$ to the answer set $\mathbb{I}$ (Line 1). At each iteration, we select the next candidate index—each candidate I corresponds to a query–label set L_q in the workload—based on its benefit score

---

Algorithm 1: Greedy Algorithm

Input: The Universal Index Set $\mathcal{I}$, The Elastic Factor c
Output: The Selected Index Set $\mathbb{I}$

1   $\mathbb{I} \leftarrow \{I_\top\}$;
2   **while** *Stop Condition Not Met* **do**
3     $I \leftarrow I \in \mathcal{I} \setminus \mathbb{I}$ such that $B(I, \mathbb{I})$ is maximum;
4     $\mathbb{I} \leftarrow \mathbb{I} \cup \{I\}$;
5   **if** For all $L_q \in \mathcal{L}$, $e(S(L_q), \mathbb{I}) \geq c)$ **then Break**;
6   **return** $\mathbb{I}$;   // Selected Index Set

Table 3: The Benefits of Each Candidate Index in Each Round

	Init Round	Second Round	Third Round
I_1	49 / 17 = 2.88		
I_2	23 / 10 = 2.30	7 / 10 = 0.70	0 / 10 = 0.00
I_3	19 / 7 = 2.71	12 / 7 = 1.71	5 / 7 = 0.71
I_4	23 / 9 = 2.55	14 / 9 = 1.55	5 / 9 = 0.55
I_5	7 / 4 = 1.75	7 / 4 = 1.75	
I_6	9 / 6 = 1.50	3 / 6 = 0.50	0 / 6 = 0.00
I_7	8 / 5 = 1.60	8 / 5 = 1.60	5 / 5 = 1.00
I_8	3 / 3 = 1.00	3 / 3 = 1.00	0 / 3 = 0.00

(Lines 3–4). The process stops when all label sets in $\mathcal{L}$ are covered, yielding a valid solution (Line 5).

EXAMPLE 3. *Using the example dataset in Fig. 4, we evaluate the benefit of different index choices, with Table 3 reporting the benefit ratio for each candidate index. Note that the top index $I_1 = I_\top$ is always selected first, regardless of its benefit. Selecting I_1 covers the vectors in I_2, I_3, I_4, I_6, and itself (with overlap ratio at least the elastic–factor threshold $c = 0.3$), for a total of $17 + 10 + 7 + 9 + 6 = 49$ vectors at a cost of 17, yielding a benefit of $49/17 \approx 2.88$. After I_1 is selected, the benefits of the remaining candidates change. Initially, I_2 covers I_2, I_5, I_6, and I_8, i.e., 23 vectors at a cost of 10, for a benefit of 2.3. In the second round, as I_1 already covers I_2 and I_6, I_2 can now cover only I_5 and I_8, reducing its benefit to 0.7. The greedy algorithm next selects the index with the highest updated benefit (Fig. 4(d)), which is I_5. At this point, I_1 and I_5 cover all vectors except those in I_7. The third round selects I_7, with a cost of 5. The greedy solution, therefore, has a total cost of $17 + 4 + 5 = 26$, which is not optimal. As shown in Fig. 4(e), the optimal solution instead selects I_3 in the second round. Although I_3 covers only I_5, I_7, and I_8 with a lower benefit of 1.71 (due to overlap with I_1), it achieves full coverage using just two indexes, I_1 and I_3, for a total of $17 + 7 = 24$, outperforming the greedy approach.*

Implementation Details and Cost Analysis. To make the greedy method practical, we first create an index in $\mathcal{I}$ for each query–label set in $\mathcal{L}$. In theory, there can be up to $2^{|\Sigma|}$ query–label sets—and thus the same number of indexes—in the worst case. However, given a query workload $\mathcal{L}$, we only need to compute the sizes of $O\left(\min\left(2^{|\Sigma|}, |\mathcal{I}|\right)\right)$ indexes during preprocessing, where $|\mathcal{I}| = |\mathcal{L}|$. When the labels in the alphabet Σ follow a power–law distribution (as in many real-world datasets), this preprocessing cost is empirically a small fraction of the total runtime. For other label distributions, however, the overhead can be substantial. To handle such cases, following [23], we estimate the size of some index I in

$\mathcal{I}$ under a query–label set L_q by sampling or advanced cardinality–estimation techniques [24], without fully materializing I. Once index sizes are approximated, we can efficiently compute the benefit of each candidate and apply the greedy algorithm. To further improve performance, we maintain a max-heap to track the index with the highest benefit at each iteration. As selecting an index can affect the benefits of up to $|\mathcal{I}|$ related indexes, we check whether the heap's top entry has become stale; if so, we update its benefit and reinsert it. This strategy keeps the practical runtime well below $O(|\mathcal{I}|^2)$ — the cost incurred by the greedy method.

4.2 Method for Space-Constrained Variant

In the previous section, we addressed the efficiency-constrained index-selection problem. We now turn to selecting indexes that maximize query efficiency while satisfying a space constraint. This optimization variant of the EIS problem follows Definition 3.3, but focuses on computing the following output:

Output   A subset $\mathbb{I} \subseteq \mathcal{I}$ such that the total cost $\sum_{I \in \mathbb{I}} |I| \leq \tau$, and the minimum elastic factor $\min(e(S(L_q), e(\mathbb{I}))$ over all $L_q \in \mathcal{L}$ is maximized.

Insight. We observe that the elastic–factor threshold c is monotone for the EIS problem. For example, if an index set $\mathbb{I}$ satisfies a threshold of 0.5 for a query workload $\mathcal{L}$, it also satisfies any smaller threshold $c' < 0.5$. This holds because the threshold c bounds the overlap between the indexes in $\mathbb{I}$ and any label set in $\mathcal{L}$, requiring the overlap to be at least c. Thus, if the *stricter* bound c is met, any *looser* bound c' is automatically satisfied.

Method. Leveraging this property, we reduce the space–constrained problem to its efficiency–constrained counterpart and apply a binary search to find a set $\mathbb{I}$ whose elastic factor exceeds a given c while keeping the total cost within the budget τ. We perform the binary search over $c \in (0, 1]$ in decreasing order. At each step, for a candidate value of c, we invoke Algorithm 1 to check if the returned index set $\mathbb{I}$ has total cost at most τ. The search stops once such a set is found, yielding the maximum achievable elastic factor under the space constraint. Overall, this procedure requires $O\left(\log \frac{1}{\varepsilon}\right)$ calls to Algorithm 1, where ε is the resolution for the binary search (set to 0.001 in our paper). In practice, the overhead of combining binary search with the greedy method is negligible—typically under 1% of the index-construction time, or about 1–2 seconds even for large workloads $\mathcal{L}$.

4.3 Discussions

Limitation. For either efficiency- or space-constrained EIS problem, our greedy methods output a subset of indexes $\mathbb{I}$ from the universal index set $\mathcal{I}$, where each index corresponds to a query–label set in the workload $\mathcal{L}$. The efficiency of our approach for AKNN search stems from controlling either space or query time through index sharing based on label-containment relationships. In extreme cases, however, index sharing may be ineffective. For example, if the query workload consists of label sets that are completely disjoint, there is no containment relationship between them. We must then either create a separate index for each disjoint query–label set

(resulting in high space cost) or build a single index covering all vectors in the database (resulting in a small elastic factor). Fortunately, in real-world scenarios, when query–label sets are disjoint, their base vectors are typically disjoint as well, so building an index for each query–label set remains affordable, requiring only $O(N)$ space. By contrast, algorithms such as UNG, ACORN, and FILTER-DISKANN are unaffected by this limitation as they do not rely on index sharing and may perform better in such cases.

Extensions to Dynamic Scenarios. In real-world deployments, both the database vectors and the query workload may evolve over time. We describe how our method accommodates such updates.

Database Vector Dynamics. Vectors can be inserted into or deleted from the database. We focus on insertions, since deletions can be handled by marking a vector as ignored and continuing the search until all results are found [10, 29, 36]. When new vectors arrive, we insert each into every index I_i (built on query–label set L_i) whose labels match the vector. For queries, we continue using the original selected index set $\mathbb{I}$, but periodically check for data changes and update the indexes once 10–20% new vectors have been added.

Query Workload Dynamics. When new label sets L_q appear in the query workload, we first collect all matching vectors for each L_q. If fewer than 4,000 (a threshold determined by experiments) are collected, we scan them directly to answer the new query. Otherwise, we select indexes from the set $\mathbb{I}$ built on label sets that are subsets of L_q. Note that the top index $I_\top$ is always available (as $\emptyset$ is a subset of any query-label set). If multiple indexes qualify, we choose the one with the maximum overlap with the collected vectors to answer the query. We selectively create an index for the new label set in the query workload (e.g., when the number of collected vectors exceeds 4,000), but adjust set $\mathbb{I}$ only periodically to maintain efficiency.

Index Updates. For both types of dynamics, we postpone structural updates to the current selected index set $\mathbb{I}$ and process them in batches. (1) For the efficiency-constrained variant: We maintain the current set $\mathbb{I}$ as long as it satisfies the elastic–factor constraint. If vector insertions or newly created indexes (from new queries) violate this constraint, we reapply Algorithm 1 to add the necessary indexes to enlarge $\mathbb{I}$. (2) For the space-constrained variant: We use the existing $\mathbb{I}$ and its current elastic factor as the starting point. If additional space is available, we attempt to increase the target elastic factor. Otherwise, we lower the target, prune redundant indexes to free space, and continue the binary-search process.

Throughout these updates, we support non-blocking operations: The index set need not be rebuilt from scratch, and incoming queries are routed dynamically to the index with the highest elastic factor, ensuring uninterrupted service. Empirical results in Section 5 confirm the efficiency of this update strategy.

5 EXPERIMENT

5.1 Experiment Settings

Datasets. We evaluate on 8 publicly available datasets (see Table 4) that are standard benchmarks for AKNN search, as well as datasets generated by state-of-the-art embedding models such as MSMARCO, OpenAI-1536 [53], TripClick, and Arxiv [26]. To examine scalability, we also sample 100M vectors (denoted as DEEP) from
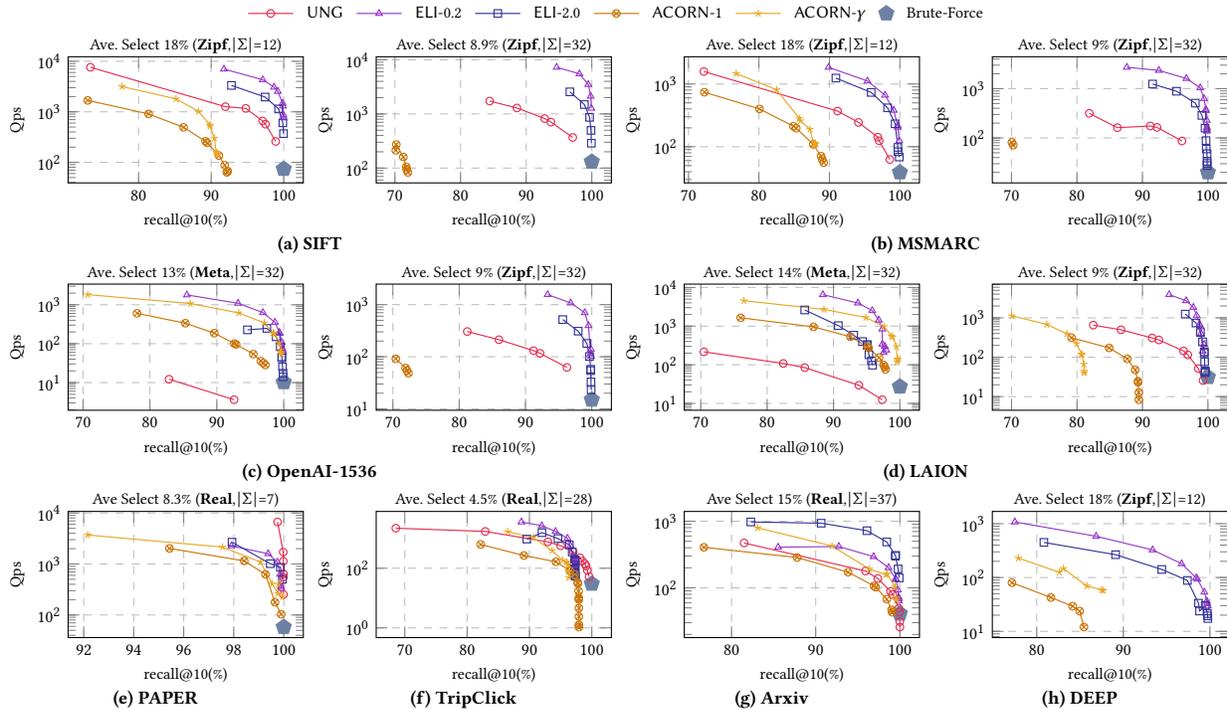
Figure 7: The Efficiency Comparison Across Methods

Table 4: The Statistics of Datasets

Dataset	Dimension	Size	Query Size	Type	Label
SIFT	128	1,000,000	10000	Image	Synthetic
MSMARCO	1024	1,000,000	1000	Text	Synthetic
LAION	512	1,182,243	1000	Image	Meta
OpenAI-1536	1536	999,000	1000	Text	Meta
PAPER	200	2,029,997	10000	Text	Real
TripClick	768	1,055,976	1000	Text	Real
Arxiv	4000	100,000	1000	Text	Real
DEEP	96	100,000,000	1000	Image	Synthetic

the Deep1B [12, 53][1]. For label annotations, datasets PAPER, LAION, TripClick, OpenAI-1536, and Arxiv supply real labels or metadata. For others, we synthesize labels following prior work [5, 18]. Label sources are grouped into three categories:

(1) **Real** labels (Paper, TripClick, Arxiv): For Paper, each vector is labeled by the paper's venue, research field, and institution type. For TripClick, we use the 28 most frequent clinic categories, consistent with prior works [5, 40]. For Arxiv, we assign the main subject categories of each vector as its label.

(2) **Metadata-derived** labels (LAION, OpenAI-1536): Using vector-associated metadata (e.g., text or images), we follow the procedure of [40] to construct a candidate label set and then assign to each vector its top-3 most relevant labels.

(3) **Synthetic** labels (SIFT, MSMARCO, DEEP): Since keywords and tags in real systems typically follow heavy-tailed distributions, prior work models label frequency using Zipf (power-law) distributions [5, 18]. We reuse publicly available code to generate labels

[1]www.tensorflow.org/datasets/catalog/deep1b

for both base and query vectors under this distribution. In subsequent experiments, we also evaluate alternative distributions (Uniform, Poisson, and Multinomial) and vary the alphabet size $|\Sigma| \in \{8, 12, 24, 32\}$ to assess robustness [54].

Metrics. The evaluation covers both efficiency and accuracy. For the search efficiency metric, we use Queries Per Second(Qps), which indicates the number of queries processed by the algorithm per second, as it is most commonly used in benchmarks. For the search accuracy, we use *recall* defined in § 2.1 as the metric to align with the baselines [5, 40]. All results are reported as averages. We also report the **average selectivity** of a query workload. For each query (q, L_q), the **selectivity** is defined as $|S(L_q)|/|S|$, i.e., the fraction of base vectors whose label sets contain L_q. We compute the average selectivity (denoted *Ave. Select*) across all test queries in the workload and annotate it at the top of each figure to highlight this value.

Algorithms. We compare the following algorithms:
• ELI-0.2: Our greedy method for the efficiency-constrained variant, where 0.2 denotes the elastic–factor threshold.
• ELI-2.0: Our method for the space-constrained variant, where 2.0 indicates the space bound (at most twice the original dataset size).
• UNG: Unified Navigating Graph for AKNN search [5].
• ACORN-1: ANN Constraint-Optimized Retrieval Network with low construction overhead [40].
• ACORN-γ: ANN Constraint-Optimized Retrieval Network for high-efficiency search [40].
• **Brute-Force:** Scan all label-matched vectors to find the results.

Implementation Details. All code was implemented in C++ and compiled with GCC 9.4.0 using the -Ofast optimization flag. Experiments were conducted on a workstation equipped with Intel(R) Xeon(R) Platinum 8352V CPUs @ 2.10 GHz and 512 GB of memory. We employed multi-threading (144 threads) for index construction and a single thread for search evaluation. For the underlying index, we used HNSW [37] with parameters $M = 16$ and *efconstruct* = 200. For ELI-0.2 (< 1.0), we applied the index-selection method to achieve a fixed elastic factor of 0.2. For ELI-2.0 (> 1.0), we used the fixed-space method, allowing at most double the original database size to maximize efficiency. For other baselines, such as ACORN and UNG, we adopted the default parameters from their respective papers: $\alpha = 1.2$, $L = 100$ for UNG; $\gamma = 80$, $M = 128$, $M_B = 128$ for ACORN-γ on the LAION and TripClick datasets; and $\gamma = 12$, $M = 32$, $M_B = 64$ for the remaining datasets.

5.2 Experiment Results

Exp-1: Comparison of Query Efficiency. We begin by evaluating query efficiency (Qps) across different methods. Fig. 7 compares UNG, ACORN-1, ACORN-γ, and our methods (ELI-0.2, ELI-2.0) under varying alphabet sizes $|\Sigma|$ and label distributions. Here, *Real*, *Meta*, and *Zipf* denote the real, meta, and synthetic Zipf-based label distributions, respectively. Points toward the upper right indicate better performance. Our methods achieve the best trade-off between search efficiency and accuracy. For example, at 95% recall with $|\Sigma| = 12$, ELI-0.2 delivers a 4× speedup over UNG on SIFT and a 10× speedup on MSMARCO. Performance remains stable across different $|\Sigma|$ values and real label distributions, whereas UNG's retrieval efficiency drops sharply as $|\Sigma|$ grows.

Thanks to the elastic factor, ELI-0.2 sustains nearly 12× higher query speed than UNG when $|\Sigma| = 32$ on the OpenAI-1536 dataset. In contrast, ACORN plateaus in accuracy for large $|\Sigma|$ due to limitations in its PreFiltering strategy. For example, on Fig. 8(a), as $|\Sigma|$ increases, ACORN accuracy declines and UNG efficiency degrades, while ELI-0.2 maintains consistently high efficiency. Although larger $|\Sigma|$ lowers average query selectivity, our design keeps runtime stable. A modest gap appears between ELI-0.2 and ELI-2.0. For example, on Fig. 8(b), under a Zipf distribution, the elastic factor of ELI-2.0 is typically below 0.2 for large $|\Sigma|$, so ELI-0.2 often performs slightly better due to its higher elastic-factor threshold.

Exp-2: Effect of Query Selectivity. To further compare methods, we vary query selectivity and measure its impact on query efficiency. We evaluate on Arxiv with four settings: *No Filter* (100% selectivity), 30%, 10%, and 5% selectivity, as shown in Fig. 8. In the No-Filter setting, the problem reduces to a standard AKNN search without label constraints. Here, ELI-0.2 and ELI-2.0 match the performance of HNSW-PostFiltering (i.e., HNSW with the PostFiltering strategy) and outperform other baselines. At 30% selectivity, ELI-2.0 consistently surpasses UNG and ACORN across the full recall range, while HNSW-PostFiltering performs nearly the same as ELI-0.2. When selectivity drops to 10% and 5%, the advantage of our methods widens: they maintain high Qps even at high recall and consistently outperform all other methods across different recall levels. These results validate the elastic-factor cost model: with sufficiently large c (e.g., $c \geq 0.2$), the additional computation scales as k/c. We also compare with brute-force search, which becomes competitive when very

Table 5: The Comparison of Indexing Time (s)

	UNG	ACORN-1	ACORN-γ	ELI-0.2	ELI-2.0
SIFT	405	7	62	34	25
MSMARCO	459	29	290	148	87
PAPER	65	18	128	39	64
LAION	27	105	3238	121	63
TripClick	283	137	4289	127	81
OpenAI-1536	218	47	568	409	166
Arxiv	116	9	50	23	25
DEEP	-	912	11328	8864	3991

Table 6: The Comparison of Index Size (Mb)

	Base	UNG	ACORN-1	ACORN-γ	ELI-0.2	ELI-2.0
SIFT	488	186	442	485	634	382
MSMARCO	3906	233	442	485	634	382
PAPER	1548	372	898	986	699	650
LAION	2309	204	2038	1820	871	405
TripClick	3093	192	1820	1625	814	540
OpenAI-1536	5853	261	442	485	782	342
Arxiv	1525	20	80	85	26	30
DEEP	36621	-	44262	48591	85293	37902

few vectors match the labels (e.g., fewer than 4,000). In practice, we therefore build indexes only for query–label sets containing at least about 4,000 vectors and use brute force otherwise.

Exp-3: Index Selection Statistics. Recall that our methods aim to select a subset of indexes $\mathbb{I}$ from the universal index set $\mathcal{I}$. To illustrate how the selection works, we report both the number of indexes chosen and the total number of vectors covered by the indexes selected by our methods (ELI-0.2 and ELI-2.0). We evaluate on real datasets (OpenAI-1536, Laion, TripClick, Arxiv, and Paper). For Paper, we also vary the label distribution (e.g., Poisson) as an additional test. The results are presented in Fig. 9. The top panel of Fig. 9 presents the total number of candidate indexes $|\mathcal{I}|$ as well as the number selected by ELI-0.2 and ELI-2.0. The bottom panel shows the total size of all candidate indexes (i.e., total number of vectors across indexes) and the size of the indexes actually selected by ELI-0.2 and ELI-2.0.

From Fig. 9, we observe: (1) **Compact label-set selection.** Both methods select up to an order of magnitude fewer indexes than the full set $\mathcal{I}$, verifying the effectiveness of index sharing in reducing index count. (2) **Significant reduction in indexed size.** The bottom panel shows that ELI-0.2 indexes only a small fraction of the total base vectors, while ELI-2.0 indexes roughly twice the number of base vectors. (3) **Robustness across distributions.** Under various label distributions for Paper (Uniform, Poisson, Multinomial, and Zipf), our methods always achieve substantial reductions in both the number of selected indexes and their total size.

Exp-4: Indexing Time and Space. We compare indexing time in Table 5 and index size in Table 6. For datasets with real labels, results are reported under their native label distributions. For synthetic datasets, we report results with $|\Sigma| = 32$, consistent with the cmdefficiency comparison in Fig. 7. (1) **Indexing time.** Table 5 shows that our methods (especially ELI-2.0) are far more efficient than UNG and ACORN-γ. Specifically, UNG requires nearly 20× the indexing time on SIFT and 5× on MSMARCO, while ACORN-γ averages about 2× the indexing time of our methods. Although
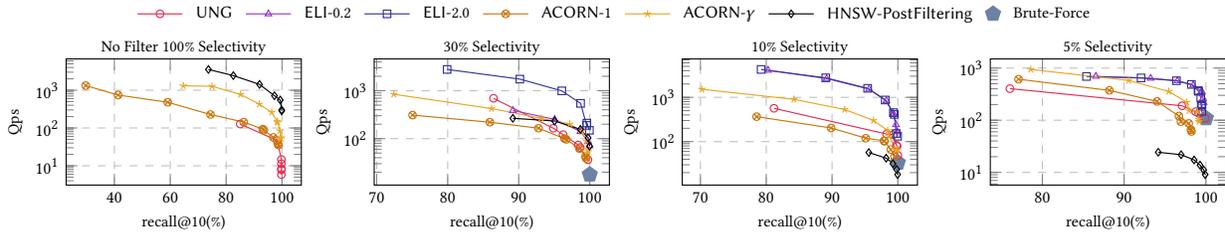
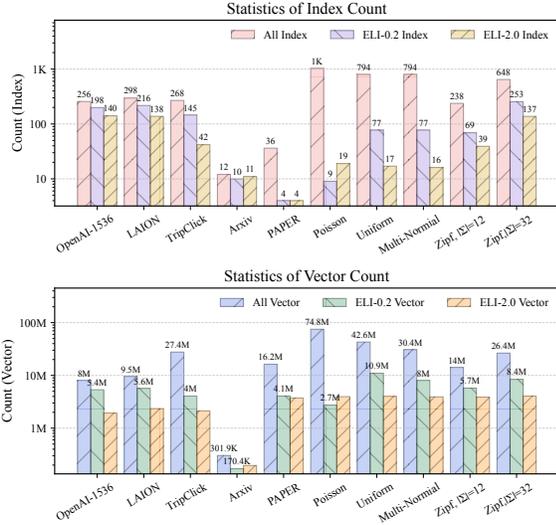Figure 8: The Effect of the Query Selectivity (on Arxiv)

Figure 9: The Statistics of Index Selection by Our Methods

ACORN-1 builds indexes quickly, its omission of label information results in poorer search efficiency. (2) **Index size.** Table 6 reports index sizes, where *Base* denotes the size of the raw vectors. On the OpenAI-1536 dataset, all methods require only about one-tenth of the total storage space. Our methods achieve the best search performance with a comparable index size. In particular, both ACORN and ELI-2.0 use roughly twice the space of the original vectors, yet ELI-2.0 achieves strong search performance with much smaller additional space. UNG saves the most space, but in high-dimensional settings, its index size is only a small fraction of the base vector dataset, leading to lower search efficiency.

Exp-5: Varying Label Distributions. Label distributions influence label-hybrid AKNN search, so we evaluate performance under different distributions—Zipf, Uniform, Poisson, and Multinomial—with $|\Sigma| = 12$. Experiments are conducted on the PAPER dataset, and results are shown in Fig. 10. (1) **Stable performance.** Our methods perform consistently across all tested distributions. In contrast, the search accuracy of other methods varies significantly with the distribution: under Uniform and Multinomial settings, ACORN fails to reach 80% recall, while UNG and our methods maintain high accuracy. (2) **High efficiency.** Across all distributions, our methods remain highly competitive. They achieve recall comparable to UNG at high thresholds and deliver 3×–5× better search efficiency.

Exp-6: Index Update Evaluation. We evaluate the effectiveness of our index-update strategy on the Arxiv and SIFT datasets. Each dataset is partitioned into two halves. The first 50% of the data, with label-set size $|\Sigma| = 12$, is used to construct the initial index set $\mathbb{I}$. The remaining 50%, featuring an expanded label set of $|\Sigma| = 24$, simulates the insertion of both new vectors (50% more vectors) and an enlarged query workload (12 additional labels).

Query Efficiency. Fig. 11 reports query efficiency. ELI-0.2 and ELI-2.0 denote the methods rebuilt from scratch using the full dataset, whereas ELI′-0.2 and ELI′-2.0 represent our batch-update strategy, which first builds indexes on the initial 50% of data and then incrementally updates them. The results show that periodic updates incur less than a 5% drop in query efficiency compared with full re-computation, demonstrating that the batch update mechanism preserves practical performance.

Update Cost. Figure 12 compares the update cost of ELI′-0.2 and ELI′-2.0 against rebuilding from scratch. The left panel shows update time: the rightmost bar corresponds to full re-computation on 100% of the data, while the three bars to the left represent incremental insertions of 10%, 20%, and 50% new vectors into the initial index built on the first 50% of the data. Update time naturally increases with the number of inserted vectors but remains well below the cost of rebuilding, even when 50% of the data is added. The right panel reports space usage (in units of updated vectors). As more vectors are inserted, space usage grows slightly but remains smaller than that of full re-computation. Notably, both our methods, especially ELI′-2.0, use significantly less space than their rebuilt counterparts.

Overall, these experiments show that our incremental update strategy achieves significantly lower update time and space overhead compared with rebuilding from scratch(Fig. 12), while incurring only a negligible loss in query efficiency (Fig. 11).

6 RELATED WORK

Attribute-Hybrid AKNN Search. When vectors carry label attributes (as defined in this paper) or numerical attributes (e.g., [50]), an **attribute-hybrid** AKNN search can be defined to return approximate k-nearest neighbors whose attributes satisfy given constraints. Such searches can be handled using either attribute-free or attribute-dependent solutions.

Attribute-Free Solutions. These methods ignore attributes during index construction and consider them only at query time. Specifically, the PreFiltering and PostFiltering search strategies are widely used, each performing differently under varying query workloads. Systems such as ADB [48], VBASE [57], CHASE [35], and Milvus [43]
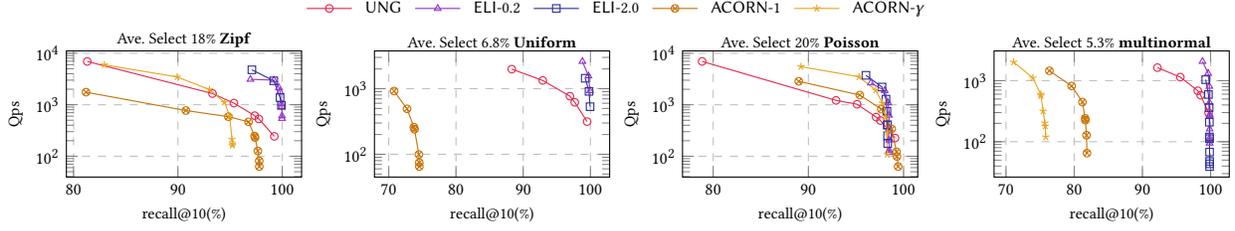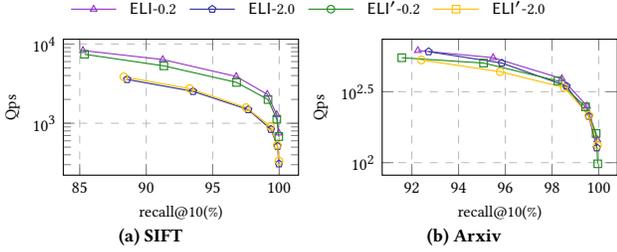
Figure 10: Varying Label Distributions

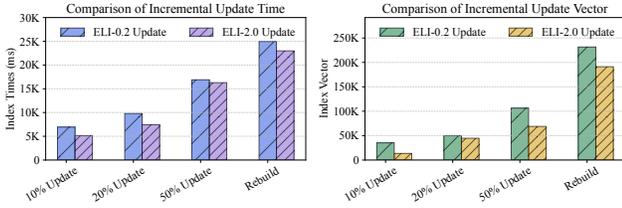Figure 11: Query Performance Drop Caused by Incremental Updates

Figure 12: The Time and Space Cost of Incremental Updates

choose between these strategies using cost models. ACORN also follows the PreFiltering strategy but builds a denser graph index to mitigate its connectivity issues. Because these approaches ignore attributes during index construction, their search accuracy and efficiency lag behind methods that incorporate attributes directly.

Attribute-Dependent Solutions. These methods build indexes by considering the attribute type. (1) For attribute-hybrid AKNN search with *numerical* attributes, recent work [9, 50, 52] organizes query-specified numeric intervals using segment trees, while SeRF [59] further compresses indexes across overlapping intervals. (2) For attribute-hybrid AKNN search with *label* attributes, UNG [5] exploits label-set inclusion, adding cross-group edges so the graph index searches only filtered vectors. NHQ [44] incorporates attribute similarity into vector similarity, requiring manual tuning of vector and attribute weights.

Data Cube Computation. Existing studies on OLAP and data-cube computation select group-by aggregates over the subset lattice of dimensions, exploiting the fact that coarser aggregates (supersets) can be derived from finer-grained materializations (subsets) [20, 23, 38]. Our setting shares this lattice structure and the reuse principle: an index built for a query-label set can serve queries with superset labels via label containment. However, key differences remain: (1) Cube computation relies on algebraic properties of aggregates (e.g., distributive or algebraic measures) to derive superset results exactly

from subsets, whereas our reuse is containment-based, yielding a query cost bounded by $O(C + k/c)$ on graph-based indexes, where c is the elastic factor and k/c is the additional cost. (2) Cube/view selection minimizes aggregate-query latency subject to storage limits, while our Elastic Index Selection (EIS) problem selects a subset of query-label sets to index so as to satisfy both space budgets and workload-wide elastic-factor coverage, ensuring predictable AKNN efficiency under label constraints. (3) Although both problems are NP-hard and typically approached with greedy heuristics, our benefit function and feasibility constraints are tailored to vector-search semantics and the label-containment relationship.

Frequent Itemset Mining. Our work is conceptually related to Frequent Itemset Mining (FIM), where labels correspond to items and label sets to transactions [21, 22, 56]. Both exploit the anti-monotone property of support (the Apriori principle), with classic algorithms such as Apriori and FP-Growth widely used for FIM [21, 22]. The objectives, however, differ fundamentally. FIM finds all itemsets whose support exceeds a threshold for pattern discovery, whereas our ELI task is an optimization problem: selecting a subset of query-label sets on which to build indexes. This selection is guided not by absolute support but by a novel *elastic factor* modeling query efficiency. Thus, while FIM outputs frequent patterns, our method produces a concrete indexing plan for AKNN search, making the formulation and goals distinct.

7 CONCLUSION

This paper investigates the label-hybrid AKNN search problem and shows that an index built for a query–label set remains effective for all its supersets with bounded query time. Building on this insight, we formalize the **Elastic Index Selection** (EIS) problem, prove that EIS is NP-complete, and introduce two optimization variants solved by efficient greedy algorithms. Extensive experiments demonstrate the effectiveness and superiority of our approach. As future work, we plan to explore hardware acceleration for further speedups.

REFERENCES

[1] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache locality is not enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan. *Proceedings of the VLDB Endowment* 9, 4 (2015).

[2] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.

[4] Alina Beygelzimer, Sham Kakade, and John Langford. 2006. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*. 97–104.

[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] Thomas Cover and Peter Hart. 1967. Nearest neighbor pattern classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27.

[7] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[8] Rong-chii Duh and Martin Fürer. 1997. Approximation of k-set cover by semi-local optimization. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 256–264.

[9] Joshua Engels, Benjamin Landrum, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2024. Approximate Nearest Neighbor Search with Window Filters. *ICML 2024* (2024).

[10] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.* 44, 8 (2022), 4139–4150.

[11] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.

[12] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2024. Practical and Asymptotically Optimal Quantization of High-Dimensional Vectors in Euclidean Space for Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2409.09913* (2024).

[13] Jianyang Gao and Cheng Long. 2023. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. *Proc. ACM Manag. Data* 1, 2 (2023), 137:1–137:27. https://doi.org/10.1145/3589282

[14] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[15] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 4 (2014), 744–755.

[16] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.

[17] Michel X Goemans and David P Williamson. 1995. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)* 42, 6 (1995), 1115–1145.

[18] 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.

[19] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. 2025. SymphonyQG: Towards Symphonious Integration of Quantization and Graph for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[20] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* 1, 1 (1997), 29–53.

[21] Jiawei Han, Jian Pei, and Hanghang Tong. 2022. *Data mining: concepts and techniques*. Morgan kaufmann.

[22] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. *ACM sigmod record* 29, 2 (2000), 1–12.

[23] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. 1996. Implementing data cubes efficiently. *Acm Sigmod Record* 25, 2 (1996), 205–216.

[24] Hazar Harmouch and Felix Naumann. 2017. Cardinality estimation: An experimental survey. *Proceedings of the VLDB Endowment* 11, 4 (2017), 499–512.

[25] 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.

[26] Patrick Iff, Paul Bruegger, Marcin Chrapek, Maciej Besta, and Torsten Hoefler. 2025. Benchmarking Filtered Approximate Nearest Neighbor Search Algorithms on Transformer-based Embedding Vectors. arXiv:2507.21989 [cs.DB] https://arxiv.org/abs/2507.21989

[27] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.

[28] Piotr Indyk and Haike Xu. 2023. Worst-case performance of popular approximate nearest neighbor search implementations: Guarantees and limitations. *Advances in Neural Information Processing Systems* 36 (2023), 66239–66256.

[29] 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).

[30] Yannis Kalantidis and Yannis Avrithis. 2014. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2321–2328.

[31] Richard M Karp. 2009. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*. Springer, 219–241.

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

[33] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *IEEE Trans. Knowl. Data Eng.* 32, 8 (2020), 1475–1488.

[34] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2021. HVS: Hierarchical Graph Structure Based on Voronoi Diagrams for Solving Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 15, 2 (2021), 246–258.

[35] Rui Ma, Kai Zhang, Zhenying He, Yinan Jing, X Sean Wang, and Zhenqiang Chen. 2025. CHASE: A Native Relational Database for Hybrid Queries on Structured and Unstructured Data. *arXiv preprint arXiv:2501.05006* (2025).

[36] Yu A Malkov and Dmitry A Yashunin. [n.d.]. hnswlib. https://github.com/nmslib/hnswlib.

[37] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.

[38] Imene Mami and Zohra Bellahsene. 2012. A survey of view selection methods. *Acm Sigmod Record* 41, 1 (2012), 20–29.

[39] Gonzalo Navarro. 2002. Searching in metric spaces by spatial approximation. *The VLDB Journal* 11 (2002), 28–46.

[40] 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.

[41] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proc. ACM Manag. Data* 1, 1 (2023), 54:1–54:27. https://doi.org/10.1145/3588908

[42] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. 2007. Collaborative filtering recommender systems. In *The adaptive web: methods and strategies of web personalization*. Springer, 291–324.

[43] 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.

[44] 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).

[45] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proc. ACM Manag. Data* 2, 1 (2024), V2mod014:1–V2mod014:27. https://doi.org/10.1145/3639269

[46] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 14, 11 (2021), 1964–1978.

[47] Runhui Wang and Dong Deng. 2020. DeltaPQ: lossless product quantization code compression for high dimensional similarity search. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3603–3616.

[48] 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.

[49] 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.

[50] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. *arXiv preprint arXiv:2409.02571* (2024).

[51] Mingyu Yang, Wentao Li, Jiabao Jin, Xiaoyao Zhong, Xiangyu Wang, Zhitao Shen, Wei Jia, and Wei Wang. 2024. Effective and General Distance Computation for Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2404.16322* (2024).

[52] Mingyu Yang, Wentao Li, Zhitao Shen, Chuan Xiao, and Wei Wang. 2025. ESG: Elastic Graphs for Range-Filtering Approximate k-Nearest Neighbor Search. *arXiv preprint arXiv:2504.04018* (2025).

[53] Mingyu Yang, Wentao Li, and Wei Wang. 2024. Fast High-dimensional Approximate Nearest Neighbor Search with Efficient Index Time and Space. *arXiv preprint arXiv:2411.06158* (2024).

[54] Mingyu Yang, Wenxuan Xia, Wentao Li, Raymond Chi-Wing Wong, and Wei Wang. 2025. *Technical Report*. https://github.com/mingyu-hkustgz/LabelANN/blob/master/technical_report.pdf

[55] Ziqi Yin, Jianyang Gao, Pasquale Balsebre, Gao Cong, and Cheng Long. 2025. DEG: Efficient Hybrid Vector Search Using the Dynamic Edge Navigation Graph.

[56] Mohammed Javeed Zaki. 2002. Scalable algorithms for association mining. *IEEE transactions on knowledge and data engineering* 12, 3 (2002), 372–390.

[57] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, et al. 2023. {VBASE}: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 377–395.

[58] Bolong Zheng, Xi Zhao, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S. Jensen. 2020. PM-LSH: A Fast and Accurate LSH Framework for High-Dimensional Approximate NN Search. *Proc. VLDB Endow.* 13, 5 (2020), 643–655.

[59] 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.