TrieHNSW: Effective Dynamic Indexing for Label-Filtered Approximate Nearest Neighbor Search

Mengxu Jiang
mxjiang@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR, China

Zhi Yang
zyang@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR, China

Fangyuan Zhang
zhang.fangyuan@huawei.com
Huawei Hong Kong Research Center
Hong Kong SAR, China

Yin Yang
yyang@hbku.edu.qa
Hamad Bin Khalifa University
Qatar

Sibo Wang
swang@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR, China

ABSTRACT

Given an object dataset O in which each object contains a d-dimensional vector where d is often large, a query $\mathbf{q} \in \mathbb{R}^d$, and an integer k, an approximate nearest neighbor search (ANNS) aims to accurately and efficiently retrieve the top-k best matches of $\mathbf{q}$ in terms of distances in the high-dimensional vector space. Recent advances in large language models have dramatically increased the demand for large-scale, low-latency ANNS over real-world entities such as products. In these applications, objects in O are often also associated with labels (e.g., attributes or keywords), and it is desirable to augment the ANNS with explicit label-based filtering, which necessitates the task of *label-filtered ANNS*.

Existing solutions face two major challenges: first, label-based filtering and ANNS require very different index structures, and it is highly non-trivial to create a unified index that is simultaneously effective at label filtering and ANNS. Second, when the dataset O is updated, the corresponding index needs to be efficiently maintained, while retaining its high query effectiveness. To address these, we propose *TrieHNSW*, a novel dynamic composite index that (i) builds a trie to organize label relationships, and (ii) uses multi-layer graph indices embedded in the trie that capture global vector connections. The query algorithm searches multiple graphs guided by the trie to achieve high efficiency and result recall. Further, TrieHNSW can be updated with low amortized cost. Extensive experiments confirm that TrieHNSW consistently outperforms existing approaches in terms of accuracy, efficiency, and update overhead.

PVLDB Reference Format:
Mengxu Jiang, Zhi Yang, Fangyuan Zhang, Yin Yang, and Sibo Wang. TrieHNSW: Effective Dynamic Indexing for Label-Filtered Approximate Nearest Neighbor Search . PVLDB, 19(X): XXX-XXX, 2026.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CUHK-DBGroup/TrieHNSW.

1 INTRODUCTION

Recent advances in machine learning, especially large language models (LLMs), have significantly increased the demand for managing high-dimensional vector data. Specifically, through a machine learning model, an unstructured object, such as an image, a full-text article, or a graph, can be converted to a high-dimensional vector representation called its *embedding*. Such embeddings capture the semantic information of their underlying data objects, in the sense that when two embeddings are close to each other in the vector space, their corresponding data objects are usually semantically relevant, and vice versa. For instance, the image of a pill and the textual instructions of the corresponding medicine are expected to map to relatively close-by embedding vectors with a vision-text model such as CLIP [26]. This technique enables semantic search of unstructured objects, often performed together with an LLM. For example, imagine that a user uploads a query image of a pill, and asks an AI agent about the potential adverse effects of the medicine; the AI agent then retrieves the top-k medical articles most relevant to the query image, and generates a response to the user through the LLM grounded by these articles. Here, the matching of the query image and the medical articles is performed in the embedding space, and the retrieval of the top-k articles is usually done via *an approximate nearest neighbor search* (*ANNS*) [17], which can be processed effectively and efficiently in a vector database system [10, 33, 34].

In practice, each data object is often also associated with a set of *labels*, such as categories, tags, or keywords. Continuing the above example, medical articles can be tagged as "papers in reputed medical journals", "manufacturer instructions", or "social posts". Accordingly, an ANNS can be augmented with explicit label-based filtering (e.g., retrieve only papers in reputed medical journals and manufacturer instructions), to give the user stronger control over the source of information. Such an augmented ANNS query is referred to as *label-filtered ANNS* [16].

Main challenges. Although both ANNS and label-based filtering have been studied extensively with mature solutions, their combination, i.e., label-filtered ANNS, still faces significant challenges. At the core of the problem is that the indices for label filtering (often based on the classic inverted list [39]) and ANNS (usually graph-based, e.g., HNSW [22]) are very different data structures. A simple idea would be to apply a cost-based optimizer to select

Proceedings of the VLDB Endowment, Vol. 19, No. X ISSN 2150-8097.
doi:XX.XX/XXX.XX

one of these indices, which leads to two common solutions (i) *pre-filtering*, which applies the label index to select candidate objects with the required label sets, and subsequently performs ANNS on the resulting candidate set without an index, and (ii) *post-filtering*, which performs index-based ANNS to retrieve a larger candidate set (i.e., more than k objects), and then filters them by labels in a postprocessing step. Both are suboptimal: the former incurs a costly scan of inverted lists, whereas the latter often requires a large candidate set to avoid missing relevant objects.

Another idea (called *in-filtering* [25]) is to perform filtering while searching an ANNS graph index, e.g., HNSW [22] (elaborated later in Section 2.2). When processing a label-filtered ANNS query, as we traverse the graph index, we only follow edges that point to vectors that satisfy the filter. An immediate problem with this approach is that the filter (especially a highly selective one) may eliminate many edges of the index, effectively sparsifying it, leading to unreachable qualified vectors and, thus, false dismissals and low result recall. In the literature, several solutions aim to mitigate this problem by following two-hop neighbors at each traversal step in addition to direct edges [25, 30], which enlarges the candidate vector set at the cost of increased traversal time at each step. Note that the in-filter approach assumes a generic filter rather than specifically a label-based filter. In particular, such *label-agnostic* methods fail to index label set relationships, thereby limiting their effectiveness.

Existing label-aware solutions. To our knowledge, currently there is only a narrow selection of *label-aware* ANNS indices. First, Filtered-DiskANN [16] is a single-graph indexing scheme that injects label information into both index construction and search. Similar to generic in-filtering techniques described above, at query time, Filtered-DiskANN expands only label-compatible neighbors; during index building, its label-aware pruning rule preserves edges that remain useful under filter constraints. However, FilterDiskANN is mainly tailored to disjunctive label filters (i.e., retrieve objects that contain at least one query label), and does not naturally extend to other predicates such as containment (i.e., find objects that contain *all* query labels). This limits its practical applicability.

RWalks [3] guides filtered retrieval by optimizing a hybrid distance that jointly accounts for vector proximity and label similarity. Instead of enforcing a hard admissibility check at every expansion, it ranks and expands candidates using an aggregate score (e.g., one that combines embedding distance with a label-based similarity term), so that the walk is simultaneously pulled toward semantically close vectors and toward regions more likely to satisfy the filter. Index update is not explicitly addressed in [3], since RWalks essentially performs refinement on an already constructed graph.

A common issue with Filtered-DiskANN and RWalks is that when the label filter is highly selective, they struggle to achieve high recall, as shown in our experiments. One main reason is that their indices *fail to include edges that guide the search from one qualified object to another*, e.g., with similar label sets, leading to *low connectivity of the admissible subgraph* under a strong label filter.

Finally, UNG [9] builds a multi-graph index by partitioning the dataset by distinct label sets, and building a separate graph index for each label set. The benefit of this design is that queries within closely related label sets can be answered efficiently; the downside, however, is that none of these subgraph indices captures the global structure in the vector space of the whole dataset. To mitigate this, UNG constructs a label navigating graph (LNG), which connects two subgraphs via cross-edges when one's corresponding label set is a minimal superset of the other's. This preserves global proximity across different subgraph indices to a certain degree; however, when the search span multiple subgraphs, the lack of sufficient connections between them may still lead to low result recall, as shown in our experiments. Further, it is unclear how to update the LNG efficiently, since inserting even a single object may trigger widespread changes,often requiring expensive LNG reconstruction.

Our solution. Motivated by the limitations of existing techniques, we propose *TrieHNSW*, an indexing scheme carefully designed for effective label-filtered ANNS, while supporting efficient updates. Specifically, a TrieHNSW index is built in two phases. The first constructs a *label trie T* to capture the relationships of all label sets of the data. Each node $u \in T$ is associated with a label, as well as a *prefix* formed by the labels on the root-to-node path ending at u. Meanwhile, each data object is assigned to a leaf node whose prefix matches the object's label set. The second phase selectively builds multiple HNSW [22]-style *graph indices embedded in T*. Graph indices near the root cover objects with shorter common prefixes, which capture more global vector relationships. In contrast, graphs near the leaves cover objects with highly similar label sets, which facilitate denser connections among vectors with closely related labels. Through this composite trie-graph structure, *TrieHNSW simultaneously captures both label-set similarity and global vector proximity of the data objects*, within feasible space overhead.

Given a query $\mathbf{q}$, we first identify a set $\mathcal{N}$ of nodes in the label trie, whose label prefixes jointly cover $\mathbf{q}$. Then, we locate common ancestors of the nodes in $\mathcal{N}$, and perform a multi-graph search with their corresponding embedded HNSW graphs. In particular, for each visited vertex v in a graph G, we examine not only its neighbors in G, *but also edges in the graph indices of the common ancestors*, which effectively connects the otherwise isolated graphs. Moreover, we perform the search in two passes. The first pass searches over the graphs that cover the majority of the relevant objects, marking all indices visited. The second pass then searches the remaining, unvisited indices, further improving recall.

Since both the label trie and HNSW naturally support updates, our index can be updated efficiently without compromising its search capabilities. In particular, we employ a lazy strategy that rebuilds an index only when the portion of irrelevant objects in it grows beyond a threshold, to limit the amortized cost for updates. Extensive experiments on multiple real high-dimensional vector datasets demonstrate that TrieHNSW achieves high query efficiency and result quality, consistently beats its competitors by a large margin, and supports efficient updates without compromising search effectiveness. Our main contributions are as follows.

- We propose the TrieHNSW index, which integrates a label trie with HNSW graphs to capture both vector-space proximity and label-set similarity of data objects.
- We design an effective query algorithm for label-filtered ANNS that fully utilizes the TrieHNSW index to achieve a favorable balance of result accuracy and query costs.
- We introduce an index update scheme for TrieHNSW with low amortized costs, while preserving the index's search capabilities.

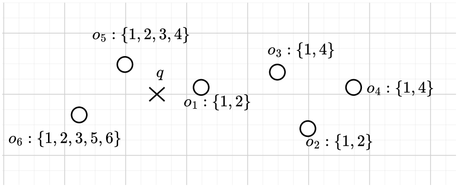


Figure 1: Example of an label-filtered nearest neighbor search

2 PRELIMINARIES

2.1 Problem Definition

Let O be a dataset containing n vectors in $\mathbb{R}^d$. Given two vectors $\mathbf{p}, \mathbf{q} \in \mathbb{R}^d$, let $\delta(\mathbf{p}, \mathbf{q})$ represents their metric distance, e.g., Euclidean distance in $\mathbb{R}^d$. We define nearest neighbor search as follows.

Definition 2.1 (Nearest Neighbors Search (NNS)). Given an object set O consisting of n vectors, a query vector $\mathbf{q} \in \mathbb{R}^d$, and a positive integer $k \leq n$, the nearest neighbor search returns a vector set $R^* = \{\mathbf{o}_1^*, \mathbf{o}_2^*, \ldots, \mathbf{o}_k^*\} \subseteq O$ with the top-k smallest distances to $\mathbf{q}$, such that $\forall \mathbf{o}^* \in R^*, \forall \mathbf{o} \notin R^*$, we have $\delta(\mathbf{o}^*, \mathbf{q}) \leq \delta(\mathbf{o}, \mathbf{q})$.

Next, we focus on the situation where each object in O contains both a vector and a label set. For instance, Figure 1 shows a set of objects $O = \{\mathbf{o}_1, \mathbf{o}_2, \ldots, \mathbf{o}_6\}$ where each object is associated with a set of integer-valued labels, e.g., $\mathbf{o}_1$ has a label set $\{1, 2\}$. We define label-filtered NNS as follows.

Definition 2.2 (Label Filtered Nearest Neighbor Search). Given an object set O where each $\mathbf{o} \in O$ is associated with a label set $f_\mathbf{o}$, a query consisting of a vector $\mathbf{q} \in \mathbb{R}^d$ and a label set f_q, a positive integer k, and a filter constraint S, the label-filtered NNS returns the k-nearest neighbors to $\mathbf{q}$ within the subset $O_S = \{\mathbf{o} \in O \mid f_\mathbf{o} \models_S f_\mathbf{q}\}$, i.e., the objects satisfying the label filter constraint S.

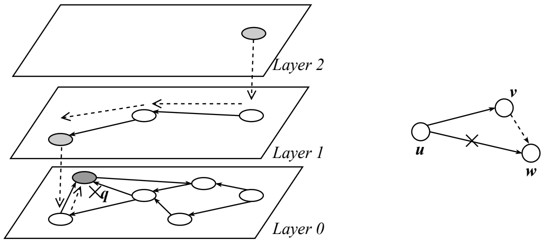
In the above definition, common choices for the filter constraint S (e.g., used in [9]) include (i) *containment*, i.e., a vector $\mathbf{o}$ with label set $f_\mathbf{o}$ satisfies $f_\mathbf{o} \models_S f_\mathbf{q}$ iff $f_\mathbf{q} \subseteq f_\mathbf{o}$, (ii) *overlap*, i.e., $f_\mathbf{o} \models_S f_\mathbf{q}$ iff $f_\mathbf{q} \cap f_\mathbf{o} \neq \emptyset$, and (iii) *equality*, i.e., $f_\mathbf{o} \models_S f_\mathbf{q}$ iff $f_\mathbf{q} = f_\mathbf{o}$.

In the example of Figure 1, given a query vector $\mathbf{q}$ (represented by a cross) and $k = 2$, an NNS on O (Definition 2.1) returns $\mathbf{o}_1$ and $\mathbf{o}_5$, which are closest to $\mathbf{q}$. Meanwhile, when the query contains a label set $f_\mathbf{q} = \{1, 3\}$, and the constraint S is containment, label-filtered NNS (Definition 2.2) returns $\mathbf{o}_5$ and $\mathbf{o}_6$, which are the closest objects to q whose label sets contain both query labels 1 and 3.

Exact NNS in high-dimensional spaces is computationally expensive due to the curse of dimensionality [17]. Hence, approximate nearest neighbor search (ANNS) is widely used [3, 9], which provides a tradeoff of query efficiency and result quality, which is usually measured by recall since the result set size is fixed to k. In particular, let R^* be the exact NNS result and R is the ANNS result, we define $\text{Recall}(R) = \frac{|R \cap R^*|}{|R^*|} = \frac{|R \cap R^*|}{k}$.

2.2 HNSW

Among existing ANNS algorithms, graph-based methods [14, 22, 31], prominently hierarchical navigable small world (HNSW) [22], have demonstrated superior performance in terms of the trade-off between query time and accuracy. Specifically, given a set O of n vectors in a d-dimensional space $\mathbb{R}^d$, such a method typically builds a directed *proximity graph* G, in which each vertex corresponds to a vector in O, and each directed edge connects a pair of vertices according to a pre-defined proximity rule, based on the given distance metric δ as explained in Section 2.1. In the particular case of HNSW, G is a complex structure with a given number (let L) of layers, each of which contains a proximity graph known as a navigable small world (NSW). Let $H[i]$ be the graph at layer i. The number of vertices in G decreases exponentially from the bottom layer $H[0]$ to the top layer $H[L-1]$.



(a) ANNS in HNSW (b) Domination

Figure 2: Example of an HNSW index.

To process an ANNS query $\mathbf{q}$, a graph-based method such as HNSW typically employs greedy search, which starts from an entry vertex in the proximity graph G and iteratively traverses G. In each iteration, the algorithm identifies the data object $\mathbf{o}$ (which corresponds to a vertex in G) closest to the query $\mathbf{q}$ among all vertices visited so far, and then visits all out-neighbors of $\mathbf{o}$. In the case of HNSW (illustrated in Figure 2a), the search starts from an entry vertex in the highest layer $H[L-1]$, and iteratively descends the layers until reaching the bottom layer $H[0]$. In each iteration, the algorithm computes the vertex closest to the query $\mathbf{q}$ at the current layer, and uses it as the entry point for the next layer.

During the search, the algorithm maintains a candidate set C of up to ef closest vectors to the query $\mathbf{q}$ seen so far. The search terminates when none of the neighbors of the vertices in C are closer to $\mathbf{q}$ than the farthest vector in C, and the top-k vectors in C are returned as the result of $\mathbf{q}$. A larger value of system parameter ef increases the number of vertices visited before termination, leading to higher result recall with increased query cost, and vice versa.

The proximity graph G is usually built incrementally, by sequentially inserting vertices, starting from an empty G. In the following, we focus on the construction of the HSNW index. It suffices to clarify the insertion of a given vertex u, which needs to be performed without degrading the search effectiveness and efficiency of the index. Specifically, the algorithm first determines its highest level L_u using a truncated geometric distribution. Then, u is inserted into the proximity graphs from level $H[L_u]$ down to the bottom level $H[0]$. The maximum out-degree is set to $2M$ at $H[0]$, and to M at all higher levels, where M is a system parameter.

For insertion into the graph at a given level, the process first identifies the ef_c nearest vertices to u as candidate neighbors, then prunes them to at most M edges to form the out-neighbor set $N(u)$. For each selected neighbor $v \in N(u)$, u is added to $N(v)$. If $|N(v)|$ exceeds M, a pruning step is applied to the neighbors of v. During pruning, candidate edges are sorted by their distance to u, and non-dominated edges (as shown in Figure 2b, an edge (u, v) is dominated by another (u, w) iff. $\delta(u, w) < \delta(u, v)$ and

Table 1: Frequently used notations

Symbol	Meaning
O, n	Given set of objects and its cardinatlity
$f_{\mathbf{o}}$	Label set of object $\mathbf{o}$
$\delta(u, v)$	Metric distance between vectors u and v
(f_{q}, S), $\mathbf{q}$	Query label filter and the query vector
$O_{f_{\mathrm{q}},S}$	Set of objects whose label sets satisfy (f_{q}, S)
M	Maximum out-degree of HNSW (except vertices in the bottom layer whose out-degree is $2M$)
ef, ef_c	Size of the candidate set during graph search and graph construction, respectively
T	Label trie in TrieHNSW
O_u	Set of objects falling into the subtree of $u \in T$
$N_G(v)$	Out-neighbor set of vertex v in graph index G
$\mathrm{log\_size}(u)$	$\lfloor \log_2 \lvert O_u \rvert \rfloor$
$u.p, u.ch$	Parent and children of node $u \in T$
$u.index$	Node index of $u \in T$
$u.\mathrm{ch}_{\mathrm{heavy}}$	Child with the same log_size as node $u \in T$
$belong_{\mathbf{o},i}$	Graph of log size i that object $\mathbf{o}$ belongs to

$\delta(v, w) < \delta(u, v)$) are iteratively added to the result set until the out-degree limit is reached. Similar to skip lists, the highest layer L_u for each point in HNSW has a constant expected value. The time and space complexities of building an HNSW graph of n vectors are $O(n \cdot c_{\mathrm{upd}})$ and $O(nM)$, respectively, where c_{upd} denotes the insertion cost of a single vector.

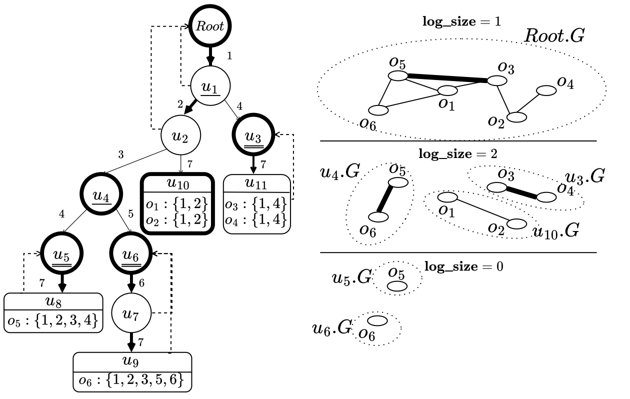
Regarding index updates, insertions are already covered since the graph index is constructed through sequential insertions, each of which is performed efficiently as explained above. For deletions, graph indices such as HNSW commonly employ a lazy approach: that the vertex is simply marked as deleted without being removed from the graph structure, and it is skipped in subsequent searches.

3 TRIEHNSW

Next, we present the proposed TrieHNSW indexing scheme. Section 3.1 introduces the label trie structure. Sections 3.2 and 3.3 describes the query processing algorithm and the update mechanism of the index, respectively. Table 1 summarizes the common notations used throughout the paper.

3.1 Index Structure

Label trie. The proposed TrieHNSW index employs a novel *label trie* T structure to organize labels of the given object set O. To avoid confusion, in the following we use different terms for to distinguish elements of the trie and the HNSW graph: in particular, a *node* always resides in the trie, whereas a *vertex* appears in the graph. Let $\mathcal{L} = \{l_0, l_1, \ldots, l_{|\mathcal{L}|-1}\}$ be the set of all possible labels associated with objects in O, i.e., $\mathcal{L} = \{f_{\mathbf{o}} | \mathbf{o} \in O\}$. Meanwhile, given a label l_j, let $freq(l_j) = \left|\{\mathbf{o} \in O \mid l_j \in f_{\mathbf{o}}\}\right|$ be its frequency in the dataset O. Without loss of generality, assume that labels in $\mathcal{L}$ are sorted in descending order of frequency, i.e., $\forall 0 \le i < j < |\mathcal{L}|, freq(l_j) \ge freq(l_j)$. To simplify our notations, we map all labels in $\mathcal{L}$ to integers $1, 2, \ldots, |\mathcal{L}|$. We further expand the label set $\mathcal{L}$ to include a *special label* $|\mathcal{L}| + 1$ with frequency 0, whose purpose will become clear soon.



(a) (b)

Figure 3: The TrieHNSW index for object set O in Figure 1.

Then, the label trie T is defined as follows.

Definition 3.1 (Label Trie). Given the collection of label sets $\mathcal{L}$ for the dataset O, its label trie is a rooted tree where each non-root node u corresponds to a label $l \in \mathcal{L}$. The children of a node u are ordered, each corresponding to a label with a larger integer identifier (i.e., with a lower frequency) than the label of u. Each leaf node in T corresponds to the special label $|\mathcal{L}| + 1$.

Given node $u \in T$, let $prefix(u)$ denote the ordered sequence of labels along the path from the root to u. Next, we assign objects in O to leaf nodes in T. To do so, we augment the label set for each object in O by adding the special label $|\mathcal{L}| + 1$. Let O_u be the set of objects stored in all leaf nodes of the subtree rooted at u. By construction, every object in O_u has a label set whose prefix matches $prefix(u)$. We use $u.p$ to refer to the parent of u. The children of u are maintained as a set of pairs $u.ch = \{(l_1, c_1), \ldots\}$, where the pair (l_i, c_i) signifies that child c_i corresponds to label l_i and satisfies $prefix(c_i) = prefix(u) \cup \{l_i\}$.

As an example, consider again the object set O in Figure 1. Assuming the maximum possible number of original labels is 6, we add a special label 7 to each object's object set, as described above. Figure 3a illustrates the label trie T in the proposed TrieHNSW index. Node u_2, for instance, corresponds to label 2 and $prefix(u_2) = \{1, 2\}$, following the path root-u_1-u_2. Its parent is $u_2.p = u_1$, and its two children $u_2.ch$ are u_4 (with label 3) and u_{10} (with special label 7, signaling that it is a leaf node). Meanwhile, we have $O_{u_2} = \{\mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_5, \mathbf{o}_6\}$, which all contain prefix $\{1, 2\}$, and, thus are assigned to leaf nodes in the subtree rooted at u_2. Specifically, objects $\mathbf{o}_1$ and $\mathbf{o}_2$ are assigned to leaf u_{10}, $\mathbf{o}_5$ is assigned to u_8, and $\mathbf{o}_6$ to u_9.

Graph indices integrated in T. As will be clarified later in Section 3.2, given a query label filter (f_{q}, S), we decompose the filtered object set $O_{f_q,S}$ into the union of subsets, each corresponding to a subtree rooted at a node in T. Formally, we can find a node set $\mathcal{N}$ in T, where $\bigcup_{u \in \mathcal{N}} O_u = O_{f_q,S}$. We then aim to build graph indexes on T such that for every $u \in \mathcal{N}$, the object set O_u can be searched efficiently via ANNS, within an acceptable total space overhead. To achieve this, we construct indices for *selected* subtrees, ensuring that each relevant O_u can utilize an index that guarantees both search quality and space efficiency. We formalize this requirement by introducing the concept of an *α-approximate index*:

Definition 3.2 (α-Approximate Index). Given a graph index G built on an object set O_G and a target object set O, we say G is an α-approximate index for O iff. $O \subseteq O_G$ and $|O_G|/|O| \leq \alpha$.

If we ensure that each O_u has an α-approximate index, the graph will contain at most $\alpha|O_u|$ vertices. When α is a small constant, the fraction of irrelevant points in the index is low. By simply skipping vertices that do not belong to O_u during the search, we can perform efficient ANNS over O_u with only a small additional overhead. For each node u, let $\text{log\_size}(u)$ denote $\lfloor \log_2(|O_u|) \rfloor$. If u is the root or $\text{log\_size}(u) \neq \text{log\_size}(u.p)$, we construct an HNSW index $u.G$ on the set O_u and set the index-node pointer $u.index$ to u itself. Otherwise, if $\text{log\_size}(u) = \text{log\_size}(u.p)$, we set $u.index = u.p.index$, thereby letting u share the index of its parent. This design leads to the following lemma on the quality of the index.

Lemma 3.3. *For any node u in T, the graph index $u.index.G$ is a 2-approximate index for O_u.*

The proof of the above lemma follows directly from Definition 3.2, which is omitted for brevity.

If a node u shares its graph index with one of its children, we record this child using the pointer $u.\text{ch}_{\text{heavy}}$. The uniqueness of such a child is guaranteed by the following lemma.

Lemma 3.4. *For any node u in the Trie T, there exists at most one child c of u such that* $\text{log\_size}(c) = \text{log\_size}(u)$.

For each object o, we maintain an array $belong_{o,i}$, which indicates to which graph of logarithmic size i the object o belongs. In other words, $belong_{o,i}$ specifies which set O_u (with $\text{log\_size}(u) = i$) contains o. This information helps quickly locate candidate graphs during query processing. We also maintain an inverted list for each label. Specifically, for every label l, we keep a list I_l that records all nodes in the label trie T where l appears along the path from the root to that node. More precisely, if a node u has a child edge labeled l leading to child c, then node c is inserted into I_l. These two auxiliary structures, i.e., the *belong* array and the label IVF lists, facilitate efficient search, elaborated later in Section 3.2.

Continuing the example in Figure 3, for each node $u \in T$, a graph index is built only if $\text{log\_size}(u)$ differs from that of its parent. For instance, in Figure 3a, the sizes of O_{u_3} and O_{u_1} are 6 and 2, with logarithmic sizes 2 and 1, respectively. Accordingly, we build an index $u_3.G$ on O_{u_3}, as shown in Figure 3b. For clarity, we only display the bottom-layer graph for each HNSW index. In the figure, an edge between u and v indicates that there exists a directed edge from u to v as well as a directed edge from v to u. The child u_{11} of u_3 has log_size 1, which equals that of its parent; thus, no separate index is built for u_{11}. Instead, we set $u_{11}.index$ to point to u_3 and mark u_{11} as the *heavy child* of u_3 (denoted $u_3.ch_{heavy} = u_{10}$).

We also maintain two auxiliary structures: (i) the *belong* array, e.g., $belong_{\mathbf{o}_5,1}$ refers to $u_4.G$, while $belong_{\mathbf{o}_5,2}$ refers to Root.G, and (ii) the inverted list I for each label is also stored, e.g., $I_4 = \{u_3, u_5\}$ indicates that label 4 appears on the paths to nodes u_3 and u_5.

Index construction. To build the TrieHNSWindex, we first build the label trie T, assigning each object to its corresponding leaf in T, and then construct the associated graphs. The detailed procedure is outlined in Algorithm 1. Specifically, starting with an empty trie, for each object o we append a new label $|\mathcal{L}|+1$ to its label set f_o and insert it into the Trie (Lines 1-4). During the insertion, we traverse the trie T following the order of labels in f_o, creating new nodes when necessary, and updating the corresponding inverted list I_l for each label l (Lines 7-15). After all objects are inserted and assigned to leaves, we recursively build the graph indexes (Lines 16-26). In particular, if a node u has a different $\lfloor \log_2(|O_u|) \rfloor$ value than its parent, we construct an HNSW index $u.G$ on the object set O_u and update the belong array accordingly (Lines 17-21). Otherwise, u shares the index with its parent and is recorded as the parent's heavy child (Lines 22-24). Lemma 3.5 and Theorem 3.6 establish the time and space complexity of Algorithm 1[1].

Algorithm 1: TrieHNSW index construction

```
Input: Object set $O$ with label set $\mathcal{L}$, HNSW parameters $M$ and $ef_c$
Output: Root node of the TrieHNSW
1  index
2  Initialize $Root \leftarrow$ NULL, $belong \leftarrow$ empty array
3  for $\mathbf{o} \in O$ do
4      Add special label $|\mathcal{L}|+1$ to $f_{\mathbf{o}}$ $TrieInsert(Root, \mathbf{o})$
5  $buildGraph(Root, O, ef_{con})$
6  return $Root$
7  procedure $TrieInsert(Root, \mathbf{o})$:
8      Initialize node $u \leftarrow Root$
9      for $idx \leftarrow 1$ to $|f_{\mathbf{o}}|$ do
10         if $\nexists(l, c) \in u.ch, l = f_{\mathbf{o}}[idx]$ then
11             Create new child $c$, and set its parent $c.p \leftarrow u$
12             $I_{f_{\mathbf{o}}[idx]} \leftarrow I_{f_{\mathbf{o}}[idx]} \cup c$
13             $u.ch \leftarrow u.ch \cup (f_{\mathbf{o}}[idx], c)$
14         $u \leftarrow$ the child of $u$ corresponding to $f_{\mathbf{o}}[idx]$
15     assign $\mathbf{o}$ to $u$
16 procedure $buildGraph(u, O, M, ef_c)$:
17     if $u = Root$ or $\text{log\_size}(u) < \text{log\_size}(u.p)$ then
18         for $\mathbf{o} \in O$ do
19             $HNSWInsert(u.G, \mathbf{o}, M, ef_c)$
20             $belong_{\mathbf{o},\text{log\_size}(u)} \leftarrow u.G$
21         $u.index \leftarrow u$
22     else
23         $u.index \leftarrow u.p.index$
24         $u.p.ch_{\text{heavy}} \leftarrow u$
25     for $(l, c) \in u.ch$ do
26         $buildGraph(c, O_c, M, ef_c)$
```

Lemma 3.5. *For an object o, it appears in $O(\min\{|f_o|, \log n\})$ distinct graph nodes u where $u.index = u$ and $o \in O_u$.*

Theorem 3.6. *Algorithm 1 constructs a TrieHNSW index on O with $O(n \log n + \Sigma_{\mathbf{o} \in O}|f_o| + \min\{|f_o|, \log n\}M)$ space complexity in $O(n \log n + \Sigma_{\mathbf{o} \in O}|f_o| + \min\{|f_o|, \log n\}c_{upd})$ time, where c_{upd} is the cost of inserting an object into the HNSW graph.*

3.2 Query Processing

First, we clarify the processing of queries with an *equality* filter constraint S (explained in Section 2.1). Specifically, the search first locates the node u in the label trie T corresponding to the exact

[1] All proofs can be found in the full version of the paper [1].

query label set, and then simply performs ANNS on the graph index associated with node u. In the case that u does not have a directly associated graph $u.G$, we perform ANNS using the graph index $u.index.G$ of the ancestor node pointed by $u.index$, and filter the results based on label equality during postprocessing.

Answering queries with *containment* and *overlap* predicates, however, is significantly more challenging. The proposed query algorithm first computes a disjoint cover set of label trie nodes $\mathcal{N}$ as well as a set of graph indices $\mathcal{G}$, and apply a multi-graph search algorithm to answer the query, described below.

Cover set and index set computation. Given a label filter $\{f_q, S\}$, we identify the disjoint subset of nodes whose subtrees collectively contain all objects $O_{f_q,S}$ that satisfy the filter, defined as follows.

Definition 3.7 (Disjoint Cover Set). A set of nodes $\mathcal{N} = \{u_1, u_2, \ldots, u_\kappa\}$ in label trie T *disjointly covers* a label filter $\{f_q, S\}$ if $\bigcup_{u \in C} O_u = O_{f_q,S}$ and $O_{u_i} \cap O_{u_j} = \emptyset$ for all $i \neq j$, where O_u is the set of objects in the subtree rooted at u.

Revisiting the example in Figure 3, suppose that the query's predicate S is *overlap*, with label set $f_q = \{4, 5\}$, meaning that the result objects must contain either label 4 or 5. Through an inverted index, we locate nodes u_3, u_5 (both corresponding to label 4), u_6 (label 5). Since these nodes have disjoint subtrees of the label trie T, they form a disjoint cover set $\mathcal{N} = \{u_3, u_5, u_6\}$.

Once a disjoint cover set $\mathcal{N}$ is computed (clarified soon), it is still challenging to process the query, since each node (say, $u \in \mathcal{N}$) corresponds to an HNSW graph $u.index.G$; thus, we need to search multiple such graphs effectively to answer the query. A naive approach would be to perform a separate ANN search on each graph and subsequently merge the results, which is clearly inefficient as the cost grows with the number of nodes in $\mathcal{N}$. Another idea is to perform a single HNSW search over all graphs, with a shared priority queue initiated with the entry point of each graph. However, in the absence of connecting edges between the different graphs, this strategy is prone to local optima: if an object from one graph that is relatively close to the query enters the queue early, the search may become confined to that graph, potentially overlooking closer candidates in other, disconnected graphs.

The composite structure of TrieHNSW provides a natural solution to the above problem. In particular, if a node u in the label trie T is a common ancestor of two or more nodes in the disjoint cover set $\mathcal{N}$, the HNSW graph corresponding to u contains edges that span across the subtrees of its descendants, which effectively connect the otherwise isolated graphs during the search, guiding the traversal across different coverage regions to avoid getting tripped in local optima. We call these *key ancestor nodes*, defined below.

Definition 3.8 (Key Ancestor Node). Let $\mathcal{N} = \{u_1, u_2, \ldots, u_\kappa\}$ be a disjoint node set of the label trie T. A node $a \in T$ is a *key ancestor node* with respect to $\mathcal{N}$, if (i) a is a common ancestor of at least two nodes in $\mathcal{N}$, and (ii) these nodes reside in the subtrees of at least two distinct children of a.

Intuitively, the graph index at a common ancestor node a contains edges that bridge the search spaces of its descendant nodes in $\mathcal{N}$. The second condition in the above definition ensures that a is a lowest common ancestor of the covered nodes in $\mathcal{N}$.

Algorithm 2: Node set and graph index set computation

```
   Input: Label trie $T$, inverted list $I$, query label filter $(f_q, S)$
1  Initialize cover set $\mathcal{N}$, ancestor node set $\mathcal{A}$, and graph set $\mathcal{G}$ to $\emptyset$
2  if $S$ is containment then
3      $l \leftarrow max\{f_q\}$
4      for $u \in I_l$ do
5          if $check\_contain(\text{prefix}(u), f_q)$ then
6              $\mathcal{N} \leftarrow \mathcal{N} \cup u$; add ancestors of $u$ to $\mathcal{A}$
7  if $S$ is overlap then
8      for $l \in f_q$ do
9          for $u \in I_l$ do
10             if $check\_not\_contain(\text{prefix}(u), f_q - l)$ then
11                 $\mathcal{N} \leftarrow \mathcal{N} \cup u$; add ancestors of $u$ to $\mathcal{A}$
12 for $a \in \mathcal{A}$ do
13     if $\exists c_1, c_2 \in a.ch, n_1, n_2 \in \mathcal{N}, n_1 \in O_{c_1}, n_2 \in O_{c_2}$ then
14         $\mathcal{G} \leftarrow \mathcal{G} \cup a.index.G$
15 for $u \in \mathcal{N}$ do
16     $\mathcal{G} \leftarrow \mathcal{G} \cup u.index.G$
17 return $\mathcal{N}, \mathcal{G}$
```

Algorithm 2 shows the pseudo-code for computing both the disjoint cover set $\mathcal{N}$ and the chosen key ancestor nodes $\mathcal{A}$. For a *containment* query, we first identify the label l in the query label set f_q that has the largest index, i.e., the least frequent label (Line 3). Using the inverted list I_l, we retrieve all nodes u where l appears on the path from the root to u. Since every object satisfying the containment filter must contain l, the set $O_{f_q,S}$ is fully covered by the union of the subtrees rooted at these disjoint nodes. For each candidate node u, we examine its label prefix; if prefix(u) contains every label in f_q, we add u to the disjoint cover set $\mathcal{N}$ (Lines 8-11).

For an *overlap* query, we process each label $l \in f_q$ separately. Using the inverted list I_l, we select all nodes whose prefix contains l. The union of the subtrees of these nodes is exactly $O_{f_q,S}$. To obtain a disjoint exact cover $\mathcal{N}$, we apply a refinement step: for each node u in the candidate set, we check whether prefix(u) contains any label from f_q other than l. If not, u is added to $\mathcal{N}$ (Lines 7-11).

After constructing $\mathcal{N}$, we identify all key ancestors of the nodes in $\mathcal{N}$, and obtain their corresponding graph indices (Lines 12-14). These, together with graph indices of the nodes in $\mathcal{N}$, form the index set $\mathcal{G}$ (Lines 15-16), which is returned together with $\mathcal{N}$ to facilitate query processing, explained soon.

Theorem 3.9. *Algorithm 2 runs in* $O\left(\sum_{u \in I_{l_{\max}}} |\text{prefix}(u)|\right)$ *time for a containment query, and* $O\left(\sum_{l \in f_q} \sum_{u \in I_l} |\text{prefix}(u)|\right)$ *time for an overlap query.*

In the running example of Figure 3, we identify u_4 and u_1 as key ancestor nodes for $\mathcal{N} = \{u_3, u_5, u_6\}$. Then, we add $u_4.G$ and Root.G (since $u_1.index = Root$) to the index set $\mathcal{G}$. Observe that the graph indices of nodes u_5, u_6, and u_3 in $\mathcal{N}$ are isolated; meanwhile, the edge from o_5 to o_3 in Root.G and the edge from u_5 to u_6 in $u_4.G$ connect the three isolated indices, thereby improving search effectiveness, explained next.

Multi-graph search. After obtaining the node set $\mathcal{N}$ and the index set $\mathcal{G}$, we proceed to answering the label-filtered ANNS query through the proposed *multi-graph search* algorithm, presented in Algorithm 3. First of all, following the HNSW search algorithm

Algorithm 3: Multi-Graph Search

```
Input: Label trie $T$, distinct cover set $\mathcal{N}$, graph index set $\mathcal{G}$, query
       ($\mathbf{q}$, $f_{\mathrm{q}}$, $S$), search parameter $ef$
1  $entries \leftarrow \emptyset$
2  for $u \in \mathcal{N}$ do
3      $entries \leftarrow entries \cup \{HNSW\_entry(u.index.G)\}$
4  Mark vectors in $entries$ as visited
5  $C \leftarrow entries$; $W \leftarrow entries$
6  while $|C| > 0$ do
7      $v \leftarrow$ nearest vector to $\mathbf{q}$ in $C$
8      $f \leftarrow$ farthest vector from $\mathbf{q}$ in $W$
9      if $\delta(v, q) > \delta(f, q)$ then break;
10     $M_{\mathrm{cnt}} \leftarrow 0$
11     for $i \leftarrow$ log_size($Root$) $downto$ 0 do
12         if $belong_{v,i} \neq NULL$ and $belong_{v,i} \in \mathcal{G}$ then
13             $G \leftarrow belong_{v,i}[0]$ // only use the bottom layer
14             for $w \in N_G(v)$ do
15                 if $w$ is not visited and $f_w \models_S f_{\mathrm{q}}$ then
16                     $M_{\mathrm{cnt}} \leftarrow M_{\mathrm{cnt}} + 1$
17                     Mark $v$ and $belong_{v,i}$ as visited
18                     if $\delta(w, \mathbf{q}) < \delta(f, \mathbf{q})$ or $|W| < ef$ then
19                         $C \leftarrow C \cup \{w\}$
20                         $W \leftarrow W \cup \{w\}$
21                     if $|W| > ef$ then remove $f$ from $W$ ;
22             if $M_{cnt} > 2M$ then break ;
23 return $W$
```

reviewed in Section 2.2, we utilize the upper layers of the graph indices corresponding to nodes in $\mathcal{N}$ to locate the entry points (Lines 1-3). In particular, we initialize two priority queues: a result queue W of size ef, and a candidate queue C. The search follows a similar procedure to HNSW-based ANNS. In each step, we extract the vertex v closest to the query vector $\mathbf{q}$ from the candidate set C. If the distance between v and $\mathbf{q}$ is larger than the distance between the farthest result point $f \in W$ and $\mathbf{q}$, the search terminates (Lines 7-9). Otherwise, we continue the search by traversing the graph indices where v appears according to the belong array, in descending order of the logarithmic size (Lines 12-23).

Specifically, if v belongs to an index of logarithmic size i that is included in the current graph set $\mathcal{G}$, we visit all neighbors of v in that index that satisfy the filter and have not been visited yet (Lines 15-22). By utilizing edges from the indices of key ancestor nodes, we incorporate global connectivity information, thereby linking points from different disjoint coverage nodes, which enhances search effectiveness. Whenever a neighbor is closer to $\mathbf{q}$ than the current farthest result f, it is inserted into both C and W (Lines 19-21). When W reaches its maximum size of ef objects, the farthest object f is removed (Line 21).

In addition, we maintain a counter M_{cnt} to track the total number of visited neighbors that satisfy the label filter ((Lines 10 and 16). The search terminates when M_{cnt} exceeds $2M$, which is the typical maximum number of base-layer points in standard HNSW, to avoid exploring an excessive number of neighbors (Line 22). Finally, the set W is returned as the query result.

Algorithm 4: TrieHNSW-Query

```
Input: Label trie $T$, inverted index $I$, query label set $f_{\mathrm{o}}$, query ($\mathbf{q}$, $f_{\mathrm{q}}$,
       $S$), search parameters $ef$, $k$, percentage $\tau$
1  $\mathcal{N}, \mathcal{G} \leftarrow Algorithm\ 2(T, I, f_{\mathrm{q}}, S)$
2  $sort(\mathcal{N})$; $\mathcal{N}_1 \leftarrow \emptyset$
3  for $u \in \mathcal{N}$ do
4      $\mathcal{N}_1 \leftarrow \mathcal{N}_1 \cup \{u\}$
5      if $\frac{|O_{\mathcal{N}_1}|}{|O_{\mathcal{N}}|} > \tau$ then break ;
6  $results \leftarrow multi\_graph\_search(T, \mathcal{N}_1, \mathcal{G}, f_{\mathrm{q}}, S, ef)$
7  $\mathcal{N}_2 \leftarrow \{u | u \in \mathcal{N}, u.index.G$ is unvisited$\}$
8  $results \leftarrow results \cup multi\_graph\_search(T, \mathcal{N}_2, \mathcal{G}, f_{\mathrm{q}}, S, ef)$
9  return top-k of results
```

Two-pass implementation. In our implementation, instead of performing the query in a single multi-graph search across all nodes, we adopt a *two-pass* strategy. A key observation is that since we re-index labels based on their frequency, the sizes of the graphs corresponding to the disjoint cover set nodes $\mathcal{N}$ for a given query can be highly imbalanced, i.e., a small number of nodes in $\mathcal{N}$ may contain the vast majority of the relevant objects. If we search across all nodes in $\mathcal{N}$ simultaneously, the search may become inefficient and poorly guided, since the smaller graphs contain limited structural information, leading to many unproductive exploratory steps.

To address this, we answer the query in two passes, as shown in Algorithm 4. In the first pass, we sort all nodes in $\mathcal{N}$ in descending order of the sizes of their associated graphs, and perform a search starting from the few largest nodes that collectively cover, say $\tau = 90\%$ of the total objects (Lines 1-6). During this pass, we also leverage edges from key ancestors and mark nodes whose graphs are traversed. The second pass is then a follow-up search on the graphs of the remaining, unvisited nodes in $\mathcal{N}$. This retrieves missed candidates, thereby further improving result recall (Lines 7-9).

In the running example in Figure 3, assuming $\tau = 0.6$, the first pass starts from $u_3.G$ and $u_5.G$. When the search reaches o_5, we access o_6 via the edge from u_5 to u_6 in $u_4.G$ and mark the index $u_6.G$ (which contains o_6) as visited. Hence, in the second phase, we do not need to search $u_6.G$ and can terminate the search directly.

3.3 Index Update

The key to updating the TrieHNSW index lies in ensuring that the graph index maintained at each label trie node remains up-to-date. For deletions, similar to most ANNS indexes such as HNSW described in Section 2.2, we apply soft deletion and simply mark the corresponding vertices as invalid, which are skipped during search. In the following, we focus on objection insertions, which trigger non-trivial index maintenance.

The insertion of an object $\mathbf{o}$ consists of three main steps: (i) inserting $\mathbf{o}$ into the label trie T, (ii) updating the affected graph indices along the insertion path in T, and (iii) adjusting which graph index each node uses, and rebuilding an index if it becomes insufficiently effective for its associated node. For (i), all operations are lightweight, requiring only $O(|f_{\mathbf{o}}|)$ time, where $|f_{\mathbf{o}}|$ is the number of labels in $\mathbf{o}$. For step (ii), since an object belongs to at most $O(\min\{|f_{\mathrm{o}}|, \log n\})$ graphs, the cost is bounded by $O(\min\{|f_{\mathrm{o}}|, \log n\}\cdot$

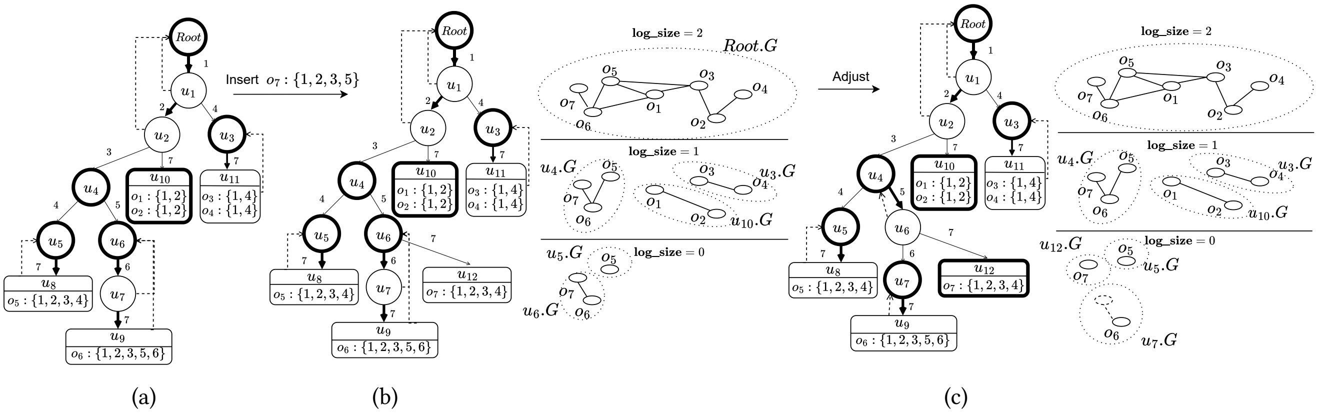


Figure 4: An example of the object insertion with the TrieHNSW index from Figure 3.

Algorithm 5: TrieHNSW-Insert

```
   Input: object $\mathbf{o}$, label set $f_\mathbf{o}$, HNSW construction parameters $M$, $ef_c$
1  $f_\mathbf{o} \leftarrow f_\mathbf{o} \cup \{|\mathcal{L}| + 1\}$
2  $TrieInsert(Root, \mathbf{o}, f_\mathbf{o})$
3  $InsertNode(Root, \mathbf{o}, f_\mathbf{o}, M, ef_c)$
4  procedure insertNode(node, $\mathbf{o}$, $f_\mathbf{o}$, $M$, $ef_c$):
5      $u \leftarrow node$
6      for $idx \leftarrow 1$ to $|f_\mathbf{o}|$ do
7          if $u.index = u$ then
8              $HNSWInsert(u.G, \mathbf{o}, M, ef_c)$
9              if log_size$(u) \neq \lfloor \log_2 (|O_u| - 1) \rfloor$ then
10                 if $u.ch_{\text{heavy}} \neq NULL$ then
11                     $u.ch_{\text{heavy}}.G \leftarrow CopyGraph(u.G)$
12                     $c \leftarrow u.ch_{\text{heavy}}$
13                     $changeIndexNode(c)$
14                     if log_size$(c.G) >$ log_size$(c) + 1$ then
15                         $Rebuild(c.G)$
16                     $u.ch_{\text{heavy}} \leftarrow NULL$
17                 if $u \neq Root$ and log_size$(u) =$ log_size$(u.p)$
                     then
18                     $DestoryGraph(u.G)$
19                     $u.p.ch_{\text{heavy}} \leftarrow u$; $u.index \leftarrow u.p.index$
20                 update $belong$
21         $u \leftarrow$ the child node of $u$ corresponding to $f_\mathbf{o}[idx]$;
```

c_{upd}), where n is the number of objects in the dataset O, and c_{upd} is the cost for inserting an object into an HNSW index.

Regarding step (iii), adjusting which index a node uses may involve copying or deleting indices, and rebuilding an index entirely is an expensive operation. To maintain update efficiency, we adopt a lazy update strategy: we allow the effective index for each node to be a 4-approximate index (see Definition 3.2). By tolerating a slightly weaker index guarantee, we postpone expensive rebuilds until the accumulated changes exceed a predefined threshold. The cost of rebuilding is then amortized over subsequent updates, which bounds the *amortized* time complexity per insertion to $O\big(|f_o|M + \min\{|f_o|, \log n\} \cdot c_{\text{upd}}\big)$.

Algorithm 5 outlines the insertion procedure. First, we append the special label $|\mathcal{L}| + 1$ to the label set $f_\mathbf{o}$ of the new object $\mathbf{o}$, and insert $\mathbf{o}$ into the trie T, updating the corresponding inverted lists (Lines 1-2). Next, we incorporate the object into the affected graph indexes, adjusting their structures accordingly. Specifically, we traverse the root-to-leaf path of o in T (Lines 5-21). At each visited node u, we check whether an index $u.G$ exists. If so, we insert o into $u.G$, following the standard HNSW insertion algorithm (Lines 7-8). Then, we check whether u.log_size has increased. If so, then u is no longer a suitable index node for its current logarithmic size (Lines 9-10). When this happens, if u has a heavy child ch, we copy the index $u.G$ to $ch.G$, and update the *index* pointers of all descendants that originally used $u.G$ (Lines 10-16).

Note that the copied index contains not only objects from O_{ch} but also objects from other subtrees of u. We opt for copying rather than rebuilding because reconstruction is significantly more expensive. However, when the number of irrelevant objects in the copied index becomes too large, a rebuild is necessary to preserve search quality. In particular, we trigger a full rebuild when the difference between the logarithmic size of the graph and that of the node reaches 2 (Lines 14-15). If the new log_size of u equals that of its parent, then u itself no longer needs to maintain a separate index, and can simply point to its parent's index. In this situation, we delete $u.G$, and adjust the relevant pointers (Lines 17-19). The following theorem guarantees the efficiency of the insertion operation and the validity of the resulting index structure.

Theorem 3.10. *Algorithm 5 inserts each new object in amortized $O\big(|f_o|M + \min\{|f_o|, \log n\} \cdot c_{upd}\big)$ time while ensuring that for every node u, the associated index remains a 4-approximate index for O_u.*

Consider again the example TrieHNSW index shown in Figure 3. Suppose we insert a new object $\mathbf{o}_7$ with label set $\{1, 2, 3, 5\}$. As illustrated in Figure 4b, we first insert $\mathbf{o}_7$ into the trie T, and add it to the graph indices along the insertion path: $Root.G$, $u_4.G$, and $u_6.G$. Then, we adjust the index assignments in T as shown in Figure 4c. After the insertion, the size of O_{u_6} becomes 2, which changes its logarithmic size. Consequently, we copy the index $u_6.G$ to its heavy child u_7, and update the index pointers so that both $u_7.index$ and $u_9.index$ point to u_7. Meanwhile, since u_6 now has the same logarithmic size as its parent u_4, we discard $u_6.G$, and set $u_6.index = u_4$, while marking u_6 as the heavy child of u_4. Note that copying $u_6.G$ to u_7 means that the new $u_7.G$ contains vertices that do not belong to O_{u_7}. To preserve update efficiency, we keep these stale edges and postpone a full rebuild until the fraction of irrelevant vertices exceeds a threshold, according to Algorithm 5.

Table 2: Summary of datasets in the experiments.

Dataset	Vector Type	Dim.	# Vector	# Query	Attribute Type	$\lvert\mathcal{L}\rvert$	Avg. Selectivity	
							Containment	Overlap
arXiv	text	768	132,678	200	year, month, task, etc	4231	0.09	0.19
TripClick	text	768	1,055,976	1000	clinical area	29	0.07	0.19
LAION1M	image	512	1,000,448	1000	entities, locations, etc	30	0.09	0.19
YFCC	image+audio	192	1,000,000	1000	classes	181,931	0.038	0.322
YTB-Audio	audio	128	5,000,000	1000	classes	3862	0.09	0.19
YTB-Video	audio	1024	1,000,000	200	classes	3862	0.09	0.19

4 EXPERIMENTS

We experimentally compare the proposed solution TrieHNSW against the current state of the art label-filtered ANNS methods. TrieHNSW is implemented using C++. We have enabled all optimizations in the code of the competitors. All experiments are conducted on a Linux machine with an Intel Xeon Gold 5320 CPU @ 2.20GHz and 1TB of memory, using a single thread.

4.1 Experimental Settings

Datasets and queries. The experiments involve six real datasets and their query settings tested in related research for filtered ANNS [3, 9, 25, 30]: (i) arXiv[23], containing papers with methodology embeddings; (ii) TripClick[28], a large-scale health information retrieval dataset, comprising click logs from the Trip Database health web search engine; (iii) LAION1M [29], a large-scale, publicly available dataset of CLIP-filtered [27] image-text pairs; (iv) YFCC [32], a standard CV dataset with image/video embeddings; (v) YTB-Audio and (vi) YTB-Audio [2], a large-scale benchmark of millions of videos annotated with thousands of machine-generated topical entities. Table 2 provides key statistics for each each dataset.

We generate three types of queries for each dataset, with *containment*, *overlap*, and *equality* label constraints, respectively. For all datasets, we generate queries covering the full range of possible selectivities. Except for YFCC and TripClick, whose label distributions are too skewed for precise control, we maintain an average selectivity of approximately 0.09 for queries with a *containment* label predicate, and 0.19 for *overlap* queries, reflecting the stricter constraint and typically higher selectivity of *containment*.

Evaluation metrics. Following prior work [3, 9, 16], we use queries per second (QPS) to measure query efficiency in terms of throughput, and recall to measure result quality. In particular, let R^* be the exact k nearest neighbors and R is the ANNS result, recall is define as $Recall(R) = \frac{|R \cap R^*|}{|R^*|} = \frac{|R \cap R^*|}{k}$. We set $k = 10$ for all experiments. Additionally, we use $selectivity(O, f_q, S) = |O_{f_q,S}|/|O|$ to represent the proportion of objects in the dataset that satisfy the filter.

Competitors. We include the following methods (detailed in Section 5) in our experimental comparisons: (i) *ACORN-1* and *ACORN-γ* [25]: following the settings in [25], we set $M = 32$ and $M_\beta = 64$ for all datasets; for ACORN-γ, we set $\gamma = 100$ in across all datasets. (ii) *NaviX* [30]: following the settings in [30], we set $M = 32$ and $ef_c = 400$ for all datasets. (iii) *Filtered-DiskANN*[16]: we set $R = 128$ and $L = 180$, and extend the filter detection logic by modifying `detect_common_filters` to support label-containment and label-overlap. (iv) *Rwalks* [3]: following the settings in [3], we set $M = 32$ and $ef_c = 400$. (v) *UNG* [9]: following the settings in [9], we set $\alpha = 1.2$, $R = 32$, $L = 100$. For cross-group edge construction, we set $\delta = 6$, and use $\sigma = 16$ entry vectors for query processing. (vi) *TrieHNSW*: we set $M = 32$ (YTB-Video) and $M = 16$ (other dataset), with $ef_c = 400$, $\tau = 0.9$ for all datasets. For all methods, we change ef to control recall and QPS. Our code and more experimental details can be found in [1].

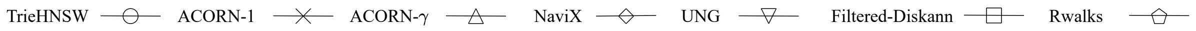
4.2 Comparison Results

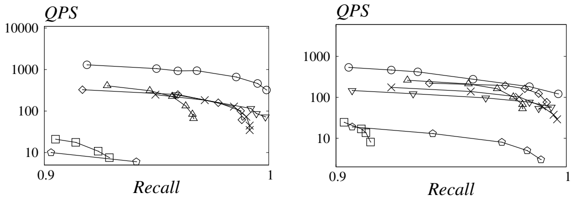
Query performance. First of all, for queries with *equality* label constraints, the search algorithm for TrieHNSW is relatively simple (see Section 3.2) and resembles that of UNG. Hence, TrieHNSW and UNG have comparable performance for such queries, and both significantly outperform all other competitors, which is consistent with results in [9]. Detailed results on *equality* queries can be found in the full version [1]. In the following, we focus on results for *containment* and *overlap* queries.

Figures 5-10 plot the QPS-recall curves for the 6 datasets in our experiments. We report methods that achieve at least 0.9 recall for arXiv, LAION1M and TripClick. For YFCC, YTB-Audio, and YTB-Video, i.e., datasets that are large, contain many label categories, and, consequently, are more challenging, most methods fail to achieve strong performance. Hence, we only report methods that achieve at least 0.8 recall. Rwalks is excluded from the experiments on YFCC because its memory consumption exceeds 1 TB.

The proposed TrieHNSW consistently yields the best performance across all datasets. On datasets with relatively few label categories and simple label sets, such as TripClick and LAION1M, TrieHNSW remains highly competitive. At 0.99 recall, it is 3× faster than the best baseline (UNG) on LAION1M and 4× faster than the best baseline (Rwalks) on TripClick for containment queries.

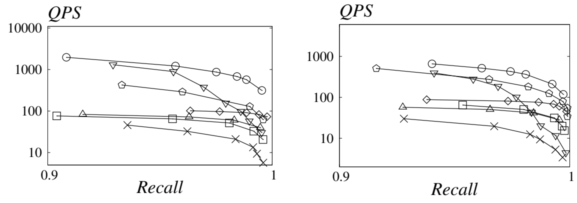
On datasets with more label categories and complex label sets, TrieHNSW demonstrates a substantial advantage over its competitors. YFCC exhibits a large number of labels with an uneven distribution. Other methods perform well only on either containment or overlap queries, but not both. For example, UNG performs well on containment queries but fails to reach high recall on overlap queries. TrieHNSW handles both query types effectively, and delivers a 3-5× speedup. On YTB-Audio, TrieHNSW is two orders of magnitude faster than its competitors. On YTB-Video, most baselines fail to reach 0.8 recall, whereas TrieHNSW not only achieves high recall but also runs an order of magnitude faster. In summary, compared to existing methods, TrieHNSW maintains consistently high efficiency across all datasets and query types, with particularly outstanding performance on datasets with complex label distributions.





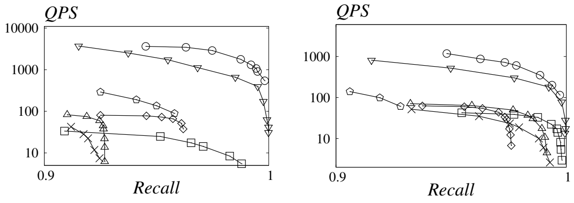
(a) Containment (b) Overlap

Figure 5: QPS vs. recall on arXiv.



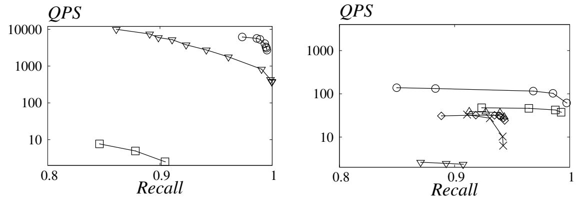
(a) Containment (b) Overlap

Figure 6: QPS vs. recall on TripClick.



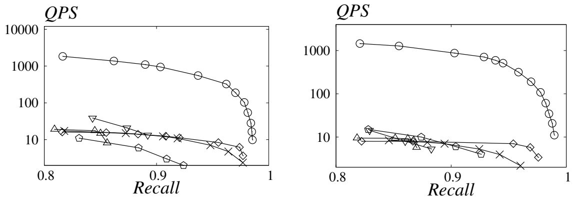
(a) Containment (b) Overlap

Figure 7: QPS vs. recall on LAION1M.



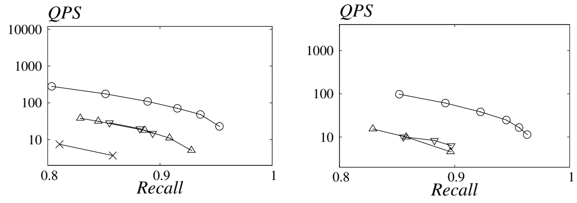
(a) Containment (b) Overlap

Figure 8: QPS vs. recall on YFCC.



(a) Containment (b) Overlap

Figure 9: QPS vs. recall on YTB-Audio.



(a) Containment (b) Overlap

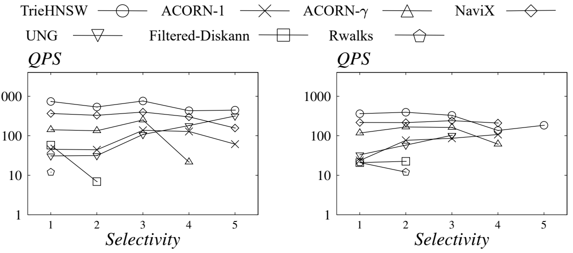
Figure 10: QPS vs. recall on YTB-Video.

Impact of query selectivity. Next, we evaluate the performance of overlap and containment queries under varying selectivity levels across all datasets. Note that the label distribution is discrete, so we cannot generate a large number of queries with a specific selectivity. For datasets except YTB-Audio and YTB-Video, we generate queries whose selectivity falls into five intervals: $[2^{-2}, 2^{-1})$, $[2^{-3}, 2^{-2})$, $[2^{-4}, 2^{-3})$, $[2^{-5}, 2^{-4})$, and $[2^{-6}, 2^{-5})$. For YTB-Audio and YTB-Video, where label frequencies are generally lower, we create queries in the intervals $[2^{-3}, 2^{-2})$ to $[2^{-7}, 2^{-6})$. For each interval, we report the QPS achieved at 0.99 recall on arXiv, TripClick, and LAION1M. Most baselines struggle to reach high recall on challenging datasets YFCC, YTB-Audio, YTB-Video. Hence, we report QPS at 0.9 recall for these. Due to the label distributions in YFCC and TripClick, it is infeasible to generate containment queries spanning a continuous range of target selectivities. Therefore, we do not include containment-query results for these two datasets; we only report overlap queries on these two datasets and defer the results to our technical report [1] for the interest of space.

Figures 11-14 present the evaluation results. On the x-axis, each coordinate value i represents the interval $[2^{-(i+1)}, 2^{-i})$. Observe that TrieHNSW maintains consistently high performance across all selectivity levels and datasets. On the LAION1M dataset, UNG slightly outperforms TrieHNSW when the selectivity is very low. This is because LAION1M has relatively few labels, and low-selectivity queries involve only a small number of highly similar label sets. UNG's static LNG can capture these relationships in advance by pre-building cross edges, whereas TrieHNSW incurs overhead to dynamically construct edges across label sets. On the other hand, UNG performs poorly on datasets with more labels. TrieHNSW shows a clear advantage on datasets with a larger number of labels. On arXiv, YTB-Audio, and YTB-Video, TrieHNSW is the only approach that maintains both high recall and high QPS across all selectivity levels and query types.

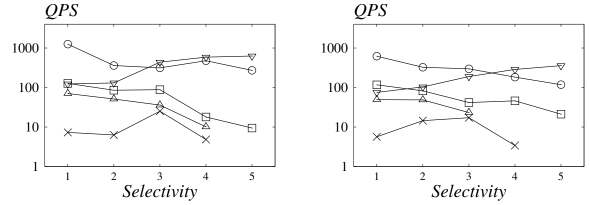
Index cost. Figures 15-16 show the memory consumption and indexing time of all solutions. Note that the RAM usage of Rwalks on YTB-Audio and YTB-Video is not included in the figure because it consumes 32GB on YTB-Audio and 147GB on YTB-Video, which would dominate the scale. Among methods specifically designed for label-filtered ANNS, index construction of TrieHNSW is faster than that of Filtered-DiskANN, but slower than UNG and Rwalks, while still remaining within the same order of magnitude. This overhead arises since we spend additional time constructing the hierarchical trie-graph structure to capture label relationships. The higher index building cost of TrieHNSW is compensated by its superior query performance, as demonstrated in the results above.

Except for YFCC and YTB-Audio, the memory usage of TrieHNSW is comparable to that of the other methods. The higher memory on YFCC and YTB-Audio is due to the larger number of label sets and more complex label relationships in these datasets, which require additional space to achieve high recall. Recall from Figure 8 that our method is the only one that achieves high performance on both query types on YFCC, outperforming other approaches by a factor of 3-5×; on YTB-Audio (Figure 9), the performance gap reaches two



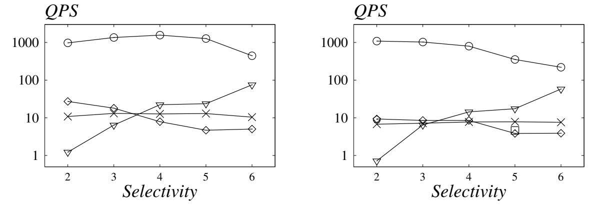
(a) Containment (b) Overlap

Figure 11: Impact of selectivity on the arXiv.



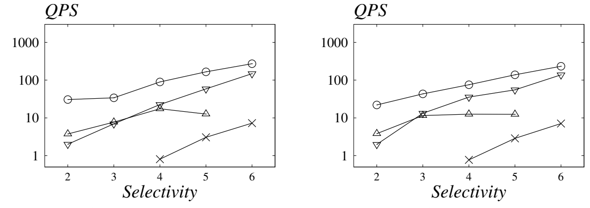
(a) Containment (b) Overlap

Figure 12: Impact of selectivity on the LAION1M.



(a) Containment (b) Overlap

Figure 13: Impact of selectivity on the YTB-Audio.



(a) Containment (b) Overlap

Figure 14: Impact of selectivity on the YTB-Video.

orders of magnitude. These results confirm that our method constructs a highly effective index within a reasonable space overhead, even for datasets with challenging label distributions.

Update cost. Among label-aware indices, only Filtered- DiskANN [16] natively supports updates. Rwalks [3], which performs additional graph refinement after constructing the complete base graph, does not support any update operations. Although UNG claims to support insertions, updating its underlying LNG structure requires a dedicated algorithm; further, modifying the LNG necessitates rebuilding the associated cross-edges, which renders the insertion time hard to predict. Since no algorithmic details or implementation for such updates are provided in [9], we do not include it in this comparison. Figure 17 compares the update time of TrieHNSW with that of Filtered-DiskANN. For each dataset, we pre-build the index on 80% of the objects and then insert the remaining 20%, reporting the average insertion time. The results show that TrieHNSW is 3–5× faster than Filtered-DiskANN. This advantage stems from the fact that Filtered-DiskANN must perform a label-filtered ANNS during each insertion, which is significantly slower than the plain greedy search employed in our update procedure. These results demonstrate the efficiency of TrieHNSW index updates, which is important for applications with dynamic data.

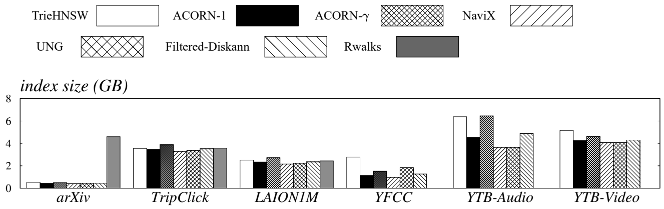


Figure 15: Memory cost.

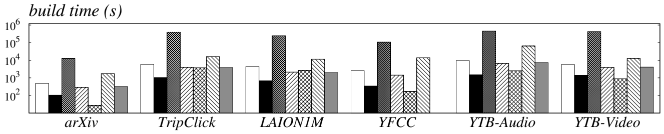


Figure 16: Indexing time.

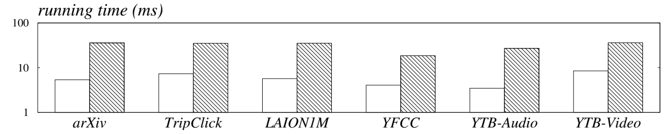


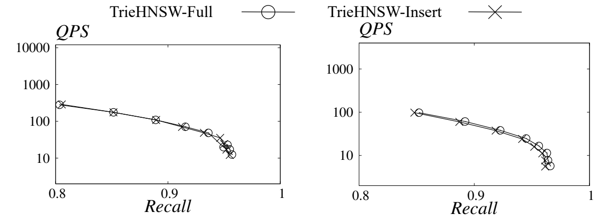
Figure 17: Insertion time.

Impact of update. For each dataset, we build two indices. The first is constructed directly on the entire dataset, while the second index is pre-built on 80% of the objects and then updated by inserting the remaining 20%. We compare the label-filtered ANNS performance on these two indices. We only report the results on YTB-Video dataset in Figure 18; the other datasets lead to similar conclusions, and can be found in the full version [1]. The results show a negligible performance gap between the two indices. This is because even with dynamic insertions, for which some graph indices may degrade from a 2-approximate to a 4-approximate index in the worst case, the performance of TrieHNSW remains stable since our search algorithm leverages multiple layers of indices. In particular, if edges in one layer become less effective, edges from other layers can compensate it, leading to strong robustness. The experiment confirms that our efficient update mechanism does not compromise the overall index quality.

Impact of parameters and ablation study. We also conduct experiments to evaluate the impact of the parameters m, τ, and ef_c on query performance, indexing time, and memory usage. Additionally, we perform an ablation study on the proposed optimizations, such as multi-graph search and two-pass search. Interested readers are referred to our technical report [1] for details.

5 RELATED WORK

Approximate nearest neighbor search. Existing ANNS solutions can be broadly categorized into three classes: *LSH-based*, *PQ-based*,



(a) Containment

(b) Overlap

Figure 18: Impact of update on the YTB-Video.

and *graph-based*. Specifically, locality-sensitive hashing (LSH) and its variants, e.g., [4, 5, 11, 17, 35], provide rigorous theoretical error guarantees, but often incur considerable time and space overhead in practice. Product quantization (PQ) [18] partitions the high-dimensional vector space into multiple low-dimensional subspaces, and quantizes each subspace to compress the vectors. PQ and its successors, e.g., [6–8, 15, 20] trade result accuracy for significantly lower memory usage, achieving high scalablility. Graph-based methods [12, 14, 21, 22, 31] construct a proximity graph over the dataset, where vertices represent vectors and edges connect similar vectors. Notable graph-based approaches include HNSW [22], NSG [14], and DiskANN [31], which are widely adopted in industry due to their favorable tradeoffs between efficiency and accuracy. None of these methods, however, considers label-based filtering.

ACORN and NaviX. ACORN [25] and NaviX [30] are label-agnostic, in-filtering methods (introduced in Section 1) that extends HNSW to filtered ANNS by (i) enforcing the filter of the retrieved vertices in each iteration, and (ii) augmenting the traversal of the proximity graph G in each iteration by (selectively) incorporating two-hop neighbors, in an effort to mitigates the problem that the filtering sparsifies G. Specifically, in each step of traversing G, ACORN considers candidates that meet the filter within two hops. As argued in [30], such indiscriminate expansion to two-hop neighbors also increases traversal cost, which may defeat the purpose when the filter is less selective. To address this, NaviX dynamically adapts the neighbors explored based on the current number of feasible candidates after filtering. Intuitively, when the candidate pool is large, NaviX reduces to HNSW and restricts the traversal to one-hop neighbors; conversely, when valid candidates are scarce, NaviX analyzes the distance profile of the one-hop neighbors, and selectively expands to promising two-hop neighbors that are more likely to reach the true nearest neighbors.

While these two-hop heuristics can be effective to a certain extent, as we mentioned in Section 1, such label-agnostic solutions cannot exploit relationships between the label sets of different objects in the dataset, which inherently limits their effectiveness when applied to label-filtered ANNS.

Filtered-DiskANN. Filtered-DiskANN adapts the DiskANN [31] / Vamana-style navigable graph to support label-filtered ANNS, by injecting label information into both search and graph pruning. During index construction, the method incrementally inserts points and invokes a label-aware pruning rule whenever the candidate proximity neighborhood exceeds a predefined degree budget. This pruning rule ensures that an edge is removed only if it can be reliably "covered" by an alternative neighbor, which considers both vector distance and label compatibility needed for future filtered searches. When processing the query, Filtered-DiskANN expands only label-compatible neighbors, ensuring that exploration stays within the admissible subgraph defined by the query filter. As mentioned in Section 1, a major limitation of Filtered-DiskANN is that it primarily targets the single-filter exact-match setting, and does not naturally generalize to conjunctive label set predicates such as *containment* without significant performance degradations.

RWalks. RWalks [3] targets filtered vector search based on existing graph indices without changing their connectivity, by adding an attribute enrichment post-processing step, plus a filter-aware search procedure. In particular, RWalks involves a twin priority queue that stores popped-but-valid candidates, and merges them back at the end. It further improves recall by biasing traversal with a hybrid distance that prioritizes visiting valid regions, and addresses attribute sparsity by transforming binary attributes into dense, neighborhood-aware representations using graph diffusion via random walks. During traversal, RWalks preferentially evaluates neighbors that either satisfy the filters or are predicted by enriched attributes to guide the walk toward valid regions.

A key limitation of RWalks is that the hybrid scoring relies on maintaining and computing label-related signals; when the label vocabulary is large, the storage overhead of the enriched label representations and the per-expansion cost of evaluating the mixed score can be substantial, which may defeat their efficiency gains.

UNG. As mentioned in Section 1, unified navigating graph (UNG) [9] exploits the relationships among labels by constructing a label navigating graph (LNG). The LNG encodes the containment relationships between label sets for all vectors, represented as a directed acyclic graph where a directed edge from one label set to another indicates the minimal superset relation. Using the LNG, UNG builds a graph among vectors within each label set, and then creates cross edges between the graphs of label sets that are directly connected in the LNG. During search, UNG starts from entry points in multiple label sets, and utilizes these cross edges to traverse across different label-set graphs. However, cross edges are only established between adjacent nodes in the LNG. Consequently, this approach is primarily efficient for queries involving a small number of labels, and/or when the label relationships are relatively simple. It does not scale well to queries with many labels, or applications involving a wide variety of label sets. In addition, UNG does not provide an effective LNG update mechanism.

Finally, we note that there exist generic solutions for attribute-filtered ANNS [13, 16, 24, 34, 38, 40], which are often less performant for specific attribute filters compared to specialized solutions, as shown in [13, 40]. Beyond label filters, range-filtered ANNS has also been studied [19, 36, 37, 40], which is orthogonal to our work.

6 CONCLUSION

This paper presents TrieHNSW, an effective indexing scheme for answering label-filtered ANNS over high-dimensional vectors, which outperforms existing approaches in terms of result accuracy, query efficiency, and update overhead, as demonstrated with extensive experiments on multiple real datasets. Regarding future work, we plan to extend our solution to on-disk indices, and study effective ANNS with both label and range filtering with multiple attributes.

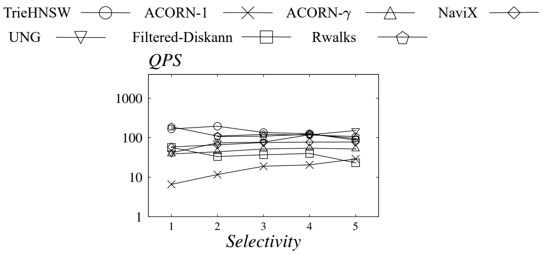


Figure 19: Impact of selectivity on the TripClick.

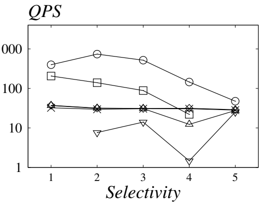


Figure 20: Impact of selectivity on the YFCC.

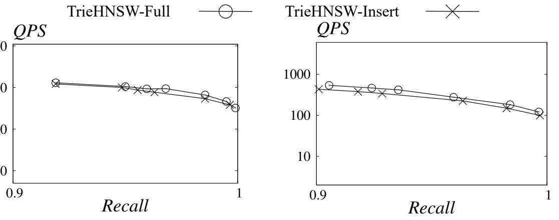
A PROOFS

Proof of Lemma 3.4. We prove by contradiction. Assume that node u has two distinct children c_1 and c_2 such that $\log_2(|O_{c_1}|) = \log_2(|O_u|)$ and $\log_2(|O_{c_2}|) = \log_2(|O_u|)$. Without loss of generality, let $|O_{c_1}| \leq |O_{c_2}|$. Since O_u is the set of all objects in the subtree of u and c_1, c_2 are disjoint subtrees, we have $|O_u| \geq |O_{c_1}| + |O_{c_2}| \geq 2|O_{c_1}|$. Consequently, $\log_2(|O_u|) \geq \log_2(2|O_{c_1}|) = \log_2(|O_{c_1}|) + 1 = \log_2(|O_u|) + 1$, which is a contradiction. Therefore, the assumption is false, and the lemma holds.

Proof of Lemma 3.5. For the object o, it only belongs to the graph indexes of its ancestors in the Trie; hence there are at most $O(|f_o|)$ such nodes. Moreover, because a node u maintains its own index ($u.index = u$) only when $\text{log\_size}(u) \neq \text{log\_size}(u.p)$, the number of distinct log_size values along the root-to-leaf path is $O(\log n)$. Therefore, o can belong to at most $O(\log n)$ such indexes. Combining the two bounds yields the result.

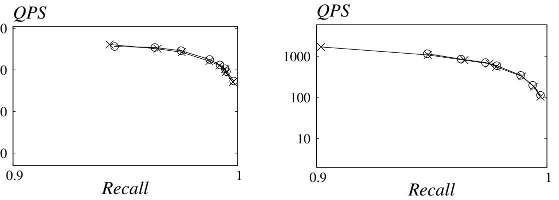
Proof of Theorem 3.6. The space complexity of our index consists of the following components: the Trie structure, the inverted lists I, the belong array, and all the HNSW graph indexes. Each object o is inserted into the Trie by traversing (or creating) at most $|f_o|$ nodes. Hence the Trie and the associated inverted lists require $O(\sum_{o \in O} |f_o|)$ space. The $belong_{\mathbf{o},i}$ indicates whether object o belongs to a graph of logarithmic size i. Since i can take at most $O(\log n)$ distinct values, the total size of the belong array is $O(n \log n)$. Through Lemma 3.5, the overall space occupied by all graph indexes is $O\Big(\sum_{o \in O} \min\{|f_o|, \log n\}\, M\Big)$ Summing the three parts, the total space complexity is $O\Big(n \log n + \sum_{o \in O} |f_o| + \sum_{o \in O} \min\{|f_o|, \log n\}\, M\Big)$.

For construction time, we first build the Trie and the inverted lists, which takes $O(\sum_{o \in O} |f_o|)$ time. Initializing and updating the belong array costs $O(n \log n)$. The dominant cost is the HNSW insertion: each object o is inserted into at most $\min\{|f_o|, \log n\}$ graphs, and each insertion has complexity c_{upd} (the time to insert one point into a plain HNSW graph). Hence the total construction time is $O\Big(n \log n + \sum_{o \in O} |f_o| + \sum_{o \in O} \min\{|f_o|, \log n\}\, c_{\text{upd}}\Big)$.



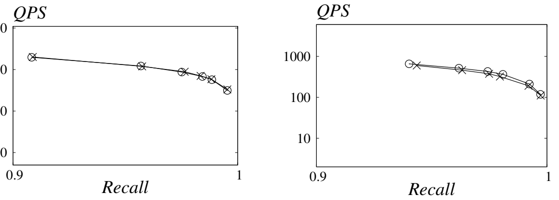
(a) Containment (b) Overlap

Figure 21: Impact of update on the arxiv.



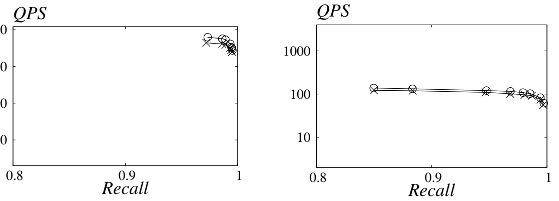
(a) Containment (b) Overlap

Figure 22: Impact of update on the LAION1M.



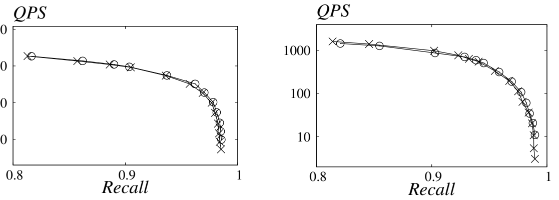
(a) Containment (b) Overlap

Figure 23: Impact of update on the TripClick.



(a) Containment (b) Overlap

Figure 24: Impact of update on the YFCC.



(a) Containment (b) Overlap

Figure 25: Impact of update on the YTB-Audio.

Proof of Theorem 3.9. For containment queries, we access ancestors of nodes in $I_{\max}$ and examine the prefix of each node. For overlap queries, we access ancestors of nodes from the IVF list of every label in f_q and also examine the prefix of each node. The number of ancestors of a node u is $|\text{prefix}(u)|$, and each ancestor can be accessed in $O(1)$ time. Therefore, the theorem holds.

Proof of Theorme 3.10. Operations on the Trie and updates to the inverted lists take $O(|f_o|)$ time. Traversing the path from the root to the leaf also costs $O(|f_o|)$. For HNSW insertion, an object

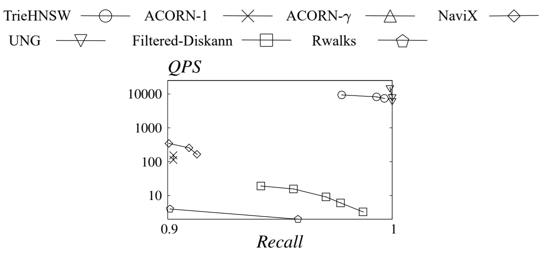


Figure 26: QPS vs. recall on arXiv (Equality).

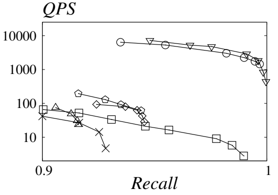


Figure 27: QPS vs. recall on TripClick (Equality).

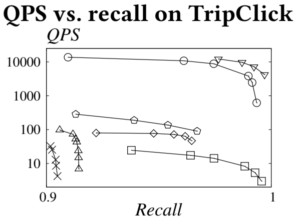


Figure 28: QPS vs. recall on LAION1M (Equality)..

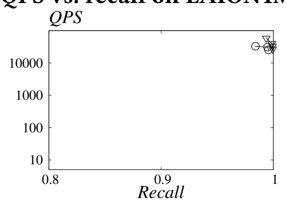


Figure 29: QPS vs. recall on YFCC (equality).

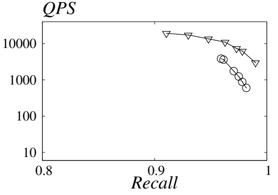


Figure 30: QPS vs. recall on YTB-Audio (equality).

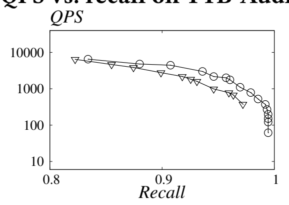
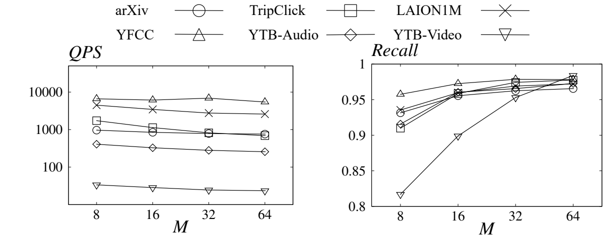


Figure 31: QPS vs. recall on YTB-Video (equality).

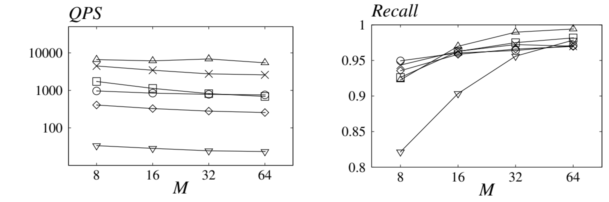
belongs to at most $O(\min\{|f_o|, \log n\})$ graphs, so the total insertion cost is $O(\min\{|f_o|, \log n\} \cdot c_{\text{upd}})$. Adjusting the index assigned to a node may involve copying or deleting an index; one such operation on node u costs $O(|O_u|M)$ and is triggered only when a change in $|O_u|$ alters $mathsflog\_size(u)$. After one such change, the next copy/delete on this index occurs after $\Omega(|O_u|)$ insertions into O_u. Hence the amortized cost per insertion contributed by copying/deleting is $O(M)$. Since an insertion affects at most $|f_o|$ subtrees, the total amortized overhead from copying/deleting is $O(|f_o|M)$.

Rebuilding the index for node u costs $O(|O_u|c_{\text{upd}})$. After a rebuild, $u.G$ contains at most $2^{\text{log\_size}(u)+1} - 1$ nodes, and the next rebuild is triggered after $\Omega(|O_u|)$ further insertions. Therefore, the amortized cost per insertion from rebuilding is $O(c_{\text{upd}})$. Because an insertion touches at most $\min\{|f_o|, \log n\}$ graphs, the total amortized rebuilding overhead is $O(\min\{|f_o|, \log n\} \cdot c_{\text{upd}})$. Summing all parts yields an amortized time complexity per update of $O\big(|f_o|M + \min\{|f_o|, \log n\} \cdot c_{\text{upd}}\big)$. Moreover, the graph u.index.G is



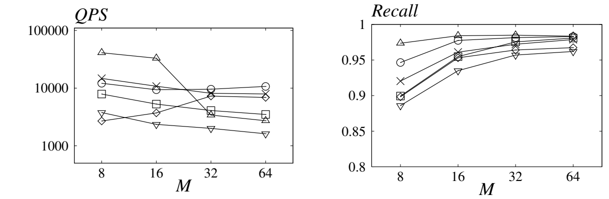
(a) Queries per second (b) Recall

Figure 32: Impact of M to query performance on containment.



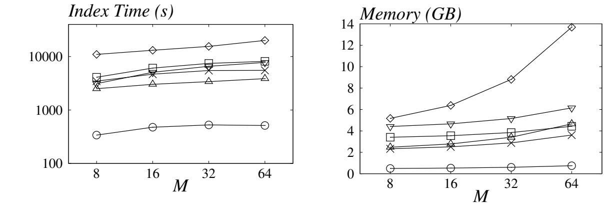
(a) Queries per second (b) Recall

Figure 33: Impact of M to query performance on overlap.



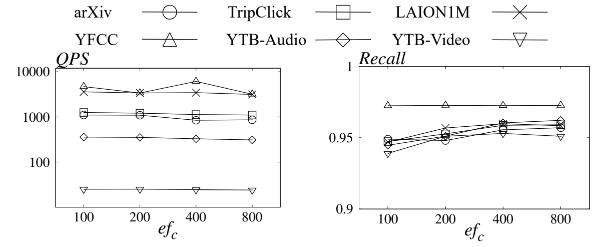
(a) Queries per second (b) Recall

Figure 34: Impact of M to query performanceo on equality.



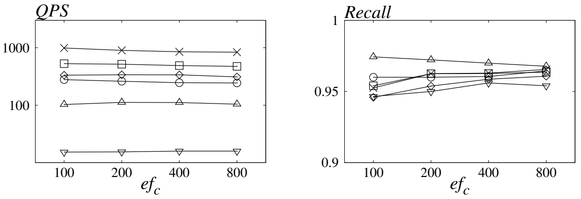
(a) Indexing time (b) Memory

Figure 35: Impact of M to indexing cost.



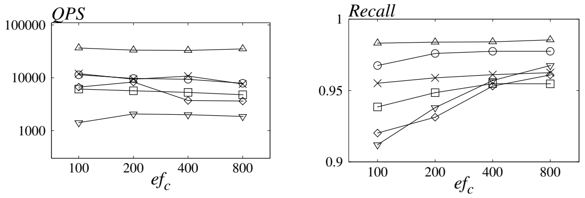
(a) Queries per second (b) Recall

Figure 36: Impact of ef_c to query performance on containment.



(a) Queries per second (b) Recall

Figure 37: Impact of ef_c to query performance on overlap.



(a) Queries per second (b) Recall

Figure 38: Impact of ef_c to query performance on equality.

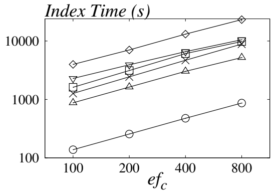


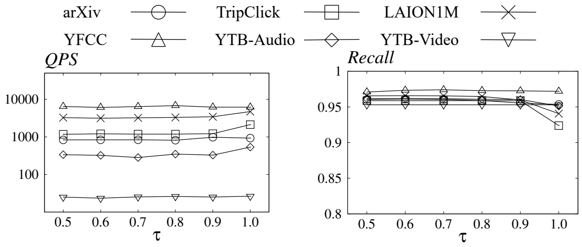
Figure 39: Impact of ef_c to indexing cost.

allowed to contain at most $2^{\text{log\_size}(u)+2} - 1$ vertices, so $\frac{|O_{u.\text{index}.G}|}{|O_u|} \le \frac{2^{\text{log\_size}(u)+2} - 1}{2^{\text{log\_size}(u)}} < 4$, which guarantees that the associated index remains a 4-approximate index for O_u.

B SUPPLEMENTARY EXPERIMENTAL RESULTS

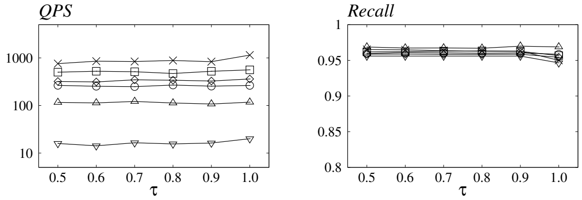
Impact of query selectivity (continued from 4.2). Figure 19-20 show the results.

Impact of update (continued from 4.2). Figure 23-25 show the results.



(a) Queries per second (b) Recall

Figure 40: Impact of τ to query performance on containment.



(a) Queries per second (b) Recall

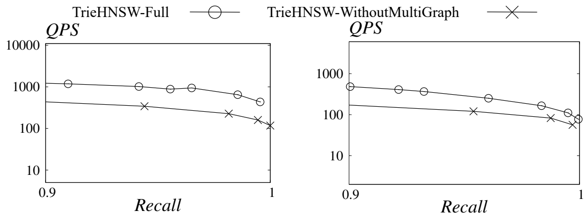
Figure 41: Impact of τ to query performance on overlap.

Query performance on equality queries. First of all, for queries with *equality* label constraints, the search algorithm for TrieHNSW is relatively simple (see Section 3.2) and resembles that of UNG. Hence, TrieHNSW and UNG have comparable performance for such queries, and both significantly outperform all other competitors, which is consistent with results in [9]. Detailed results on *equality* queries are shown in Figure 26- 31.

Impact of parameter M. In the experiment studying the impact of hyperparameters, we fix the search parameter ef for each dataset and examine how recall and QPS change as the hyperparameter varies under this fixed ef. Figure 32 - 35 shows the impact of parameter M. Increasing M leads to higher space consumption but strengthens the graph's connectivity, resulting in the exploration of more nodes during greedy search. This improvement in connectivity enhances recall but adversely affects QPS. Based on the experimental results, we set $M = 32$ for the YTB-Video dataset and $M = 16$ for other datasets as the default parameters to achieve a balance between query performance and space usage.

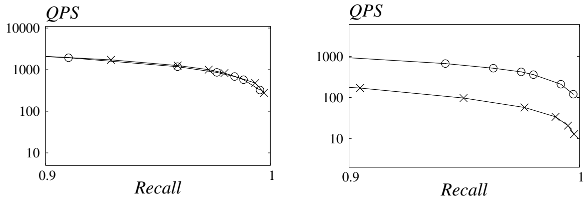
Impact of parameter ef_c. Figure 36 - 39 shows the impact of parameter ef_c. Increasing M leads to higher indexing time but strengthens the graph's quality, resulting in better performance. This improvement in connectivity enhances recall but adversely affects QPS. Based on the experimental results, we set $ef_c = 400$ as the default parameter, as it achieves a good balance between recall and QPS.

Impact of parameter τ. In this experiment we examine the effect of τ on search performance, as shown in Figure 40-41. **Note that $\tau = 1$ corresponds to using only a single-phase search.** We observe that when $\tau = 1$, although QPS slightly increases because only one search pass is performed, recall drops significantly. The reason is that a single-phase search is forced to include many small disjoint-coverage nodes, which provide limited structural information and can misguide the search direction. In contrast, the



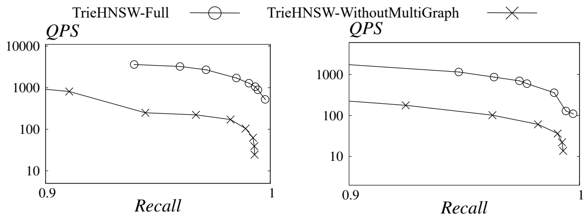
(a) Containment (b) Overlap

Figure 42: Impact of multi-graph search on arXiv.



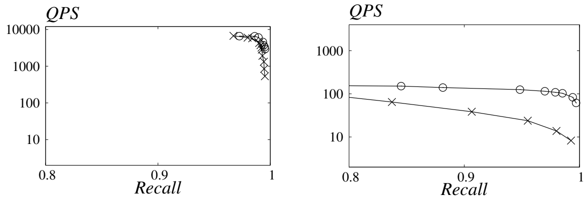
(a) Containment (b) Overlap

Figure 43: Impact of multi-graph search on TripClick.



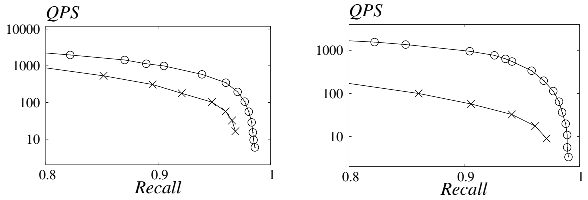
(a) Containment (b) Overlap

Figure 44: Impact of multi-graph search on LAION1M.



(a) Containment (b) Overlap

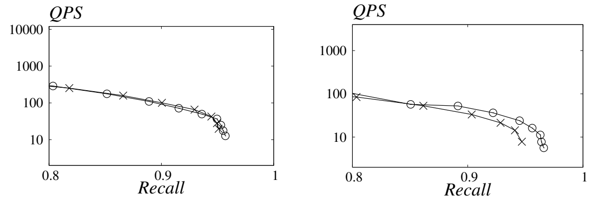
Figure 45: Impact of multi-graph search on YFCC.



(a) Containment (b) Overlap

Figure 46: Impact of multi-graph search on YTB-Audio.

two-phase search maintains consistently high recall, with peak QPS occurring in the range $\tau = 0.7$–0.9. We therefore set $\tau = 0.9$ for all datasets in our evaluation.



(a) Containment (b) Overlap

Figure 47: Impact of multi-graph search on YTB-Video.

Impact of multi-graph search. Figures 42-47 present the results of our method with and without multi-graph search. The results demonstrate that when multi-graph search is enabled, our method achieves both higher QPS and higher recall, confirming the effectiveness of this search optimization.

REFERENCES

[1] 2026. Code and technical report. https://github.com/CUHK-DBGroup/TrieHNSW.

[2] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. 2016. YouTube-8M: A Large-Scale Video Classification Benchmark. *CoRR* abs/1609.08675 (2016).

[3] Anas Ait Aomar, Karima Echihabi, Marco Arnaboldi, Ioannis Alagiannis, Damien Hilloulin, and Manal Cherkaoui. 2025. RWalks: Random Walks as Attribute Diffusers for Filtered Vector Search. *Proc. ACM Manag. Data* 3, 3 (2025), 212:1–212:26.

[4] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM* 51, 1 (2008), 117–122.

[5] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *NeurIPS*. 1225–1233.

[6] 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. *Proc. VLDB Endow.* 9, 4 (2015), 288–299.

[7] Artem Babenko and Victor S. Lempitsky. 2015. Tree quantization for large-scale similarity search and classification. In *CVPR*. 4240–4248.

[8] Davis W. Blalock and John V. Guttag. 2017. Bolt: Accelerated Data Mining with Fast Vector Compression. In *SIGKDD*. 727–735.

[9] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. 2024. Navigating Labels and Vectors: A Unified Approach to Filtered Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 6 (2024), 246:1–246:27.

[10] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proc. VLDB Endow.* 17, 12 (2024), 3772–3785.

[11] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *PoCG*. 253–262.

[12] Wei Dong, Moses Charikar, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*. 577–586.

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

[14] 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.

[15] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 3 (2024), 167.

[16] 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 *WWW*. 3406–3416.

[17] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. 604–613.

[18] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128.

[19] 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. *Proc. ACM Manag. Data* 3, 3 (2025), 148:1–148:26.

[20] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Trans. Big Data* 7, 3 (2021), 535–547.

[21] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.* 45 (2014), 61–68.

[22] 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.

[23] Malteos. 2022. Aspect Paper Embeddings. https://huggingface.co/datasets/malteos/aspect-paper-embeddings. Hugging Face Dataset.

[24] Yusuke Matsui, Ryota Hinami, and Shin'ichi Satoh. 2018. Reconfigurable Inverted Index. In *MM*. 1715–1723.

[25] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proc. ACM Manag. Data* 2, 3 (2024), 120.

[26] 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 *ICML*, Vol. 139. 8748–8763.

[27] 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 *ICML*, Vol. 139. 8748–8763.

[28] Navid Rekabsaz, Oleg Lesota, Markus Schedl, Jon Brassey, and Carsten Eickhoff. 2021. TripClick: The Log Files of a Large Health Web Search Engine. In *SIGIR*. 2507–2513.

[29] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. 2021. LAION-400M: Open Dataset of CLIP-Filtered 400 Million Image-Text Pairs. *CoRR* abs/2111.02114 (2021).

[30] Gaurav Sehgal and Semih Salihoglu. 2025. NaviX: A Native Vector Index Design for Graph DBMSs With Robust Predicate-Agnostic Search Performance. *Proc. VLDB Endow.* 18, 11 (2025), 4438–4451.

[31] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NeurIPS*. 13748–13758.

[32] 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.

[33] Jiang Wang, Xin Wen, Ruicheng Liu, Xiaoyan Tang, Yongqing Xie, Pinghua Wang, Ji Sun, Yanfeng Zhang, Yong Li, Guoliang Li, Xiaoguang Ren, Wen Nie, and Chunbo Li. 2025. GaussDB-Vector: A Large-Scale Persistent Real-Time Vector Database for LLM Applications. *Proc. VLDB Endow.* 18, 12 (2025), 4951–4963.

[34] 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 *SIGMOD*. 2614–2627.

[35] Jiuqi Wei, Botao Peng, Xiaodong Lee, and Themis Palpanas. 2024. DET-LSH: A Locality-Sensitive Hashing Scheme with Dynamic Encoding Tree for Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 17, 9 (2024), 2241–2254.

[36] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 6 (2024), 239:1–239:26.

[37] 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. *Proc. ACM Manag. Data* 3, 3 (2025), 152:1–152:26.

[38] 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 *OSDI*. 377–395.

[39] Justin Zobel and Alistair Moffat. 1998. Inverted files versus signature files for text indexing. *TODS* 23, 4 (1998), 453–490.

[40] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 1 (2024), 69:1–69:26.