Efficient Approximate Nearest Neighbor Search under Multi-Attribute Range Filter

Yuanhang Yu
Tongji University
Shanghai, China
yuanhangyu@tongji.edu.cn

Dawei Cheng*
Tongji University
Shanghai, China
dcheng@tongji.edu.cn

Ying Zhang
Zhejiang Gongshang University
Hangzhou, China
ying.zhang@zjgsu.edu.cn

Lu Qin
University of Technology Sydney
Sydney, Australia
lu.qin@uts.edu.au

Wenjie Zhang
The University of New South Wales
Sydney, Australia
wenjie.zhang@unsw.edu.au

Xuemin Lin
Shanghai Jiao Tong University
Shanghai, China
xuemin.lin@sjtu.edu.cn

Abstract

Nearest neighbor search on high-dimensional vectors is fundamental in modern AI and database systems. In many real-world applications, queries involve constraints on multiple numeric attributes, giving rise to range-filtering approximate nearest neighbor search (RFANNS). While there exist RFANNS indexes for single-attribute range predicates, extending them to the multi-attribute setting is nontrivial and often ineffective. In this paper, we propose KHI, an index for multi-attribute RFANNS that combines an attribute-space partitioning tree with HNSW graphs attached to tree nodes. A skew-aware splitting rule bounds the tree height by $O(\log n)$, and queries are answered by routing through the tree and running greedy search on the HNSW graphs. Experiments on four real-world datasets show that KHI consistently achieves high query throughput while maintaining high recall. Compared with the state-of-the-art RFANNS baseline, KHI improves QPS by 2.46× on average and up to 16.22× on the hard dataset, with larger gains for smaller selectivity, larger k, and higher predicate cardinality.

PVLDB Reference Format:
Yuanhang Yu, Dawei Cheng, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. Efficient Approximate Nearest Neighbor Search under Multi-Attribute Range Filter. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yuyuanhang/KHI.

1 Introduction

Nearest neighbor search (NNS) is a fundamental problem in high-dimensional similarity search that has been extensively studied [7, 22, 29, 44]. Given a query object q and a set O of data objects in a metric space (S, δ), nearest neighbor search aims to find an object $o^* \in O$ that minimizes $\delta(q, o^*)$. Due to the curse of dimensionality [22], exact NNS is often impractical on large high-dimensional datasets, motivating extensive work on approximate nearest neighbor (ANN) search [9, 13, 20, 23, 32, 33, 48]. In modern systems, objects such as images, text, and videos are often represented as high-dimensional vectors. Consequently, ANN has become a fundamental operation in a wide range of applications, including information retrieval [27, 31], machine learning [19, 28], and vector database systems [26, 42].

Beyond vanilla ANN, there is increasing demand for ANN queries augmented with constraints on numeric attributes. This trend is evident both in recent research [50, 54] and in industrial systems from Apple [34], Zilliz [42], and Alibaba [46], where ANN queries are routinely combined with metadata filters on attributes such as time and popularity. In these setting, each data object consists of a feature vector together with a tuple of numeric attribute values, and queries are issued as a query vector accompanied by a range predicate over these attributes. For example, in a scholarly search engine, a user may submit a paper abstract as a query vector and retrieve similar publications whose publication year and citation count both fall within user-specified numeric ranges. In this paper, we study multi-attribute range-filtering approximate nearest neighbor search (RFANNS) in Euclidean space.

Existing RFANNS indexes [50, 54] are primarily tailored to single-attribute range predicates. The multi-attribute setting is more challenging and is not handled effectively by existing RFANNS indexes. Specifically, the method in [54] does not admit a straightforward extension to multiple attributes, while the index proposed in [50] explicitly supports multi-attribute range predicates but suffers from efficiency limitations in this regime. To address this challenge, one natural approach is to first partition the low-dimensional attribute space and then build graph indexes over the objects associated with each partition. R-trees are a well-established choice for the partitioning structure, as they are widely used for multidimensional spatial indexing. However, we argue that the substantial overlap among R-tree nodes degrades the quality of the graph indexes constructed over these partitions.

Motivated by these limitations, we propose KHI, an RFANNS index tailored for multi-attribute numeric range predicates. Specifically, KHI combines a KD-tree–like partitioning tree T over the attribute space with graph indexes attached to its nodes. The partitioning tree recursively splits the data along individual attribute

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

dimensions, producing non-overlapping partitions that adapt to attribute distributions. To keep the tree height under control in the presence of skewed attributes, we adopt a skew-aware splitting rule that only accepts reasonably balanced splits, guided by a balance threshold τ and a small per-node exclusion set of problematic dimensions. We show that this rule guarantees an $O(\log n)$ bound on the tree height, where n is the number of data objects. On top of this tree, each node $p \in T$ stores a single-level HNSW graph G_p built on the object subset $\mathcal{O}(p)$ covered by its partition.

To answer RFANNS queries efficiently, KHI leverages both the partitioning tree and the HNSW graphs. Given a query $Q = (q, B)$, where q is the query vector and B is a range predicate that constrains the attribute values, we first traverse the tree to identify nodes whose attribute-space partitions intersect B and select a small subset of them as entry points. Starting from these entry points, the algorithm performs a greedy search. When a candidate object is expanded, we reconstruct its neighborhood by aggregating incident edges from all HNSW graphs containing that object and retain only in-range neighbors whose attribute tuples satisfy B. Together, these components enable efficient retrieval of the k nearest neighbors satisfying the range predicate B.

To construct KHI, we first build the partitioning tree T in the attribute space using the skew-aware splitting strategy. Given T, we then generate single-level HNSW graphs for all tree nodes in a bottom-up fashion: at each leaf node p, we build the graph G_p directly over its object set $\mathcal{O}(p)$, and at internal nodes we reuse and incrementally merge the child graphs instead of rebuilding them from scratch. To further reduce index construction time, we employ a hybrid parallelization strategy that combines level-wise parallelism across tree nodes with intra-node parallelism during graph merging within individual tree nodes.

In summary, our main contributions are as follows.

- To the best of our knowledge, this is the first work that specifically studies RFANNS with multi-attribute numeric range predicates in Euclidean space.
- We propose KHI, a new RFANNS index that combines an attribute-space partitioning tree with HNSW graphs anchored at tree nodes, and design an efficient RFANNS query algorithm that uses only in-range neighbors while still achieving high recall and high query throughput (Section 4).
- We conduct an extensive experimental study on four real-world datasets. The results demonstrate that at recall 0.95 (or 0.9), KHI improves QPS over a state-of-the-art RFANNS baseline by an average of 2.46×, and up to 16.22× on the hard dataset. (Section 5).

In addition, Section 2 presents the problem definition and basic background, Section 6 provides a detailed discussion of related work, and Section 7 concludes the paper.

2 Preliminaries

2.1 Problem Definition

Let S be the d-dimensional space and δ: $S \times S \to \mathbb{R}$ be the distance function over S. The k-nearest neighbor (k-NN) search problem with numerous real-world applications is defined as follows:

Definition 1. (k-NN). Given a finite set $X \subseteq S$, a query vector $q \in S$, a distance function δ over S, and a positive integer $k \leq n$, the k-nearest neighbor search returns a subset $R \subseteq X$ with $|R| = k$ such that $\forall x \in R, \forall y \in X \setminus R$: $\delta(x, q) \leq \delta(y, q)$.

In practice, k–NN search is often performed over vectors whose dimensionality ranges from tens to hundreds. A notable example is retrieval-augmented generation (RAG) [28]. As dimensionality increases, the curse of dimensionality [22] renders exact nearest neighbor search computationally prohibitive. Therefore, approximate nearest neighbor (ANN) search has attracted significant interest for achieving high efficiency at the cost of only a slight loss in accuracy. This loss is commonly quantified by the recall metric: given a set $\hat{R}$ of size k returned by an ANN algorithm, recall is defined as $|R \cap \hat{R}|/k$, where R denotes the true k nearest neighbors.

In real-world applications, objects represented as vectors are often associated with structured attributes. For example, each product on e-commerce platforms can be represented by an image embedding and structured attributes (e.g., price and rating). Unless otherwise stated, we use vector and embedding interchangeably for the high-dimensional representation of an object. In this setting, one may desire to retrieve products that are similar in the embedding space while satisfying attribute filters, such as a given price range and a rating above 4. This type of search is known as the range-filtering nearest neighbor search (RFNNS). To formally define RFNNS, we first present a schema-style description of the object set, analogous to a relation schema in relational databases.

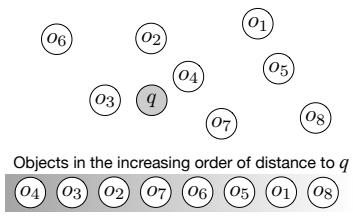
Definition 2. (Object Schema). An object schema is defined as a pair (S, A), where (1) $S \subseteq \mathbb{R}^d$ denotes the embedding space; (2) $A = (a_1, \ldots, a_m)$ specifies the attribute schema, where each a_i is a numeric attribute.

Given an object schema (S, A), an object o under this schema is represented as a pair (x, t), where $x \in S$ is a d-dimensional vector and $t = (t_1, \ldots, t_m)$ is a tuple attribute values conforming to A. An object set O is an instance of (S, A) if every $o \in O$ follows this representation. Now, we formally define RFNNS as follows:

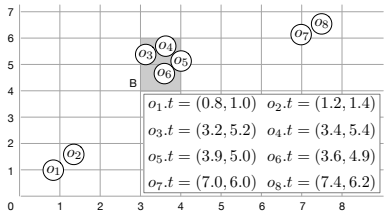
Definition 3. (RFNNS). Given an object schema (S, A), its instance O, a distance function δ over S, and a positive integer k, an RFNNS query is defined as $Q = (q, B)$, where $q \in S$ is a query vector and B is a range predicate. Specifically, $B = \{b_i = [l_i, r_i] \mid i \in \mathcal{J}\}$ where $\emptyset \neq \mathcal{J} \subseteq \{1, \ldots, m\}$ specifies a non-empty set of constrained attributes. An object o satisfies B, denoted as $o \vDash B$, if $l_i \leq o.t.t_i \leq r_i, \forall i \in \mathcal{J}$. Let $O_B = \{o \in O \mid o \vDash B\}$ denote the filtered object set of O under B. The RFNNS problem is to find a set $R_Q \subseteq O_B$ with $|R_Q| = k$ such that for any $o_x = (x, t_x) \in R_Q$ and $o_y = (y, t_y) \in O_B \setminus R_Q$, $\delta(x, q) \leq \delta(y, q)$.

We refer to $|B| = |\mathcal{J}|$ as the cardinality of B, i.e., the number of constrained attributes. To illustrate the above definitions, we consider Example 1.

Example 1. Given an object schema (S, A) with $m = 2$, its instance $O = \{o_i \mid i \in [1, 8]\}$ is presented in Figure 1. Consider an RFNNS query $Q = (q, B)$ with $k = 2$, where q and $B = \{b_1 = [3.0, 4.0], b_2 = [4.0, 6.0]\}$ are illustrated in Figure 1a and Figure 1b, respectively. $O_B = \{o_3, o_4, o_5, o_6\}$ based on their attribute tuples in Figure 1b. For example, $o_3 \vDash B$ because $3.0 \leq o_3.t.t_1 \leq 4.0$ and $4.0 \leq o_3.t.t_2 \leq 6.0$. As depicted in Figure 1a, since the vectors of objects o_3 and o_4 are closer to q than that of o_5 and o_6, $R_Q = \{o_3, o_4\}$ is returned for the query Q.



(a) Vectors of an example object set and a query vector q.



(b) Tuples of an example object set and a range predicate B.

Figure 1: An example object set O and an example query Q.

Since RFNNS faces challenges analogous to exact k-NN, i.e., high computational cost at scale, we focus on the approximate variant of RFNNS, referred to as RFANNS, and throughout this paper we use the Euclidean distance as the distance function δ.

2.2 Graph-based ANN Solutions

We briefly review graph-based ANN methods, which are adopted as core modules of RFANNS methods due to their ability to achieve high recall while probing only a small fraction of the dataset.

Graph-based methods construct a proximity graph G where each vertex u corresponds to a data vector in the dataset. Given a query vector q, they perform a greedy graph traversal, starting from an entry point and iteratively moving to neighbors that are closer to q until no closer candidate can be found. In practice, the search quality is controlled by a parameter ef, which limits the size of closest elements maintained during search. A larger ef typically explores more vertices, thus improving recall at the cost of higher query time. Note that ef is independent of k: k defines the query result size, while ef controls the search breadth. Existing graph-based methods [13, 20, 29, 32, 33] primarily differ in their graph construction strategies, which in turn affect the efficiency and effectiveness of greedy navigation. Among them, NSW [32] and HNSW [33] have been widely adopted, as they offer an excellent trade-off between search accuracy and efficiency at scale and often exhibit near-logarithmic search path length in practice.

Navigable small world graphs. NSW builds the proximity graph G by inserting vertices corresponding to data vectors in sequence. For each inserted vertex u, it performs a greedy search on the current graph to obtain an approximate neighbor set $\hat{R}$ of size M, and then adds symmetric edges between u and each $v \in \hat{R}$. Here, the parameter M specifies the number of neighbors linked for each inserted vertex, thus controlling the graph density. Similarly, the result quality of construction-time search is controlled by the parameter ef_b, which bounds the size of the candidate set explored during insertion. After inserting all vertices, the construction terminates. This incremental strategy yields a mix of two critical link types: short-range links that approximate local Delaunay neighborhoods [4] to facilitate fine-grained local exploration around the query vector, and long-range links formed during early insertions to enable fast navigation from distant regions toward the vicinity of the query vector, which together promote small-world navigability and lead to strong empirical efficiency.

Hierarchical navigable small world graphs. Building upon incremental construction of NSW, HNSW further introduces a hierarchical structure to improve scalability. Inspired by skip lists, HNSW organizes the proximity graph into L layers, where each layer is an NSW-like graph. The expected number of vertices increases exponentially from the top layer to the bottom layer, enabling a coarse-to-fine greedy traversal. To reduce redundant connections, HNSW adopts a heuristic neighbor selection strategy inspired by the Relative Neighborhood Graph (RNG) [41]. Specifically, an edge (u, v) is retained only if there exists no neighbor v' of u such that $\delta(u, v') < \delta(u, v)$ and $\delta(v, v') < \delta(u, v)$. To control graph density, HNSW bounds the maximum degree of each vertex, using M for the upper layers and $2M$ for the bottom layer.

2.3 Existing RFANNS Solutions

Existing RFANNS solutions [50, 54] primarily focus on the single-attribute setting, i.e., $m = 1$, where the range predicate B reduces to a query range $[l, r]$. In this setting, these methods follow an intuitive paradigm that supports efficient search over the filtered object set O_B under an arbitrary B. Specifically, motivated by the strong empirical performance of HNSW, they seek to enable greedy search over O_B as if an HNSW graph were built on the vectors in O_B. We denote this HNSW graph as $G_h(O_B)$ and refer to any HNSW graph built on a subset of O as a filtered HNSW graph. Since constructing a filtered HNSW graph for every possible query range is prohibitive in both space and build time, these methods typically (i) locate an entry point within O_B, and (ii) recover, exactly or approximately, the neighbor list of each visited vertex u on the fly, matching the neighbors that u would have in $G_h(O_B)$. Similarly, each vertex in the graph corresponds to an object in O. Let O be the object set with $|O| = n$. Without loss of generality, we assume that its one-dimensional attribute domain is discretized and normalized to the integer range $[0, n-1]$.

SeRF [54]. The discretized attribute domain yields $O(n^2)$ possible query ranges. SeRF aims to losslessly compress the filtered HNSW graphs associated with these $O(n^2)$ query ranges, thereby enabling exact on-the-fly reconstruction of the neighbor list for any visited vertex. This is achieved by compressing neighbor lists across adjacent ranges with the same left boundary.

For a fixed left boundary l, as the right boundary r increases from l to $n-1$, the membership of a vertex v in the neighbor list of another vertex u is non-reentrant, i.e., v enters and eventually exits without reappearing. This behavior enables compact encoding via an inner interval $[t_s, t_e]$, such that v is a neighbor of u for all ranges $[l, r]$ with $r \in [t_s, t_e]$. Moreover, an outer interval $[t_l, t_r]$ specifies the range of left boundaries l for which the inner interval $[t_s, t_e]$ remains valid. That is, for any $l \in [t_l, t_r]$, v is a neighbor of u in all query ranges $[l, r]$ where $r \in [t_s, t_e]$.

During query processing, SeRF leverages these intervals to exactly recover, on the fly, the neighbor list of each visited vertex for an arbitrary query range $B = [l, r]$, while an entry point in O_B can be efficiently located by ordering objects according to their attribute values.

iRangeGraph [50]. iRangeGraph supports approximate, rather than exact, on-the-fly neighbor reconstruction. To this end, iRangeGraph constructs HNSW graphs over a moderate number of ranges, achieving provable bounds on index space and construction time. Let us refer to a neighbor v of a vertex u as an in-range neighbor with respect to B if $v \models B$; otherwise, v is an out-of-range neighbor.

iRangeGraph organizes the attribute domain into subranges using a segment tree with $O(n)$ nodes, where each node corresponds to a subrange of $[0, n-1]$. For each subrange, the corresponding filtered HNSW graph is built. Given a query range $[l, r]$, iRangeGraph reconstructs the neighbor list of each visited vertex u by aggregating its neighbors from the filtered HNSW graphs associated with the segment-tree nodes containing u, and retaining only in-range neighbors. An entry point in O_B can be efficiently obtained from the segment-tree nodes covered by $[l, r]$.

To extend its single-attribute index to multiple attributes, iRangeGraph introduces a probabilistic neighbor-reconstruction rule that retains out-of-range neighbors from each filtered HNSW graph with a decaying probability. This relaxation enlarges the search space and thus improves result quality under multi-attribute range predicates, although the index is built on a single attribute.

3 Motivation

3.1 Limitations of Existing RFANNS Solutions

Existing RFANNS methods have demonstrated strong performance in the single-attribute setting. However, their extensions to multiple attributes are either impractical or ineffective. We detail these limitations below.

Limitations of SeRF. SeRF achieves strong performance in the single-attribute setting by enabling exact on-the-fly neighbor reconstruction for each visited vertex. However, this idea does not scale straightforwardly to multiple attributes. As the number of attributes in A increases, the space of possible range predicates B can grow exponentially, which would lead to prohibitive index construction time and space overhead. Therefore, SeRF becomes impractical in the multi-attribute setting.

Limitations of iRangeGraph. iRangeGraph supports the multi-attribute setting through a probabilistic rule that allows the search to access out-of-range neighbors with a decaying probability. This strategy lies between in-filtering and post-filtering: it relaxes the strict in-range constraint of in-filtering, while remaining more selective than post-filtering in exploring all out-of-range neighbors.

However, this relaxation also causes iRangeGraph to inherit a key drawback of post-filtering. For practical multi-attribute RFANNS queries where a subset of attributes are constrained, the filtered set O_B is often sparse in O, i.e., the selectivity $\sigma = |O_B|/|O|$ is low, so the neighbor lists reconstructed on the fly by iRangeGraph contain only a small fraction of in-range neighbors. As a result, the search traverses many out-of-range objects, which slows the refinement of the best-so-far candidates and can even cause the number of visited objects to exceed $|O_B|$ when σ is sufficiently low. Our experiments in Section 4.2 confirm that this makes iRangeGraph less effective for multi-attribute RFANNS queries, despite its practical utility as an extension of a single-attribute index.

Implications. The above discussion shows that the multi-attribute setting poses two intertwined challenges that existing RFANNS methods do not sufficiently address. First, the space of possible range predicates grows rapidly with the number of attributes, making exact on-the-fly neighbor reconstruction for arbitrary B unsustainable. This suggests approximating the neighbor lists in $G_h(O_B)$ using only a moderate number of selectively constructed filtered HNSW graphs. Second, when the selectivity σ is relatively low, out-of-range expansions tend to dominate the search cost. Thus, an effective index should preserve strong search quality even when traversal is restricted to in-range neighbors.

These observations motivate the following design. We treat the tuples of objects in O as points in the attribute space and organize them with a spatial partitioning structure. This structure induces a collection of subsets of O, and on each subset we build a corresponding filtered HNSW graph. By maintaining a moderate number of such graphs, we aim to obtain a close approximate to, for an arbitrary multi-attribute range predicate B, the HNSW graph $G_h(O_B)$ and thereby achieve a practical balance between query performance and index cost.

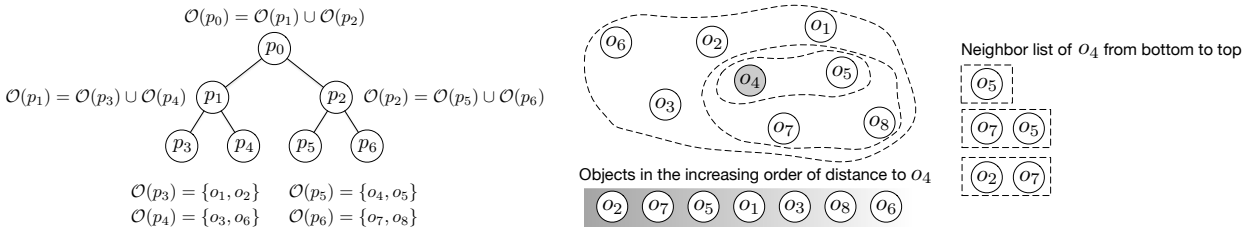
3.2 Limitations of R-tree

Given the above implications, our next step is to identify a spatial index that can partition the attribute space into subsets. The R-tree is a natural candidate, as it is a classic index for organizing multi-dimensional points and supporting range queries. We thus examine whether an R-tree over the tuples can induce partitions that are well aligned with our partition-driven graph construction. As we show next, however, R-tree partitions do not consistently satisfy this requirement. To avoid confusion, we use node for tree structures and vertex for graph structures.

Overlapping MBRs. In an R-tree, each internal node stores the minimum bounding rectangles (MBRs) of its children, and the MBRs stored in the same internal node are allowed to overlap in order to maintain balance and compact bounding rectangles. Thus, the portion of the attribute space indexed by an internal node is represented by overlapping MBRs rather than disjoint cells, and two objects that are close in the attribute space can be assigned to different child nodes of the same parent.

Neighbor quality issues. In general, distances in the attribute space and in the embedding space are not necessarily aligned: objects that are far apart in the attribute space may still be close in the embedding space, and vice versa. Since sibling MBRs in an internal R-tree node are allowed to overlap, consider a case where two objects that are close in the attribute space, say o_u and o_v, fall into different child nodes p_1 and p_2 of the same parent p_0, while another object o_w that is farther from o_u in the attribute space falls into the same node p_1 as o_u. At the same time, o_w may be much closer to o_u than o_v in the embedding space.

Let $\mathcal{O}(p)$ be the set of objects whose tuples fall within the MBR of the tree node p. For brevity, we write $G_p = G_h(\mathcal{O}(p))$. When we build G_{p_1}, the neighbors of o_u are restricted to the objects in



(a) Example R-tree on the example object set. **(b) Neighbor lists of o_4 across R-tree levels.**

Figure 2: Running example illustrating neighbor lists of o_4 induced by R-tree partitions over the object set O.

$\mathcal{O}(p_1)$. Since o_w is among the closest objects to o_u in the embedding space within $\mathcal{O}(p_1)$, o_w becomes a neighbor of o_u. When G_{p_0} is constructed, o_w, which is close to o_u in the embedding space, tends to remain in one of the neighbor slots of o_u. In contrast, o_v, which resides in the different child node p_2, is pruned due to the limited neighbor-list capacity of o_u and will no longer be considered as a neighbor of o_u in subsequent graph construction. As a result, the index systematically misses attribute-close neighbors of o_u, such as o_v, degrading the quality of the local neighborhood around objects. For a given range predicate B, queries may either fail to retrieve certain relevant objects in O_B or reach them only via longer paths, thereby increasing search cost.

Example 2. Given the object set O shown in Figure 1, an example R-tree built over O is depicted in Figure 2a. Since sibling MBRs in the R-tree may overlap, the closest pair of objects in the attribute space, o_3 and o_4, is assigned to two different leaf nodes, p_4 and p_5. We focus on o_4 and its neighbor lists across the R-tree levels.

As shown in Figure 2b, at node p_5 we have $\mathcal{O}(p_5) = \{o_4, o_5\}$, and the neighbor list of o_4 is $\{o_5\}$. At its parent p_2, additional objects such as o_7 and o_8 are present; because o_7 and o_5 are closer to o_4 than o_8 in the embedding space, the neighbor list of o_4 becomes $\{o_7, o_5\}$. At the root p_0, more objects including o_2 are considered, and the neighbor list of o_4 is updated to $\{o_2, o_7\}$. Note that o_3 and o_4 first co-occur in the same object set only at p_0, i.e., $o_3, o_4 \in \mathcal{O}(p_0)$. However, by that time the two neighbor slots of o_4 are already occupied by o_2 and o_7, which are farther from o_4 than o_3 in the attribute space. Consequently, the attribute-close neighbor o_3 is excluded from the neighbor list of o_4 at all levels. For a range predicate B such that $o_3, o_4 \in O_B$, a graph-based search that starts from o_4 and expands along its neighbors may therefore fail to retrieve o_3, even though o_3 is the closest attribute neighbor of o_4.

Implications. These observations indicate that overlapping partitions in the attribute space distort the neighborhood structure of filtered HNSW graphs and harm search quality, as confirmed by our experiments in Appendix A. Thus, for RFANNS, spatial indexes that induce non-overlapping partitions in the attribute space are therefore preferable.

4 Our Solution

4.1 Index Overview

We propose a KD-tree–HNSW hybrid index, denoted by KHI, which integrates a KD-tree–like partitioning tree on attribute tuples with filtered HNSW graphs over vectors.

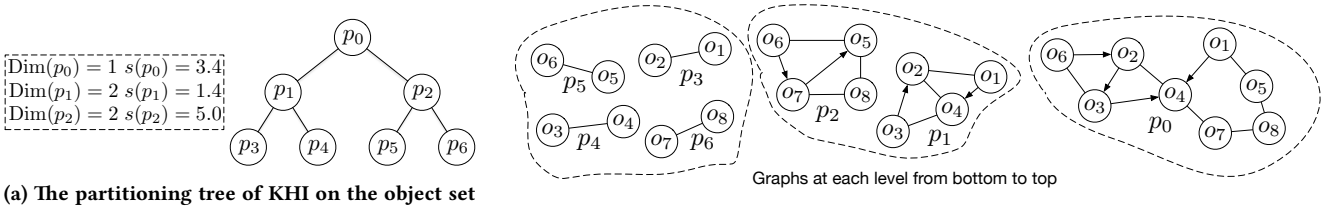
Attribute-space partitioning. KHI maintains a binary partitioning tree T on the attribute tuples of O. Each node p in T is associated with (1) an axis-aligned hyper-rectangle $\mathcal{R}(p)$ in the m-dimensional attribute space, and (2) an object subset $\mathcal{O}(p) \subseteq O$, defined as $\mathcal{O}(p) = \{o \in O \mid o.t \in \mathcal{R}(p)\}$. For the root node p_r, $\mathcal{R}(p_r)$ covers all attribute tuples and $\mathcal{O}(p_r) = O$. A node p is a leaf if $|\mathcal{O}(p)| \leq c_l$, where c_l is the leaf-capacity parameter (set to 2 in our implementation).

When splitting an internal node p, KHI proceeds as follows. KHI first chooses a splitting dimension $\mathsf{Dim}(p) \in \{1, \ldots, m\}$ and a split value $s(p)$. Along dimension $\mathsf{Dim}(p)$, we split the region $\mathcal{R}(p)$ at $s(p)$ into two disjoint child regions $\mathcal{R}(p_\ell)$ and $\mathcal{R}(p_r)$. Objects with $o.t.t_{\mathsf{Dim}(p)} \leq s(p)$ are assigned to the left child, and those with $o.t.t_{\mathsf{Dim}(p)} > s(p)$ to the right. To keep the tree reasonably balanced, we follow a round-robin rule for choosing the splitting dimension: $\mathsf{Dim}(p)$ is set to the next dimension after the splitting dimension of its parent node (wrapping around after m). Given $\mathsf{Dim}(p)$, we collect the values $\{o.t.t_{\mathsf{Dim}(p)} \mid o \in \mathcal{O}(p)\}$ and set $s(p)$ to their median. However, this median-based choice of $s(p)$ may still lead to a highly unbalanced partition on skewed data. To mitigate this, we enforce a balance threshold $\tau > 1$. Let n_L and n_R denote the numbers of objects assigned to the left and right child of p, respectively. If $\frac{\max\{n_L, n_R\}}{\min\{n_L, n_R\}} \geq \tau$, we regard dimension $\mathsf{Dim}(p)$ as skewed at p, , exclude this dimension from consideration, and retry the split using the next available dimension. Once excluded at p, a dimension is no longer used as a splitting dimension for any descendant of p.

Lemma 1. The height of T is $O(\log \frac{n}{c_l})$.

Proof sketch. Let $N = |\mathcal{O}(p)|$ for an internal node p, and assume $n_L \geq n_R$. From $\frac{n_L}{n_R} < \tau$ and $n_L + n_R = N$, we obtain $n_L < \frac{\tau}{\tau+1} N$. Hence, after each recursive split, the larger child has size at most ρN, where $\rho = \frac{\tau}{\tau+1} < 1$. Starting from n objects at the root, the size of any node is at most $\rho^h n$ after h levels. The recursion stops once $|\mathcal{O}(p)| \leq c_l$, which implies $\rho^h n \leq c_l$ and therefore $h \leq \log_{1/\rho} \frac{n}{c_l} = O(\log \frac{n}{c_l})$. □

Embedding-space graphs. For each node p in T, KHI builds a filtered HNSW graph G_p. The vertex set of G_p consists of the objects in $\mathcal{O}(p)$, and edges are constructed by the standard HNSW procedure. In our implementation, each G_p is a single-level HNSW graph ($L = 1$): the KD-tree–like hierarchy already provides the coarse-to-fine search structure, so additional HNSW layers are unnecessary. We use M to bound the maximum degree of each vertex in all single-level HNSW graphs.



(a) The partitioning tree of KHI on the object set O.

(b) Graphs of KHI on the object set O.

Figure 3: Example of the KHI index over the object set O.

Example 3. Consider the object set O in Figure 1 and the leaf capacity is $c_l = 2$. The partitioning tree T of KHI on O is illustrated in Figure 3a. At the root p_0, KHI splits on the first attribute, i.e., $\mathsf{Dim}(p_0) = 1$, with split value $s(p_0) = 3.4$, the median of the first-attribute values in $\mathcal{O}(p_0)$. Objects with $o.t.t_1 \leq 3.4$ are assigned to the left child p_1, and the rest to the right child p_2, resulting in $|\mathcal{O}(p_1)| = |\mathcal{O}(p_2)| = 4$. For p_1 and p_2, the splitting dimension advances to the second attribute, i.e., $\mathsf{Dim}(p_1) = \mathsf{Dim}(p_2) = 2$, and the split values are set to $s(p_1) = 1.4$ and $s(p_2) = 5.0$, respectively. Both splits are balanced, so no dimension is excluded, and all children p_3–p_6 satisfy $|\mathcal{O}(p)| \leq c_l$ and become leaves.

For each node p in T, KHI builds a node-level graph G_p. Figure 3b shows these graphs for $M = 2$: four attached to the leaves p_3–p_6 (each over two objects), two for p_1 and p_2 (each over four objects), and one for the root p_0 over all objects.

From the above example, we see that under the partitioning strategy of KHI, each leaf-level graph is built over a small subset of objects that are close in the attribute space. For instance, o_3 and o_4, the closest pair in the attribute space, fall into the same leaf node and are linked as mutual neighbors in G_{p_4}. When neighbor reconstruction is performed for a range predicate B, the locality of leaf-level graphs in attribute space makes them more likely to contribute in-range neighbors, which can, in many cases, help the reconstructed neighbor lists of visited objects more closely approximate those in the HNSW graph $G_h(O_B)$.

Lemma 2. The space complexity of KHI is $O(nM \cdot \log \frac{n}{c_l})$.

Proof. By Lemma 1, the tree T has height $O(\log \frac{n}{c_l})$. Since each object belongs to exactly one node at each level of T, it participates in $O(\log \frac{n}{c_l})$ node-level filtered HNSW graphs in total. In each single-level filtered HNSW graph, every object stores at most M neighbors in its adjacency list. Hence, the total space required by all filtered HNSW graphs is $O(nM \cdot \log \frac{n}{c_l})$. The tree T contains $O(n)$ nodes and therefore occupies $O(n)$ space. Thus, the overall space complexity of KHI is $O(nM \cdot \log \frac{n}{c_l})$. □

4.2 Query Processing

Given an RFANNS query $Q = (q, B)$, KHI performs a greedy search over O_B. Starting from an entry point in O_B, the search iteratively expands from the current object by aggregating its in-range neighbors from the filtered HNSW graphs.

Entry point selection. With a range predicate B and an entry-point budget c_e as input, Algorithm 1 constructs a set $\mathcal{P}$ of at most c_e entry points for the subsequent greedy search. It performs a depth-first traversal of T to collect a small set of candidate nodes

Algorithm 1: RangeFilter (T, B, c_e)

Input: Partitioning tree T of KHI with root p_r, range predicate B, and entry-point budget c_e
Output: A set $\mathcal{P}$ of entry points

1 $\mathcal{C}, \mathcal{P} \leftarrow \emptyset$; stack $\leftarrow \{(p_r, \emptyset)\}$;
2 **while** stack $\neq \emptyset \wedge |\mathcal{C}| < c_e$ **do**
3 $\quad (p, D) \leftarrow$ stack.pop();
4 $\quad D \leftarrow D \cup \mathsf{BL}(p)$;
5 $\quad$ **if** $|D| = m$ **then** $\mathcal{C} \leftarrow \mathcal{C} \cup \{p\}$; **continue** ;
6 $\quad$ **if** *p is a leaf* **then continue**;
7 $\quad$ **if** $\mathsf{Dim}(p) \in D$ **then**
8 $\quad\quad$ **foreach** *child p_c of p* **do** stack.push(p_c, D) ;
9 $\quad$ **else**
10 $\quad\quad$ **foreach** *child p_c of p* **do**
11 $\quad\quad\quad$ $[l_c, r_c] \leftarrow \pi_{\mathsf{Dim}(p)}(\mathcal{R}(p_c))$;
12 $\quad\quad\quad$ **if** $l_c > r_{\mathsf{Dim}(p)} \vee r_c < l_{\mathsf{Dim}(p)}$ **then continue** ;
13 $\quad\quad\quad$ **else if** $l_c \geq l_{\mathsf{Dim}(p)} \wedge r_c \leq r_{\mathsf{Dim}(p)}$ **then**
14 $\quad\quad\quad\quad$ stack.push$(p_c, D \cup \{\mathsf{Dim}(p)\})$;
15 $\quad\quad\quad$ **else** stack.push(p_c, D) ;
16 **foreach** $p \in \mathcal{C}$ **do**
17 $\quad$ **foreach** $o \in \mathcal{O}(p)$ **do**
18 $\quad\quad$ **if** $o.t \vDash B$ **then** $\mathcal{P} \leftarrow \mathcal{P} \cup \{o\}$; **break** ;
19 **return** $\mathcal{P}$;

$\mathcal{C} = \{p\}$ such that objects satisfying B can be efficiently located within their object sets $\mathcal{O}(p)$. It then scans $\mathcal{O}(p)$ for each tree node $p \in \mathcal{C}$, adding at most one object $o \vDash B$ from each $\mathcal{O}(p)$ to $\mathcal{P}$. For notational convenience, any attribute not constrained in B is treated as having the trivial query range $(-\infty, +\infty)$.

At initialization, the algorithm creates an empty candidate–node set $\mathcal{C}$, an empty entry–point set $\mathcal{P}$, and a stack seeded with the root node p_r and an empty set of covered dimensions (Line 1). In Lines 2–15, the algorithm performs a depth-first traversal over T while the stack is non-empty and $|\mathcal{C}| < c_e$. In each step, it pops a pair (p, D) from the stack (Line 3). Here, for each node p, the set D consists of attribute dimensions $a_i \in A$ that fall into two categories: (1) dimensions already fully covered by the range predicate, i.e., $\pi_i(\mathcal{R}(p)) \subseteq b_i$, where $\pi_i(\mathcal{R}(p))$ denotes the projection of the rectangle $\mathcal{R}(p)$ onto dimension a_i; and (2) dimensions identified as severely skewed and therefore excluded from further splitting. The algorithm then augments D with the set of excluded dimensions $\mathsf{BL}(p)$ (Line 4). Intuitively, dimensions in $\mathsf{BL}(p)$ have been identified as highly skewed at p, so further partitioning along them is discouraged. If all dimensions are covered, i.e., $|D| = m$, the node p is added to $\mathcal{C}$ and its subtree is not explored (Line 5). If p is a

Algorithm 2: ReconsNbr($\mathcal{J}, o, B, c_n$, visited)

Input: Index KHI $\mathcal{J}$, object o, range predicate B, neighbor budget c_n, and visited set visited
Output: Reconstructed neighbor list $\mathcal{N}(o)$

1 $\mathcal{N}(o) \leftarrow \emptyset$;
2 **for** $\ell \leftarrow \mathcal{J}.T.h - 1$ downto 0 **do**
3 $p \leftarrow$ node at level ℓ such that $o \in \mathcal{O}(p)$;
4 **foreach** neighbor o_v of o in $\mathcal{J}.G_p$ **do**
5 **if** visited$[o_v]$ **then continue**;
6 visited$[o_v] \leftarrow$ true;
7 **if** $\neg(o_v \vDash B)$ **then continue**;
8 $\mathcal{N}(o) \leftarrow \mathcal{N}(o) \cup \{o_v\}$;
9 **if** $|\mathcal{N}(o)| = c_n$ **then return** $\mathcal{N}(o)$;
10 **return** $\mathcal{N}(o)$;

leaf and $|D| < m$, the node is skipped and the traversal continues with the remaining stack entries (Line 6), since not all dimensions in $\mathcal{R}(p)$ w.r.t. B are sufficiently refined. Otherwise, the algorithm proceeds to the children of p.

If $\mathsf{Dim}(p) \in D$, then the splitting dimension has already been treated as covered by B, and no further refinement along $\mathsf{Dim}(p)$ is needed. Consequently, all children of p are pushed onto the stack with the same set D (Lines 7–8). If $\mathsf{Dim}(p) \notin D$, then for each child p_c of p, the algorithm compares the interval $[l_c, r_c] = \pi_{\mathsf{Dim}(p)}(\mathcal{R}(p_c))$ with the range $[l_{\mathsf{Dim}(p)}, r_{\mathsf{Dim}(p)}]$ of B on dimension $\mathsf{Dim}(p)$: If $[l_c, r_c]$ does not intersect $[l_{\mathsf{Dim}(n_i)}, r_{\mathsf{Dim}(n_i)}]$, child p_c is discarded (Line 12). If $[l_c, r_c]$ is fully contained in $[l_{\mathsf{Dim}(n_i)}, r_{\mathsf{Dim}(n_i)}]$, then p_c is pushed with the updated set $D \cup \{\mathsf{Dim}(p)\}$ (Lines 13–14); otherwise, p_c is pushed with the original D (Lines 15). This depth-first exploration continues until the stack becomes empty or the number of candidate nodes reaches the budget c_e. For each candidate node $p \in \mathcal{C}$, the algorithm designates the first object satisfying B from $\mathcal{O}(p)$ as an entry point (Lines 16–18). The resulting set $\mathcal{P} \subseteq O_B$ is well spread over O_B in the attribute space, providing diverse starting points for the subsequent greedy search and mitigating the risk of being trapped in poor local optima.

On-the-fly neighbor reconstruction. Algorithm 2 reconstructs neighbors of an object o w.r.t. a range predicate B. The reconstructed neighbor list $\mathcal{N}(o)$ is initialized as empty (Line 1). Starting from the root, it descends the tree; at each level ℓ, it locates the unique node p such that $o \in \mathcal{O}(p)$ and scans the neighbors of o in G_p. If a neighbor o_v has already been marked as visited in previous expansions, it is skipped (Line 5); otherwise, o_v is marked as visited and appended to $\mathcal{N}(o)$ if it satisfies B (Lines 6-8). The procedure terminates once $|\mathcal{N}(o)| = c_n$ or the leaf level is reached, and returns the neighbor list $\mathcal{N}(o)$.

This reconstruction strategy is deliberately simple: more sophisticated variants would add complexity, and their benefits tend to diminish as the number of attributes m increases. In contrast, aggregating in-range neighbors along the path of o provides a direct and effective way to exploit the multi-level structure of KHI.

Query algorithm. Algorithm 3 summarizes the query procedure based on KHI $\mathcal{J}$. Given a query $Q = (q, B)$ with target size k, exploration factor ef, entry-point budget c_e, and neighbor budget c_n, the algorithm maintains two priority queues: a candidate queue $\mathcal{C}_q$

Algorithm 3: Query($\mathcal{J}, Q, k, ef, c_e, c_n$)

Input: Index KHI $\mathcal{J}$, query $Q = (q, B)$, target size k, exploration factor ef, entry-point budget c_e, and neighbor budget c_n
Output: An object set $\hat{R}$

1 $\hat{R}, \mathcal{C}_q$, visited $\leftarrow \emptyset$;
2 $\mathcal{P} \leftarrow$ RangeFilter($\mathcal{J}.T, B, c_e$);
3 **foreach** $o \in \mathcal{P}$ **do**
4 $dist \leftarrow \delta(o.x, q)$;
5 $\hat{R}$.push($o, dist$); $\mathcal{C}_q$.push($o, dist$);
6 visited$[o] \leftarrow$ true;
7 **while** $\mathcal{C}_q \neq \emptyset \wedge (|\hat{R}| < ef \vee \mathcal{C}_q.top().dist \leq \hat{R}.top().dist)$ **do**
8 $o_u \leftarrow \mathcal{C}_q$.pop();
9 $\mathcal{N}(o_u) \leftarrow$ ReconsNbr($\mathcal{J}, o_u, B, c_n$, visited);
10 **foreach** $o_v \in \mathcal{N}(o_u)$ **do**
11 $dist \leftarrow \delta(o_v.x, q)$;
12 $\hat{R}$.push($o_v, dist$); $\mathcal{C}_q$.push($o_v, dist$);
13 **if** $|\hat{R}| > ef$ **then** $\hat{R}$.pop();
14 **return** *the k closest objects in* $\hat{R}$;

and a result queue $\hat{R}$ (Line 1). Both queues store pairs $(o, \delta(o.x, q))$, ordered by distance to q; $\mathcal{C}_q$ is a min-heap, whereas $\hat{R}$ is a bounded max-heap of capacity ef.

The algorithm first invokes Algorithm 1 to obtain a set $\mathcal{P}$ of entry points in O_B (Line 2). For each $o \in \mathcal{P}$, it computes $\delta(o.x, q)$, inserts (o, *dist*) into both $\hat{R}$ and $\mathcal{C}_q$, and marks o as visited (Lines 3–6). While $\mathcal{C}_q$ is non-empty and either $|\hat{R}| < ef$ or the best candidate in $\mathcal{C}_q$ is no farther than the worst object in $\hat{R}$, the algorithm extracts from $\mathcal{C}_q$ the current closest object o_u (Line 8). Its neighbors $\mathcal{N}(o_u)$ under B are reconstructed by Algorithm 2 (Line 9). For each $o_v \in \mathcal{N}(o_u)$, the distance $\delta(o_v.x, q)$ is computed and (o_v, *dist*) is inserted into both $\hat{R}$ and $\mathcal{C}_q$ (Lines 10–12). If $|\hat{R}|$ exceeds ef, the farthest object in $\hat{R}$ is removed to keep the result queue bounded (Line 13). After the greedy search terminates, the k closest objects in $\hat{R}$ are returned as the query result to Q (Line 14).

In our implementation, we set $c_e = k$, since larger values bring only limited gains in search quality. The neighbor budget is set to $c_n = M$, matching the maximum degree bound in the filtered HNSW graphs for the same reason.

4.3 Index Construction

Considering the structure of KHI, index construction proceeds as follows. First, we build the partitioning tree T over all objects in the attribute space. Then, leveraging the hierarchy of T, we construct a single-level filtered HNSW graph at each node to capture proximity relationships among the corresponding embeddings.

Partitioning tree construction. We construct the partitioning tree T in a top-down manner, as shown in Algorithm 4. Starting from the root node p_r, each node p stores a splitting dimension $\mathsf{Dim}(p)$ and a set of excluded dimensions $\mathsf{BL}(p) \subseteq \{1, \ldots, m\}$. The construction proceeds through a stack-based traversal.

When a node p is popped from the stack, partitioning stops if $|\mathcal{O}(p)| \leq c_l$ or all m dimensions have been included in $\mathsf{BL}(p)$

Algorithm 4: BuildTree (O, τ, c_l)

Input: Object set O, balance threshold τ, leaf capacity c_l
Output: Partitioning tree T

1 $p_r \leftarrow$ the root node of T;
2 $\mathcal{O}(p_r) \leftarrow O$; $\mathsf{Dim}(p_r) \leftarrow 1$; $\mathsf{BL}(p_r) \leftarrow \emptyset$;
3 stack.push(p_r);
4 **while** stack $\neq \emptyset$ **do**
5 $\quad p \leftarrow$ stack.pop();
6 $\quad$ **if** $|\mathcal{O}(p)| \leq c_l \vee |\mathsf{BL}(p)| = m$ **then continue**;
7 $\quad$ **while** $\mathsf{Dim}(p) \in \mathsf{BL}(p)$ **do**
8 $\quad\quad \mathsf{Dim}(p) \leftarrow (\mathsf{Dim}(p) \bmod m) + 1$;
9 $\quad$ sort objects in $\mathcal{O}(p)$ increasingly by $o.t.t_{\mathsf{Dim}(p)}$;
10 $\quad$ set $s(p)$ as the lower median;
11 $\quad \mathcal{O}(p_l) \leftarrow \{o \mid o.t.t_{\mathsf{Dim}(p)} \leq s(p)\}$;
12 $\quad \mathcal{O}(p_r) \leftarrow \mathcal{O}(p) \setminus \mathcal{O}(p_l)$;
13 $\quad$ **if** $\tau \cdot \min(|\mathcal{O}(p_l)|, |\mathcal{O}(p_r)|) \leq \max(|\mathcal{O}(p_l)|, |\mathcal{O}(p_r)|)$ **then**
14 $\quad\quad \mathsf{BL}(p) \leftarrow \mathsf{BL}(p) \cup \{\mathsf{Dim}(p)\}$;
15 $\quad\quad$ stack.push(p); **continue**;
16 $\quad$ **else**
17 $\quad\quad \mathsf{Dim}(p_l), \mathsf{Dim}(p_r) \leftarrow (\mathsf{Dim}(p) \bmod m) + 1$;
18 $\quad\quad \mathsf{BL}(p_l), \mathsf{BL}(p_r) \leftarrow \mathsf{BL}(p)$;
19 $\quad\quad$ the left and right child of $p \leftarrow (p_l, p_r)$;
20 $\quad\quad$ stack.push(p_l); stack.push(p_r);
21 **return** T;

(Line 6). Otherwise, we advance $\mathsf{Dim}(p)$ in a round-robin manner until we find an eligible (non-excluded) dimension (Lines 7-8), and then determine the split value $s(p)$ on dimension $\mathsf{Dim}(p)$ (Lines 9-10). Concretely, let $\{o.t.t_{\mathsf{Dim}(p)} \mid o \in \mathcal{O}(p)\}$ be the multiset of attribute values on this dimension, sorted in ascending order. We define $\mathrm{mid} = \lfloor (|\mathcal{O}(p)| - 1)/2 \rfloor$ and choose $s(p)$ as the value at position mid, which induces two subsets $\mathcal{O}(p_l)$ and $\mathcal{O}(p_r)$ for the left and right child, respectively (Lines 11–12). We then check the balance condition. If $\tau \cdot \min(|\mathcal{O}(p_l)|, |\mathcal{O}(p_r)|) \leq \max(|\mathcal{O}(p_l)|, |\mathcal{O}(p_r)|)$, the split is considered skewed: $\mathsf{Dim}(p)$ is added to $\mathsf{BL}(p)$ and p is pushed back onto the stack to retry with another dimension (Lines 14–15). Otherwise, we accept the split, create the two children, assign to each the next splitting dimension in the round-robin order, inherit the set of excluded dimensions, and push them onto the stack to continue partitioning (Lines 17–20).

Filtered HNSW graph construction. Given the partitioning tree T, we construct for each node p a single-level filtered HNSW graph G_p in a bottom-up manner (Algorithm 5). We first build a queue of tree nodes ordered from leaves to root: nodes are grouped by depth, and the levels are enqueued from the deepest level up to the root, ensuring that every child is processed before its parent (Line 1).

If the dequeued node p is a leaf, we build G_p by invoking the standard HNSW building procedure with maximum degree bound M (Lines 4–5). Otherwise, we set G_p to G_{p_l} (Line 8), and then incrementally merge the objects in $\mathcal{O}(p_r)$ into G_p. For each object $o \in \mathcal{O}(p_r)$, we run a greedy search on the current G_p to obtain a candidate set $\hat{R}$ of up to ef_b objects (Line 10). Its neighbor list in G_p is then determined by applying the RNG-based heuristic from Section 2.2 to the union of candidates from the parent and

Algorithm 5: BuildGraph(T, M, ef_b)

Input: Partitioning tree T, maximum degree bound M, and build exploration factor ef_b
Output: Filtered HNSW graphs in KHI
// tree nodes ordered from leaves to root

1 queue $\leftarrow$ BottomUpLevelTraversal(T);
2 **while** queue $\neq \emptyset$ **do**
3 $\quad p \leftarrow$ queue.pop();
4 $\quad$ **if** *p is a leaf* **then**
5 $\quad\quad$ build the filtered HNSW graph G_p;
6 $\quad$ **else**
7 $\quad\quad$ $(p_l, p_r) \leftarrow$ the left and right child of p;
8 $\quad\quad$ $G_p \leftarrow G_{p_l}$;
9 $\quad\quad$ **foreach** $o \in \mathcal{O}(p_r)$ **do**
10 $\quad\quad\quad \hat{R} \leftarrow \mathsf{GreedySearch}(o, ef_b, G_p)$;
11 $\quad\quad\quad \mathcal{N}(o)$ in $G_p \leftarrow \mathsf{Prune}(\hat{R} \cup \mathcal{N}(o)$ in $G_{p_r})$;
12 $\quad\quad\quad$ **foreach** $\hat{o} \in \mathcal{O}(p_l) \cap \mathcal{N}(o)$ *in* G_p **do**
13 $\quad\quad\quad\quad \mathcal{N}(\hat{o})$ in $G_p \leftarrow \mathsf{Prune}(\{o\} \cup \mathcal{N}(\hat{o})$ in $G_p)$;
14 **return** $\{G_p\}$;

right-child graphs, obtaining an M-bounded neighbor list while retaining high-quality neighbors (Line 11). Finally, we update neighbor lists of affected objects in $\mathcal{O}(p_l)$: for each $\hat{o} \in \mathcal{O}(p_l) \cap \mathcal{N}(o)$ in G_p, we refine $\mathcal{N}(\hat{o})$ in G_p by applying the same RNG-based pruning to $o \cup \mathcal{N}(\hat{o})$ in G_p (Lines 12–13). Processing all nodes in this bottom-up order yields the collection $\{G_p\}$ of graphs attached to the nodes of KHI (Line 14). In our implementation, we set $ef_b = M$.

Parallel graph construction. In the overall index construction, building the filtered HNSW graphs dominates the runtime because it requires many distance computations between high-dimensional embeddings (see Appendix B). To reduce construction time, we parallelize graph construction while keeping the resulting graphs identical to the sequential version.

Recall that the nodes of T are grouped by depth in a bottom-up order. Since the construction of graph G_p at a node p only depends on the graphs of its children, all nodes on the same level can be processed independently, so we first build graphs in a level-wise parallel fashion. However, near the upper levels of the tree, the number of nodes per level becomes small and the workload across nodes can become imbalanced, limiting the benefit of purely level-wise parallelization. To address this, we introduce a threshold τ_p on the number of nodes per level: when a level contains fewer than τ_p nodes, we switch from level-wise parallelism to intra-node parallelism. In this case, each remaining node p typically has a large $\mathcal{O}(p_r)$, and we parallelize the merge step for objects in $\mathcal{O}(p_r)$ (Lines 9–13 of Algorithm 5) by distributing these objects across worker threads, each independently running the greedy search and RNG-based pruning to update the corresponding neighbor lists in G_p. In our implementation, we set $\tau_p = 100$.

Discussion. Our partitioning tree T is inspired by classical KD-trees but departs from standard variants in several key aspects. Each node maintains a set of excluded dimensions $\mathsf{BL}(p)$ and uses a balance threshold τ to avoid highly skewed splits, which yields a provable bound on the tree height. Moreover, T is tailored to RFANNS: it is built purely in the attribute space, and its nodes

Table 1: Datasets.

Dataset	n	d	m	Vector Type	Attributes
Youtube	3,650,716	1,024	4	Video	PublishYear, #Views, #Likes, #Comments
DBLP	6,357,867	768	4	Text	PublishYear, #Citations, #References, #Authors
MSMarco	8,000,000	384	5	Text	#Words, #Chars, #Sentences, #UniqueWords, TFIDF
LAION	9,636,707	512	3	Image	Width, Height, Similarity

serve as anchors for filtered HNSW graphs rather than for traditional point-location queries as in standard KD-trees. These design choices are driven by the requirements of RFANNS and, to the best of our knowledge, have not been explored in existing KD-tree–based indexes.

5 Experiments

5.1 Experimental Setup

Our experimental evaluation focuses on two aspects: query performance and index efficiency. For query performance, we compare our method against baselines under various query settings. For index efficiency, we report the index construction time and the empirical space usage of our method and baselines.

Datasets. To evaluate the performance of all methods, we use four real-world public datasets. Since few public corpora provide both multi-attribute numeric metadata and corresponding vector representations, we construct our evaluation datasets by encoding the raw content with open-source embedding models and pairing the resulting vectors with the original attributes. The resulting datasets are summarized in Table 1. Laion[1] contains image–text pairs with three numeric attributes: image width, image height, and a similarity score. MSMarco[2] and DBLP[3] are text collections with document-level statistics such as the number of words, sentences, and citations. Youtube[4] consists of video records with temporal and popularity metadata, including publication year, the number of views, and the number of likes.

Algorithms. Among existing RFANNS indexes, we consider two representative methods, SeRF and iRangeGraph, as introduced in Section 2.3. SeRF is designed for single-attribute range filters. Although its paper outlines a possible extension to multi-attribute RFANNS, this variant is only described at a high level and, to the best of our knowledge, has not been implemented. Since our focus is on multi-attribute RFANNS, we do not evaluate SeRF experimentally and instead use iRangeGraph as the main RFANNS baseline.

In addition to iRangeGraph, we also include a simple pre-filtering baseline. Given a query $Q = (q, B)$, Prefiltering first scans all objects to materialize the subset O_B. It then performs exact nearest neighbor search for q over O_B in the embedding space by exhaustively computing distances and returning the k closest objects.

Queries. For each dataset, we generate 1,000 RFANNS queries of the form $Q = (q, B)$. During dataset construction, we randomly sample 1,000 raw objects from the original corpus and encode them with the same embedding model used to build the dataset; these vectors are stored separately and used only as query embeddings.

The range predicate B is generated in the attribute space with a target selectivity σ and a relative tolerance parameter tol. Following iRangeGraph, we parameterize the target selectivity as $\sigma = 1/2^i$. Since multi-attribute range predicates often result in relatively small selectivities in practice, we focus on $i \in \{4, 6, 8\}$, i.e., $\sigma \in \{1/16, 1/64, 1/256\}$.

To instantiate B, we first draw a random sample of attribute tuples and, for each attribute, build a sorted list of finite values to support quantile queries. For each query, we derive per-attribute intervals by selecting lower and upper quantiles on the sampled tuples so that the empirical selectivity of the resulting B lies within $[\sigma(1 - \text{tol}), \sigma(1 + \text{tol})]$. By default, we set tol $= 0.5$.

Equipment. All algorithms are implemented in C++, compiled with the g++ compiler at -O3 optimization level, and run on a Linux machine equipped with an Intel Xeon Gold 6230 CPU @ 2.10 GHz and 256 GB of RAM.

5.2 Query performance

Overall performance. We evaluate query efficiency using queries per second (QPS), defined as the number of RFANNS queries processed per second, and report the trade-off between recall and QPS. Figure 4 summarizes the query performance of the three methods on all four datasets. For each figure, the curve is obtained by varying the exploration factor *ef* on a fixed index, yielding a series of points with different recall–QPS combinations. Both KHI and iRangeGraph use a maximum degree bound $M = 32$. In this experiment, we set the cardinality of B to m (i.e., each range predicate constrains all attributes) and fix the target size $k = 10$. We make the following observations:

(1) The two graph-based methods, KHI and iRangeGraph, achieve high recall across all datasets and selectivities: on Laion, MSMarco, and DBLP the recall exceeds 0.99, and on the more challenging Youtube it remains above 0.9. However, their query throughput shows clear differences.

(2) When $\sigma \in \{1/16, 1/64, 1/256\}$, KHI consistently outperforms iRangeGraph in QPS at comparable recall levels. At recall 0.95, the QPS gains on Laion are 1.66×, 2.34×, and 3.99× over iRangeGraph as σ decreases from $1/16$ to $1/256$; on MSMarco, the corresponding speedups are 1.20×, 1.67×, and 4.92×; and on DBLP, they are 1.76×, 1.48×, and 3.15×, respectively. On the more challenging Youtube dataset, KHI yields 1.68×, 8.89×, and 38.08× speedups at recall 0.9 for $\sigma = 1/16$, $1/64$, and $1/256$, respectively. Averaged over Laion, MSMarco, and DBLP and the three selectivities, KHI attains an overall speedup of 2.46× in QPS compared with iRangeGraph, while on Youtube the average speedup reaches 16.22×. Overall, the advantage grows more pronounced for lower selectivities and more challenging datasets.

(3) When the target recall is relaxed below 1.0, both KHI and iRangeGraph yield notable speedups over Prefiltering. At recall

[1] https://laion.ai/blog/laion-400-open-dataset/
[2] https://microsoft.github.io/msmarco/
[3] https://open.aminer.cn/open/article?id=655db2202ab17a072284bc0c
[4] https://research.google.com/youtube8m/

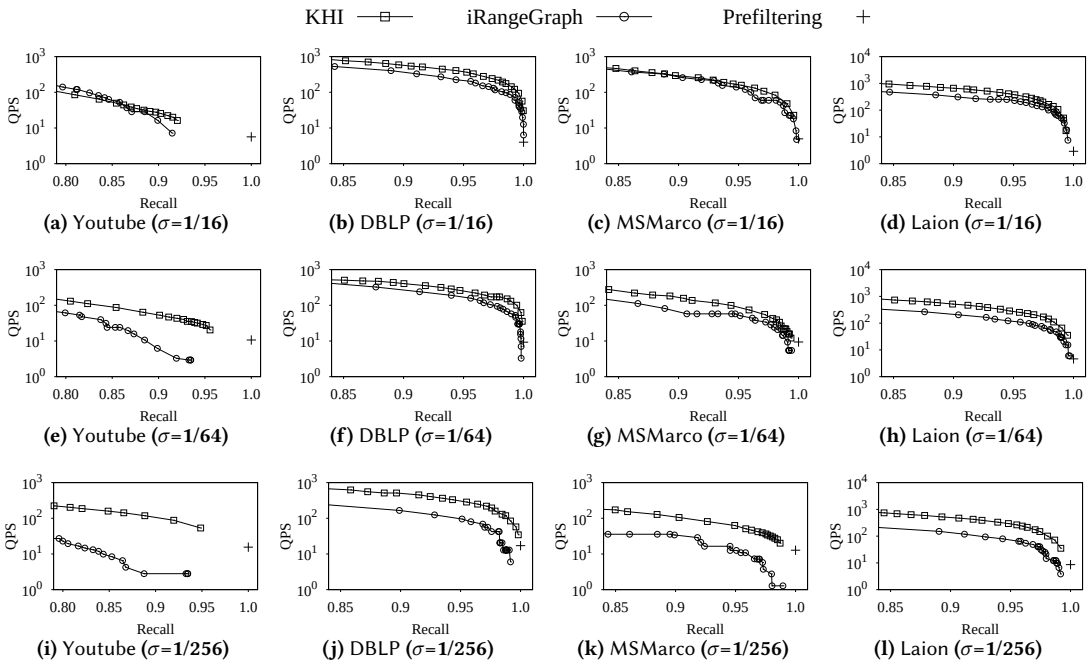


Figure 4: Overall query performance.

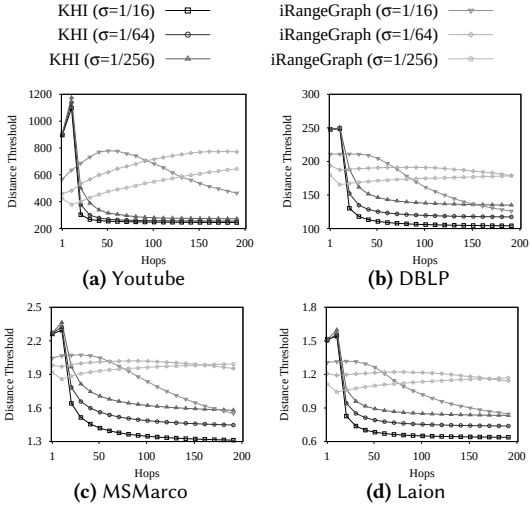


Figure 5: Evolution of distance threshold during search.

0.95 on Laion, MSMarco, and DBLP (and 0.9 on Youtube), KHI consistently achieves higher QPS than Prefiltering on all datasets, with an average speedup of 35.59×. In the same setting, iRangeGraph also outperforms Prefiltering on all datasets except Youtube, with an average speedup of 24.91×; on Youtube, however, iRangeGraph falls short of Prefiltering, as shown in Figure 4e and Figure 4i.

Convergence of the distance threshold. To better understand how the graph-based methods behave, we further examine the evolution of the distance threshold during greedy search. We measure the search progress in hops, where one hop corresponds to expanding the neighbors of a candidate object. At a given hop, we define the distance threshold as the distance from q to the farthest object in $\hat{R}$, where $\hat{R}$ denotes the current set of best-so-far candidates. In this experiment, we use the same settings as in the overall query performance study. We report the evolution of the distance threshold at $ef = 300$ for $\sigma \in \{1/16, 1/64, 1/256\}$.

Figure 5 shows how this threshold evolves with the number of hops across all datasets and selectivities. For KHI, across all datasets and selectivities, the distance threshold decreases rapidly in the first few hops and then quickly stabilizes. This suggests that KHI is able to tighten the distance threshold more aggressively and prune inferior candidates earlier, while still preserving high recall. In contrast, for iRangeGraph, the distance threshold decays much more slowly and often remains high over many hops, especially for smaller selectivities, e.g., $\sigma = 1/64$ and $\sigma = 1/256$. As discussed in Section 3.1, this behavior stems from its reliance on many out-of-range neighbors. Thus, iRangeGraph tends to explore more candidates during search. These search dynamics are consistent with the higher QPS achieved by KHI, compared with iRangeGraph at comparable recall levels in Figure 4.

Varying target size k. We study the impact of the target size k on query performance and present the results on Laion as a representative example. We fix the target recall at 0.95 and evaluate QPS while varying k from 20 to 100, as shown in Figure 6; the case $k = 10$ has already been reported in Figure 4. All other settings follow the overall query performance study.

We fix the target recall at 0.95 and report the QPS achieved by KHI and iRangeGraph on Laion while varying $k \in \{10, 20, 50, 100\}$. When $\sigma = 1/16$, KHI achieves speedups over iRangeGraph, with QPS improvements of 1.66×, 1.91×, 2.38×, and 2.60× at $k = 10$, 20, 50, and 100, respectively. For $\sigma = 1/64$, the gains become larger, with speedups of about 2.34×, 2.51×, 4.81×, and 5.10× for $k = 10$, 20, 50, and 100, respectively. For the case $\sigma = 1/256$, the advantage of KHI is even more pronounced, reaching 3.99×, 4.90×, 7.20×, and

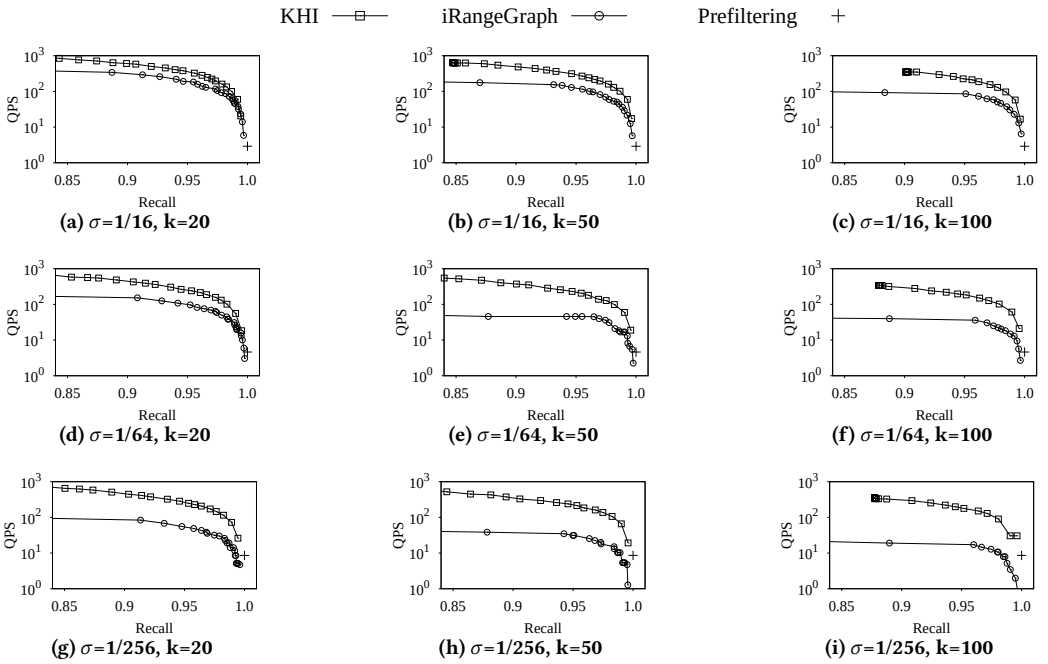


Figure 6: Query performance by varying target size k.

10.52× for $k = 10$, 20, 50, and 100, respectively. Overall, the performance gap between KHI and iRangeGraph widens as k increases, especially under low selectivity. This effect may be attributed to the fact that larger k forces the search to explore a broader local neighborhood; in this regime, KHI benefits from higher-quality in-range neighbors, whereas iRangeGraph spends more hops on out-of-range candidates, so KHI can obtain the k nearest neighbors with fewer expansions and consequently higher QPS. We also observe that, when the recall is relaxed to 0.95, both KHI and iRangeGraph achieve substantially higher QPS than Prefiltering. On Laion, averaging over $k \in \{10, 20, 50, 100\}$, their QPS speedups over Prefiltering are 62.76× and 24.16×, respectively.

Varying cardinality of B. We study the effect of the cardinality of B on query performance. We report the results on DBLP; the other datasets exhibit similar trends. We fix the target recall at 0.95 and evaluate QPS while varying the cardinality of B from 2 to 4. All other settings follow the overall query performance study. When the cardinality of B is less than m, for each query we choose the constrained attributes uniformly at random from the m attributes. In Figure 7, we plot the cases where the cardinality of B is 2 and 3; the case where the cardinality of B equals m has already been reported in Figure 4 and is omitted here for brevity.

When $\sigma = 1/16$, KHI achieves about 1.06×, 1.29×, and 1.76× higher QPS than iRangeGraph when the cardinality of B is 2, 3, and 4, respectively. For $\sigma = 1/64$, the speedup ranges from roughly parity when the cardinality of B is 2 to about 1.43× and 1.48× when it is 3 and 4. For $\sigma = 1/256$, the gains become more pronounced, reaching around 1.53×, 2.47×, and 3.15× for cardinalities 2, 3, and 4, respectively. These results indicate that, as B involves more attributes, KHI increasingly outperforms iRangeGraph, especially under low selectivity, highlighting the benefit of its attribute-space partitioning and and the resulting high-quality graphs. Moreover, compared with the exhaustive Prefiltering baseline on DBLP,

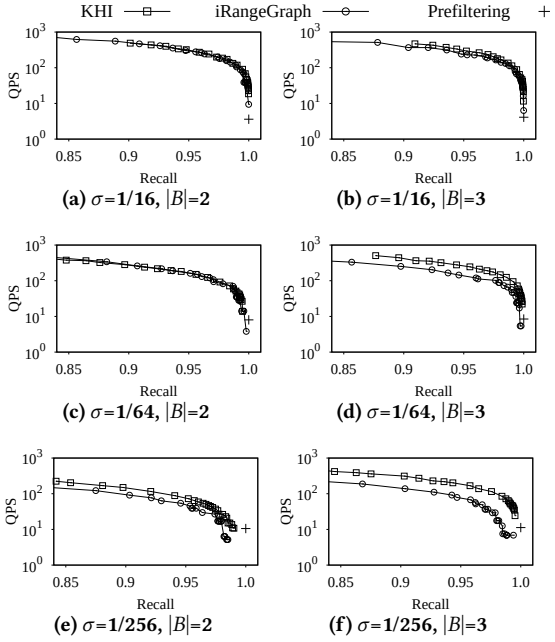


Figure 7: Query performance by varying cardinality of B.

KHI and iRangeGraph achieve average QPS speedups of 39.46× and 27.81×, respectively, across these settings.

5.3 Index construction

Index construction time. Since single-threaded index construction runs slow on our datasets, we report multi-threaded construction times under a 16-thread configuration with maximum degree bound $M = 32$ for both KHI and iRangeGraph. As shown in Table 2, across the four datasets, KHI can be built in about 2,000–5,500 s,

Table 2: Index construction time.

Dataset	KHI (s)	iRangeGraph (s)
Youtube	1,991.75	6,675.85
DBLP	4,642.37	13,973.42
MSMarco	5,507.91	17,841.86
Laion	4,661.85	16,250.53

Table 3: Index size.

Dataset	KHI (GB)	iRangeGraph (GB)
Youtube	3.27	2.89
DBLP	9.74	10.14
MSMarco	18.49	16.89
Laion	15.85	13.97

whereas iRangeGraph requires roughly 6,700–17,800 s. These correspond to speedups of 3.35×, 3.01×, 3.24×, and 3.49×, respectively, i.e., an average reduction of construction time by about a factor of 3.27. This confirms that our parallelization strategy, which combines level-wise and intra-node parallelism, substantially accelerates index construction compared with the pure level-wise parallelism used in iRangeGraph.

Index size. Table 3 reports the index sizes on all datasets, with $M = 32$ for both indexes. Overall, the two indexes occupy comparable space: across all datasets, the size of KHI differs from that of iRangeGraph by less than 15%. Although our partitioning tree T is not strictly balanced, the skew-aware splitting strategy keeps most objects at moderate depths, so that only a small fraction reside in deeper levels. As a result, the additional nodes and their filtered HNSW graphs incur only limited extra space. Further details on the empirical tree height are provided in Appendix C.

6 Related Work

Approximate nearest neighbor search. Approximate nearest neighbor search in high-dimensional Euclidean spaces has been extensively studied. Existing methods can be broadly categorized into graph-based [9, 13, 20, 23, 29, 32, 33, 48], quantization-based [1, 5, 6, 15, 18, 24], and hashing-based [10, 14, 21, 39, 40]; we refer readers to recent tutorials [11, 38], surveys [29, 44, 45], and benchmark studies [2, 3] for a comprehensive overview of this literature. Among these categories, graph-based methods have demonstrated particularly strong performance: HNSW [33], NSG [13], and DiskANN [23] have been widely adopted in industry [42]. More recently, several GPU-accelerated graph-based ANN methods [17, 35, 51, 53] have been proposed to increase throughput by exploiting massive parallelism. These methods are highly optimized for ANN without attribute constraints. In contrast, we focus on extending ANN to handle multi-attribute range filters efficiently, a setting not addressed in the above line of work.

Range-filtering nearest neighbor search. Motivated by the practical importance of attribute-filtered ANN queries, many algorithms and systems have been proposed [8, 12, 16, 34, 36, 42, 43, 46, 49, 50, 54]. We refer readers to the survey [30] for a comprehensive overview. Across different application scenarios, attribute predicates can be broadly grouped into equality predicates, numeric range predicates, and more general comparison predicates. For equality predicates [8, 16, 43], objects carry categorical labels and the goal is to retrieve the k nearest neighbors whose labels exactly match the query label(s). This setting differs from ours, where queries impose multi-dimensional numeric constraints. Range predicates enforce lower and/or upper bounds on numeric attributes. Most existing RFANNS methods for range predicates focus on single-attribute filters [12, 50, 54], where the condition is specified on a single numeric dimension. In contrast, we study RFANNS with multi-attribute numeric ranges, filling this gap. For general comparison predicates, several systems support attribute-filtered ANN queries over heterogeneous attributes, including numeric, categorical, and other application-specific fields [34, 36, 42, 46, 49]. Prior work has shown that different attribute types exhibit markedly different properties and that strong performance typically requires index designs tailored to a specific predicate class [50]. Since our method is specifically designed for numeric range predicates, we therefore compare primarily against specialized RFANNS indexes rather than these general-purpose systems. In addition, GPU-based methods for attribute-filtered ANN queries have recently been explored [47]; these techniques are largely orthogonal to our contribution and could in principle be combined with our index.

Dynamic range-filtering nearest neighbor search. Several works [25, 37, 52] have been proposed to support RFANNS in dynamic settings. DIGRA [25] organizes objects in a B-tree over attributes and maintains HNSW graphs at tree nodes, both of which admit efficient maintenance under insertions and deletions. DSG [37] proposes a dynamic segment graph structure, while requiring only $O(\log n)$ new edges in expectation per insertion.

7 Conclusion

In this paper, we study RFANNS in high-dimensional Euclidean spaces with multi-attribute numeric range predicates, a setting that is increasingly important in modern vector databases but has been insufficiently explored by existing work. We propose KHI, a tailored RFANNS index that combines an attribute-space partitioning tree with filtered HNSW graphs anchored at tree nodes. The partitioning tree employs a skew-aware splitting strategy, which adapts to attribute distributions while still admitting provable bounds on the tree height. On top of this structure, we design an efficient query algorithm that relies only on in-range neighbors yet still achieves high recall and strong query performance.

We conduct extensive experiments on four real-world datasets and compare KHI with iRangeGraph and a prefiltering baseline. The results demonstrate that KHI consistently achieves better query throughput than the baselines. Specifically, KHI achieves an average QPS speedup of 2.46× over iRangeGraph on Laion, DBLP, and MSMarco, and 16.22× on the more challenging Youtube, with gains further increasing for smaller selectivities, larger k, and higher predicate cardinality. When the recall is relaxed to 0.95 (or 0.9 on Youtube), KHI also consistently outperforms Prefiltering in QPS across all datasets, with an average speedup of 35.59×. In terms of index construction, KHI is on average 3.27× faster than iRangeGraph under a 16-thread configuration, while incurring at most roughly a 15% space overhead compared with iRangeGraph. Future work includes supporting fully dynamic workloads and exploring GPU implementations for large-scale deployments.

References

[1] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. In *42nd International Conference on Very Large Data Bases*, Vol. 9. VLDB Endowment, New Delhi, India, 1–12.

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

[3] Martin Aumüller and Matteo Ceccarello. 2023. Recent Approaches and Trends in Approximate Nearest Neighbor Search, with Remarks on Benchmarking. *IEEE Data Eng. Bull.* 46, 3 (2023), 89–105.

[4] Franz Aurenhammer. 1991. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM computing surveys (CSUR)* 23, 3 (1991), 345–405.

[5] Artem Babenko and Victor Lempitsky. 2014. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Columbus, Ohio, USA, 931–938.

[6] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence* 37, 6 (2014), 1247–1260.

[7] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

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

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

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

[11] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2021. New trends in high-d vector similarity search: al-driven, progressive, and distributed. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3198–3201.

[12] Joshua Engels, Benjamin Landrum, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2024. Approximate nearest neighbor search with window filters. In *Proceedings of the 41st International Conference on Machine Learning*. PMLR, Vienna, Austria, 12469–12490.

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

[14] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. ACM New York, NY, USA, Scottsdale, Arizona, USA, 541–552.

[15] Jianyang Gao and Cheng Long. 2024. Rabitq: Quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

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

[17] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik PA Lensch. 2022. Ggnn: Graph-based gpu nearest neighbor search. *IEEE Transactions on Big Data* 9, 1 (2022), 267–279.

[18] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, PMLR, Vienna, Austria, 3887–3896.

[19] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In *International conference on machine learning*. PMLR, PMLR, Vienna, Austria, 3929–3938.

[20] Ben Harwood and Tom Drummond. 2016. Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Las Vegas, Nevada, USA, 5713–5722.

[21] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 9, 1 (2015), 1–12.

[22] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM New York, NY, USA, Dallas, Texas, USA, 604–613.

[23] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing Systems* 32 (2019), 13766–13776.

[24] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

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

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

[27] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering.. In *EMNLP (1)*. ACL, Punta Cana, Dominican Republic, 6769–6781.

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

[29] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2019), 1475–1488.

[30] Yanjun Lin, Kai Zhang, Zhenying He, Yinan Jing, and X Sean Wang. 2025. Survey of Filtered Approximate Nearest Neighbor Search over the Vector-Scalar Hybrid Data. *arXiv preprint arXiv:2505.06501* (2025).

[31] Ying Liu, Dengsheng Zhang, Guojun Lu, and Wei-Ying Ma. 2007. A survey of content-based image retrieval with high-level semantics. *Pattern recognition* 40, 1 (2007), 262–282.

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

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

[34] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-throughput vector similarity search in knowledge graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[35] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2024. Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, IEEE, Utrecht, Netherlands, 4236–4247.

[36] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. Acorn: Performant and predicate-agnostic search over vector embeddings and structured data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[37] Zhencan Peng, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2025. Dynamic Range-Filtering Approximate Nearest Neighbor Search. *Proceedings of the VLDB Endowment* 18, 10 (2025), 3256–3268.

[38] Jianbin Qin, Wei Wang, Chuan Xiao, Ying Zhang, and Yaoshu Wang. 2021. High-dimensional similarity query processing for data science. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. ACM New York, NY, USA, Singapore, 4062–4063.

[39] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proceedings of the VLDB Endowment* 8 (2014), 1–12.

[40] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2010. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Transactions on Database Systems (TODS)* 35, 3 (2010), 1–46.

[41] Godfried T Toussaint. 1980. The relative neighbourhood graph of a finite planar set. *Pattern recognition* 12, 4 (1980), 261–268.

[42] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 international conference on management of data*. ACM New York, NY, USA, Xi'an, China, 2614–2627.

[43] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2023. An efficient and robust framework for approximate nearest neighbor search with attribute constraint. *Advances in Neural Information Processing Systems* 36 (2023), 15738–15751.

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

[45] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Graph-and Tree-based Indexes for High-dimensional Vector Similarity Search: Analyses, Comparisons, and Future Directions. *IEEE Data Eng. Bull.* 46, 3 (2023), 3–21.

[46] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment*

13, 12 (2020), 3152–3165.

[47] Jingyi Xi, Chenghao Mo, Ben Karsin, Artem Chirkin, Mingqin Li, and Minjia Zhang. 2025. VecFlow: A High-Performance Vector Data Management System for Filtered-Search on GPUs. *Proceedings of the ACM on Management of Data* 3, 4 (2025), 1–27.

[48] Jiadong Xie, Jeffrey Xu Yu, and Yingfan Liu. 2025. Graph Based K-Nearest Neighbor Search Revisited. *ACM Transactions on Database Systems* 40 (2025), 1–30.

[49] Jiadong Xie, Jeffrey Xu Yu, Siyi Teng, and Yingfan Liu. 2025. Beyond Vector Search: Querying With and Without Predicates. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–26.

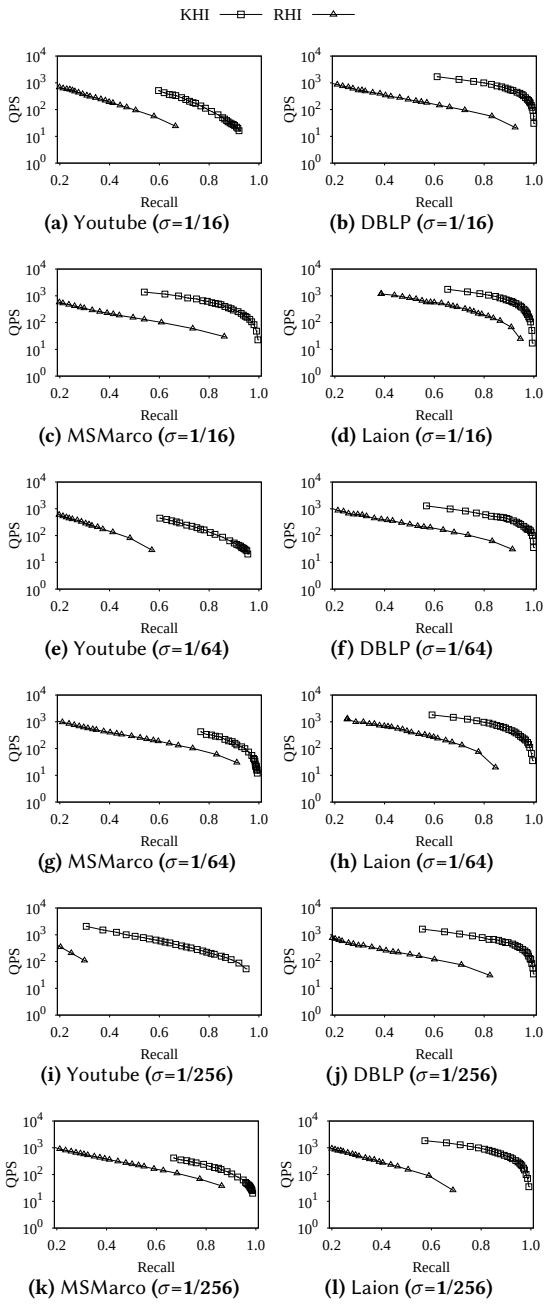
[50] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S Jensen. 2024. irangegraph: Improvising range-dedicated graphs for range-filtering nearest neighbor search. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–26.

[51] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2022. GPU-accelerated proximity graph approximate nearest Neighbor search and construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, IEEE, Kuala Lumpur, Malaysia, 552–564.

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

[53] Weijie Zhao, Shulong Tan, and Ping Li. 2020. Song: Approximate nearest neighbor search on gpu. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, IEEE, Dallas, Texas, USA, 1033–1044.

[54] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: segment graph for range-filtering approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.



(a) Youtube (σ=**1/16**) **(b)** DBLP (σ=**1/16**)

(c) MSMarco (σ=**1/16**) **(d)** Laion (σ=**1/16**)

(e) Youtube (σ=**1/64**) **(f)** DBLP (σ=**1/64**)

(g) MSMarco (σ=**1/64**) **(h)** Laion (σ=**1/64**)

(i) Youtube (σ=**1/256**) **(j)** DBLP (σ=**1/256**)

(k) MSMarco (σ=**1/256**) **(l)** Laion (σ=**1/256**)

Figure 8: Query performance of KHI and RHI.

A Experimental Evaluation of R-tree-based Partitioning

In our method, each object maintains a neighbor list at each level of partitioning tree. A deeper partitioning tree can offer more candidate neighbors to reconstruct neighbor list on-the-fly, which in turn can improve search quality. Therefore, we evaluate R-tree–based partitioning with node fan-outs as small as possible. When the node fan-out is set to 2 or 3, we build an R-tree on a two-million-object subset of DBLP using a commonly used fill factor of 0.5. The resulting trees are extremely deep (thousands of levels for fan-out 2 and hundreds of levels for fan-out 3), indicating near-degenerate structures. Based on this observation, we fix the node fan-out to 4 and retain the commonly used fill factor of 0.5 in all R-tree–based experiments. We refer to the R-tree–based variant as RHI.

Figure 8 reports the recall–QPS trade-off of KHI and RHI on all four datasets for selectivities $\sigma \in \{1/16, 1/64, 1/256\}$, under the setting where $M = 32$, $k = 10$, and the cardinality of B is m. Overall, RHI is consistently dominated by KHI, with both recall and QPS lower across all tested settings. Moreover, as the selectivity decreases, the maximum recall attainable by RHI drops markedly, suggesting that R-tree–based partitioning is poorly suited to highly selective multi-attribute range predicates. These results align with our discussion in Section 3.2: overlapping and unstable partitions lead to filtered HNSW graphs with weaker neighbor lists and less effective routing, which in turn degrades both efficiency and accuracy in the multi-attribute RFANNS setting.

B Breakdown of KHI Construction Time

Table 4: Breakdown of KHI construction time.

Dataset	Tree (s)	Graphs (s)
Youtube	8.92	1,982.83
DBLP	9.66	4,632.71
MSMarco	22.11	5,485.80
Laion	25.11	4,636.74

Table 4 reports the breakdown of KHI construction time into building the partitioning tree and constructing the filtered HNSW graphs. In these experiments, the partitioning tree is built with a single thread, the filtered HNSW graphs are constructed with 16 threads, and the maximum degree bound is set to $M = 32$. Across all datasets, tree construction takes only 8.92–25.11 s, whereas graph construction dominates the cost at 1,982.83–5,485.80 s. The tree-building stage accounts for less than 0.6% of the total construction time on every dataset, confirming that the construction cost of KHI is overwhelmingly dominated by graph construction.

C Tree Height of KHI and iRangeGraph

Table 5: Tree height of KHI and iRangeGraph.

Dataset	KHI	iRangeGraph
Youtube	29	23
DBLP	34	24
MSMarco	33	24
Laion	30	25

Table 5 compares the height of the partitioning tree in KHI with the segment tree used by iRangeGraph. Since the partitioning tree in KHI is not strictly balanced, its height is consistently but only moderately larger. However, this does not translate into a substantial space overhead. As shown in Table 3, the overall index sizes of KHI and iRangeGraph remain close, with the size of KHI exceeding that of iRangeGraph by less than 15% on all datasets. This suggests that the additional depth is concentrated on a small fraction of objects residing at deeper levels, while the majority of objects are distributed over well-balanced levels, so the extra nodes and their graphs contribute only modest space overhead.