iRangeGraph

Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search

Xu, Yuexuan; Gao, Jianyang; Gou, Yutong; Long, Cheng; Jensen, Christian S.

Published in:
Proceedings of the ACM on Management of Data

DOI (link to publication from Publisher):
10.1145/3698814

Publication date:
2024

Document Version
Publisher's PDF, also known as Version of record

iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search

YUEXUAN XU[*], Nanyang Technological University, Singapore
JIANYANG GAO[*], Nanyang Technological University, Singapore
YUTONG GOU, Nanyang Technological University, Singapore
CHENG LONG[†], Nanyang Technological University, Singapore
CHRISTIAN S. JENSEN, Aalborg University, Denmark

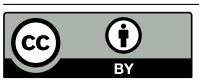
Range-filtering approximate nearest neighbor (RFANN) search is attracting increasing attention in academia and industry. Given a set of data objects, each being a pair of a high-dimensional vector and a numeric value, an RFANN query with a vector and a numeric range as parameters returns the data object whose numeric value is in the query range and whose vector is nearest to the query vector. To process this query, a recent study proposes to build $O(n^2)$ dedicated graph-based indexes for all possible query ranges to enable efficient processing on a database of n objects. As storing all these indexes is prohibitively expensive, the study constructs compressed indexes instead, which reduces the memory consumption considerably. However, this incurs suboptimal performance because the compression is lossy. In this study, instead of materializing a compressed index for every possible query range in preparation for querying, we materialize graph-based indexes, called elemental graphs, for a moderate number of ranges. We then provide an effective and efficient algorithm that during querying can construct an index for any query range using the elemental graphs. We prove that the time needed to construct such an index is low. We also cover an experimental study on real-world datasets that provides evidence that the materialized elemental graphs only consume moderate space and that the proposed method is capable of superior and stable query performance across different query workloads.

CCS Concepts: • **Theory of computation → Data structures design and analysis**; • **Information systems → Information retrieval query processing**.

Additional Key Words and Phrases: Range-Filtering Approximate Nearest Neighbor Search, High-Dimensional Vector

ACM Reference Format:
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 (SIGMOD), Article 239 (December 2024), 26 pages. https://doi.org/10.1145/3698814

[*]Both authors contributed equally to this research.
[†]Corresponding author.
Authors' addresses: Yuexuan Xu[*], Nanyang Technological University, Singapore, yuexuan001@e.ntu.edu.sg; Jianyang Gao[*], Nanyang Technological University, Singapore, jianyang.gao@ntu.edu.sg; Yutong Gou, Nanyang Technological University, Singapore, yutong003@e.ntu.edu.sg; Cheng Long[†], Nanyang Technological University, Singapore, c.long@ntu.edu.sg; Christian S. Jensen, Aalborg University, Aalborg, Denmark, csj@cs.aau.dk.



ACM 2836-6573/2024/12-ART239
https://doi.org/10.1145/3698814

1 INTRODUCTION

Nearest neighbor (NN) search in high-dimensional Euclidean spaces occurs in a broad range of settings, including in information retrieval [57], and machine learning systems (e.g., retrieval-augmented generative AI) [54] and in database systems [75]. The curse of dimensionality reduces the utility of exact NN search on large datasets, primarily due to long response time. To obtain better response time at the cost of reduced accuracy, approximate nearest neighbor (ANN) search [3, 36, 48, 56, 59, 78] had received attention. Among existing ANN methods, graph-based methods have shown promising performance in terms of the trade-off between response time and result accuracy in in-memory settings [17, 18, 34–36, 45, 50, 51, 56, 58, 59, 73, 78].

There has been a recent growing interest in academia [29, 65] and industry (including Apple [62], Zilliz [75], and Alibaba [81]) in supporting ANN queries that also involve constraints on numerical data attributes. Here, a data object consists of two components: (1) a vector and (2) one or more numeric attribute values. A query then takes two parameters: a query vector and constraints on the numeric attribute values. In this paper, we study the query where a data object has one numeric attribute value and the constraint is a range, called the range-filtering ANN (RFANN) query [29, 89]. As mentioned in industrial papers [62, 75, 81], this query is important in many real-world systems. For example, on an e-commerce platform, a user could upload an image of a product, encoded as a vector, to find similar products within a certain price range. A data object is *in-range* if its numeric attribute value is in the query range; otherwise, it is *out-of-range*.

Three strategies that differ in when the range filtering is done can be adopted to process RFANN queries. `Pre-filtering` first eliminates out-of-range data objects and finds the nearest neighbor of the query vector among the remaining objects that are not indexed. `Post-filtering` first conducts (graph-based) ANN search to find data objects whose vectors are close to the query vector and then eliminates any out-of-range data objects. `In-filtering` integrates the range filtering into the (graph-based) ANN search process that then only visits in-range objects (for details, see Section 2.2). While these strategies are intuitive, they have the inherent issue that they are each efficient for certain query workloads only. For example, when the predicate is unselective (i.e., a large fraction of data objects have the attribute value within the query range), `Pre-filtering` degenerates to sequential search on a large dataset. `Post-filtering` and `In-filtering` are suboptimal when the predicate is very selective. In particular, `Post-filtering` will visit many out-of-range objects in order to find the nearest neighbor. As for `In-filtering`, as it only visits in-range objects during the graph-based ANN search, the nearest neighbor might be unreachable during the search process, which reduces search performance.

Most existing proposals [29, 62, 75, 81, 89] address the shortcomings of the three strategies. Milvus [75] and ADBV [81] propose to automatically select a strategy on a per query basis using a cost model. Although this avoids using the strategies where they are distinctively disadvantageous, the inherent issues of the strategies have yet to be fully resolved. In particular, especially when the predicate is neither extremely selective or unselective, no matter which strategy is adopted, the inherent issues remain, causing suboptimal performance.

To address the inherent issues of the basic strategies, a recent study [89] considers building a dedicated graph-based index for every possible query range. To process a query, the index matching the query range is used, thereby reducing the query to an ANN query. However, as also observed in the study, building an index for each possible query range is not feasible in practice because for a database of n objects, $O(n^2)$ dedicated indexes would be built (each corresponding to an index on a sub-interval of the list of objects sorted wrt their numeric values), resulting in a space complexity of $O(n^3m)$ for storing the indexes. The study thus proposes to compress the $O(n^2)$ indexes, to reduce substantially the memory consumption, yielding an empirical memory consumption after

compression that is much smaller than $O(n^3m)$. However, the lossy nature of the compression has the effect of reducing index quality, in turn reducing query performance. Thus, as reported in the study [89] and verified in Section 5.2.1, for queries with short ranges, the method cannot achieve >0.8 recall.

In this paper, we follow the idea of performing search on a dedicated graph-based index for a given query range to avoid the issues inherent to the three basic strategies. However, unlike the existing proposal [89] that builds compressed graphs for all possible query ranges, we materialize graphs for only a moderate number of ranges in preparation for querying. These graphs, called *elemental graphs*, are then used to construct dedicated graphs for any query range seen during querying. The construction occurs on the fly in the sense that edges of an object are only constructed when having to visit its neighbors during query processing. The algorithm for constructing the dedicated graph in the query phase incurs only a low overhead compared to the cost of searching the dedicated graph. We call this new method, which aims to improvise range-dedicated graph indexes during querying, `iRangeGraph`. We further extend `iRangeGraph` to support multi-attribute range-filtering ANN querying. Moreover, we propose a simple yet effective strategy to further enhance the search performance on the multi-attribute query.

The paper makes the following main contributions.

(1) We propose the `iRangeGraph` method for computing the RFANN query. It materializes a moderate number of elemental graphs prior to querying and constructs a dedicated graph index for any given query range based on these elemental graphs on the fly during querying. The overhead of constructing dedicated graph during querying is low, meaning that the RFANN query can be reduced to an ANN query on a dedicated graph index with little overhead. (Section 3)
(2) We extend `iRangeGraph` to support multi-attribute range-filtering ANN queries that involve conjunctive predicates on multiple numeric attributes. We also propose a simple yet effective strategy to further enhance the performance of the multi-attribute RFANN query. (Section 4)
(3) We report on extensive experiments on real-world datasets. The observations are summarized as follows. (a) In general, our method achieves superior search performance with moderate memory footprint on all the datasets and query workloads for RFANN query, e.g., it outperforms the most competitive baseline method by 2x-5x in qps (query per second) at 0.9 recall with consistently smaller memory footprint. (b) The extension to support multi-attribute RFANN queries also exhibits state-of-the-art performance. It outperforms the most competitive baseline by 2x–4x in qps at 0.9 recall. (c) We measure the gap in search performance between our method and the dedicated graph-based indexes which are materialized for the given query ranges. While the latter is expected to enable ideal search performance with impractical space consumption, we find that our method is only slower than the ideal performance by less than 2x at 0.9 recall. Thus, the proposed method is capable of performance very close to the ideal performance with much lower space consumption, i.e., $O(nm \log n)$ vs. $O(n^3m)$. (Section 5)

In addition, Section 2 presents the range-filtering ANN query and existing studies, while Section 6 covers related work and Section 7 concludes the paper.

2 ANN AND RANGE-FILTERING ANN

2.1 ANN and Graph-based Algorithms

Approximate Nearest Neighbor Search. Given a query vector q and a set of data objects where each object has a vector v, the nearest neighbor (NN) query finds the object whose vector has the smallest distance to q. Due to the curse of dimensionality [48], the exact NN query often takes

unacceptably long response time to compute. Thus, it is often relaxed to an approximate NN (ANN) query in order to trade improved response time for reduced result accuracy [36, 38, 48, 51, 59, 78, 79]. Furthermore, the ANN query is usually extended to return k approximate nearest neighbors. For ease of presentation, we assume $k = 1$ in algorithm descriptions, while we note that all the techniques in this paper are adapted and evaluated for a general k in Section 5. Further, we focus on the in-memory setting, where the vectors and indexes are in RAM [3, 36, 59, 78].

Graph-based ANN Methods. Among the proposed ANN algorithms [1, 5, 8, 36, 38, 47, 51, 52, 59], graph-based methods exhibit superior performance in terms of the time-accuracy trade-off [3, 36, 56, 59, 78, 79]. Prior to querying, these methods all build a graph index on the data objects, but they differ in how they construct the edges in the graph. During querying, they perform greedy search on the graph in iterations. Specifically, the algorithms start with an entry point in the graph. Then in each iteration, they identify the data object nearest to the query that has been visited so far, and then visit all its neighbors. If the vectors of all the neighbors are further away from the query vector than the currently nearest vector, the algorithm terminates. In order to achieve practical accuracy, the algorithm is usually extended to the greedy beam search which tunes a parameter named the beam size b to control the time-accuracy trade-off. Specifically, different from the greedy search, the greedy beam search terminates when the neighbors of the first b nearest objects that have been visited so far all have their distances to the query larger than the distances of the first b nearest objects. In particular, the greedy search is a special case of the greedy beam search where $b = 1$. Clearly, a larger beam size will increase the number of the visited data objects before termination, and thus, would lead to better accuracy with higher time cost.

Most of the state-of-the-art graph-based methods, including HNSW [59], NSG [36], and DiskANN [51] [1], construct their graphs by computing an approximate variant of a so-called RNG [17, 35, 36, 51, 56, 59, 73, 78]. This graph is defined as follows.

Definition 2.1 (RNG [73]). Given a set of vectors $\mathcal{D}$, let (u, v) be a pair of vectors in $\mathcal{D}$ and let $\delta(u, v)$ be the distance between the vectors. RNG is a graph where the set of vertices represent vectors in set $\mathcal{D}$ and where edges are included as follows: an edge from u to v is included in the RNG if and only if $\forall u' \in \mathcal{D}$, u' cannot prune the edge (u, v). In particular, u' can prune (u, v) if and only if u' is closer to u than v is and also closer to v than u is, i.e., $\delta(u, u') < \delta(u, v)$ and $\delta(v, u') < \delta(u, v)$.

The RNG guarantees that the edges of one data object [2] that are retained after pruning have diversified directions, i.e., an edge is pruned if there exists a shorter edge that has similar direction to it in the high-dimensional space. Since computing an RNG graph for n data objects takes $O(n^3)$ time [51], which is prohibitively expensive when n is large, existing methods mostly construct approximate RNG-based graphs [17, 35, 36, 51, 56, 59]. Specifically, when including the edges for a data object, they first search for a few candidates in the database (e.g., approximate nearest neighbors). Then they apply the pruning rule of RNG to the candidates. The candidates that cannot be pruned are retained, and edges are built from the data object to the candidates. To prevent a data object from having too many edges, a maximum out-degree m is imposed. According to a recent benchmark [78], the RNG-based graphs have shown highly promising ANN query performance on many real-world datasets, although we note that the algorithms for candidate generation differ from method to method.

[1]The classical library hnswlib has updated its pruning rules with that of RNG due to its better empirical performance. The pruning rule of DiskANN is generalized from RNG's pruning rule by introducing a new parameter α. When $\alpha = 1$, it is identical to the original RNG's pruning rule.

[2]Without further specification, the edges are all directed in this paper. The edges of a data object refer to its outgoing edges.

2.2 Range-Filtering ANN Query

As already explained in the introduction, RFANN queries, which combine ANN querying on vectors with range querying on numeric attributes, have gained attention in both academia and industry. We focus on the setting where a data object has a vector and one numeric attribute and where a query retrieves the data object with a vector that is nearest to a query vector and a numeric value that is in a query range. This is the range-filtering ANN (RFANN) query [29, 89], which is defined as follows.

Definition 2.2 (Range-filtering ANN (RFANN) query [29, 89]). Formally, a data object is defined as $O = (v, a)$, where v is a vector and a is a numeric attribute value. Let $\mathcal{D} = \{O_1, O_2, ..., O_n\}$ be the set of the data objects. We define an RFANN query as $Q = (q, a_l, a_r)$, where q is a vector, and $[a_l, a_r]$ denotes a range. The RFANN query returns the result $\arg\min_{O \in \mathcal{D}, Q.a_l \leq O.a \leq Q.a_r} \delta(Q.q, O.v)$, where δ is a distance function.

A data object is *in-range* if the value of its numeric attribute is in the query range; otherwise, it is *out-of-range*. Without loss of generality, we assume that the data objects are given in an ascending order with respect to their numeric attribute values, i.e., $\forall i \leq j$, we have $O_i.a \leq O_j.a$. Otherwise, we initially sort the dataset and build a one-to-one mapping between the ranking and the attribute values of the objects. When a query arrives, we can easily find the smallest ranking L and the largest ranking R (using binary search) such that the in-range objects are exactly the objects with indices (rankings) between these two values. Then the RFANN query is reduced to finding $\arg\min_{L \leq i \leq R} \delta(Q.q, O_i.v)$. Going forward, we use $[L, R]$, to denote a query range and i to denote O_i. We note that the number of distinct attribute values of the dataset is upper-bounded by the size of the dataset. To ease the presentation, we first assume that all data objects have distinct attribute values. At the end of Section 3, after presenting our method, we present a more detailed discussion of the cases where duplicate attribute values exist. Thus, after the above operation is performed, the attribute values of the data objects are all mapped to an integer that is no larger than n. The raw query ranges are all mapped to a range of indices $[L, R]$, which is a sub-interval of $[1, n]$. Therefore, in the following sections, we use n to denote both the size of the dataset and the cardinality of the attribute. In addition, we note that because the attribute values are all mapped to their rankings, the distribution of the attribute values does not impact the performance of an algorithm. Related work on queries with other types of attributes and constraints is covered in Section 6.

We proceed to revisit the three basic strategies for processing RFANN queries. **`Pre-filtering`** first filters out the out-of-range objects (e.g., using binary search) and then iterates through the remaining objects to find the nearest neighbor. For unselective queries, this degrades to a linear scan of most of the data. **`Post-filtering`** first conducts ANN search on the dataset (e.g., with graph-based algorithms) and then finds the nearest neighbor among the in-range objects visited. Given a highly selective query, this strategy visits many out-of-range objects, which is suboptimal. **`In-filtering`** integrates the range filtering into the (graph-based) ANN search. Specifically, unlike vanilla graph-based methods that visit *all* neighbors [3] of an object during querying (Section 2.1), the `In-filtering` strategy only visits the *in-range* neighbors. This strategy cannot handle different query ranges with the fixed graph built for ANN search on the whole database. In particular, when query ranges are small, a data object is likely to have only a few, or even no, in-range neighbors. If we then increase the number of edges in the graph-based indexes to solve the issue of short query ranges, then when considering queries with long ranges, a data object may have too many in-range

[3]Without further specification, by "neighbors," we mean neighbors of a node in a graph index, not the nearest neighbors in high-dimensional space.

neighbors. Overall, with the `In-filtering` strategy, the quality of a fixed graph-based index is suboptimal for some query ranges.

Most existing proposals aim to mitigate the issues inherent to the three strategies [29, 62, 75, 81, 89]. In particular, `ADBV` [81] and `Milvus` [75] use a cost model to automatically select the best strategy for a given query. `Milvus` [75] further partitions a dataset into several subsets with consecutive attribute values. During querying, it conducts RFANN search separately on the subsets that intersect with the query range and then merges their results. Another study [29] proposes `SuperPostfiltering` that presets multiple overlapping ranges and builds a graph-based index for each range separately. During querying, it first finds the shortest range that covers the query range. Then it conducts RFANN search on the range with the `Post-filtering` strategy. Although these proposals successfully avoid the distinctively disadvantageous cases of the basic strategies, we note that the inherent issues of the strategies have yet to be fully resolved. For example, for a query range of size s, `SuperPostfiltering` [29] performs `Post-filtering` on a subset of the database where there are at most $4s$ data objects. However, among them, there may still be $3s$ out-of-range objects, which causes suboptimal performance.

To address the inherent issues of the basic strategies, a recent study [89] considers building a dedicated graph-based index for each possible query range. During querying, a method can then find and use the graph-based index corresponding the range of a query. As the graph includes exactly all in-range data objects, RFANN query is reduced to an ANN query. However, as explained already, the number of possible query ranges, each in the format of $[L, R]$ with $1 \leq L \leq R$, is $O(n^2)$. Therefore, compression is applied. However, the lossy compression reduces the quality of the indices, causing suboptimal query performance.

3 METHODOLOGY

3.1 Overview

We follow the idea of performing the search process on a dedicated graph-based index for a given query range as the existing method [89] does. However, while that method considers all $O(n^2)$ possible ranges and materializes compressed graphs for these ranges, we build graphs for only a moderate number of ranges before the querying. We call these graphs *elemental* graphs. Then when a query is processed, the edges in elemental graphs are used as the basic elements to construct a dedicated graph for the query's range. The intuition is that a query range will intersect with some ranges for which elemental graphs exist. The edges in these graphs are leveraged to construct the dedicated graph. Dedicated graphs are constructed *on-the-fly* in that we determine the edges of a data object only when we are to visit its neighbors during the search process. The construction process incurs only a modest overhead compared to the cost of searching the dedicated graph. Overall, the dedicated graph for a query range is not materialized but is constructed on-the-fly with little overhead. This eliminates the huge space cost of storing all dedicated graphs but can still leverage a dedicated graph for any query range in the search process. Two questions still need answers: (1) How to select the ranges for which elemental graphs are materialized? (2) How to construct a dedicated graph for any given query range based on elemental graphs? In particular, the main consideration when answering these questions is the trade-off between a) the number of ranges for which elemental graphs should be built and b) the number of elemental graphs the algorithm needs to access when constructing the out-edges for a vertex on-the-fly for a given query range. The former determines the space complexity of the index, while the latter determines the time complexity of querying. We notice that the classical segment tree has promising properties related to the handling of range queries and strikes a good balance between these two costs, as illustrated next.

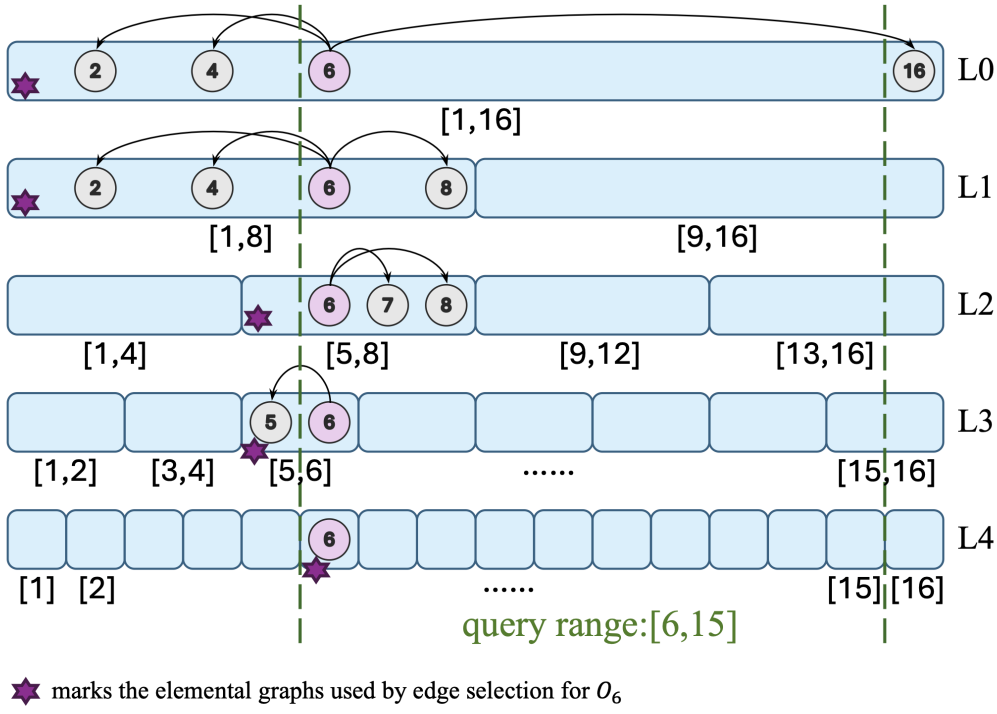


Fig. 1. The iRangeGraph index applied to 16 data objects. It is based on a segment tree with 5 layers, namely L0 to L4. L0 has one segment corresponding to the range [1, 16]. L1 has two segments corresponding to the ranges [1, 8] and [9, 16] respectively, etc. The elemental graphs are materialized for each segment with respective data objects (e.g., an elemental graph based on $O_1, O_2, ..., O_8$ is materialized for segment [1, 8] and the out-going edges of node O_6 are represented by the arrows). In all elemental graphs, the maximum out-degree m of a node is 3 in this example.

For the **first** question, inspired by the classical segment tree [24], we build a multi-layer index structure (see Figure 1). In the ith layer ($i = 0, 1, ..., \log n$), the full range $[1, n]$ (corresponding to the set of all objects $O_1, O_2, ..., O_n$) is partitioned into 2^i disjoint segments (i.e., ranges), each of length $\frac{n}{2^i}$. For each segment, we materialize an RNG-based elemental graph for the corresponding range [4]. The rationale for choosing the segments indexed by the segment tree and build graphs for them is three-fold. (1) The space consumption is moderate. In the multi-layer index, a data object appears only once in each layer, and it has a maximum out-degree of m in a graph of each layer. Thus, the space complexity of storing the graphs for n data objects is $O(nm \log n)$, which is much lower than the prohibitive space consumption of storing graphs for all possible query ranges, i.e., $O(n^3 m)$. (2) These elemental graphs enable an effective and efficient algorithm for constructing a dedicated graph for any given query range during querying (details are given in Section 3.3).(3) The recursive structure of the segment tree enables an efficient algorithm for materializing the graphs in the index phase. It can be shown that the asymptotic time complexity of constructing the entire multi-layer index differs by up to a sub-logarithmic factor from that of constructing a single graph-based index (HNSW) for the dataset. Section 3.2 elaborates the index phase.

[4]To ease the presentation, we assume that n is power of 2.

To answer the **second** question, we design an effective and efficient algorithm for constructing an RNG-based dedicated graph for any given query range on the fly. In particular, we construct the graph by selecting the edges for an in-range data object from the elemental graphs when we need to visit its neighbors during the search process. Each object is covered by a segment in our index at each of the $O(\log n)$ layers, and hence the object is involved in $O(\log n)$ elemental graphs. (1) The algorithm is effective at constructing an RNG-based dedicated graph because an RNG is defined by a pruning rule (Section 2.1), and the relationship between the query range and a certain segment enables determining whether an edge in an elemental graph of the segment should be retained in the dedicated graph for the given query range. For example, when a segment covers the given query range, the edges (connecting two in-range objects) that are retained in the elemental graph of the segment must be retained in the dedicated graph of the query range (since if an edge cannot be pruned from a set of objects, it can not be pruned for a subset of objects, either). (2) Our algorithm is efficient because the index is based on the segment tree whose recursive structure allows us to skip the edge selection in some layers without affecting the dedicated graph. Based on the idea of skipping layers, we design an algorithm for selecting edges for an object, which has amortized time complexity $O(m + \log n)$ - as a comparison, the time complexity is $O(m \log n)$ when enumerating the edges in $O(\log n)$ layers without skipping layers. This is low compared to the cost of visiting the neighbors of an object in the dedicated graph, which is $O(md)$, where d is the dimensionality of the vectors. Section 3.3 presents the details of the algorithm for constructing a dedicated graph and the search strategies in the query phase.

We summarize the resulting method, called `iRangeGraph` and provide an analysis of it in Section 3.4.

3.2 Materializing Elemental Graphs

3.2.1 The Structure of the Index. Our index is based on segment tree [24], which is a powerful data structure for supporting various range-based queries (e.g., range maximum query, range sum query, etc). Specifically, it is a balanced binary tree of $O(\log n)$ layers (see Figure 1). Each node in the tree represents a segment (i.e., a range), which is defined recursively. In particular, the root node is defined to represent the full range of the dataset, i.e., $[1, n]$ (i.e., corresponding to the set of all objects $O_1, O_2, ..., O_n$). Then recursively, for a node whose corresponding segment is $[l, r]$, its left and right children are defined to represent the segments of $[l, mid]$ and $[mid + 1, r]$ respectively where $mid = \lfloor \frac{l+r}{2} \rfloor$. Those nodes whose segments have the length of 1 are defined to be the leaf nodes (i.e., corresponding to a single object).

Based on the recursive structure of the segment tree, we note that a data object appears only once in a single segment in each of the $O(\log n)$ layers. Recall that a graph-based index has the maximum out-degree of m for each data object. Thus, when we build a graph for every segment in the segment tree, the space complexity of storing the index is $O(nm \log n)$, which is moderate compared with that of storing the graphs for all possible query ranges, i.e., $O(n^3 m)$. In other word, it is feasible to materialize all the graphs for the segments in the segment tree in the index phase. These materialized graphs which we call the *elemental graphs*, will be later used in the query phase for constructing a dedicated graph for any given query range (details will be presented in Section 3.3).

For the graph indexes that are built for the segments (i.e., elemental graphs), we follow state-of-the-art graph-based algorithms [36, 51, 59] and adopt approximate RNG-based graphs. Recall that the time complexity of exactly computing an RNG is prohibitive [36, 51]. We follow the convention of most of the existing methods [36, 51, 59] which constructs approximate RNGs by searching for a

few candidates, applying the pruning rule of RNG and cutting off excessive edges (see Section 2.1 for detailed description).

3.2.2 The Algorithm for Materializing the Index. Next, to materialize the index, we note that it can be achieved trivially by running the construction algorithm of any existing graph indexes [36, 51, 59] for every segment independently. The good news is that, due to the recursive structure of the segment tree, the construction can be done in a bottom-up manner, which is more efficient (e.g., it can be proven that the time complexity for constructing the entire index differs by up to a sub-logarithmic factor from that for constructing HNSW on the same dataset, details are given in Section 3.4), as follows.

Let $[l, r]$ be a segment in the tree and $[l, mid]$, $[mid + 1, r]$ be its left and right child segment respectively. Consider the procedure of constructing the approximate RNG of $[l, r]$, for which the graphs for $[l, mid]$ and $[mid + 1, r]$ are available given that we construct the index in a bottom-up manner. Recall that for constructing the edges of a data object u, we first find some candidates (e.g., approximate nearest neighbors) and then apply the pruning rule of RNG to them. Without loss of generality, assume that u is covered by the child segment $[l, mid]$ (the case where u is covered by the other child segment $[mid + 1, r]$ is symmetric and thus omitted). Note that the candidates are from either $[l, mid]$ (the child segment that contains u) or $[mid + 1, r]$ (the child segment that does not contsain u). We discuss these two cases separately.

First, for the candidates that are from the child segment that contains u, i.e., $[l, mid]$, we can infer whether a candidate of object u will be pruned in the RNG of $[l, r]$ based on the RNG of its child segment $[l, mid]$. In particular, we note that a candidate may not be pruned in the RNG of $[l, r]$ only if it is retained in the RNG of the child segment $[l, mid]$. Note that the set of objects of the child segment $[l, mid]$ is a subset of that of the segment $[l, r]$. Thus, if a candidate can be pruned by an object in the subset, it can also be pruned by the same object in the full set. Thus, for generating the candidates that are from $[l, mid]$, it suffices to consider those that cannot be pruned in the RNG of the child segment only (i.e., to copy the neighbors of u in the elemental graph of $[l, mid]$ that have been constructed). It is unnecessary to consider other objects in $[l, mid]$ as the candidates since otherwise they would be pruned. This operation avoids the cost of searching for candidates of u in $[l, mid]$ from scratch. Second, for the candidates that are from the child segment that does not contain u, i.e., $[mid + 1, r]$, we must search for candidates for u as we have no clue which candidates from $[mid + 1, r]$ must be pruned in this case. In particular, in this case, we take the approximate nearest neighbors as the candidates by following the well-known HNSW [59] method.

After the candidates are generated, finally the pruning of RNG is applied to the candidates, and the edges are built accordingly. In summary, in the index phase, we materialize the RNG-based elemental graphs for all the segments in the segment tree bottom-up, which is conducted recursively.

3.3 Constructing and Searching Dedicated Graphs

3.3.1 Constructing Dedicated Graphs. Let $[L, R]$ be a query range given in the query phase. Recall that we target to construct a dedicated RNG-based graph for any given query range on the fly for the RFANN query. Basically, the construction of a graph is to provide the set of edges for every data object $u \in [L, R]$. In our index, note that u is included in a single segment in each of the $O(\log n)$ layers (see Figure 1). As in the elemental graph of each segment, a data object has up to m edges, a data object has in total $O(m \log n)$ edges in the index. To construct the dedicated graph for the query range, the question is how we should select up to m edges out of the $O(m \log n)$ edges in the

elemental graphs. For example, in Figure 1, the object O_6 has 9 edges in total from all layers, and in order to construct the dedicated graph, we need to select among them up to 3 edges for O_6.

The Strategy of Edge Selection. The relationship between a segment and the query range can help to decide whether an edge in the elemental graph should be selected in the dedicated graph for the query range. For example, the segments that contain an object u (in Figure 1, the purple stars mark all the segments that contain data object O_6) all intersect with the query range, and the larger the intersection is, the more strict the pruning would be (meaning that it is more likely that an edge is pruned). Specifically, if an edge (u, v) is retained in the elemental graph of the segment $[l, r]$, this indicates that the edge would not be pruned by any of the objects in the subset $[L, R] \cap [l, r]$ of the query range $[L, R]$. In the extreme case, where $[L, R] \cap [l, r] = [L, R]$, i.e., the segment covers the query range, it is guaranteed that the edge would not be pruned by any of the objects in the query range. In other word, if an edge is retained in the elemental graph that has larger intersection with the query range, it is more robust against pruning by the objects in the given query range. As a result, given that we only select m edges out of the $O(m \log n)$ edges, we give priority to selecting the edges in the segment that has larger intersection with the query range. In our index, they correspond to the edges in the upper layers of the segment tree.

The Efficient Algorithm for Edge Selection. Based on the above strategy of edge selection, for a data object u, we select the edges in a top-down way. Specifically, we start the process from the root node of the segment tree. Then iteratively, we select all the in-range edges (u, v) in the elemental graph of a layer and move to the segment that contains u in the next layer. The edge selection terminates after selecting edges of an elemental graph whose segment is covered by the query range. This is because if an edge can be pruned by an object in this segment, then as the object is also in the query range, the edge is also pruned in the RNG-based graph on the query range by the same object.

Although this algorithm is effective in constructing a dedicated graph for a given query range, its time cost in the query phase is unignorable. Specifically, in the worst-case, the cost of selecting the edges for an object is $O(m \log n)$: it scans m edges of the object at each of the $O(\log n)$ segments covering the object and thus takes $O(m \log n)$ time. For example, for the query range [6, 15] and data object O_6 shown in Figure 1, in order to select up to 3 edges, it scans the edges of O_6 in every layer, i.e., it scans 9 edges at 4 layers in total. Recall that our strategy is to prioritize the edge selection in the segment that has the larger intersection with the query range. In particular, when a segment has exactly the same intersection with the query range as its child segment does, the edges in their elemental graphs have exactly the same robustness against pruning by the objects in the query range. In this case, we can skip the edge selection at this layer and directly move to the next layer. For example, in Figure 1, to select the edges for O_6, the segments in L1 and L2 that contain O_6 have exactly the same intersection with the query range, and thus we can skip L1 to select edges from L2.

We present our edge selection algorithm based on the strategy of skipping layers in Algorithm 1. We note that it is provable that the amortized time complexity of the algorithm is $O(m + \log n)$, which is much smaller than the time complexity of an edge selection algorithm that does not skip layers (i.e., $O(m \log n)$). In addition, the time complexity of our edge selection algorithm is negligible compared with the time cost of visiting the m neighbors, which is $O(md)$, where d is the dimensionality (see details in Section 3.4).

3.3.2 Searching on Dedicated Graphs. For a given RFANN query, we search on the dedicated graph, which we construct on the fly for the query range, as we do on existing graph-based indexes for the ANN query. Specifically, we employ greedy beam search on the dedicated graph, where the edges for a data object are constructed on the fly when we access its neighbors during the search process.

Algorithm 1 Edge Selection

Input: u: a data object; $[L, R]$: the query range; m: the maximum out-degree; $\mathcal{N}_{lay,u}$: the set of neighbors of u at layer lay
Output: a set of neighbors selected for u

$l \leftarrow 1, r \leftarrow n, lay \leftarrow 0, \mathcal{S} \leftarrow \emptyset$
while $|\mathcal{S}| < m$ **do**
  $[l_c, r_c] \leftarrow$ the child segment of $[l, r]$ which contains u
  if $[l_c, r_c] \cap [L, R] = [l, r] \cap [L, R]$ **then**
    $l \leftarrow l_c, r \leftarrow r_c, lay \leftarrow lay + 1$
  else
    $\mathcal{S} \leftarrow \mathcal{S} \cup (\mathcal{N}_{lay,u} \cap [L, R])$
    Cut off $\mathcal{S}$ to the size of m
    if $[l, r] \subseteq [L, R]$ **then**
      break
    $l \leftarrow l_c, r \leftarrow r_c, lay \leftarrow lay + 1$
return $\mathcal{S}$

3.4 Summary and Discussion

In summary, the proposed method initially builds a multi-layer index based on the segment tree, where for each segment, we materialize an approximate RNG-based graphs, called an elemental graph. During query processing, the method leverages the elemental graphs to construct a dedicated graph for the query being processed on the fly (Algorithm 1) and conduct an ANN query on it.

Analysis. First, **the space complexity** of our method is of $O(nm \log n)$ which is moderate compared with that of storing the dedicated graphs for all possible query ranges, i.e., $O(n^3 m)$. Next, **the time complexity of the index phase** is a somewhat controversial topic. This is because the construction algorithm involves nearest neighbor search. It can be proven that no algorithm can be guaranteed to achieve sub-linear time complexity for nearest neighbor search [48, 69] (including graph-based methods [49]). However, due to the promising performance of the graph-based methods in practice on real-world datasets, it is often the case that these methods have sub-linear performance on real-world datasets [36, 51, 59, 78]. Thus, instead of presenting an asymptotic time complexity for the construction algorithm, we argue that the time complexity of constructing the entire multi-layer index differs by up to a sub-logarithmic factor from that of constructing a single graph-based index (e.g., HNSW) for the same dataset. The conclusion holds regardless of the above issue (see Theorem 3.1 below). Empirically, the time cost for constructing the entire index of our method is no more than 3x of that for constructing HNSW over the set of all objects (note that HNSW only supports queries without range constraints). The detailed proof and empirical verification are in the technical report [83] due to the page limitation.

Theorem 3.1. *The time complexity of the algorithm for materializing the entire index of our method differs by up to a sub-logarithmic factor from that for constructing HNSW on the set of all objects.*

As for **the time complexity of querying**, due to the same issue as above, it is not informative to compare two graph-based methods based on their big-O time complexity (because for all methods, the time complexity is $O(n)$). We note that for the RFANN query, our method is superior because (1) it constructs a dedicated RNG-based graph for any given query range in the query phase, and (2) the time cost of doing so is low (see Theorem 3.2 below, the detailed proof is given in the technical

report [83]). We refer to the empirical study in Section 5.2.1 that provides evidence that our method is capable of state-of-the-art performance.

Theorem 3.2. *The amortized time complexity of the algorithm (Algorithm 1) for constructing the edges for a data object in the dedicated graph on the fly is* $O(m + \log n)$.

Limitation. Nevertheless, we note that the dedicated graph constructed by our algorithm might not approximate an RNG perfectly based on all in-range objects for a query range, i.e., the graph constructed using elemental graphs is not necessarily identical to the corresponding dedicated graph built from scratch. The space complexity of storing the graphs that are explicitly materialized on the in-range objects for all possible query ranges is $O(n^3 m)$, while the space complexity of our method is only $O(nm \log n)$. We observe that the elemental graphs may miss some edges that should have been retained in the approximate RNG of the query range. Specifically, it is possible that an edge that should be retained in the RNG of the query range is pruned in the elemental graph of a segment. This is because besides the in-range objects, a segment also involves some out-of-ranges objects, which may accidentally prune the edges. Despite this limitation, our method achieves search performance that is very close to the ideal search performance, as verified empirically in Section 5.2.4. Specifically, in Section 5.2.4, we explicitly materialize an HNSW for each query range in a query workload. These HNSWs are called Oracle-HNSW because materializing them for all possible query ranges is impractical. We measure the search performance of our method and the Oracle-HNSW, finding that the queries per second of the impractical Oracle-HNSW is no more than 2x that of ours when reaching 0.9 recall. At the same time, the space complexity of our method is $O(nm \log n)$, which is much smaller than that of the Oracle-HNSW, which is $O(n^3 m)$.

Comparison with `SuperPostfiltering` [29]. We notice that a concurrent study [29], which was recently posted on arXiv, proposes a method called `SuperPostfiltering` for the RFANN query with a method also inspired by the segment tree. Unlike our method that aims to build a dedicated graph for any given query range on the fly, `SuperPostfiltering` [29] builds graph-based indexes for multiple ranges in the index phase. In the query phase, it then adopts the corresponding graph of the smallest range that covers the query range for conducting `Post-filtering` RFANN search. As a result, this method still suffers from the inherent issues of `Post-filtering` (see Section 2.2), i.e., it may visit many out-of-range objects for finding the nearest in-range object. Furthermore, the concurrent study [29] only applies the segment tree for determining the ranges to build graphs for. In our method, the segment tree is applied in a more integrated way, i.e., the recursive structure of the segment tree helps both the efficient construction of our index structure in the index phase (Section 3.2) and the effective and efficient strategy of edge selection for constructing dedicated graphs on the fly in the query phase (Section 3.3). According to the empirical study in Section 5.2.1, our method is in general superior to `SuperPostfiltering` in terms of search performance, memory footprint, and indexing time.

Impact of Duplicate Attribute values and Cardinality of Attribute. So far, we have assumed that all data objects have distinct attribute values. However, it is easy to extend the algorithm to handle duplicate attribute values. Specifically, we can simply put the data objects with identical attribute values into the same node in every layer of the segment tree. All other operations in the algorithm remain unchanged. Furthermore, the RFANN query becomes easier when duplicate attribute values exists. Specifically, when the number c of distinct attribute values, i.e., the cardinality of the attribute, is significantly smaller than n, the space complexity of our index becomes $O(nm \log c)$, and the time complexity in Theorem 3.2 becomes $O(m + \log c)$. On the other hand, we note that even when c is small, the cost of storing a graph-based index for every possible query range remains impractical. This is because the space complexity of storing indexes for all possible

query ranges is $O(nmc^2)$. For example, when $c = 100$, 5,050 indexes must be stored, which incurs impractical costs.

4 MULTI-ATTRIBUTE RANGE-FILTERING ANN SEARCH

We have developed an algorithm for the RFANN that involves a predicate on a single numeric attribute. In real-world scenarios, queries that involve conjunctive predicates [5] on multiple numeric attributes are also common [62, 75, 86]. For example, in a passage retrieval system, clients could set constraints on publish time and passage length in order to find desired results. In particular, to support multi-attribute queries, during the index phase, we can build our index with respect to one of the numeric attributes, e.g., A_1. Then, during the query phase, we can construct a dedicated graph-based index for the query range on A_1. Then for a multi-attribute query, we can conduct the graph-based ANN search with the dedicated graph built based on A_1 and adopt the `In-filtering` or `Post-filtering` strategy for handling the predicates on other attributes (see Section 2.2).

While with this simple extension, our method can support multi-attribute queries, we note that due to the inherent issues of the basic strategies, the extended algorithm may suffer from degenerated performance for some workloads. For example, as discussed in Section 2.2, the `In-filtering` strategy may have the issue that a data object has few (or no) in-range neighbors. On the other hand, the `Post-filtering` strategy may have the issue that it may visit a large number of out-of-range objects to find an in-range object. To mitigate these issues, we propose a simple idea that generalizes the `In-filtering` and `Post-filtering` strategies. Note that the data objects in the dedicated graph-based index are all in-range in terms of A_1. Next, by "in-range" and "out-of-range", we refer to the "in-range" and "out-of-range" objects w.r.t. the attributes other than A_1. Specifically, when conducting graph-based ANN search, unlike the `In-filtering` (resp. `Post-filtering`) strategy that visits none (resp. all) of the out-of-range neighbors, we visit the out-of-range neighbors with probability of p where $0 \leq p \leq 1$. When $p = 0$, the method is exactly equivalent to the `In-filtering` strategy, i.e., it does not visit the out-of-range objects. When $p = 1$, the method is equivalent to the `Post-filtering` strategy, i.e., it visits all the neighbors regardless of their attribute values during the graph-based ANN search and finds the nearest neighbors from only the in-range objects that are visited. Thus, when $0 < p < 1$, it corresponds to an intermediate strategy in-between `In-filtering` and `Post-filtering`, which may mitigate the disadvantages of the two extreme strategies. Our experimental results in Section 5.2.5 indicate that this simple technique is capable of a 70% speed-up at 0.9 recall.

5 EXPERIMENTAL STUDY

5.1 Experimental Setup

Our experiments involve four studies. (1) We compare our method with the existing methods for the single-attribute RFANN query (Section 5.2.1). (2) We verify the effectiveness of the proposed techniques via ablation studies (Section 5.2.2) and the scalability study (Section 5.2.3). (3) Recall that the target of our method is to build a dedicated graph for any given query range. We measure the performance gap between our method and the dedicated graph which is explicitly materialized for the given query ranges (Section 5.2.4). (4) We evaluate the extension of our method on the multi-attribute RFANN query (Section 5.2.5). In all of the experiments, the RFANN query targets to find the first 10 nearest neighbors that satisfy the predicates.

[5] It refers to the queries that have multiple constraints on multiple attributes. A data object is said to satisfy the conjunctive predicates if and only if it satisfies all the constraints.

Datasets. For evaluating the performance of the methods, We adopt five real-world public datasets including WIT-Image [6] (WIT in short), TripClick [7], Redcaps [8], YouTube-Rgb [9] (YT-Rgb in short) and YouTube-Audio [10] (YT-Audio in short). These datasets have been used in existing studies on RFANN [29, 65, 89]. For each dataset, one million objects which involve both real-world vectors and numeric attributes are extracted to be the data objects. Another 1,000 vectors are extracted to be the query vectors. For these query vectors, we would assign different query ranges to them in order to test the performance of the methods for different workloads. The details will be specified in the next paragraph. The properties of the datasets are summarized in Table 1. Note that YT-Rgb and YT-Audio involve two attributes. We use the first attribute (i.e., # of likes for YT-Rgb and publish time for YT-Audio) for the evaluation of the single-attribute RFANN query and use both for the evaluation of the multi-attribute RFANN query. The detailed descriptions of the datasets are left in the technical report [83] due to the page limit.

Table 1. Datasets. All the datasets involve one million data objects and one thousand queries.

	Vector Type	Dim.	Attribute Type
WIT	image	2,048	image size
TripClick	text	768	publication date
Redcaps	multi-modality	512	timestamp
YT-Rgb	video	1,024	# of likes, # of comments
YT-Audio	audio	128	publish time, # of views

Query Ranges. We follow several studies [62, 75, 89] by adopting datasets that feature real-world attributes and evaluating query performance across varying query workloads. We note that although it is mentioned in many industrial studies that the RFANN query is an important feature in real-world systems (e.g., Apple [62], Milvus [75], Alibaba [81]). However, possibly due to privacy issues, no publicly available datasets with range-filters have been provided from industry, to the best of our knowledge. Instead, we evaluate the search performance of the methods under different query ranges by following the existing study [62]. In particular, we say a query has the *range fraction* of 2^{-i} if its query range covers $n/2^i$ of the data objects. Based on the range fractions, the queries can be divided into three scales including large ($i \in [0, 3]$), moderate ($i \in [4, 6]$), and small ($i \in [7, 9]$). As for even smaller query ranges, we note that these cases are inherently simple for the RFANN query. They can be handled efficiently with the simple `Pre-filtering` strategy. Thus, we exclude them from the evaluation. We evaluate the methods in two types of workload including (1) the workload with a fixed range fraction and (2) the workload with mixed range fractions. For the fixed workload, when the length of the query range (i.e., the range fraction) is fixed, the specific locations of the query ranges are generated randomly for the query vectors. For the mixed workload, we first randomly partition the query vectors into 10 subsets. For the ith subset ($i \in [0, 9]$), we assign a query range with the range fraction of 2^{-i} to the queries and generate the specific query ranges randomly.

Performance Metrics. Following existing benchmarks [3, 78], we primarily measure the efficiency by qps, i.e., the number of queries responded per second, and we measure the accuracy by $recall = \frac{|G \cap S|}{K}$, where G is the groundtruth of the K nearest neighbors, and S is the results produced by an

[6] https://github.com/google-research-datasets/wit
[7] https://tripdatabase.github.io/tripclick/
[8] https://redcaps.xyz/
[9] https://research.google.com/youtube8m/download.html
[10] https://research.google.com/youtube8m/download.html

algorithm. In addition to qps and recall, we note that there are other metrics which provide valuable insight into the evaluation of ANN algorithms [66], e.g., the number of distance computations and average distance ratio. For brevity, we include experimental results regarding these metrics in the technical report [83].

Methods and Parameters. We study 6 methods, including our method and 5 existing methods as follows. For the methods which provide default parameters, we adopt the default parameters; for others we use grid search to find the optimal parameters for building the indexes. We vary the parameter named the beam size for all the graph-based methods (i.e., except `Pre-filtering`) to control the qps-recall trade-off in the query phase. (1) `iRangeGraph` is our method. For index construction, there are two parameters, maximum out-degree m, and the number of candidates generated for constructing the graphs EF (see Section 3.2). m is set to 16 for TripClick, YT-Audio and 64 for WIT, Redcaps and YT-Rgb. EF is set to 100 for WIT, TripClick and YT-Audio and 400 for Redcaps and YT-Rgb. (2) `2DSegmentGraph` [89] builds compressed dedicated graphs for all possible query ranges. In particular, it provides three versions of index construction named MaxLeap, MidLeap and MinLeap. As reported in the paper, the MaxLeap version shows optimal search performance with significantly smaller index time and space. Therefore, we evaluate `2DSegmentGraph` with MaxLeap. We follow the parameter setting in [89] for WIT and YT-Audio, i.e., $M = 8$, $K = 100$ for YT-Audio, and $M = 64$, $K = 100$ for WIT, where M denotes the maximum out-degree and K is a parameter which controls the index construction. We set $K = 100$, $M = 32$ for all the rest datasets based on grid search. (3) `Filtered-DiskANN` [41] proposes two methods for index construction, namely `FilteredVamana` and `StitchedVamana`. We note that they are originally designed for label filtering instead of range filtering. Following the prior work [89], to adapt them to RFANN query, we evenly divide the full range $[1, n]$ into 10 consecutive buckets and assign each bucket a label. The buckets which have overlap with the query range are used as the query labels. We use the default parameters [61], i.e., $R = 64$, $L = 100$ for both `FilteredVamana` and `StitchedVamana`, and set $SR = 64$ for `StitchedVamana`. (4) `Milvus` [75] is a vector database system that supports range filtering. We choose HNSW as its index. With grid search, we fix $EF = 400$ for all datasets, and set $M = 16$ for YT-Audio, $M = 32$ for Redcaps, TripClick and YT-Rgb, and $M = 64$ for WIT. (5) `SuperPostFiltering` [29] is a method based on post-filtering. The detailed discussion about the method can be found in Section 2.2 and Section 3.4. We use its recommended parameters, i.e., $\beta = 2, EF = 500, m = 64$ for all datasets. (6) `Pre-filtering` conducts a linear scan with objects satisfying the range constraint. Specifically, it first applies binary search to eliminate the out-of-range objects. Then, among the remaining objects, it computes the distances between their vectors and the query vector to find the NN. The time cost of the first step is almost negligible, while that of the second step is high as it computes many distances between high-dimensional vectors. No parameter tuning is needed. `VBASE` [86] is excluded from the comparison because it has been reported that it, as a system for generic attribute-filtering ANN queries has suboptimal search performance for the RFANN query [29].

Platform. All experiments are conducted on a server with Intel(R) Xeon(R) Gold 6418H CPU@4GHz and 1TB of RAM under Ubuntu 22.04.4 LTS. All methods are implemented in C++ and are compiled using GCC 11.4.0 with `-O3 -march=native`. The search performance is evaluated using a single thread, and the indexing time is measured using 32 threads. The source code is available at https://github.com/YuexuanXu7/iRangeGraph.

5.2 Experimental Results

5.2.1 The Results of the Single-Attribute RFANN Query. This section compares our `iRangeGraph` method with the baseline approaches for single-attribute RFANN query.

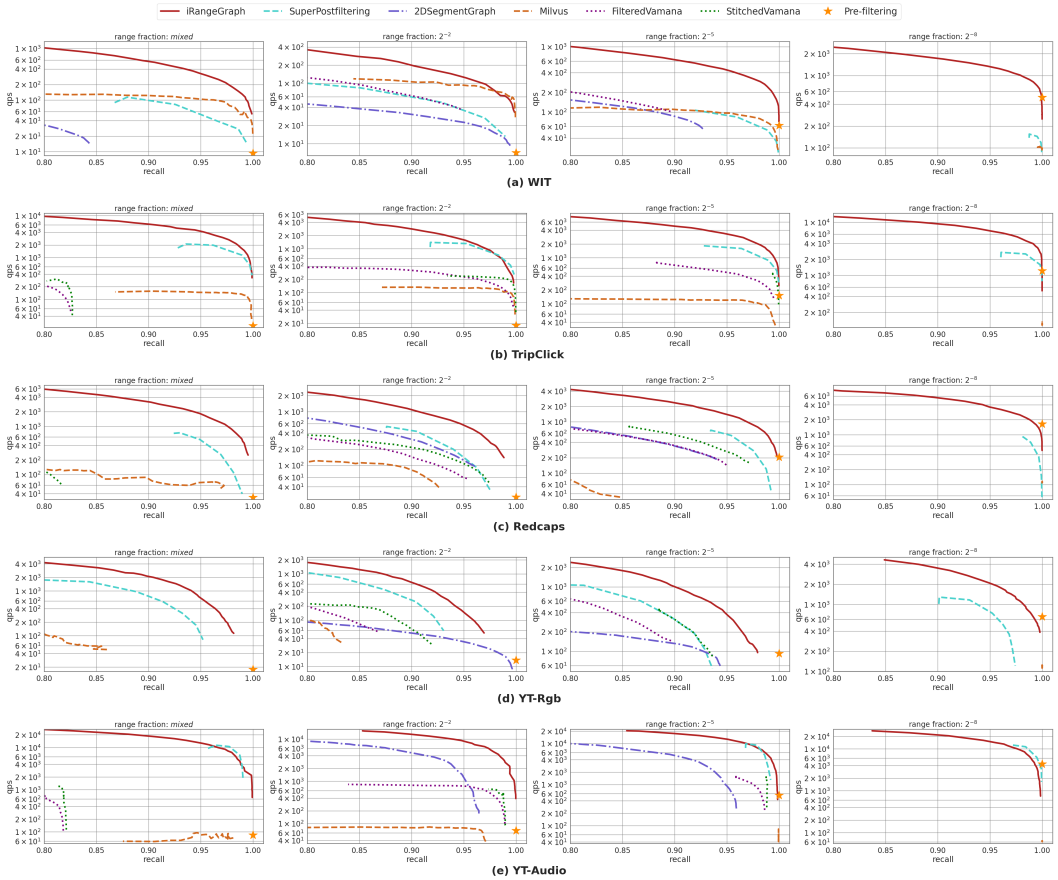


Fig. 2. Comparison of all methods on the single-attribute RFANN query with different datasets and query workloads of mixed, large, moderate and small range fractions. The curve of a method is missing for a certain dataset and query workload indicates that it fails to achieve at least 0.8 recall.

The Search Performance for Single-Attribute RFANN Query. We first evaluate the search performance of all methods on RFANN queries. In particular, the search performance is evaluated under four different query workloads which correspond to the query ranges with different range fractions, including 2^{-2} (large scale), 2^{-5} (moderate scale), 2^{-8} (small scale) and the mixed. Figure 2 plots the qps-recall curves (upper-right is better) by varying the parameter named beam size for the graph-based methods (i.e., the methods except for `Pre-filtering`). We have the following observations. (1) Only our `iRangeGraph` method, `Pre-filtering`, `SuperPostFiltering` and `Milvus` could stably achieve reasonable recall in almost all datasets under the query workload of all the fixed and mixed range fractions. Other methods fail to achieve over 0.8 recall in the workload of the mixed range fractions in most of the datasets. Note that in real-world scenarios, it is likely that a query workload involves query ranges with varying lengths. (2) `2DSegmentGraph`, `FilteredVamana` and `StitchedVamana` can only handle the query workload of large and moderate range fractions while failing to achieve 0.8 recall on the query workload of small and mixed range fractions. `2DSegmentGraph` may also fail in large and moderate range fractions on some datasets (e.g., on TripClick). Note that the phenomenon that `2DSegmentGraph` cannot produce reasonable

recall for small query ranges is consistent with the results in its paper [89]. The reduced query performance of 2DSegmentGraph may be, to some extent, caused by its aggressive compression of dedicated graphs. In particular, based on the analysis of its compression strategy [89], the performance is only guaranteed on the queries with *half-bounded* query ranges (i.e., $L = 1$ or $R = n$). For other general ranges, the compression strategy is likely to cause reduced performance. (3) In general, iRangeGraph shows clear superiority in the search performance over all baseline methods. Compared with the baselines, iRangeGraph achieves the best qps-recall trade-off in almost all datasets and query workloads. In particular, iRangeGraph surpasses the most competitive baseline SuperPostfiltering by 2x-5x in qps at 0.9 recall on most of the datasets (including WIT, TripClick, Redcaps, and YT-Rgb), and achieves even more improvement compared to other baselines, (e.g., 2x-300x speed-up over Milvus). On YT-Audio, it has comparable search performance with the most competitive baseline SuperPostfiltering. This might be due to the dimensionality of the dataset, i.e., it has 128 dimensions only. In particular, when the dimensionality is small, the overhead of constructing the dedicated graph on the fly could somewhat affect the search performance, although the overhead has been reduced from $O(m \log n)$ to $O(m + \log n)$. (4) For small range fractions, Pre-Filtering has the optimal efficiency if the target recall is 1. However, we note that our method still provides a better time-accuracy trade-off when the requirement on the recall is relaxed. For example, iRangeGraph achieves 3x-110x speedup over Pre-filtering at 0.9 recall among different datasets. Additional experimental results and analyses related to the single-attribute RFANN query are placed in the technical report [83] due to the page limit.

The Memory Footprint and Indexing Time. We then measure the memory footprint and indexing time of all methods. We report the memory footprint of all the methods for RFANN in Table 2. We also include the sizes of the raw vectors for reference. The overall memory footprint of a method minus that of the raw vectors equals to the size of its index. The indexing time is reported in Table 3 [11]. Note that only our method, SuperPostfiltering, Milvus, and Pre-filtering can stably produce reasonable recall for most of the datasets and query workloads. Thus, we focus the comparisons on these methods only. We note that the memory footprint of our method is smaller than that of SuperPostfiltering and larger than those of Pre-filtering and Milvus. Recall that our method outperforms Milvus and Pre-filtering by orders of magnitudes in terms of query efficiency across many datasets and query workloads. Thus, compared with these methods, our method strikes a good trade-off between the search performance and the (time and space) costs of the index. On the other hand, as is reflected in Table 2, it is not guaranteed that iRangeGraph has smaller memory footprint than 2DSegmentGraph although iRangeGraph has the lower space complexity of $O(nm \log n)$. This is because 2DSegmentGraph applies an aggressive compression strategy (MaxLeap) that produces an index whose empirical size is significantly smaller than $O(n^3 m)$. However, due to the compression, 2DSegmentGraph cannot enable reasonable query performance on many datasets and query workloads. In contrast, our method achieves superior query performance with a stable and moderate memory footprint.

Based on the experimental results above, we proceed to characterize the settings in which deployment of iRangeGraph is appropriate. Overall, iRangeGraph is suited in setting where the query workload includes mixed query ranges and the dimensionality is high. In particular, iRangeGraph offers competitive query performance at the cost of moderate (but unignorable) memory consumption. Its memory overhead is caused by its storage of the edges of elemental graphs. When the dimensionality is high, the memory cost is dominated by the cost of storing the vectors. Thus, this overhead is relatively low. For example, the WIT dataset has 2,048 dimensions, and the memory

[11] For all methods except for 2DSegmentGraph, 32 threads are used; for 2DSegmentGraph, a single thread is used since its multi-thread implementation is not available.

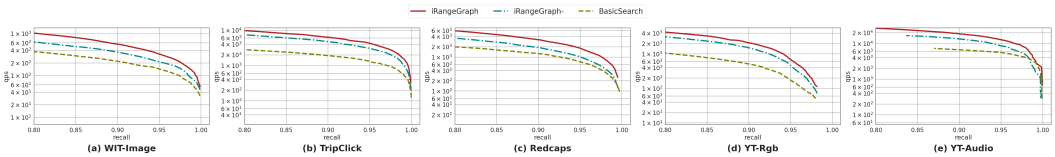


Fig. 3. The ablation study of our core algorithm (constructing the dedicated graph on the fly) and edge selection algorithm 1.

consumption of our method is 1.6x that of the raw vectors. As for the query performance, unless the query ranges are consistently small (in this case, the RFANN query is intrinsically simple and `Pre-filtering` is the optimal solution), `iRangeGraph` shows competitive search performance. We note that the methods designed for label filtering do not quite suit the RFANN query. Recall that these methods partition the dataset into several consecutive buckets and assign each bucket a label. The problem is that it is hard to decide the length of the bucket for different query ranges. For example, when the query range is much larger than the bucket length, it is necessary to conduct separate ANN queries on many buckets, which causes reduced performance.

Table 2. Memory footprint (GB).

	WIT	TripClick	Redcaps	YT-Rgb	YT-Audio
Raw Vectors	7.7	2.9	2.0	3.9	0.47
iRangeGraph	12.87	4.34	7.14	9.05	1.95
SuperPostfiltering	28.98	14.67	11.8	17.53	7.50
Milvus	8.38	3.23	2.21	4.28	0.66
Pre-filtering	7.7	2.9	2.0	3.9	0.47
2DSegmentGraph	9.35	4.23	3.19	5.22	1.61
FilteredVamana	8.17	3.29	2.31	4.18	0.84
StitchedVamana	8.15	3.22	2.21	4.06	0.72

Table 3. Indexing time (s).

	WIT	TripClick	Redcaps	YT-Rgb	YT-Audio
iRangeGraph	3,776	621	1,851	4,719	236
SuperPostfiltering	11,603	5,140	1,765	4,735	1,206
Milvus	784	476	174	593	49
Pre-filtering	<10	<10	<10	<10	<10
2DSegmentGraph	16,865	3,165	1,412	3,038	381
FilteredVamana	358	144	80	110	35
StitchedVamana	450	54	41	85	16

5.2.2 The Results of the Ablation Study. In this section, we investigate the effect of the components in our search algorithm on the search performance. The qps-recall curves are presented in Figure 3 with the query workloads of mixed range across all datasets. The ablation involves two folds.

In order to evaluate the effectiveness of our core algorithm for constructing a dedicated graph for any query range for RFANN, we compare it with a trivial baseline, which we call `BasicSearch`. With the structure of segment tree, any query range can be expressed as the union of $O(\log n)$ non-overlapping segments in the tree. For example in Figure 1, the query range $[6, 15]$ can be exactly expressed by the union of 5 disjoint segments, i.e., segment $[9, 12]$ at L2, segments $[7, 8]$, $[13, 14]$ at L3, and segments $[6]$, $[15]$ at L4. The `BasicSearch` baseline independently conducts ANN search on the elemental graphs of the $O(\log n)$ segments and finally merges the results. We note that it

is the way the segment tree is used for handling other types of range-based queries, e.g., range maximum query and range sum query [24]. Figure 3 shows that our method outperforms this trivial baseline by 2x to 4x in efficiency on all workloads at 0.9 recall.

We next evaluate the effectiveness of the efficient algorithm for edge selection. Note that the trivial algorithm of edge selection without skipping some layers has the time complexity of $O(m \log n)$ while the efficient algorithm has the amortized time complexity of $O(m + \log n)$ (Section 3.3). The former introduces unignorable overhead in the search algorithm. According to Figure 3, we observe that the search performance based on the efficient algorithm (`iRangeGraph`, the red curve) achieves consistent improvement over the trivial algorithm (`iRangeGraph-`, the blue curve), which verifies the effectiveness of the efficient algorithm for edge selection.

5.2.3 Scalability Results. We evaluate the scalability of our method with a larger public dataset DEEP [12]. The results show that as the dataset scales, the indexing cost remains moderate and the search performance remains promising. For more details, we refer to our technical report [83].

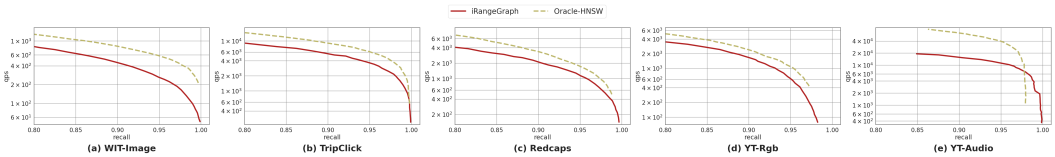


Fig. 4. Comparison between `iRangeGraph` and `Oracle-HNSW` under mixed range fraction.

5.2.4 Comparison with `Oracle-HNSW`. Here, we evaluate the performance gap between our dedicated graph and the graph that is explicitly materialized for the given query ranges. Specifically, in this experiment, we explicitly build an HNSW graph for every given query range and measure its search performance. We call this method `Oracle-HNSW`. Note that `Oracle-HNSW` is not practical since it has space complexity $O(n^3 m)$. This experiment is only intended to quantify the performance gap between our method and the one that materializes all possible dedicated graphs for RFANN queries (in an impractical way). In practice, query ranges are not known during the index phase, so it is necessary to materialize the HNSWs for all possible query ranges. Thus, the memory consumption of Oracle-HNSW in this experiment does not reflect its memory consumption in practice. We use a mixed query workload in this study. Specifically, we randomly generate a specific query range for each of the subset of 100 query vectors (but not a query range for a query as we did in Section 5.2.1); since otherwise, we would have to build 1,000 HNSWs for the 1,000 distinct query ranges, which renders the study infeasible. According to Figure 4, on most of the datasets (except for YT-Audio), our practical algorithm has its search performance very close to `Oracle-HNSW`, e.g., `Oracle-HNSW` only outperforms our method by less than 2x in efficiency when reaching 0.9 recall. On YT-Audio, `Oracle-HNSW` outperforms our method for many recall values, but not at very high recalls. This might be explained by the fact that both our method and HNSW involve heuristic approximation of the RNG graph, and it is possible that a method does not produce stable performance across different datasets, as has been observed in a recent benchmark study [78].

5.2.5 The Results of the Multi-Attribute RFANN Query. This section assesses the effectiveness of the extension of `iRangeGraph` (Section 4) to the RFANN query with conjunctive predicates on multiple attributes. Recall that the query aims to find the nearest neighbor that satisfies *all* the constraints on the attributes. We note that it is likely that only a few data objects satisfy the conjunctive predicates (i.e., the selectivity of the conjunctive predicates is high), especially

[12]https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search

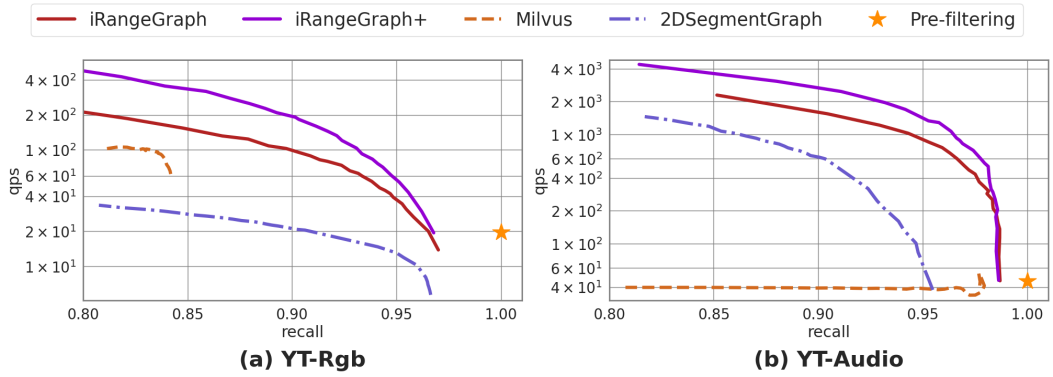


Fig. 5. Multi-attribute RFANN query performance.

when the conjunctive predicates involve many attributes. Note that this is an inherently simple workload for the query as it can be efficiently handled by the `Pre-filtering` strategy. Therefore, we evaluate our method on non-trivial query workloads, where the selectivity of the conjunctive predicates is moderate. Specifically, note that each data object in the YT-Rgb and YT-Audio dataset involve two numeric attributes. For each query vector, we randomly assign a query range with the expected range fraction of 2^{-2} to each of the attributes so that for the conjunctive predicates, the selectivity is moderate. As for the baseline methods, we note that `SuperPostfiltering` and `Filtered-DiskANN` (including `FilteredVamana` and `StitchedVamana`) are not included in this experiment because they do not support the query with multiple range constraints. Considering the baseline methods `2DSegmentGraph`, `Milvus`, and `Pre-filtering`, we note that they either provide extensions to the multi-attribute query (`2DSegmentGraph` [89] and `Milvus` [75]) or can be extended to the query trivially (`Pre-filtering`). For multi-attribute queries, `iRangeGraph` and `2DSegmentGraph` build indexes based on one of the attributes. For both methods, we build indexes based on the first attribute of the dataset (i.e., # of likes for YT-Rgb and the publish time for YT-Audio) and adopt `Post-filtering` for handling the second attribute (according to our experiments, using `In-filtering` in this case cannot achieve >0.8 recall). Besides, recall that we have proposed a simple idea (in Section 4) that generalizes `In-filtering` and `Post-filtering` by allowing the search algorithm to visit the out-of-range neighbors with probability p. In order to prevent the algorithm from visiting too many out-of-range objects, p is in practice set to $\exp(-t)$, where t is the number of consecutive out-of-range objects that have been visited in the search path. We evaluate the effectiveness of this technique by equipping `iRangeGraph` with it, denoted as `iRangeGraph+`.

As Figure 5 indicates, (1) the extension of `iRangeGraph` can effectively cope with multi-attribute RFANN query workloads with moderate selectivity. It outperforms `2DSegmentGraph` by more than 2x speed-ups at 0.9 recall on both datasets. Further, `iRangeGraph` also outperforms `Milvus` with a much better qps-recall trade-off. (2) We note that the extension of `iRangeGraph` is not perfect for the multi-attribute RFANN query. It is shown that unlike `Pre-filtering`, it cannot achieve 1.0 recall on these datasets. However, `iRangeGraph` still provides a better qps-recall trade-off if the requirement on recall is relaxed, e.g., `iRangeGraph` offers 5x higher qps at 0.9 recall on YT-Rgb, and 35x higher qps at 0.9 recall on YT-Audio than `Pre-filtering`. (3) The idea of visiting out-of-range neighbors with probability p helps enhance the search performance. For example, on both datasets, `iRangeGraph+` improves the qps by more than 70% for `iRangeGraph` at 0.9 recall.

6 RELATED WORK

Approximate Nearest Neighbor Search. The existing methods for ANN query in high-dimensional Euclidean space can be generally divided into four categories: the graph-based, the quantization-based, the hashing-based and the tree-based. We refer readers to recent tutorials [28, 68] and benchmarks/surveys [3, 4, 26, 56, 78, 80] for a comprehensive review. Graph-based methods [17, 18, 34–36, 45, 50, 51, 56, 58, 59, 73, 78] use a graph to connect data vectors and a heuristic search algorithm, usually greedy beam search, on the graph to find vectors that are close to query vector. Among the graph-based methods, HNSW [59], NSG [36] and DiskANN [51], which are all approximate RNG-based graphs [78], have been widely used in industry [27, 36, 51, 75]. In our paper, we also build approximate RNG-based graphs as the indexes. Quantization-based methods [1, 5, 6, 39, 40, 43, 52] accelerate searching by reducing the cost of distance computation and memory usage. Hashing-based methods [23, 37, 47, 48, 71, 72] offer a theoretical guarantee on the probability of finding approximate nearest neighbors. Tree-based methods [2, 8, 42] are powerful in searching the nearest neighbor in low-dimensional space, but they suffer from the curse of dimensionality.

Attribute-filtering ANN Search. Various algorithms and systems are developed for attribute-filtering ANN query [29, 41, 44, 55, 60, 62, 65, 75, 77, 81, 82, 87, 89]. Besides the range-filtering ANN query [29, 89], there are also many studies targeting the ANN query with different attributes and predicates [41, 44, 60, 62, 65, 77, 82, 87]. For example, Filtered-DiskANN [41] is developed for predicates on categorical attributes [41, 77, 82, 87]. Some systems and algorithms (e.g., Milvus [75]) are developed to support predicates on generic attributes [62, 65, 75, 86]. Because different types of attributes involve highly diversified properties, for each type of attribute, it often entails specialized design in order to achieve the most competitive search performance on their corresponding attribute-filtering ANN queries. Among those which support predicates on categorical attributes, we adapt the state-of-the-art method Filtered-DiskANN [41] as the baseline in our experiments. Besides, the prevalent system Milvus [75] that supports predicates on generic attributes is also included in our experiments. The experimental results (Section 5.2.1) show that they indeed incur suboptimal performance on the range-filtering ANN query compared with the specialized methods. We note that HQI [62] is a method which further optimizes the generic attribute-filtering ANN query based on prior knowledge on the query workloads and conducts batched query execution offline, whose problem setting is different from ours. There are also studies which research on the attribute-filtering ANN query for multi-dimensional vectors [31]. Due to the curse of dimensionality, these methods cannot be easily adapted to the query for high-dimensional vectors and achieve competitive performance with the graph-based methods.

Similarity Search in General Metric Spaces. Beyond similarity search in Euclidean space, a large number of studies consider similarity search in general metric spaces [7, 9–16, 19–22, 25, 30, 32, 33, 46, 53, 63, 64, 66, 67, 70, 74, 76, 84, 85, 88]. In particular, a metric space is defined by a function with the properties of symmetry, positivity and triangle inequality. Euclidean space is an instance of metric spaces. The techniques proposed for similarity search in metric spaces usually do not rely on the additional properties of a specific space. Next, studies on similarity search in metric spaces can be generally divided into two threads, the exact and the approximate. The exact methods mostly conduct partitioning (of the metric space or dataset) and pruning (based on the triangle inequality) [7, 9–13, 15, 19, 21, 25, 32, 33, 53, 63, 64, 70, 74, 84] to accelerate the search. The approximate methods usually target better efficiency by allowing some sacrifice on accuracy [10, 14, 22, 30, 76, 85]. In particular, we note that graph-based methods that are popular for the ANN query in Euclidean space are also being studied widely in general metric spaces [11–13, 32, 33, 64]. In addition, in general metric spaces, attribute-filtering queries are also valuable as

users may pose hard constraints on their targeted data objects. We note that the method proposed in this paper is oblivious to the specific data type and metric function. This suggests that it has potential for being used in metric spaces as well (e.g., in combination with the graph-based methods).

7 CONCLUSION

In conclusion, in this paper, for the range-filtering ANN query, we propose a method named `iRangeGraph`. It constructs elemental graphs with moderate space consumption in the index phase and uses them to improvise range-dedicated graphs on the fly in the query phase. The algorithm `iRangeGraph` is further extended to support multi-attribute range-filtering ANN query. Extensive experiments on real-world datasets confirm its superior search performance over existing methods and moderate space consumption.

We would like to mention the following extensions and possible directions as future work. (1) The memory footprint of `iRangeGraph` could be further reduced by building the index based on a multi-branch tree. As more branches lead to fewer layers of the index, and as each object appears once in a layer, the memory footprint will be further reduced when a multi-branch tree is used. (2) For the attribute-filtering ANN queries with other types of predicates and attributes, there are also well-established classical algorithms which can handle the filtering (e.g., segment tree can handle the range filtering). It is worth exploring to extend the idea of `iRangeGraph` to the attribute-filtering for other types of attributes and predicates. (3) It would be an interesting research topic to further explore the dynamic updates, insertion and deletion of the `iRangeGraph` index given its promising search performance.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for providing constructive feedback and valuable suggestions. This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund (Tier 2 Award MOE-T2EP20221-0013, Tier 2 Award MOE-T2EP20220-0011, and Tier 1 Award (RG77/21)). This research is also supported by the Innovation Fund Denmark centre, DIREC. C- S. Jensen was supported in part by the Innovation Fund Denmark centre, DIREC. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2016. 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. 12.

[2] Sunil Arya and David M Mount. 1993. Approximate nearest neighbor queries in fixed dimensions.. In *SODA*, Vol. 93. Citeseer, 271–280.

[3] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. *Inf. Syst.* 87, C (jan 2020), 13 pages. https://doi.org/10.1016/j.is.2019.02.006

[4] Martin Aumüller and Matteo Ceccarello. 2023. Recent Approaches and Trends in Approximate Nearest Neighbor Search, with Remarks on Benchmarking. *Data Engineering* (2023), 89.

[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*. 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] Ricardo Baeza-Yates, Walter Cunto, Udi Manber, and Sun Wu. 1994. Proximity matching using fixed-queries trees. In *Combinatorial Pattern Matching: 5th Annual Symposium, CPM 94 Asilomar, CA, USA, June 5–8, 1994 Proceedings 5*. Springer, 198–212.

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

[9] Walter A. Burkhard and Robert M. Keller. 1973. Some approaches to best-match file searching. *Commun. ACM* 16, 4 (1973), 230–236.

[10] Domenico Cantone, Alfredo Ferro, Alfredo Pulvirenti, Diego Reforgiato Recupero, and Dennis Shasha. 2005. Antipole tree indexing to support range search and k-nearest neighbor search in metric spaces. *IEEE transactions on knowledge and data engineering* 17, 4 (2005), 535–550.

[11] Edgar Chavez, Stefan Dobrev, Evangelos Kranakis, Jaroslav Opatrny, Ladislav Stacho, Héctor Tejeda, and Jorge Urrutia. 2006. Half-space proximal: A new local test for extracting a bounded dilation spanner of a unit disk graph. In *Principles of Distributed Systems: 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers 9*. Springer, 235–245.

[12] Edgar Chávez, Verónica Luduena, Nora Reyes, and Patricia Roggero. 2014. Faster Proximity Searching with the Distal SAT. In *Similarity Search and Applications: 7th International Conference, SISAP 2014, Los Cabos, Mexico, October 29-31, 2104, Proceedings*, Vol. 8821. Springer, 58.

[13] Edgar Chávez, Verónica Luduena, Nora Reyes, and Patricia Roggero. 2016. Faster proximity searching with the distal SAT. *Information Systems* 59 (2016), 15–47.

[14] Edgar Chávez and Gonzalo Navarro. 2003. Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. *Inform. Process. Lett.* 85, 1 (2003), 39–46.

[15] Lu Chen, Yunjun Gao, Xinhan Li, Christian S Jensen, and Gang Chen. 2015. Efficient metric indexing for similarity search and similarity joins. *IEEE Transactions on Knowledge and Data Engineering* 29, 3 (2015), 556–571.

[16] Lu Chen, Yunjun Gao, Xuan Song, Zheng Li, Yifan Zhu, Xiaoye Miao, and Christian S Jensen. 2022. Indexing metric spaces for exact similarity search. *Comput. Surveys* 55, 6 (2022), 1–39.

[17] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. *SPTAG: A library for fast approximate nearest neighbor search.* https://github.com/Microsoft/SPTAG

[18] 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.

[19] Paolo Ciaccia and Marco Patella. 1998. Bulk loading the M-tree. In *Proceedings of the 9th Australasian Database Conference (ADC'98)*. Citeseer, 15–26.

[20] Paolo Ciaccia and Marco Patella. 2001. Approximate similarity queries: A survey. *University of Bolognia: Bologna, Italy* (2001).

[21] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 426–435.

[22] Kenneth L Clarkson. 1997. Nearest neighbor queries in metric spaces. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 609–617.

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

[24] Mark De Berg. 2000. *Computational geometry: algorithms and applications*. Springer Science & Business Media.

[25] FKHA Dehne and Hartmut Noltemeier. 1987. Voronoi trees and clustering problems. *Information Systems* 12, 2 (1987), 171–175.

[26] Magdalen Dobson, Zheqi Shen, Guy E Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2023. Scaling Graph-Based ANNS Algorithms to Billion-Size Datasets: A Comparative Analysis. *arXiv preprint arXiv:2305.04359* (2023).

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

[28] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2021. New Trends in High-D Vector Similarity Search: Al-Driven, Progressive, and Distributed. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3198–3201. https://doi.org/10.14778/3476311.3476407

[29] Joshua Engels, Benjamin Landrum, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2024. Approximate Nearest Neighbor Search with Window Filters. *arXiv preprint arXiv:2402.00943* (2024).

[30] Christos Faloutsos and King-Ip Lin. 1995. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. 163–174.

[31] Hakan Ferhatosmanoglu, Ioanna Stanoi, Divyakant Agrawal, and Amr El Abbadi. 2001. Constrained nearest neighbor queries. In *International Symposium on Spatial and Temporal Databases*. Springer, 257–276.

[32] Omar U Florez and SeungJin Lim. 2008. HRG: A graph structure for fast similarity search in metric spaces. In *Database and Expert Systems Applications: 19th International Conference, DEXA 2008, Turin, Italy, September 1-5, 2008. Proceedings*

19. Springer, 57–64.
[33] Cole Foster, Berk Sevilmis, and Benjamin Kimia. 2022. Generalized Relative Neighborhood Graph (GRNG) for Similarity Search. In *International Conference on Similarity Search and Applications*. Springer, 133–149.
[34] Cong Fu and Deng Cai. 2016. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv preprint arXiv:1609.07228* (2016).
[35] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 8 (2022), 4139–4150. https://doi.org/10.1109/TPAMI.2021.3067706
[36] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143* (2017).
[37] 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*. 541–552.
[38] Jianyang Gao and Cheng Long. 2023. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
[39] 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.
[40] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2013), 744–755.
[41] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. 2023. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*. 3406–3416.
[42] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. 2022. Parallel cover trees and their applications. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 259–272.
[43] 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, 3887–3896.
[44] Gaurav Gupta, Jonah Yi, Benjamin Coleman, Chen Luo, Vihan Lakshman, and Anshumali Shrivastava. 2023. CAPS: A Practical Partition Index for Filtered Similarity Search. *arXiv preprint arXiv:2308.15014* (2023).
[45] Ben Harwood and Tom Drummond. 2016. Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5713–5722.
[46] Gisli R Hjaltason and Hanan Samet. 2003. Index-driven similarity search in metric spaces (survey article). *ACM Transactions on Database Systems (TODS)* 28, 4 (2003), 517–580.
[47] 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.
[48] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.
[49] Piotr Indyk and Haike Xu. 2023. Worst-case Performance of Popular Approximate Nearest Neighbor Search Implementations: Guarantees and Limitations. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=oKqaWlEfjY
[50] Masajiro Iwasaki. 2016. Pruned bi-directed k-nearest neighbor graph for proximity search. In *International Conference on Similarity Search and Applications*. Springer, 20–33.
[51] 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).
[52] 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.
[53] Iraj Kalantari and Gerard McDonald. 1983. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering* 5 (1983), 631–634.
[54] 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.
[55] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. 2018. The design and implementation of a real time visual search system on JD E-commerce platform. In *Proceedings of the 19th International Middleware Conference Industry*. 9–16.
[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] Yusuke Matsui, Ryota Hinami, and Shin'ichi Satoh. 2018. Reconfigurable Inverted Index. In *Proceedings of the 26th ACM international conference on Multimedia*. 1715–1723.

[61] Microsoft. 2024. https://github.com/microsoft/DiskANN.

[62] 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.

[63] Juraj Moško, Jakub Lokoč, and Tomáš Skopal. 2011. Clustered pivot tables for I/O-optimized similarity search. In *Proceedings of the Fourth International Conference on SImilarity Search and APplications*. 17–24.

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

[65] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *arXiv preprint arXiv:2403.04871* (2024).

[66] Marco Patella and Paolo Ciaccia. 2008. The many facets of approximate similarity search. In *First International Workshop on Similarity Search and Applications (sisap 2008)*. IEEE, 10–21.

[67] Marco Patella and Paolo Ciaccia. 2009. Approximate similarity search: A multi-faceted problem. *Journal of Discrete Algorithms* 7, 1 (2009), 36–48.

[68] 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 and Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 4062–4063. https://doi.org/10.1145/3447548.3470811

[69] Aviad Rubinstein. 2018. Hardness of approximate nearest neighbor search. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing* (Los Angeles, CA, USA) *(STOC 2018)*. Association for Computing Machinery, New York, NY, USA, 1260–1268. https://doi.org/10.1145/3188745.3188916

[70] Enrique Vidal Ruiz. 1986. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters* 4, 3 (1986), 145–157.

[71] 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* (2014).

[72] 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.

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

[74] Jeffrey K Uhlmann. 1991. Satisfying general proximity/similarity queries with metric trees. *Information processing letters* 40, 4 (1991), 175–179.

[75] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[76] Jason Tsong-Li Wang, Xiong Wang, Dennis Shasha, and Kaizhong Zhang. 2005. Metricmap: an embedding technique for processing distance-based queries in metric spaces. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 35, 5 (2005), 973–987.

[77] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2024. An efficient and robust framework for approximate nearest neighbor search with attribute constraint. *Advances in Neural Information Processing Systems* 36 (2024).

[78] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 14, 11 (2021), 1964–1978. http://www.vldb.org/pvldb/vol14/p1964-wang.pdf

[79] Yifan Wang. 2022. A Survey on Efficient Processing of Similarity Queries over Neural Embeddings. *arXiv preprint arXiv:2204.07922* (2022).

[80] 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. *Data Engineering* (2023), 3–21.

[81] 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.

[82] Wei Wu, Junlin He, Yu Qiao, Guoheng Fu, Li Liu, and Jin Yu. 2022. HQANN: Efficient and robust similarity search for hybrid queries with structured and unstructured constraints. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 4580–4584.

[83] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. 2025. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search (Technical Report). https://github.com/YuexuanXu7/iRangeGraph.

[84] Peter N Yianilos. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Soda*, Vol. 93. 311–21.

[85] Pavel Zezula, Pasquale Savino, Giuseppe Amato, and Fausto Rabitti. 1998. Approximate similarity retrieval with M-trees. *The VLDB Journal* 7, 4 (1998), 275–293.

[86] 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 *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 377–395. https://www.usenix.org/conference/osdi23/presentation/zhang-qianxi

[87] Weijie Zhao, Shulong Tan, and Ping Li. 2022. Constrained Approximate Similarity Search on Proximity Graph. *arXiv preprint arXiv:2210.14958* (2022).

[88] Yifan Zhu, Lu Chen, Yunjun Gao, and Christian S Jensen. 2022. Pivot selection algorithms in metric spaces: a survey and experimental study. *The VLDB Journal* 31, 1 (2022), 23–47.

[89] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

Received April 2024; revised July 2024; accepted August 2024