SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search

CHAOJI ZUO*, Rutgers University, USA
MIAO QIAO, The University of Auckland, New Zealand
WENCHAO ZHOU, Alibaba Group, China
FEIFEI LI, Alibaba Group, China
DONG DENG†, Rutgers University, USA

Effective vector representation models, e.g., word2vec and node2vec, embed real-world objects such as images and documents in high dimensional vector space. In the meanwhile, the objects are often associated with attributes such as timestamps and prices. Many scenarios need to jointly query the vector representations of the objects together with their attributes. These queries can be formalized as range-filtering approximate nearest neighbor search (ANNS) queries. Specifically, given a collection of data vectors, each associated with an attribute value whose domain has a total order. The range-filtering ANNS consists of a query range and a query vector. It finds the approximate nearest neighbors of the query vector among all the data vectors whose attribute values fall in the query range. Existing approaches suffer from a rapidly degrading query performance when the query range width shifts. The query performance can be optimized by a solution that builds an ANNS index for every possible query range; however, the index time and index size become prohibitive – the number of query ranges is quadratic to the number n of data vectors. To overcome these challenges, for the query range contains all attribute values smaller than a user-provided threshold, we design a structure called the segment graph whose index time and size are the same as a single ANNS index, yet can losslessly compress the n ANNS indexes, reducing the indexing cost by a factor of $\Omega(n)$. To handle general range queries, we propose a 2D segment graph with average-case index size $O(n \log n)$ to compress n segment graphs, breaking the quadratic barrier. Extensive experiments conducted on real-world datasets show that our proposed structures outperformed existing methods significantly; our index also exhibits superior scalability.

CCS Concepts: • **Information systems → Nearest-neighbor search**; • **Theory of computation → Nearest neighbor algorithms**.

Additional Key Words and Phrases: Similarity Search, Multi-Model Search, High-Dimensional Vector

ACM Reference Format:
Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 69 (February 2024), 26 pages. https://doi.org/10.1145/3639324

*Work performed during an internship at Alibaba Group
†Corresponding author

Authors' addresses: Chaoji Zuo, Rutgers University, Piscataway, USA, chaoji.zuo@rutgers.edu; Miao Qiao, The University of Auckland, Auckland, New Zealand, miao.qiao@auckland.ac.nz; Wenchao Zhou, Alibaba Group, Hangzhou, China, zwc231487@alibaba-inc.com; Feifei Li, Alibaba Group, Hangzhou, China, lifeifei@alibaba-inc.com; Dong Deng, Rutgers University, Piscataway, USA, dong.deng@rutgers.edu.

ACM 2836-6573/2024/2-ART69
https://doi.org/10.1145/3639324

1 Introduction

In recent years, various machine learning models, e.g., word2vec [33, 35], doc2vec [25], and node2vec [16], have been developed to effectively represent real-world objects such as images, documents, and graphs as high-dimensional feature vectors. In the meanwhile, real-world objects are often associated with structured attributes/fields such as timestamps, prices, and quantities. In many scenarios, the feature vectors and the structured attributes of the objects need to be jointly queried, as illustrated in the motivation examples below.

Example 1: Product Search. On e-commerce platforms like Amazon, a customer may search for a wardrobe with conditions on price (less than $200) and style (visually similar to the one in an image). The condition on the style can be formulated as an approximate nearest neighbor search (ANNS) over the image feature vectors. However, the search should be conducted not over all the wardrobes but over a subset of the wardrobes whose prices are less than $200.

Example 2: Vehicle Search. Consider a traffic camera that monitors cars passing on a state highway. The camera detects each car in the video stream, extracts a feature vector to represent it, and stores the feature vector along with its corresponding timestamp in a database. When a query arrives with a specific car image and a specified time interval, the query processing performs an ANNS on the feature vectors within the specified time interval.

The above queries can be formulated as the *range-filtering approximate nearest neighbor search* queries. Consider a collection of objects where each object is represented as a pair of a data vector (i.e., feature vector) and an attribute value whose domain has a total order. A range-filtering ANNS query consists of a query vector and a query range. The query reports the approximate nearest neighbors of the query vector among all the data vectors whose attribute values fall in the query range. The scenarios of range-filtering ANNS can be found in many applications such as face recognition [15], person/vehicle re-identification [27, 47], and recommendation [39].

Existing methods for range-filtering ANNS rely on two simple strategies, ANNS-first and range-first [40]. ANNS-first builds an ANNS index over all the data vectors during the offline indexing phase. In the online querying phase, when a range-filtering ANNS query arrives, it progressively finds data vectors nearest to the query vector using the ANNS index. It stops when a data vector whose attribute value falls in the query range. However, *ANNS-first is very slow especially when the query range is small*: the portion of the data vectors falling into a small query range is small and it is thus hard to meet one in the ANNS. On the other hand, the range-first strategy filters the data vector with the query range first based on their attribute values. Since there is no ANNS index over the remaining data vectors available, it can only linearly scan all the data vectors in the query range to find the nearest neighbor, *which is rather inefficient, especially when the query range is large.*

Hierarchical Navigable Small World graph (HNSW) [30] is an efficient index for ANNS query, which has been widely used in both academia and industry. For example, the multi-modal feature vector dataset LAION-5B, which contains 5.85 billion vectors [37], comes with an HNSW index for efficient ANNS. Lucene 9.0 implements HNSW to support ANNS [1]. Moreover, HNSW index also serves as the backbone of various vector databases such as Weaviate [3], Zilliz [4], and Pinecone [2]. Using HNSW as a building block, one can build one ANNS index (HNSW) for every possible query range: when a range-filtering ANNS query arrives, use the corresponding ANNS index of the query range to find and report a set of approximate nearest neighbors of the query vector. This solution is query-efficient, yet it faces the challenge of the excessively large index size and long index time: the total number of possible query ranges is quadratic to the number of data vectors.

To address the above challenges, for a set of n data vectors, we first consider *half-bounded range queries* where the query range includes all the attribute values smaller than a user-provided value.

Note that there are n possible half-bounded ranges which can be trivially answered if one constructs n HNSW indexes. We propose a structure called *segment graph* which *losslessly compresses* all the n HNSW indexes with indexing time and space the same as that of constructing a single HNSW index. In other words, the segment graph reduces the indexing cost for half-bounded queries by a factor of $\Omega(n)$. Note the construction of the segment graph does not necessitate the construction of the n HNSW graphs.

When it comes to queries with general ranges, constructing n segment graphs is sufficient for query processing, but the quadratic index size and time are not scalable to large datasets. We propose *2D segment graph* to compress the n segment graphs. *The average-case index size of 2D segment graph is $O(n \log n)$, breaking the quadratic barrier.* We propose optimizations to further reduce index time and size. Our experiments show that our 2D segment graph can achieve superior empirical performance.

In summary, this paper makes the following contributions.

- We propose, for half-bounded range-filtering ANNS queries, a structure called *segment graph* to losslessly compress n HNSW indexes with the time and space complexity that of a single HNSW index, i.e., we reduce the index cost by a factor of $\Omega(n)$.
- We propose, for general range-filtering ANNS queries, an index called *2D segment graph* which compresses n segment graphs with average-case index size breaking the quadratic barrier.
- Experiments on real-world datasets show that our indexes significantly outperformed existing methods and are highly scalable.

The paper is organized as follows. Section 2 defines the problem and introduces our building block HNSW. Section 3 introduces segment graph while Section 4 presents the 2D segment graph and index optimization techniques. Section 5 shows experimental results. Section 6 explains related work. Section 7 concludes the paper.

2 Preliminary

This section introduces the notions that shall be used throughout the paper. Let $d > 0$ be an integer and δ a distance metric (e.g., the Euclidean distance metric) on the d-dimensional space of $\mathbb{R}^d$. Let $n > 0$ be an integer and $\mathcal{D}$ be a set of n points in $\mathbb{R}^d$.

Definition 1 (Nearest Neighbors Search). *Given a query point $q \in \mathbb{R}^d$ and an integer $k > 0$, the k-nearest neighbors of q, denoted as $\mathsf{kNN}_\delta(q, \mathcal{D})$, is the set of k points in $\mathcal{D}$ with the smallest distances to q under metric δ. Formally, $\mathsf{kNN}_\delta(q, \mathcal{D})$ is a set $\mathcal{S} \subseteq \mathcal{D}$ of k points in $\mathcal{D}$ such that for $\forall u \in \mathcal{S}$ and $\forall v \in \mathcal{D} \setminus \mathcal{S}$, $\delta(u, q) \leq \delta(v, q)$.*

In particular, $\mathsf{1NN}_\delta(q, \mathcal{D}) = \{\arg\min_{v \in \mathcal{D}} \delta(v, q)\}$. We omit the subscription δ when the context is clear. Due to the notorious "curse of dimensionality" [19], a large body of existing nearest neighbor search studies approximate nearest neighbors search (ANNS) [26] which reports a set $\mathsf{kANN}(q, \mathcal{D})$ of k points aiming at an optimized recall $\frac{1}{k}|\mathsf{kANN}(q, \mathcal{D}) \cap \mathsf{kNN}(q, \mathcal{D})|$ for a point q and integer k.

2.1 Problem Definition

This paper considers $\mathcal{D}$ with **attributed** data points and the nearest neighbor search with attribute constraints. Specifically, let A be an attribute (e.g., dates, prices, quantities, etc.) whose domain $Dom(A)$ has a *total order*. Each point v in $\mathcal{D}$ is associated with an A-attribute value $v[A]$. Let I_A be any interval on $Dom(A)$. Denote by $\mathcal{D}[I_A] = \{v \in \mathcal{D} | v[A] \in I_A\}$ the set of points in $\mathcal{D}$ whose A-attribute value falls in I_A. Given a query point q and an attribute interval I_A on $Dom(A)$, *we aim at reporting the nearest neighbors of q in $\mathcal{D}[I_A]$.*

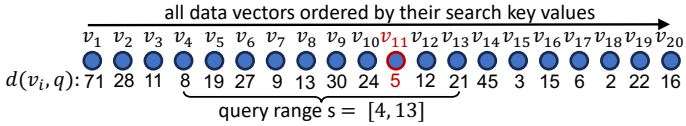


Fig. 1. An example of the range-filtering ANNS query.

To simplify the search, we sort the data points in $\mathcal{D}$ in ascending order of their A-attribute values $\mathcal{D} = \{v_1, v_2, \cdots, v_n\}$ and call the index of each point its **key**. In other words, for any $v_i, v_j \in \mathcal{D}$, their keys $i < j$ implies $v_i[A] \leq v_j[A]$. For an A-attribute interval I_A, we can find two integers $i, j \in [n]$ such that $\mathcal{D}[I_A] = \mathcal{D}_{i,j} = \{v_x \in \mathcal{D} | x \in [i, j]\}$ based on the *predecessor operation*: for any attribute value $a \in Dom(A)$, the point v in $\mathcal{D}$ with the largest key such that $v[A] < a$ (or $v[A] \leq a$) can be reported on $\mathcal{D} = \{v_1, v_2, \cdots, v_n\}$ in $O(\log(n))$ time. The nearest neighbor search with the above attribute constraints then boils down to the problem defined below.

Definition 2 (Range-Filtering Nearest Neighbors Search). *Let $\mathcal{D} = \{v_1, v_2, \cdots, v_n\}$ be a set of n points in $\mathbb{R}^d$. The key of the point $v_i \in \mathcal{D}$ is i, for each $i \in [n]$. A range-filtering nearest neighbors search query $Q = (q, [i, j], k)$ with integers $i, j \in [n]$ and $0 < k \leq j - i + 1$ returns $\text{kNN}(q, \mathcal{D}_{i,j})$, a k-sized subset $\mathcal{R}$ of $\mathcal{D}$ s.t.*

- $\forall\, v_x \in \mathcal{R}, x \in [i, j]$, *and*
- $\forall\, v_x \in \mathcal{R}$ *and* $\forall\, v_y \in \mathcal{D} \setminus \mathcal{R}$, *either* $y \notin [i, j]$ *or* $\delta(v_x, q) \leq \delta(v_y, q)$.

Example 1. Figure 1 shows an example: Consider the set of $n = 20$ points $\mathcal{D} = \{v_1, \cdots, v_{20}\}$. The range-filtering nearest neighbors search query $Q = (q, [4, 13], 1)$ consists of a query point q, a query range $[4, 13]$, and an integer $k = 1$. We show by the number under the point v_i its distance to the query point $\delta(v_i, q)$. The query Q returns $\text{1NN}(q, \mathcal{D}_{4,13}) = \{v_{11}\}$. Though v_{18} and v_{15} are closer to the query point q than v_{11}, their keys 18 and 15 are not within the query range $[4, 13]$, thus both are filtered by the range condition.

This paper studies the range-filtering *approximate* nearest neighbors search (RFANNS) which generalizes traditional approximate k-nearest neighbors search (ANNS) with range-filtering constraints. The problem can be defined as follows.

Definition 3 (RFANNS). *Given a set $\mathcal{D} = \{v_1, v_2, \cdots, v_n\}$ of points in $\mathbb{R}^d$, a range-filtering approximate nearest neighbors search query $Q = (q, [i, j], k)$ aims at reporting $\text{kANN}(q, \mathcal{D}_{i,j})$, a subset of k points in $\mathcal{D}_{i,j}$, with an optimized recall $\frac{|\text{kANN}(q, \mathcal{D}_{i,j}) \cap \text{kNN}(q, \mathcal{D}_{i,j})|}{k}$.*

As $\mathcal{D}_{1,n}$ and $\mathcal{D}$ are identical, the traditional ANNS is essentially a special case $(q, [1, n], k)$ of RFANNS. Without loss of generality, we break distance ties with the ordering on keys, i.e., any two points will have different distances to another point.

2.2 Graph-based ANNS and HNSW

HNSW (Hierarchical Navigable Small-World graph) [30] is an efficient and widely used graph-based index for approximate nearest neighbor search [42]. This line of ANNS establishes a graph G on $\mathcal{D}$ to navigate the nearest neighbor search: G has n nodes where each node in G is a point $v_i \in \mathcal{D}$, $i \in [n]$. We define the distance between two nodes v_i and v_j in G as the δ-distance $\delta(v_i, v_j)$ of their points. In our paper, nodes and points are interchangeable, indicating both the nodes in the graph G and the points in $\mathcal{D}$. We denote the edge set of G with adjacency lists. Specifically, for each node $v_i \in \mathcal{D}$, denote by $\text{G}[v_i] \subseteq \mathcal{D}$ the set of (outgoing) neighbors of v_i, i.e., for any $i, j \in [n]$, there is a directed edge (v_i, v_j) in G iff $v_j \in \text{G}[v_i]$.

HNSW is widely adopted for real-world ANNS and serves as a building block of our solution. Next, we introduce the process of the ANNS in HNSW and the establishment of the HNSW graph.

Algorithm 1: ANNSearch(G, q, ep, K)

Input: G: HNSW graph, denote the neighbor set of a node v as G[v]; q: query point; ep: entry point ; K: an integer.
Output: *ann*: a set of K approximate nearest neighbors of q.

```
1  mark ep as visited;
2  push ep to the min-heap pool in the order of distance to q;
3  push ep to the max-heap ann in the order of distance to q;
4  while pool is not empty do
5      v ← the vector nearest to q in pool, pop pool;
6      u ← the vector farthest to q in ann;
7      if δ(q, v) > δ(q, u) then break;
8      foreach unvisited o ∈ G[v] do
9          mark o as visited;
10         u ← the vector farthest to q in ann;
11         if |ann| < K or δ(q, o) < δ(q, u) then
12             push o to pool and ann;
13             if |ann| > K then pop ann;
14 return ann;
```

Algorithm 2: EdgeInsertion(o, ep, M, K, G)

Input: o: a point in $\mathcal{D}$; ep: the entry point; M: the maximum degree; K: an integer; G: the existing HNSW graph.
Output: G: the updated HNSW graph.

```
1  ann ← ANNSearch(G, o, ep, K);
2  G[o] ← Prune(o, ann, M);
3  foreach u ∈ G[o] do
4      add o to G[u];
5      if |G[u]| > M then G[u] ← Prune(u, G[u], M);
6  return G;
```

ANN Search (Algorithm 1). Let q be the query point and K $\geq k$ be an integer. The search aims at reporting K ANNs of q with a priority search [36] over the HNSW graph G from an entry point ep. The entry point can be trivially v_1 or a point in $\mathcal{D}$ (node in G) chosen by heuristics [42] (we used v_1 in our paper). The search uses two heaps *ann* and *pool* both keeping a subset of points in $\mathcal{D}$. Initially, both *ann* and *pool* contain only the entry point ep (Lines 2-3). *ann* is a max-heap which keeps up to K visited nodes that are *nearest* – having the smallest δ-distance – to the query point q: it can pop the K-th ANN searched so far. *pool* is a min-heap that keeps all the unvisited neighbors of all the visited nodes: it can pop the closest-to-q unvisited neighbors of the visited nodes. The search repeatedly picks the closest-to-q point v from *pool* (Line 5), and adds v's unvisited neighbors o in G to *pool* (Lines 8-13) while updating *ann* with o (Lines 11-13). The search stops when all the points in *pool* are of longer distance to q than all the points in *ann* (Lines 4,7,11). The size of *ann* is kept no larger than K (Line 14). Eventually, the k points (in *ann*) that are nearest to q are regarded as the approximate k-nearest neighbors of q.

Graph Construction. HNSW graph is constructed with repetitive edge insertions. Let M be a parameter indicating the maximum outdegree of the HNSW graph nodes. Starting with a graph G on $\mathcal{D}$ without any edge, G is updated by visiting the points o in $\mathcal{D}$ one by one in an arbitrary order: the visit to o adds to G the edges incident to o which may trigger the pruning of existing edges (Algorithm 2). Specifically, when visiting o, the construction algorithm uses the current HNSW graph to find a set, denote as *ann*, of approximate nearest neighbors of o by calling Algorithm 1 (Line 1). The tentative edges incident to o are the edges between o and the points in *ann* (Lines 2-5). Note that, inserting these edges to G may trigger a pruning process (Lines 2 and 5). The pruning process (will be detailed later) selects from *ann* a subset of points that are far away from each other as o's neighbors in the HNSW graph (Lines 2). Moreover, when adding o to its neighbor u's neighbor list G[u] causes an overflow ($\geq$ M outgoing edges, Line 5), the neighbor list G[u] will also be pruned to keep only spatially diversified nodes. The spatially diversified neighbors of each node ensure better coverage/connectivity for navigation, leading to a better recall.

The pruning process is based on a key concept of domination.

Definition 4 (Domination). *Given a point $o \in \mathcal{D}$ (called the center) and two other points u and v in $\mathcal{D}$, u dominates v with respect to the center o if and only if $\delta(o,u) < \delta(o,v)$ and $\delta(u,v) < \delta(o,v)$.*

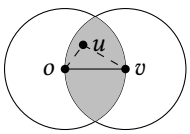


Fig. 2. Who dominates v?

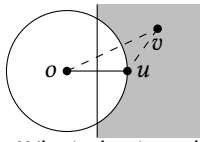


Fig. 3. Who is dominated by u?

Example 2. Consider 2-dimensional Euclidean space $\mathbb{R}^2$ on which o is the center point. Figure 2 shades all the points u that dominate point v: the intersection of two $\delta(o,v)$-radius circles centered at o and v, respectively. Figure 3 shades all the points v that are dominated by u: the area to the right of the perpendicular of o and u outside the circle centered at o of radius $\delta(o,u)$.

Algorithm 3 shows the pruning process. The tentative neighbors v in *ann* are checked one by one in the ascending order of their distances to o (Line 2). v is selected as a neighbor of o only if it is not dominated by any neighbor already selected (Lines 3-7). This pruning process is key to the scalability of the HNSW graph. The average number of neighbors in the HNSW graph only grows sublinearly with the dataset size empirically.

Example 3. Consider Figures 2-3. To include u to the neighbor list of o, there must be no selected neighbors in the shaded area of Figure 2; by adding u to the neighbor list of o, no nodes in the shaded area of Figure 3 can be selected as o's neighbors.

Complexity Analysis. As each node in the HNSW graph G has at most M neighbors. The space complexity is $O(n\mathrm{M})$.

3 RFANNS with Half-bounded Query Ranges

This section considers half-bounded (query) ranges, a special type of ranges, in range-filtering approximate nearest neighbor search (RFANNS). Compared to the range $[i, j]$ in Definition 3, a half-bounded range has one side, either the left side or the right side, aligned with the domain $[1, n]$ of the keys. Specifically, for a given key $z \in [n]$, half-bounded range is either $[1, z]$ (left-bounded) or $[z, n]$ (right-bounded). Next, we focus on left-bounded ranges $[1, z]$ because all the techniques developed for left-bounded ranges can be symmetrically applied to right-bounded ranges.

A straightforward solution to RFANNS with left-bounded ranges is to build n HNSW graphs: an HNSW graph G_x on the set of $\mathcal{D}_x = \{v_1, v_2, \cdots, v_x\}$ for each $x \in [n]$. Upon the arrival of an

Algorithm 3: PRUNE(o, *ann*, M)

Input: o: a point; *ann*: a set of o's approximate nearest neighbors; M: the max number of neighbors to keep.
Output: *neighbors* $\subseteq$ *ann*: o's neighbors after pruning.

```
1 neighbors ← ∅;
2 foreach v ∈ ann in the ascending order of δ(o, v) do
3     not_dominated ← true;
4     foreach u ∈ neighbors do
5         if u dominates v as o's neighbors then
6             not_dominated ← false and break;
7     if not_dominated then add v to neighbors;
8     if |neighbors| ≥ M then break;
9 return neighbors;
```

Algorithm 4: GRAPHCONSTRUCTION(M, K, $\mathcal{D}$)

Input: M: the maximum degree; K: an integer; $\mathcal{D}$: the ordered set of points $\{v_1, v_2, \cdots, v_n\}$
Output: $G_1, G_2, \cdots, G_n$: the HNSW graphs for $\mathcal{D}_1, \cdots, \mathcal{D}_n$.

```
1 G_1 ← a graph with only node v_1 without edges;
2 foreach i ∈ [2, n] do G_i ←EDGEINSERTION(v_i, v_1, M, K, G_{i−1});
3 return G_1, G_2, ··· , G_n;
```

RFANNS query $Q = (q, [1, z], k)$, call Algorithm 1 on the HNSW graph G_z to report the k ANNs of q in $\mathcal{D}_z$. The query processing is thus efficient. However, the drawback lies in the quadratic cost in constructing and storing the n HNSW graphs – the space complexity of $O(Mn^2)$ is prohibitive on a set $\mathcal{D}$ with millions of data points.

In the following, Section 3.1 shows how to incrementally construct n HNSW graphs $G_1, G_2, \cdots, G_n$ for $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_n$, respectively, and Section 3.2 proposes a structure called **segment graph** to losslessly compress the n HNSW graphs in one graph of $O(n\mathrm{M})$ size, saving the space complexity by a factor of n. Note that the segment graph can be constructed directly from $\mathcal{D}$ without physically compressing the n HNSW graphs, leading to efficient indexing.

3.1 HNSW Graphs for Half-bounded Ranges

Recall that in Section 2.2, the HNSW graph of $\mathcal{D}$ is constructed by i) vising the nodes o in $\mathcal{D}$ one by one in an arbitrary order and ii) when visiting o, inserting the edges incidents on o to the HNSW graph using Algorithm 2. If we use the ordering of the points in $\mathcal{D} = \{v_1, v_2, \cdots, v_n\}$ as the node visiting order, it turns out that for each $x \in [n]$, the snapshot G_x of the HNSW graph after inserting v_x is the HNSW graph of $\mathcal{D}_x$. Algorithm 4 shows the construction of the HNSW graphs $G_1, \cdots, G_n$ for point sets $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_n$, respectively. Lemma 1 shows the proof of correctness.

When sequentially inserting points, the original HNSW algorithm updates the HNSW graph in-place while Algorithm 4 does it out-of-place, i.e., it creates a new graph after each insertion (Line 2). Under the same node order, the produced graphs are identical.

LEMMA 1. *Under the same parameters* M *and* K*, the HNSW graph constructed after inserting nodes in* $\mathcal{D}_x = \{v_1, v_2, \cdots, v_x\}$ *sequentially is the graph* G_x *reported by Algorithm 4, for each* $x \in [n]$.

PROOF. G_1 is a graph with a single point v_1, it is the HNSW graph of $\mathcal{D}_1 = \{v_1\}$ (Line 1). If, for $i = x - 1$, G_{x-1} is an HNSW graph of $\mathcal{D}_{x-1} = \{v_1, \cdots, v_{x-1}\}$, then when we insert v_i (i.e., $i = x$) to

Algorithm 5: SegmentGraphConstruction

Input: $\mathcal{D}$: $\{v_1, v_2, \cdots, v_n\}$; K: an integer; M: the max degree.
Output: G: segment HNSW graph for half-bounded ranges.

```
1  G ← a graph with node v_1 without any edge;
2  foreach 1 < x ≤ n do
3      ann ← SegmentANNSearch(G, v_x, [1, x − 1], v_1, K);
4      foreach v_i ∈ Prune(v_x, ann, M) do
5          add (v_i, x, n) to G[v_x];
6          add (v_x, x, n) to G[v_i];
7          neighbors ← {v | (v, b, n) ∈ G[v_i]};
8          if |neighbors| > M then
9              nbrs ← Prune(v_i, neighbors, M);
10             foreach v ∈ neighbors \ nbrs do
11                 update (v, b, n) ∈ G[v_i] as (v, b, x − 1);
12 return G;
```

G_{x-1} by calling Algorithm 2 (Line 2), we first use v_x as the query vector and perform the ANN search (Line 1 Algorithm 2) over the HNSW graph G_{x-1} to find a set *ann* of approximate nearest neighbors of v_x in $\mathcal{D}_{x-1}$. Then we prune *ann* to get a set of spatially diversified neighbors of v_x, forming the neighbor list $G_x[v_x]$ of graph G_x. For any point $v_j \in \mathcal{D}_{x-1}$, if $v_j \notin G_x[v_x]$, its neighbor list remains the same, i.e., $G_x[v_j] = G_{x-1}[v_j]$; otherwise we append v_x to v_j's neighbor list, i.e., $G_x[v_j] = G_{x-1}[v_j] \cup \{v_x\}$. Note that if v_j has more than M neighbors, we prune its neighbor list $G_x[v_j]$. Since $\mathcal{D}_x = \mathcal{D}_{x-1} \cup \{v_x\}$, G_x is an HNSW graph of $\mathcal{D}_x$. □

Complexity Analysis. Each HNSW graph G_x takes $O(n\mathrm{M})$ space. There are n graphs in total. Thus the space complexity is $O(n^2\mathrm{M})$. This prohibitive space complexity motivates us to develop a lossless compression of the n HNSW graphs, totaling only $O(n\mathrm{M})$ space.

3.2 Segment Graph

To compress the n HNSW graphs, we make a key observation below.

Observation 1. *Consider a point v_i and its neighbor v_j in an* HNSW *graph G_x built by Algorithm 4. Formally, $v_j \in G_x[v_i]$. It can be observed that i) $1 \le i \ne j \le x$ and ii) v_j is v_i's neighbor in $G_x, G_{x+1}, \cdots, G_t$, $t \in [x, n]$ with (1) either $t = n$ or (2) upon the insertion of v_{t+1}, the neighborhood pruning process is triggered (Line 5, Algorithm 2) and v_j is dominated and pruned by another neighbor of v_i (Lines 5-6, Algorithm 3). In this case, v_j cannot be v_i's neighbor for any $G_{t'}$, $t' > t$: only the unvisited nodes can be v_i's new neighbors.*

Observation 1 shows that in Algorithm 4, once a node v_i becomes the neighbor of another node v_j, it remains to be v_j's neighbor until it is pruned from v_j's neighbor list. The prune is permanent, i.e., v_i will not be added back to v_j's neighbor list thereafter.

Example 4. Figure 4 shows the neighbors of v_9 in Algorithm 4 under the max degree $\mathrm{M} = 5$. When visiting v_9, after ANNSearch and neighborhood pruning, v_3, v_4, and v_6 become v_9's neighbors, and thus $G_9[v_9] = \{v_3, v_4, v_6\}$. When visiting v_{12}, v_9 becomes v_{12}'s neighbor and thus $G_{12}[v_9]$ becomes $\{v_3, v_4, v_6, v_{12}\}$. Similarly, $G_{14}[v_9]$ becomes $\{v_3, v_4, v_6, v_{12}, v_{14}\}$ after visiting v_{14}. When v_{16} is visited, v_9 is selected as v_{16}'s neighbor; however, adding v_{16} to v_9's neighbor list triggers v_9's neighborhood

Fig. 4. The evolution of a point's neighbors.

Algorithm 6: SegmentANNSearch($\mathbf{G}$, q, $[1, z]$, ep, K)

Input: G: segment HNSW graph; $[1, z]$: a query range.
Output: *ann*: K approximate nearest neighbors of q in $\mathcal{D}_z$.
`// replace Line 8, Algorithm 1 with the line below`
1 **foreach** unvisited o with $(o, b, e) \in \mathbf{G}[v]$ **do if** $z \in [b, e]$ **then**

pruning process: v_9 now has more than M = 5 outgoing neighbors. Three neighbors of v_9, i.e., v_3, v_4, and v_{14}, are dominated and pruned and we have $\mathrm{G}_{16}[v_9] = \{v_6, v_{12}, v_{16}\}$. Since then, only $v_{t'}$ with $t' > 16$ can be added as v_9's neighbor; v_3, v_4, and v_{14} won't be v_9's neighbors again.

Based on Observation 1, we define the **segment graph**, a structure that stores all the n HNSW graphs in a single graph where each edge is associated with a segment $[b, e]$ indicating the edge exists exclusively in G_x with $x \in [b, e]$, as formalized below.

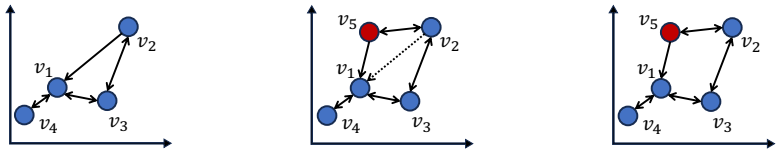
Definition 5. *Given a set $\mathcal{D} = \{v_1, v_2, \cdots, v_n\}$, denote by $\mathrm{G}_1, \mathrm{G}_2, \cdots, \mathrm{G}_n$ the HNSW graphs constructed by Algorithm 4. A* segment graph $\mathbf{G}$ *of $\mathcal{D}$ is a graph on $\mathcal{D}$. Each node v_i has an outgoing neighbor list $\mathbf{G}[v_i]$ where each neighbor is represented as a triple $(v_j, b, e) \in \mathbf{G}[v_i]$ with $1 \le b \le e \le n$ indicating that node v_j is an outgoing neighbor of v_i on the HNSW graphs of $\mathrm{G}_b, \mathrm{G}_{b+1}, \cdots, \mathrm{G}_e$ only. We call b the* beginning timestamp *and e the* ending timestamp.

It turns out that the construction of the segment graph $\mathbf{G}$ does not necessitate the construction of the n HNSW graphs of $\mathrm{G}_1, \cdots, \mathrm{G}_n$. In the following, Algorithm 5 shows the construction of $\mathbf{G}$ and Theorem 6 shows that $\mathbf{G}$ is a lossless compression of $\mathrm{G}_1, \mathrm{G}_2, \cdots, \mathrm{G}_n$.

Segment Graph Construction. Algorithm 5 shows the segment graph construction. It tracks a neighbor's beginning timestamp, i.e., when the neighbor is added to the neighbor list, and ending timestamp, i.e., when the neighbor is pruned. Specifically, when inserting a point v_x, edges (v_x, v_i) and (v_i, v_x) incident to v_x are constructed and we assign them a beginning timestamp x (Lines 5-6). If adding the edge to v_i triggers the neighborhood pruning process, we update the ending timestamp of the pruned edges as $x - 1$ (Lines 9 to 11). Note that the ending timestamp of every edge is set to be n when the edge is constructed and is updated when the edge is removed. Thus we only take into account the neighbors whose ending timestamps are n when determining whether the neighborhood pruning process should be triggered (Lines 7 to 8).

ANNSearch on Segment Graph. Algorithm 6 takes a segment graph $\mathbf{G}$, a query vector q, a left-bounded query range $[1, z]$, and an integer K as input and outputs a set of K approximate nearest neighbors of q in $\mathcal{D}_z$. The search is similar to the ordinary ANNSearch over the HNSW graph G_z (i.e., Algorithm 1). The only difference is that for each neighbor (o, b, e) in the neighbor list to explore, we visit the neighbor o only if $z \in [b, e]$ which leads to an additional check before visiting a neighbor (Line 5). This is because the edge from v to o does not exist in the graph G_z if $z \notin [b, e]$. As shall be proved in Theorem 6, the segment graph $\mathbf{G}$ is a lossless compression of the n HNSW graphs $\mathrm{G}_1, \cdots, \mathrm{G}_n$. Thus, Algorithm 6 produces the same results as Algorithm 1 on G_z.

Example 5. Consider the construction of an HNSW graph on a set of 2-dimentional points. Let G_4 in Figure 5(a) be the HNSW graph after inserting 4 nodes $\{v_1, v_2, v_3, v_4\}$ sequentially. Suppose



(a) G_4 (b) inserting v_5 (c) G_5

Fig. 5. An example of segment graph construction.

the neighbors found for v_5 after pruning are v_1 and v_2. After inserting the node v_5 to G_4, G_5 in Figure 5(b) retains all the edges in G_4 with only one exception: edge (v_2, v_1) is removed because v_1 was dominated by v_5 in the neighbor list of v_2. Three new edges (v_5, v_1), (v_5, v_2), and (v_2, v_5) are inserted. Note that node v_5 was not able to enter v_1's neighbor list as it was dominated by v_3. Combining Figure 5(a) and 5(c), we can update the segment graph $\mathbf{G}$: edge (v_2, v_1) has an ending timestamp of 4 while the three new edges have the beginning timestamp of 5. Note that all the new edges upon the insertion of v_5 are incident on v_5, therefore, after (v_2, v_1) is removed, it will not be added back in the future – both v_2 and v_1 had been inserted before the removal of the edge. In other words, each edge will be inserted to / removed from the HNSW graph by at most once during the construction.

As another example, Figure 4 shows the beginning timestamps and ending timestamps of the point v_9's neighbors before and after inserting v_{16} into the segment graph.

Theorem 6. *Consider the segment graph* $\mathbf{G}$*. For each edge from* v_i *to* v_j *in* $\mathbf{G}$*,* $\forall i, j \in [n]$*, i.e., there is a triple* (v_j, b, e) *in the neighbor list of* v_i *in* $\mathbf{G}$*, we call* $[b, e]$ *the range of edge* (v_i, v_j)*. For any* $1 \le x \le n$*, denote by* $\mathbf{G}[x]$ *the subgraph of the segment graph* $\mathbf{G}$ *containing all edges whose range has* $b \le x \le e$*; let* G_x *be the* x*-th* HNSW *graphs produced by Alg. 4. We have* $\mathbf{G}[x] = G_x$*, for all* $x \in [n]$*; in other words, the segment graph* $\mathbf{G}$ *produced by Alg. 5 is a lossless compression of the* n HNSW *graphs produced by Alg. 4.*

Proof. Consider the loop in Line 3 Alg. 4: for each $x \in [1, n]$, denote by G'_x the graph $\mathbf{G}[x]$ by the end of the iteration of x. Since Line 11 shows that setting the ending timestamp to $r < n$ only happens in iteration $r + 1$, in the iteration of x, no edge in $\mathbf{G}$ has ending timestamp in range $[x, n]$, and thus $\mathbf{G}[x] = \mathbf{G}[n]$. We prove by induction that edge-wise, $G'_x = G_x$ for all $x \in [n]$. Firstly, when $x = 1$, $G'_x = G_x$: both graph have no edges. Assume that for $x = y \in [1, n)$ we have $G'_y = G_y$, we prove below that when $x = y + 1$, $G'_x = G_x$. Denote by G^- the HNSW graph being constructed in Alg. 4, which is G_y before inserting v_x. Line 1, Alg. 2 gets the ANN set S of v_x from G_y. Line 3, Alg. 5 searches the ANN set S' of v_x on $\mathbf{G}[x] = G'_y$. Thus, $S = S'$. Line 2, Alg. 2 prunes set S and Line 5, Alg. 5 prunes S' using the same procedure, after prunning so $S = S'$. Alg. 2 set S as v_x's neighbor in G^- and Line 5, Alg. 5 set S as v_x's neighbor in $\mathbf{G}[n]$. To add edges from S to v_x, Lines 4,6-11, Alg. 4 updates G^- and Lines 3-5, Alg. 2 update $\mathbf{G}[n]$ in the same way. Thus, by the end of iteration x, $\mathbf{G}[n] = \mathbf{G}[x] = G'_x = G^- = G_x$. This completes the induction. Next we show that by the end of iteration of n, $\mathbf{G}[x] = G'_x$ edge-wise. Line 11 indicates that setting the ending timestamp to $r < n$ can only happen in iteration $r + 1$. In other words, G'_x is a subgraph of $\mathbf{G}[x]$. Lines 5-6 show that all the inserted edges in iteration x has starting timestamp x. Thus, the edge set of $\mathbf{G}[x]$ is a subset of that of G'_x, completing the proof. □

Complexity Analysis. Inserting a node to the segment graph creates no more than 2M edges; the total number of edges in a segment graph is no more than $2n$M. In contrast, in an HNSW graph, the outdegree of each node is bounded by M, and the total number of edges in an HNSW graph is no more than nM. The space complexity of both structures is $O(n\text{M})$.

Algorithm 7: 2DSegmentANNSearch($\mathbb{G}, q, [x, y], ep, \mathrm{K}$)

Input: $\mathbb{G}$: 2D segment graph; $[x, y]$: a query range.
Output: *ann*: K approximate nearest neighbors of q in $\mathcal{D}_{x,y}$.
`// replace Line 8, Algorithm 1 with the line below`
1 **foreach** unvisited o with $(l, r, o, b, e) \in \mathbb{G}[v]$ **do if** $x \in [l, r]$ and $y \in [b, e]$ **then**

4 RFANNS with Arbitrary Query Ranges

This section considers RFANNS on arbitrary ranges, i.e., the range can be $[x, y]$ with arbitrary $1 \le x \le y \le n$. A naive method is to build n segment graphs $\mathbf{G}_1, \mathbf{G}_2, \cdots, \mathbf{G}_n$. For each $z \in [1, n]$, $\mathbf{G}_z$ answers half-bounded queries on $\mathcal{D}^z = \{v_z, v_{z+1}, \cdots, v_n\}$. This is sufficient because an arbitrary range $[x, y]$ is a half-bounded range on $\mathcal{D}^x$ and thus we can use Algorithm 6 on $\mathbf{G}_x$ to process the query range $[x, y]$. The main drawback of the naive approach is the quadratic space complexity of $O(n^2\mathrm{M})$, which is excessively large.

4.1 Two Dimensional Segment Graph

We propose a structure called two dimensional (2D) segment graph that can compress the n segment graphs in the naive solution based on the observation below.

Observation 2. *Consider a point $v_i \in \mathcal{D}$. If the triple (v_j, b, e) exists in v_i's neighbor list on a consecutive list of segment graphs $\mathbf{G}_l, \mathbf{G}_{l+1}, \cdots, \mathbf{G}_r$, we can use a single tuple (l, r, v_j, b, e) to show the $r - l + 1$ copies of the triple in these segment graphs. In other words, the two segments $[l, r]$ and $[b, e]$ compress the copies of the edge from v_i to v_j in the segment graphs for* RFANNS *with arbitrary ranges.*

Based on the observation above, we first formally define a 2D segment graph $\mathbb{G}$ and then discuss its properties and construction.

Definition 6. *A* 2D segment graph $\mathbb{G}$ *of $\mathcal{D} = \{v_1, \cdots, v_n\}$ contains n nodes $v_1, v_2, \cdots, v_n$. Each node v_i has an outgoing neighbor list $\mathbb{G}[v_i]$ where each neighbor is represented as a tuple $(l, r, v_j, b, e) \in \mathbb{G}[v_i]$ where $1 \le l \le r \le b \le e \le n$ when triple (v_j, b, e) is in the neighbor list of v_i on each of the segment graphs $\mathbf{G}_l, \mathbf{G}_{l+1}, \cdots, \mathbf{G}_r$.*

Based on Definition 6, Algorithm 7 adapts SegmentANNSearch. To discuss how effectively a 2D segment graph can compress the n segment graphs, we consider the neighbor list of a node v_z in two consecutive segment graphs $\mathbf{G}_i$ and $\mathbf{G}_{i+1}$. There are essentially three steps when a point $v_x, x > z > i$, is inserted (Line 3-11, Algorithm 5) into the two segment graphs respectively. **Step 1** (Line 3) finds a list of nearest neighbors for v_x in $\mathcal{D}_{i,x-1} = \{v_i, v_{i+1}, \cdots, v_{x-1}\}$ and $\mathcal{D}_{i+1,x-1} = \{v_{i+1}, v_{i+2}, \cdots, v_{x-1}\}$, respectively. **Step 2** (Lines 4-6) prunes the nearest neighbors to form the neighbor list of v_x and insert the reverse edges to the two segment graphs, respectively. **Step 3** (Line 4, Lines 7-11) prunes the neighbor list of the neighbors of v_x in both segment graphs if necessary. For the three steps, we make the following two observations.

Lemma 2. *Based on Definition 2, for an integer* K, $\mathrm{kNN}(v_x, [i, x-1], \mathrm{K})$ *denotes the set of* K*-nearest neighbors of v_x in $\mathcal{D}_{i,x-1}$ and $\mathrm{kNN}(v_x, [i+1, x-1], \mathrm{K})$ denotes that in $\mathcal{D}_{i+1,x-1}$. If $v_i \notin \mathrm{kNN}(v_x, [i, x-1], \mathrm{K})$, we have $\mathrm{kNN}(v_x, [i, x-1], \mathrm{K}) = \mathrm{kNN}(v_x, [i+1, x-1], \mathrm{K})$.*

Proof. $\mathcal{D}_{i,x-1} = \{v_i\} \cup \mathcal{D}_{i+1,x-1}$ and thus, $v_i \notin \mathrm{kNN}(v_x, [i, x-1], \mathrm{K})$ means that there are K points in $\mathcal{D}_{i+1,x-1}$ that are closer to v_x than v_i. Based on our assumption that for each point v_x, all other points in $\mathcal{D}$ have different distances to v_x (Section 3.1), $\mathrm{kNN}(v_x, [i, x-1], \mathrm{K}) = \mathrm{kNN}(v_x, [i+1, x-1], \mathrm{K})$. □

Lemma 3. *Let i' be the minimum key of points in the* K*-nearest neighbors $\mathrm{kNN}(v_x, [i, x-1], \mathrm{K})$ in $\mathcal{D}_{i,x-1}$, we have $\mathrm{kNN}(v_x, [i, x-1], \mathrm{K}) = \mathrm{kNN}(v_x, [i+1, x-1], \mathrm{K}) = \cdots = \mathrm{kNN}(v_x, [i', x-1], \mathrm{K})$.*

Algorithm 8: 2DSegmentGraphConstruction

Input: $\mathcal{D} = \{v_1, v_2, \cdots, v_n\}$; K: an integer; M: the max degree.
Output: $\mathbb{G}$: 2D segment graph for $\mathcal{D}$.

1 **foreach** $1 < j \leq n$ **do**
2 $i = 1$;
3 **while** $i < j$ **do**
4 $ann \leftarrow$2DSegmentANNSearch($\mathbb{G}, v_j, [i, j-1], v_i$, K);
5 $i' = \min\{x | v_x \in ann\}$;
6 **foreach** $v \in$ Prune(v_j, ann, M) **do**
7 add (i, i', v, j, n) to $\mathbb{G}[v_j]$;
8 add (i, i', v_j, j, n) to $\mathbb{G}[v]$;
9 $i = i' + 1$;
10 **return** $\mathbb{G}$;

Proof. When $i' = i$, the lemma holds trivially. If $v_i \notin \text{kNN}(v_x, [i{+}1, x{-}1], \text{K})$, then $\text{kNN}(v_x, [i, x{-}1], \text{K}) = \text{kNN}(v_x, [i{+}1, x{-}1], K)$ (Lemma 2). If $v_{i+1} \notin \text{kNN}(v_x, [i{+}1, x{-}1], \text{K})$, then $\text{kNN}(v_x, [i{+}1, x{-}1], \text{K}) = \text{kNN}(v_x, [i{+}2, x{-}1], \text{K}$ (Lemma 2). As none of $v_i, v_{i+1}, \cdots, v_{i'-1}$ is in $\text{kNN}(v_x, [i{+}1, x{-}1], \text{K})$, the lemma can be proved by iteratively applying Lemma 2. □

Lemma 2 shows that if Line 3, Algorithm 5 returns accurate range-filtering nearest neighbors of v_x, then the neighbor lists of v_x in $\mathbf{G}_i$ and $\mathbf{G}_{i+1}$ must be the same, even after pruning (Line 4).

Lemma 4. *On a segment graph* $\mathbf{G}$ *constructed with Algorithm 5 opting out Step 3, in the neighbor list of any node* $v_i \in \mathcal{D}$*, every triple* (v_j, b, e) *must have* $b = \max\{i, j\}$ *and* $e = n$.

Proof. Every edge (v_i, v_j) in $\mathbf{G}$ is inserted upon the insertion of node v_b where $b = \max\{i, j\}$. Since Step 3 is opted out, once an edge is inserted, it will never be deleted, therefore, $e = n$. □

Algorithm 8 constructs a 2D segment graph $\mathbb{G}$ which compresses the n segment graphs $\mathbf{G}_1, \mathbf{G}_2, \cdots, \mathbf{G}_n$ constructed by Algorithm 5 whose Step 3 (Lines 7-11) is opted out. Note that $\mathbb{G}$ is constructed directly without constructing the n segment graphs. Specifically, the points in $\mathcal{D}$ are added into $\mathbb{G}$ sequentially. For the j-th point v_j, it first finds a set ann of approximate nearest neighbors of v_j in $\mathcal{D}_{i,j}$ (Line 4), where $i = 1$ initially (Line 2). Let i' be the smallest key of the points in ann (Line 5). We insert the edges between v_j and the nodes in the pruned ann with 2D segment $[i, i']$ and $[j, n]$ (Lemma 4) in Lines 6-8. After that, we make a "leap" by setting $i = i' + 1$ and repeat the above steps until $i \geq j$.

Lemma 5. *The worst-case size of a 2DSegmentGraph is* $O(n^2\text{M})$. *The worst-case 2DSegmentGraph construction time has two parts: 1)* $O(n^2)$ *calls of online searches (Algorithm 7), and 2) indexing time* $O(n^2\text{M}^2)$ *excluding the online search.*

Proof. For each pair of $i, j \in [n]$, 2DSegmentGraph adds up to $O(\text{M})$ edges (Lines 6-8) after an online search (Line 4) and an $O(\text{M}^2)$ time pruning (Line 6 can be done once for all nodes v). □

Theorem 7. *Under the assumption that the attribute values of points in* $\mathcal{D}$ *are independent*[1]*, the expected number of rounds in Lines 4-9 of Algorithm 8 is* $O(n \log n)$*, and thus in the average-case, the index size of 2DSegmentGraph is* $O(n\text{M} \log n)$ *and index time has two parts: 1)* $O(n \log n)$ *calls of online searches (Algorithm 7), and 2) indexing time* $O(n\text{M}^2 \log n)$ *excluding the online search.*

[1]That is, for any $i \in [1, n]$, each point in $\mathcal{D}$ appears in the i-th position of the ordering of $\mathcal{D}$ with an equal probability.

Proof. Consider the iteration (Line 1) of $j \in [1, n]$. Fix j, we analyze the expected number of rounds $z(i = 1, j)$, i.e., the number of different values i could take from initial value 1 before it reaches j in Line 9. Each value of i calls the online search once (Line 4), performs an $O(\mathrm{M}^2)$ time prunning process (Line 6) and adds $O(\mathrm{M})$ edges to the 2D segment graph (Lines 6-8). Based on our attribute value independence assumption, for each v_x with $x \in [i, j - \mathrm{K}]$, the probability of $v_x \in ann$ is $\frac{\mathrm{K}}{j-i}$; besides, $z(j-\mathrm{K}, j) = \mathrm{K}$. We have $z(i, j) = \frac{\mathrm{K}}{j-i}(z(i+1, j)+1)+(1-\frac{\mathrm{K}}{j-i})z(i+1, j) = z(i+1, j)+\frac{\mathrm{K}}{j-i}$ for every $i \in [1, j - \mathrm{K})$. Solving this recurrence, we have $z(1, j) = \mathrm{K} + \sum_{y\in[\mathrm{K},j-1]} \frac{\mathrm{K}}{y}$ if $j \geq K$; otherwise, i.e., $j < \mathrm{K}$, $z(1, j) = j$. Take K as a constant and denote by $h(y) = \sum_{x\in[1,y]} \frac{1}{y} = O(\ln y)$, we have the expected total number of rounds $\sum_{j\in[1,n]} z(1, j) = O(n + n \cdot h(n)) = O(n \ln n)$. Thus in expectation (average-case), the graph size is $O(n\mathrm{M} \ln n)$, the number of calls of online search is $O(n \log n)$ and the construction time excluding online search is $O(n\mathrm{M}^2 \log n)$, completing the proof. □

Theorem 8. *The 2D segment graph produced by Algorithm 8 is a lossless compression of n segment graphs* $\mathbf{G}_1, \mathbf{G}_2, \cdots, \mathbf{G}_n$ *constructed by Algorithm 5 with Step 3 opted out under an assumption that the procedures SegmentANNSearch and 2DSegmentANNSearch return the exact* K*-nearest neighbors of the query point.*

Proof. If all the ANN searches are exact searches and Step 3 is opted out (no edge will be removed after insertion), then consider an edge e between v_j and $v_{j'}$ with $j' < j$ in the segment graph $\mathbf{G}_{i''}$ with a segment $[b, e]$. We have $v_{j'}$ must be in the pruned kNN list of v_j on $\mathcal{D}_{i'',j-1}$, $b = j$, $e = n$ and $i'' \leq j$. We argue that there must be an edge between v_j and $v_{j'}$ (essentially undirected since Step 3 is opted out) in the 2D segment graph with tuple $(i, i', *, b', e')$ such that $i \leq i'' \leq i'$, $b' = j$ and $e' = n$ and thus the 2D segment graph achieves a lossless compression. Since in inserting v_j in the 2D segment graph, i leaps from 1 to $j - 1$, partitioning $[1, j - 1]$ into concatenated disjoint segments of $[i, i']$ that jointly cover $[1, j - 1]$. Thus, there must exist i and i' during the leap such that $i'' \in [i, i']$. Based on Lemma 3, if *ann* obtained in Line 4, Algorithm 8 is the set of exact nearest neighbors of v_j on $\mathcal{D}_{i,j-1}$, then it contains the top-K exact nearest neighbors of v_j on $\mathcal{D}_{i,j-1}, \mathcal{D}_{i+1,j-1}, \cdots, \mathcal{D}_{i',j-1}$ where i' is the smallest key of the points in *ann*. When constructing $\mathbf{G}_i, \mathbf{G}_{i+1}, \cdots, \mathbf{G}_{i'}$, upon the insertion of v_j, the nearest neighbor sets found for v_j are all *ann*. The pruned *ann* w.r.t. the center node v_j is deterministic. Thus, the neighbor list of v_j to the nodes inserted before v_j is the same (the pruned *ann*) on $\mathbf{G}_i, \mathbf{G}_{i+1}, \cdots, \mathbf{G}_{i'}$. Furthermore, because Step 3 is opted out and all the nodes in *ann* have keys smaller than j, Lemma 4 shows that $[j, n]$ is the segment for all the edges between v_j and pruned *ann* on segment graphs $\mathbf{G}_i, \mathbf{G}_{i+1}, \cdots, \mathbf{G}_{i'}$. Because $v_{j'}$ is in the pruned nearest neighbor list of v_j on set $\mathcal{D}_{i'',j-1}$, it is in the pruned *ann* on the 2D segment graph, proving the compression is lossless. □

Remarks: Due to the approximate nature of ANNS, without the assumption in Theorem 8, which is not likely to be achieved practically, it is hard to envision the relation between 2D segment graph and the n segment graphs. On the other hand, whether the assumption is upheld or not does not affect the index size of 2D segment graph: our experiment Exp-5 in Section 5.2 shows that the index size of 2D segment graph with or without assumption A1 are approximately the same (the difference is less than 2%) on all of the evaluated datasets. The actual efficiency is mainly evaluated with empirical studies following the common practice of ANNS.

Example 9. As shown in Figure 6, consider adding v_9 into the 2D segment graph. Suppose $\mathrm{K} = 3$ and $\mathrm{M} = 2$. The algorithm first finds a set of $\mathrm{K} = 3$ approximate nearest neighbors of v_9 in $\mathcal{D}_{1,8}$, which are v_2, v_5, and v_7. The smallest key of the three points is $i' = 2$. In the neighborhood pruning process, v_2 is dominated and pruned. Thus we add two neighbors $(1, 2, v_5, 9, n)$ and $(1, 2, v_7, 9, n)$ to the neighbor list $\mathbb{G}[v_9]$ and the reverse neighbor $(1, 2, v_9, 9, n)$ to two neighbor lists $\mathbb{G}[v_5]$ and

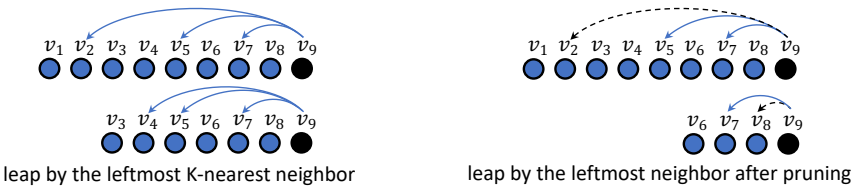


Fig. 6. Examples of leap strategies.

$\mathbb{G}[v_7]$. Then the algorithm sets $i = i' + 1 = 3$ and finds a set of 3 approximate nearest neighbors of v_9 in $\mathcal{D}_{3,8}$, which are v_4, v_5, and v_7. Next, it repeatedly prunes the three neighbors and adds edges to the 2D segment graph.

4.2 Leap Optimizations

This section discusses the techniques for further optimizing the index time and index size based on the following observation.

Observation 3. *Inserting a point into the 2D segment graph necessitates repeated "leaps" from i to $i' + 1$ where i' is determined by the* leftmost *approximate nearest neighbor, i.e., the ann node with the smallest key. Larger step sizes of the leaps lead to a shorter index time and smaller index size.*

We propose two optimizations to increase the step sizes.

Optimization 1: Leap (by setting i') to the leftmost neighbor **after** pruning the approximate nearest neighbors *ann*. The pruned *ann* is a subset of *ann*, and thus, the smallest key in the pruned *ann* is no less than the smallest key in *ann*. The step size is thus increased.

Example 10. As illustrated in Figure 6 on the right. Let the 3-nearest neighbors of v_9 in $\mathcal{D}_{1,8}$ be $\{v_2, v_5, v_7\}$. After neighborhood pruning, the neighbor list of v_9 becomes $\{v_5, v_7\}$. We propose to use $\{v_5, v_7\}$ as the neighbor list of v_9 in each and every consecutive segment graph $\mathbf{G}_1, \mathbf{G}_2, \cdots, \mathbf{G}_{i'=5}$ where $i' = 5$ is determined by the smallest key in the neighbor list after pruning. Then we repeat the process to build the neighbor lists of v_9 in the remaining segment graphs $\mathbf{G}_6, \mathbf{G}_7, \mathbf{G}_8$.

Algorithm 9 shows the optimization. When inserting a point v_j into the 2D segment graph, we first find a set *ann* of approximate nearest neighbors of v_j in $\mathcal{D}_{i,j-1}$ where $i = 1$ initially. Then we prune *ann* to get a neighbor list *nbr* of v_x (Line 1, Algorithm 9). Let i' be the smallest key in *nbr* (Line 2, Algorithm 9). Then, for every $v_x \in$ *ann* where $i \le x < i'$, v_x must be dominated and pruned as $v_{i'}$ is the leftmost neighbor after pruning. We use *nbr* as an "approximate" of the neighbor list of v_j in the segment graphs $\mathbf{G}_i, \mathbf{G}_{i+1}, \cdots, \mathbf{G}_{i'}$. We argue that *nbr* and the neighbor lists of v_j in these segment graphs are highly similar. To see this, consider the exact K-nearest neighbors N and N' of v_x in $\mathcal{D}_{i,j-1}$ and $\mathcal{D}_{i+1,j-1}$ where $v_i \in N$ and v_i is dominated and pruned during neighborhood pruning. We have $N' = N \setminus \{v_i\} \cup \{v\}$ where v must be the K-th nearest neighbor of v_j in $\mathcal{D}_{i+1,j-1}$. For example, as shown in Figure 6 on the left. Consider the 3-nearest neighbors of v_9 in $\mathcal{D}_{2,8}$ and $\mathcal{D}_{3,8}$. We have $N = \{v_2, v_5, v_7\}$ and $N' = \{v_4, v_5, v_7\}$. Then v_4 must be the 3-rd nearest neighbor of v_9 in $\mathcal{D}_{3,8}$.

On the one hand, when pruning the K-nearest neighbors N of v_j, as $v_i \in N$ is dominated and pruned by another neighbor, the neighbor list after pruning N must be identical to the neighbor list after pruning $N \setminus \{v_i\}$. On the other hand, when pruning $N' = N \setminus \{v_i\} \cup \{v\}$, if v is dominated and pruned by another neighbor, the neighbor list after pruning N' must also be identical to the neighbor list after pruning $N \setminus \{v_i\}$. Thus when v is dominated and pruned, the neighbor lists of v_j after pruning in $\mathbf{G}_i$ and $\mathbf{G}_{i+1}$ must be the same. Note that, since v is the K-th nearest neighbor, it is likely to be pruned and dominated by another neighbor as it is tested lastly in Algorithm 3 (Line 2).

Algorithm 9: 2DSegmentGraphConstructionMIN

```
// replace Lines 5-6, Algorithm 8 with lines below
1 nbr ← Prune(v_j, ann, M) ;
2 i' = min{x|v_x ∈ nbr};
3 foreach v ∈ nbr do
```

Optimization 2: Leap to the rightmost neighbor in the pruned *ann* (by setting i' to $\max\{x|v_x \in nbr\}$ in Line 2, Algorithm 9). The step sizes can be dramatically increased by leaping to the rightmost node in the pruned *ann*. While significantly reducing the index time and index size, we may need to apply a minor extension to the ANNSearch over the 2D segment graph to achieve a better recall. Specifically, for query range $[x, y]$, even if a data point v_i does not fall into the query range, i.e., $i < x$ or $i > y$, we can add it to the list-to-explore *pool* (Line 12, Algorithm 1). However, it will never be added to the candidate list *ann*. As a tradeoff, we can also use the median key in *nbr* as i' for the next leap (by setting i' to $\text{median}\{x|v_x \in nbr\}$ in Line 2, Algorithm 9).

4.3 Discussions

Until now, we have assumed that to process a RFANNS query on $\mathcal{D}$, one 2D segment graph on one attribute is used as an index. For certain queries, we may use multiple indices if they exist or one index on multiple attributes. For simplicity, we assume below that each point in $\mathcal{D}$ has two attributes A_1 and A_2; however, our discussions can be easily extended to $\mathcal{D}$ with more attributes.

Using Multiple Single-Attribute Indices. Assume that one index has been established for each attribute using our 2D segment graph. A **conjunctive query** with query point q, integer k, filters $r_1(A_1)$ AND $r_2(A_2)$ where $r_1(A_1)$ is a range on the domain of attribute A_1 and $r_2(A_2)$ that of attribute A_2, aims at reporting k ANN of q among all the points in $\mathcal{D}$ that satisfies both r_1 and r_2. To answer this query, we may select the index of one attribute (e.g., A_1), use its 2D segment graph, and adapt Line 12, Algorithm 1 to "push o to *ann* only if o satisfies query conditions $r_1(A_1)$ AND $r_2(A_2)$" to complete the search. For a **disjunctive query** with conditions $r_1(A_1)$ OR $r_2(A_2)$, use the indexes for A_1 and A_2, resp., to get R_1, the k ANN w.r.t. r_1, and R_2 that to r_2. Report the k ANN to q in the merged set $R_1 \cup R_2$ as the results. Note that when multiple indices are constructed, we can store the points in $\mathcal{D}$ with one copy, which gives each point in $\mathcal{D}$ a unique ID, and build the 2D segment graph as a secondary index for each attribute: the merge of $R_1 \cup R_2$ can be performed in the level of IDs before the distance computation.

Using One Index on Multiple Attributes. For multiple attributes that follow a *lexicographic ordering* in the search, we may formulate a composite attribute, e.g., (A_1, A_2) for two attributes A_1 and A_2. The lexicographic ordering ensures that for two composite attribute values $(b_1, b_2) < (c_1, c_2)$ if $b_1 < c_1$ or ($b_1 = c_1$ and $b_2 < c_2$), which is essentially a total ordering. Thus, building one 2D segment graph on the composite attribute is sufficient for such RFANNS queries.

Limitation 1. The use of our structure in supporting queries on multiple attributes is still primitive. For example, the conjunctive query can only choose one attribute for indexed search, leaving filter on other attributes to the online search. There lies an open question of building multi-dimensional index and developing query optimization techniques for multi-attribute RFANNS queries.

Limitation 2. Our method is closely entangled with HNSW and is suitable for static dataset $\mathcal{D}$ and/or append-only insertions in which the inserted point always has the largest attribute value. Specifically, our 2D segment graph leverages the incremental insertion of HNSW and its heuristic pruning properties. Some other graph-based indexes, such as NSG [11] and DiskANN [21],

are constructed using approximate KNN graphs or random graphs, employing similar pruning techniques. There lies an opportunity to apply our insights to these alternative indexes and coping with other types of updates. We leave these research as our future work.

5 Experiment

Environment. Our methods[2] and the baseline approaches are all implemented in C++ and compiled using GCC 9.2.0 with -O3 optimization. All experiments were run on a server with an Intel(R) Xeon(R) Platinum 8358 CPU@2.60GHz with 64 cores and 256GB of RAM. We ran every experiment multiple times and reported the average result. We used a single thread and disabled SIMD optimization.

Datasets. We used three real-world public datasets. (1) DEEP[3]: each point is a 96-dimensional image feature vector, which is acquired from the last fully-connected layer of the GoogLeNet model [9]. Each point is assigned a random number as the synthetic key. (2) YouTube-Audio: each point is a 128-dimensional audio feature vector of a YouTube video. This dataset is from YouTube8M[4]. For each video, we crawled the corresponding release time and used it as the search key. (3) WIT-Image[5] (Wikipedia-based Image Text Dataset): each point is a 2048-dimensional ResNet-50 embedding of an image from Wikipedia. We obtained the size of the image and used it as the search key. For each dataset, we randomly draw 1 million points to do the experiments by default.

5.1 Evaluating Graph Construction

Exp-1: Segment Graph. We first compare the construction cost of SegmentGraph with that of ordinary HNSW graph. Figure 7 shows index time and size when varying either parameter M (the max outdegree) or K (the number of candidate neighbors), fix the other ones. SegmentGraph and HNSW graph had almost the same index time: when M = 32 and K = 400, it took 1592s and 1572s respectively to build the SegmentGraph and HNSW graph for the DEEP dataset. The extra overhead of building the SegmentGraph is in marking the pruned edges with timestamps. Moreover, we observe the index time grew sublinearly with M and roughly linearly with K. For example, when M increased 8 times from 8 to 64, on the YouTube-Audio dataset, the graph construction time only increased 2.17 times. The reason is that M affects the neighborhood pruning. The first step of edge insertion, finding a set *ann* of K approximate nearest neighbors, is more time-consuming than the second step, neighborhood pruning. The index size of SegmentGraph was only slightly larger than that of the HNSW graph: when M = 64 and K = 400, the index sizes of SegmentGraph and HNSW graph for the YouTube-Audio dataset were 0.62GB and 0.59GB respectively. This is because they had the same node set, while the average outdegrees of SegmentGraph and HNSW graph cannot exceed 2M and M.

Exp-2: 2D Segment Graph. We then evaluate the 2DSegmentGraph construction. We implemented three leap strategies MinLeap, MidLeap, and MaxLeap, which leaps by the leftmost neighbor, the one in the middle, and the rightmost neighbor *after* pruning. As shown in Figure 8, MinLeap always has the largest index size and MaxLeap the smallest. Similarly, MinLeap has the longest index time, while MaxLeap the shortest. For example, on DEEP, when M = 32 and K = 100, it took respectively 4573s, 4988s, and 8981s to build the 2DSegmentGraph. This is because the step size of MaxLeap is large, leading to a smaller number of rounds running Lines 4-9 in Algorithm 8. The impact of M and K on the 2DSegmentGraph was similar to what we observed on the SegmentGraph.

[2]Our source code is publicly available at: https://github.com/rutgers-db/SeRF
[3]https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search
[4]https://research.google.com/youtube8m/download.html
[5]https://github.com/google-research-datasets/wit

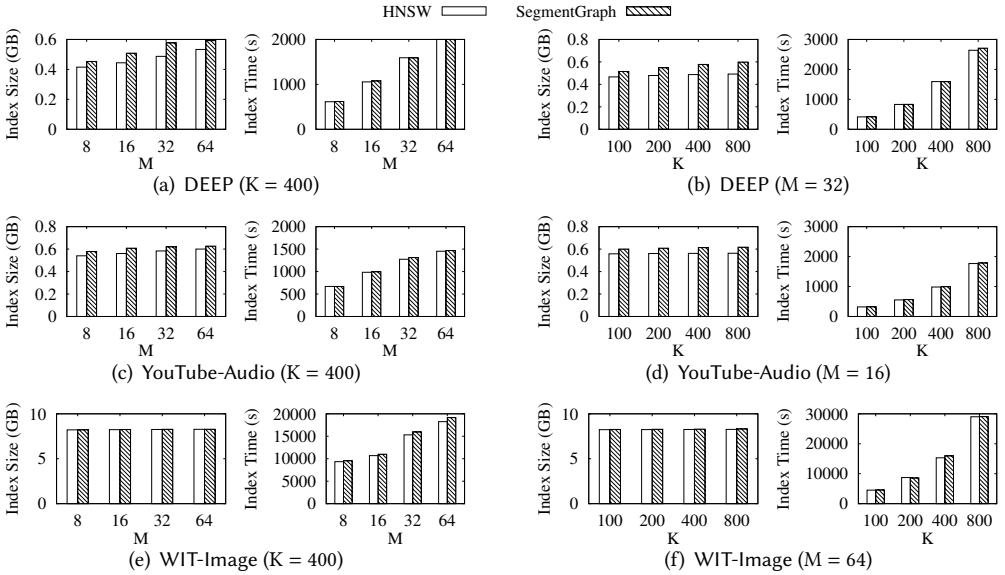


Fig. 7. Evaluating SegmentGraph construction.

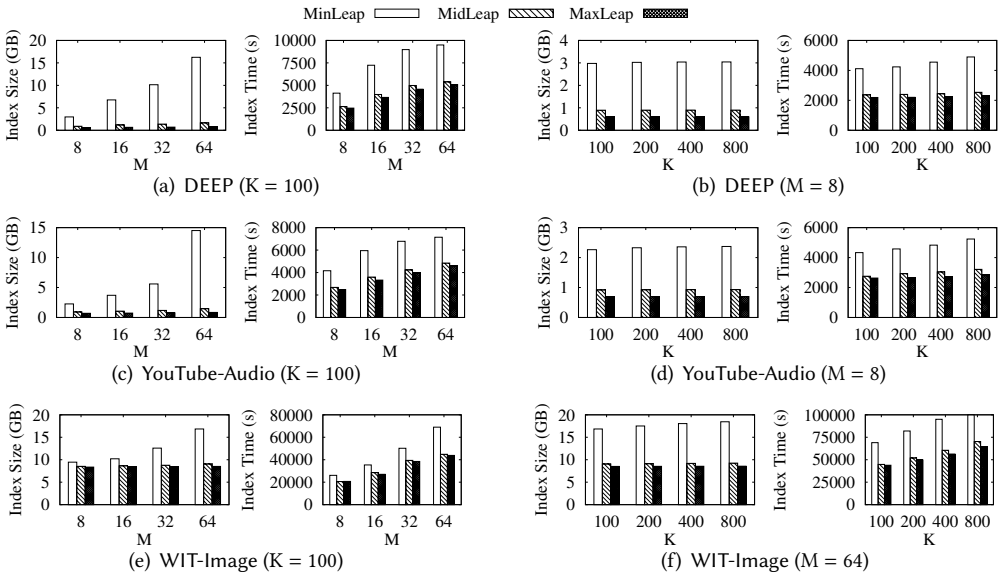
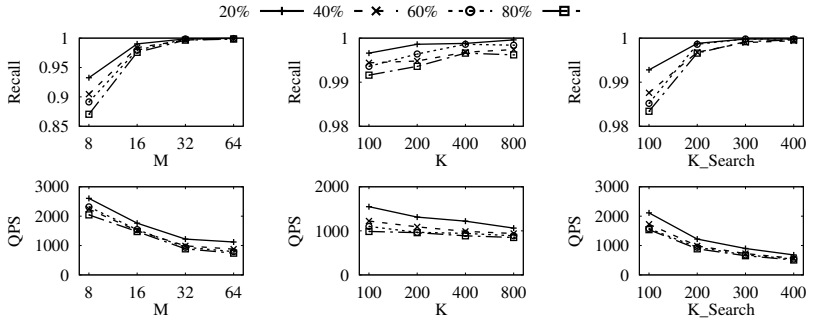


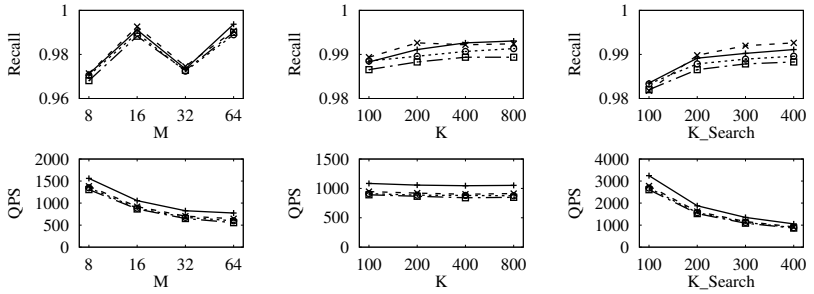
Fig. 8. Evaluating 2DSegmentGraph construction.

5.2 Evaluating Query Processing

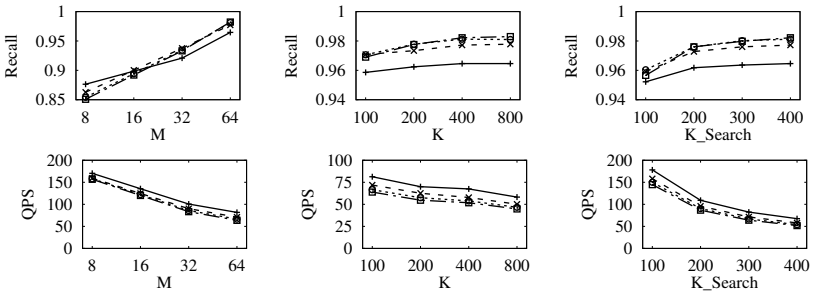
Exp-3: Half-bouned Query Ranges. In this section, we evaluate the impact of the parameters on RFANNS query performance. Note SegmentANNSearch (i.e., Algorithm 6) is used both in index construction and in query processing. To distinguish from its parameter K for index construction, we denote the parameter as K_Search during query processing. Specifically, we varied one of the three parameters M (from 8 to 64), K (from 100 to 400), and K_Search (from 100 to 400), fixed the other two (default parameters are listed in the sub-captions), and report the query-per-second (QPS) and recall. Note that k, the number of approximate nearest neighbors to report, is fixed to



(a) DEEP (M = 32, K = 400, K_Search = 200, unless otherwise specified)



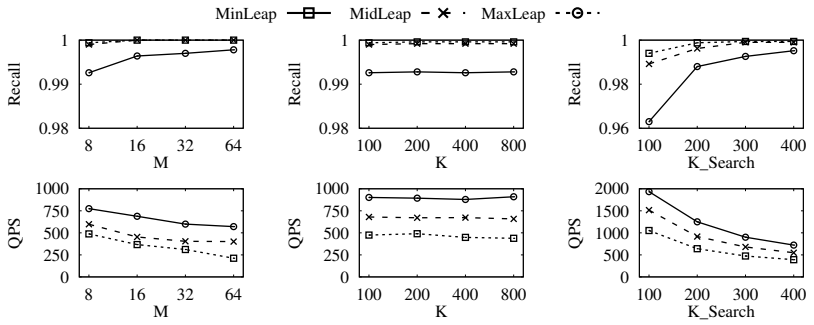
(b) YouTube-Audio (M = 16, K = 400, K_Search = 400, unless otherwise specified)



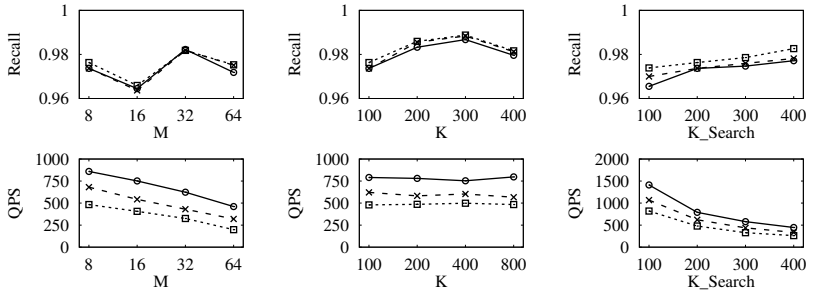
(c) WIT-Image (M = 64, K = 400, K_Search = 400, unless otherwise specified)

Fig. 9. Evaluating SegmentGraph for RFANNS with half-bounded query ranges.

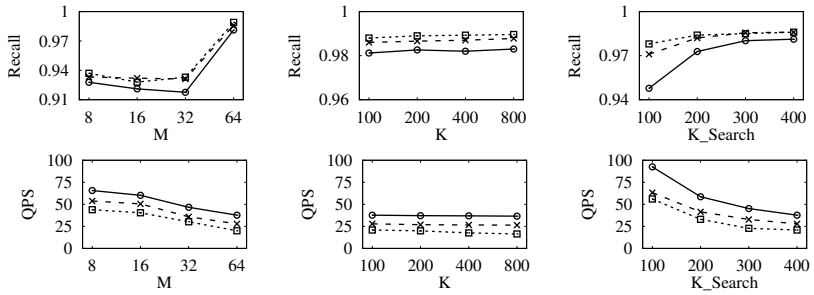
be 10. We randomly selected 500 points outside of the 1 million data points as the query points and equipped each point with a fixed-width query range (x% means the query range includes x% of data points). Figure 9 shows the results. As we can see, with the increase of M, the recall rapidly went up and approached perfect, while the QPS went down sublinearly. For example, when K_Search = 200 and M increased from 8 to 32, on DEEP dataset, at 20% query range width, the recall increased from 0.933 to 0.999, while the QPS decreased from 2606 to 1220. This is because when M increases, the SegmentGraph's average outdegree grows. Thus the SegmentGraph is more connected and the greedy search explores more data points, which benefits recall while hurting QPS. Similarly, with the increase of K, the recall slightly went up and the QPS slightly went down. This is because K had a smaller impact on the average outdegree than M, which is consistent with our previous experimental result as shown in Figure 7. The trends were similar when we increased K_Search. This is because K_Search determines when the greedy search terminates, the larger the more data points in the graph are visited. In addition, the recall and QPS of small query range widths were slightly higher than those of large query range widths. This is because query range width determines how many data points are visited during the greedy search in the SegmentGraph.



(a) DEEP (M = 8, K = 100, K_Search = 300, unless otherwise specified)



(b) YouTube-Audio (M = 8, K = 100, K_Search = 200, unless otherwise specified)



(c) WIT-Image (M = 64, K = 100, K_Search = 400, unless otherwise specified)

Fig. 10. Evaluating 2DSegmentGraph for RFANNS with arbitrary query ranges.

It is relatively easier to search a small number of data points than a large number of data points. Note that the QPS on WIT-Image was significantly lower than that on DEEP and YouTube-Audio. This is because the dimensionalities of the latter two datasets were 10× lower than that of the former dataset and it took a much shorter time to calculate the distance of two points.

Exp-4: Arbitrary Query Ranges. We then evaluate the query performance of the three leap strategies on arbitrary query ranges. We fixed the query range width as 10%. Figure 10 shows the results. We observe that, although the average outdegree of the 2DSegmentGraph constructed by MaxLeap was significantly smaller than that by MinLeap, the query performance of MaxLeap was on par with that of MinLeap. For example, when M = 32, K = 100 and K_Search = 300, on DEEP dataset, the recall of MaxLeap and MinLeap was respectively 0.997 and 1.0, while the QPS of MaxLeap and MinLeap was respectively 599 and 311. This is because MinLeap led a larger average outdegree and thus explored more data points than MaxLeap. In addition, the QPS of MaxLeap was almost always higher than that of MinLeap. This is because MaxLeap has the lowest average outdegree and index size.

Exp-5: Impact of the Assumption in Theorem 8. In this experiment, we replace the approximate nearest neighbor search in Line 4, Algorithm 8 with the brute-force exact nearest neighbor search and denote the constructed graph as exact-2DSegmentGraph. The exact-2DSegmentGraph and 2DSegmentGraph over 100,000 data points in the DEEP dataset had average outdegrees of 1459 and 1451, respectively, and index sizes 0.68GB and 0.67GB, respectively. In addition, they had similar query performances. With a query range width of 50%, both graphs achieved a recall of 0.999. The QPS was 552 for exact-2DSegmentGraph and 668 for 2DSegmentGraph.

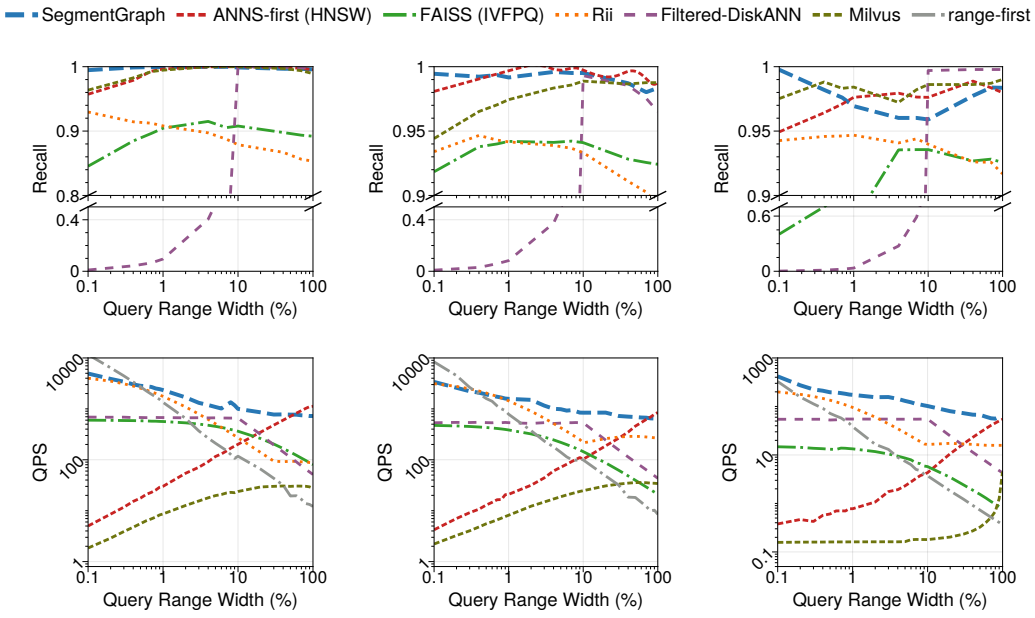
5.3 Comparing with Existing Methods

This section compares our proposed techniques with the following six baseline approaches. (1) ANNS-first: we adapted the HNSW implementation in nmslib[6] for RFANNS as discussed in Section 1. (2) range-first linearly scans the data points that satisfy the query range condition. (3) FAISS[7] is a popular library for ANNS. It has a function IDSeletorRange that supports RFANNS. Specifically, using IVFPQ, the function ignores the data point if its key is outside the query range. (4) Rii scans all the PQ codes in the query range if the query range width is within a threshold; otherwise, it filters the PQ codes using a traditional inverted index before checking the range condition [31]. We used the authors' implementation[8]. (5) Filtered-DiskANN[9] is designed for categorical attribute filtering ANNS [14]. In this scenario, vectors are assigned with label sets. It finds approximate nearest neighbors whose label sets match the query labels. We adapted Filtered-DiskANN for range-filtering ANNS by evenly dividing the keys into 10 buckets and assigning a label to each bucket. The buckets overlapping with the query range become query labels. (6) Milvus[10] [40] is a vector database that supports attribute-filtering ANNS. We used HNSW as its index.

We use grid search to find the optimal parameters. Specifically, In Rii and FAISS, the number of subspaces was 48 for DEEP, 64 for YouTube-Audio, and 1024 for WIT-Image, respectively. The number of centroids in each subspace was set to 256 for all datasets. In addition, Rii has a search parameter L, which trades off search speed and accuracy. It was 64000, 16000, and 64000 in DEEP, YouTube-Audio, and WIT-Image. The number of coarse centroids in FAISS was fixed as 256 for all datasets, while the number of partitions to probe was fixed as 40. ANNS-first, Filtered-DiskANN, and Milvus are graph-based methods, which have 3 common parameters: K, M and K_Search. We set K = 400 in all these methods for all datasets. All three methods had the same M, which was 16 for DEEP, 16 for YouTube-Audio, and 64 for WIT-Image. In ANNS-first, K_Search was set to 200, 300, and 400 for DEEP, YouTube-Audio, and WIT-Image respectively. In Milvus, K_Search was 100 for all datasets. In Filtered-DiskANN, we set K_Search as 200, the number of buckets as 10, and the neighbor pruning threshold α as 1.2 for all datasets. The parameters in the baselines were the same for half-bounded and arbitrary ranges.

Exp-6: Half-bounded Query Range. We first evaluate the half-bounded query range. We varied the query range width from 0.1% to 100% and reported the recall and QPS. Figure 11 shows the results. Note that the recall of range-first is ignored as it is always perfect (i.e., 1.0) due to the linear scan. In almost all query range widths, our SegmentGraph not only achieved the highest recall but also the highest QPS. For example, on DEEP dataset, with 20% query range width, the recall of SegmentGraph, ANNS-first,FAISS, Rii, Filtered-DiskANN, Milvus, range-first were respectively 0.999, 0.999, 0.902, 0.870, 0.999, 0.999, 1.0, while the QPS were respectively 855, 348, 261, 139, 314, 29, 65. SegmentGraph's high query performance is attributed to its ability to losslessly compress

[6]https://github.com/nmslib/hnswlib
[7]https://github.com/facebookresearch/faiss
[8]https://github.com/matsui528/rii
[9]https://github.com/microsoft/DiskANN
[10]https://milvus.io/docs/v2.0.x



(a) DEEP
M = 32, K = 400, K_Search = 200

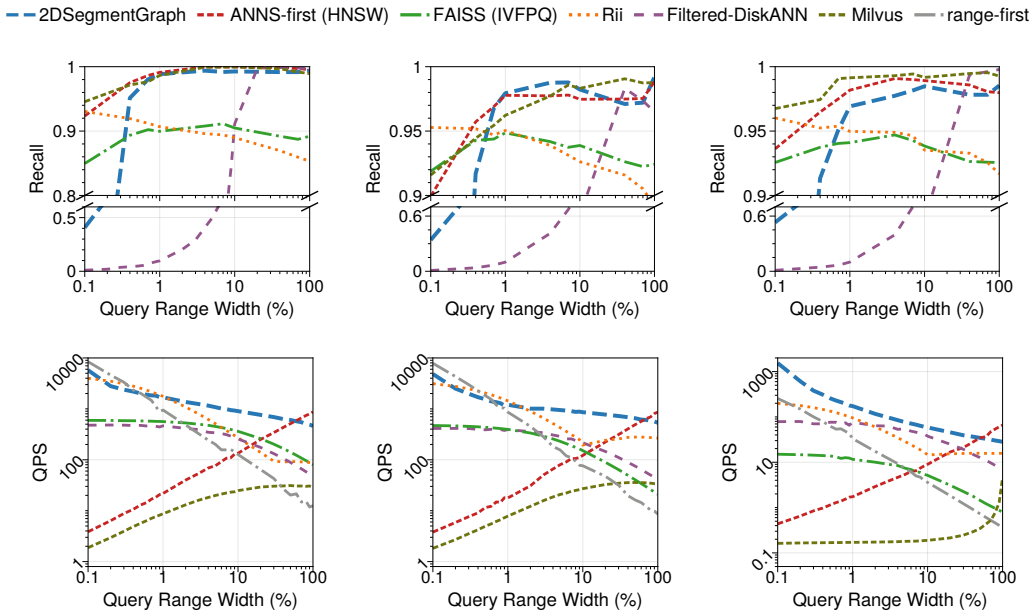
(b) YouTube-Audio
M = 16, K = 400, K_Search = 400

(c) WIT-Image
M = 64, K = 400, K_Search = 400

Fig. 11. Comparing SegmentGraph with existing methods for RFANNS with half-bounded ranges.

multiple HNSW graphs, making its search operation equivalent to a greedy search on the HNSW graph over data points in the query range. Moreover, when the query range width was very large ($\geq$ 90%), SegmentGraph was on par with ANNS-first. With the decrease of the query range width, the QPS of ANNS-first went down, while the QPS of the other methods (excluding Milvus) went up. The reason is that ANNS-first searches in the HNSW graph over all the data points, while the other methods, including SegmentGraph, focus solely on the data points falling within the query range. Milvus behaved similarly to ANNS-first, as it employs HNSW as the internal search index and utilizes a bitset to restrict results during the search process. It is worth noting that Milvus, with its system overhead and optimization, attained high recall at the expense of search speed. Conversely, when the query range width was exceedingly narrow ($\leq$ 0.5%), range-first's QPS outperformed other methods, except SegmentGraph and Rii. This is because Rii is equipped with a cost model that favors linear scanning when the query range width is extremely small. Additionally, graph-based methods (ANNS-first, SegmentGraph, and Milvus) consistently achieved higher recall than PQ-based methods (FAISS and Rii), primarily owing to the lossy nature of PQ compression. Besides, Filtered-DiskANN can only attain high recall when the query range width is large. When the query's range width exceeded 10%, Filtered-DiskANN needed to search multiple buckets, which led to a reduction in QPS.

The index size of the PQ-based methods (i.e., Rii and FAISS) depends on the code length. Rii's index sizes for DEEP, YouTube-Audio, and WIT-Image were respectively 0.20GB, 0.28GB, and 4.13GB, while the corresponding indexing time were 702s, 340s, and 6345s. FAISS's index sizes were 0.20GB, 0.28GB, and 4.23GB for DEEP, YouTube-Audio, and WIT-Image, while the indexing time were 955s, 970s, and 15977s, resp.. The index size of graph-based methods (i.e., SegmentGraph,ANNS-first, Filtered-DiskANN and Milvus) was dominated by the size of raw data because it needs to load all the raw vectors into memory, which is usually much larger than the edge set size. SegmentGraph's index sizes were 0.58GB, 0.61GB, and 8.30GB, while it took 1572s, 563s, and 18943s to construct the



(a) DEEP
M = 8, K = 100, K_Search = 300

(b) YouTube-Audio
M = 8, K = 100, K_Search = 200

(c) WIT-Image
M = 64, K = 100, K_Search = 400

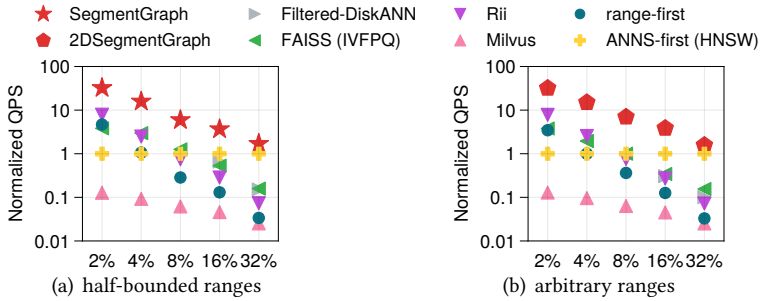
Fig. 12. Comparing 2DSegmentGraph with existing methods for RFANNS with arbitrary ranges.

index. ANNS-first's index sizes for DEEP, YouTube-Audio, and WIT-Image were 0.49GB, 0.56GB, and 8.29GB, while the indexing time were respectively 1562s, 983s, and 18262s. Filtered-DiskANN's index sizes were 0.91GB, 0.75GB, and 8.30GB for these three datasets, while the indexing time were respectively 313s, 217s, and 2905s using 24 threads. Milvus's index sizes were 1.79GB, 2.27GB, and 10.32GB, and the indexing time were respectively 4337s, 2046s, and 24488s.

Exp-7: Arbitrary Query Ranges. Next, we compare 2DSegmentGraph with the baseline approaches on RFANNS with arbitrary query ranges. Note that here we use the leap strategy MaxLeap to construct the 2DSegmentGraph as it leads to the smallest index size. Figure 12 shows the results. When the query range width was very small (< 1%), our 2DSegmentGraph had a decrease in recall with the decrement of query range width. But at this scale of range, the linear scan method range-first would have the best QPS comparing to others. For middle and large scale query range width (> 1%), 2DSegmentGraph had higher QPS than all baselines, while the recall was among the highest ones. For example, on YouTube-Audio dataset, for 20% query range width, the recall of 2DSegmentGraph, ANNS-first, FAISS, Rii, Filtered-DiskANN, Milvus and range-first were respectively 0.974, 0.959, 0.931, 0.912, 0.985, 0.990 and 1, while the QPS were respectively 530, 313, 64, 257, 110, 35 and 32. This is because 2DSegmentGraph can reconstruct HNSW graphs within arbitrary ranges, which is efficient for RFANNS, especially when the query range width is in the middle scale. 2DSegmentGraph's index sizes were 0.64GB, 0.74GB, and 8.51GB for the three datasets, while the index time was 2469s, 2487s, and 43851s.

The right most points in Figures 11 and 12 show the performance of unfiltered queries (i.e., 100% query range width). Our methods SegmentGraph and 2DSegmentGraph had very similar recall and QPS as HNSW based method (i.e., ANNS-first), while significantly outperforming other baselines.

Exp-8: Quantification Evaluation. To quantify the contributions to the performance gain of different sub-solutions, for each method, we find 8 settings in a grid search under which the query



(a) half-bounded ranges (b) arbitrary ranges

Fig. 13. The ratio of QPS of RFANNS over that of ANNS-first, varying query range width (recall ≥ 90%).

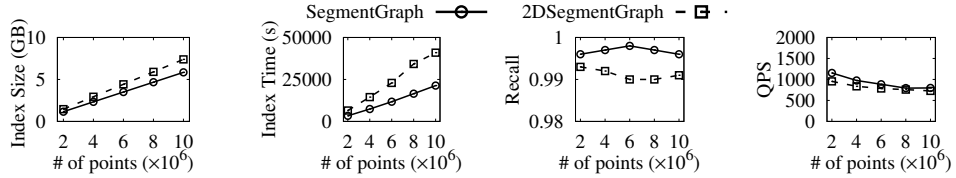


Fig. 14. Evaluating the scalability of SegmentGraph and 2DSegmentGraph on DEEP by varying the number of points from 2 million to 10 million (M = 32, K = 400, K_Search = 200).

recall is ≥ 90%, pick the best QPS, and show, in Figure 13, the ratio between the QPS of each method over the QPS of ANNS-first (HNSW-based method) on 5 query ranges. A method is not displayed if it cannot satisfy the recall requirement. We can see that the ratio of the QPSs of SegmentGraph and 2DSegmentGraph are constantly more than 1 and are up to 32.3 – these gains are contributed by our approach. Furthermore, the HNSW-based method (i.e., ANNS-first) outperformed all the other methods only on ranges larger than 8% while SegmentGraph and 2DSegmentGraph constantly outperformed all the other methods on all the ranges.

5.4 Scalability

We evaluate the scalability of our approach by varying the number of points from 2 million to 10 million on DEEP. We fixed the parameters M = 32, K = 400, and K_Search = 200 and used 10% query range width. Figure 14 shows the results. As we can see, with the increase of the dataset size, the index size and index time increased almost linearly. For example, SegmentGraph took 3340s, 7377s, 11749s, 16512s, and 21354s to build the index for 2M, 4K, 6M, 8M, and 10M data points. This is attributed to the neighborhood pruning process. As for the query performance, the recall of SegmentGraph and 2DSegmentGraph remained above 0.99 for all dataset sizes, while the QPS declined sublinearly with the increase of dataset size. For example, when the number of points increased from 2M to 10M, the QPS of SegmentGraph decreased from 835 to 719, while the QPS of 2DSegmentGraph declined from 541 to 417. This is because HNSW can achieve a high recall and QPS in ANNS and our index is a compression of multiple HNSW indexes. In summary, our approach achieved good scalability both in index construction and in query processing.

6 Related Work

Approximate Nearest Neighbor Search. Existing methods for approximate nearest neighbor search (ANNS) can be broadly classified into three categories: locality-sensitive hashing (LSH) [5, 12, 18, 19, 28, 38], product quantization [6, 13, 22, 23, 32, 43], and proximity graphs [7, 11, 17, 20, 24, 29, 30]. The seminal work by Indyk and Motwani [19] introduced LSH, where specific hash functions are designed to ensure that nearby vectors are likely to be mapped to the same bucket. Different

variations and improvements of LSH have since been proposed, including Multi-Probe LSH [28], Collision Counting LSH [12], and Query-Aware LSH [18]. Product quantization (PQ) is another popular technique for ANNS that offers a memory-efficient representation of high-dimensional vectors. It involves partitioning the vector space into subspaces and quantizing each subspace independently. Jegou et al. first introduce PQ as a way to effectively compress vectors and achieve fast search [22]. Extensions and refinements, such as OptimizedPQ [13] and PQFastScan [6], have further improved the accuracy and efficiency of product quantization-based methods. Proximity graphs provide a graph-based representation of the dataset, where nodes represent vectors, and edges encode proximity relationships. Various graph-based methods have been proposed for ANNS. The Delaunay graph [8] guarantees that for any vector, at least one of its neighbors is closer to the query. Randomized neighborhood graphs [7] offer efficient search complexity while ensuring a certain level of accuracy. Hierarchical Navigable Small World (HNSW) graphs [30] use a hierarchical structure to facilitate efficient ANNS.

Attribute-Filtering Approximate Nearest Neighbor Search. Different systems use various terms to refer to the attributed-filtering ANNS. Specifically, Ferhatosmanoglu et al. [10] study the constrained nearest neighbor queries over multi-dimensional data. It is orthogonal to our work as we focus on high-dimensional vectors. AnalyticDB-V introduces the "hybrid query", which incorporates structured attributes and vectors in an analytic database system [44]. It proposes a cost model to optimize the hybrid query. Milvus introduces the attribute-filtering ANNS and extends the query plans developed in AnalyticDB-V by introducing a partition-based query plan [40]. Matsui et al. present the "subset query" and develop the reconfigurable inverted index (Rii) to process it, employing different strategies based on the size of the query subset [31]. Mohoney et al. explore hybrid vector similarity search, combining vector similarity queries with predicates over relational attributes [34]. The proposed system enables efficient batch processing of offline hybrid query workloads through workload-aware vector data partitioning and multi-query optimization techniques. Zhao et al. investigate the constrained similarity search problem and propose three optimization strategies: starting point selection, multi-direction search, and biased priority queue selection [46]. Both NHQ [41] and HQANN [45] propose to design a fusion distance metric to support the hybrid query with unstructured and structured constraints. Gollapudi et al. propose to study ANNS with filters on disk [14]. ARKGraph [48] proposes an index to retrieve the k-nearest-neighbor graph among the given query range.

7 Conclusion

This paper studies the RFANNS query with a querying vector and a range of attribute values. Existing solutions are either ANNS-first or range-first, which are inefficient for small query ranges and large query ranges, respectively. This paper proposes a novel index that can efficiently answer ANNS for every possible query range. The index size and index time break the quadratic barrier in average-case analysis. Experimental results show that the proposed index outperforms existing methods for both small and large query ranges while maintaining a small index size. To handle updates, we can adopt a process that makes deleted vectors dummy and rebuilds the index periodically. Our future work will focus on developing more efficient update methods.

Acknowledgments

We thank all anonymous reviewers for their valuable comments. Dong Deng was supported by the National Science Foundation grant #2212629. Qiao Miao was supported by Marsden Fund UOA1732, and MBIE Catalyst: Strategic Fund NZ-Singapore Data Science Research Programme UOAX2001.

References

[1] [n. d.]. Lucene 9.0.0. https://lucene.apache.org/core/corenews.html#apache-lucenetm-900-available.
[2] [n. d.]. Pinecone.io. https://www.pinecone.io/.
[3] [n. d.]. Weaviate.io. https://weaviate.io/developers/weaviate/concepts/vector-index.
[4] [n. d.]. Zilliz. https://zilliz.com/.
[5] Alexandr Andoni and Piotr Indyk. 2006. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *FOCS*. 459–468.
[6] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache locality is not enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan. *PVLDB* 9, 4 (2015), 288–299.
[7] Sunil Arya and David M. Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *ACM/SIGACT-SIAM*. 271–280.
[8] Franz Aurenhammer. 1991. Voronoi Diagrams - A Survey of a Fundamental Geometric Data Structure. *ACM Comput. Surv.* 23, 3 (1991), 345–405.
[9] Artem Babenko and Victor S. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *CVPR*. 2055–2063. https://doi.org/10.1109/CVPR.2016.226
[10] Hakan Ferhatosmanoglu, Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. 2001. Constrained Nearest Neighbor Queries. In *SSTD*, Vol. 2121. 257–278. https://doi.org/10.1007/3-540-47724-1_14
[11] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *PVLDB* 12, 5 (2019), 461–474.
[12] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*. 541–552.
[13] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. 2946–2953.
[14] 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 *WWW*. 3406–3416.
[15] Patrick Grother, Patrick Grother, Mei Ngan, and Kayee Hanaoka. 2019. *Face recognition vendor test (FRVT) part 2: identification*. US Department of Commerce, National Institute of Standards and Technology.
[16] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *KDD*. 855–864. https://doi.org/10.1145/2939672.2939754
[17] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *CVPR*. 5713–5722.
[18] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *PVLDB* 9, 1 (2015), 1–12.
[19] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. 604–613.
[20] Jerzy W. Jaromczyk and Godfried T. Toussaint. 1992. Relative neighborhood graphs and their relatives. *Proc. IEEE* 80, 9 (1992), 1502–1517. https://doi.org/10.1109/5.163414
[21] 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. In *NeurIPS*, Vol. 32.
[22] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *TPAMI* 33, 1 (2011), 117–128.
[23] Yannis Kalantidis and Yannis Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. 2329–2336.
[24] Jon M. Kleinberg. 2000. Navigation in a small world. *Nature* 406, 6798 (2000), 845–845.
[25] Quoc V. Le and Tomás Mikolov. 2014. Distributed Representations of Sentences and Documents. In *ICML*, Vol. 32. 1188–1196. http://proceedings.mlr.press/v32/le14.html
[26] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *TKDE* 32, 8 (2020), 1475–1488. https://doi.org/10.1109/TKDE.2019.2909204
[27] Xinchen Liu, Wu Liu, Huadong Ma, and Huiyuan Fu. 2016. Large-scale vehicle re-identification in urban surveillance videos. In *ICME*. 1–6. https://doi.org/10.1109/ICME.2016.7553002
[28] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *PVLDB*. 950–961.
[29] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.* 45 (2014), 61–68.
[30] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *TPAMI* 42, 4 (2018), 824–836.

[31] Yusuke Matsui, Ryota Hinami, and Shin'ichi Satoh. 2018. Reconfigurable Inverted Index. In *MM*. 1715–1723. https://doi.org/10.1145/3240508.3240630
[32] Yusuke Matsui, Toshihiko Yamasaki, and Kiyoharu Aizawa. 2015. PQTable: Fast Exact Asymmetric Distance Neighbor Search for Product Quantization Using Hash Tables. In *ICCV*. 1940–1948.
[33] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NeurIPS*. 3111–3119.
[34] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-Throughput Vector Similarity Search in Knowledge Graphs. *SIGMOD* 1, 2 (2023), 1–25.
[35] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *EMNLP*. 1532–1543.
[36] Stuart J. Russell and Peter Norvig. 2003. *Artificial intelligence - a modern approach, 2nd Edition.* Prentice Hall.
[37] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, Patrick Schramowski, Srivatsa Kundurthy, Katherine Crowson, Ludwig Schmidt, Robert Kaczmarczyk, and Jenia Jitsev. 2022. LAION-5B: An open large-scale dataset for training next generation image-text models. arXiv:2210.08402 [cs.CV]
[38] Anshumali Shrivastava and Ping Li. 2014. Asymmetric LSH (ALSH) for Sublinear Time Maximum Inner Product Search (MIPS). In *NeurIPS*. 2321–2329.
[39] Ján Suchal and Pavol Návrat. 2010. Full Text Search Engine as Scalable k-Nearest Neighbor Recommendation System *(IFIP Advances in Information and Communication Technology, Vol. 331)*. 165–173. https://doi.org/10.1007/978-3-642-15286-3_16
[40] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD*. ACM, 2614–2627. https://doi.org/10.1145/3448016.3457550
[41] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2022. Navigable Proximity Graph-Driven Native Hybrid Queries with Structured and Unstructured Constraints. arXiv:2203.13601 [cs.DB]
[42] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *PVLDB* 14, 11 (2021), 1964–1978. https://doi.org/10.14778/3476249.3476255
[43] Runhui Wang and Dong Deng. 2020. DeltaPQ: Lossless Product Quantization Code Compression for High Dimensional Similarity Search. *PVLDB* 13, 13 (2020), 3603–3616. https://doi.org/10.14778/3424573.3424580
[44] 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. *PVLDB* 13, 12 (2020), 3152–3165. https://doi.org/10.14778/3415478.3415541
[45] 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 *CIKM*. 4580–4584. https://doi.org/10.1145/3511808.3557610
[46] Weijie Zhao, Shulong Tan, and Ping Li. 2022. Constrained Approximate Similarity Search on Proximity Graph. *CoRR* abs/2210.14958 (2022). https://doi.org/10.48550/ARXIV.2210.14958 arXiv:2210.14958
[47] Liang Zheng, Liyue Shen, Lu Tian, Shengjin Wang, Jingdong Wang, and Qi Tian. 2015. Scalable Person Re-identification: A Benchmark. In *CVPR*. 1116–1124. https://doi.org/10.1109/ICCV.2015.133
[48] Chaoji Zuo and Dong Deng. 2023. ARKGraph: All-Range Approximate K-Nearest-Neighbor Graph. *PVLDB* 16, 10 (2023), 2645–2658. https://doi.org/10.14778/3603581.3603601

Received July 2023; revised October 2023; accepted November 2023