



# Dynamic Range-Filtering Approximate Nearest Neighbor Search

Zhencan Peng\*  
Rutgers University  
zhencan.peng@rutgers.edu

Miao Qiao  
The University of Auckland  
miao.qiao@auckland.ac.nz

Wenchao Zhou  
Alibaba Group  
zwc231487@alibaba-inc.com

Feifei Li  
Alibaba Group  
lifeifei@alibaba-inc.com

Dong Deng†  
Rutgers University  
dong.deng@rutgers.edu

## ABSTRACT

Range-filtering approximate nearest neighbor search (RFANNS) has gained significant attention recently. Consider a set  $\mathcal{D}$  of high-dimensional vectors, each associated with a numeric attribute value, e.g., price or timestamp. An RFANNS query consists of a query vector  $q$  and a query range, reporting the approximate nearest neighbors of  $q$  among data vectors whose attributes fall in the query range. Existing work on RFANNS only considers a static set  $\mathcal{D}$  of data vectors while in many real-world scenarios, vectors arrive in the system in an arbitrary order. This paper studies dynamic RFANNS where both data vectors and queries arrive in a mixed stream: a query is posed on all the data vectors that have already arrived in the system. Existing work on RFANNS is difficult to be extended to the streaming setting as they construct the index in the order of the attribute values while the vectors arrive in the system in an arbitrary order. The main challenge to the dynamic RFANNS lies in the difference between the two orders. A naive approach to RFANNS maintains multiple hierarchical navigable small-world (HNSW) graphs, one for each of the  $O(|\mathcal{D}|^2)$  possible query ranges – too expensive to construct and maintain. To design an index structure that can integrate new data vectors with a low index size increment for efficient and effective query processing, we propose a structure called *dynamic segment graph*. It compresses the set of HNSW graphs of the naive approach, proven to be lossless under certain conditions, with only a linear to  $\log |\mathcal{D}|$  new edges in expectation when inserting a new vector. This dramatically reduces the index size while largely preserving the search performance. We further propose heuristics to significantly reduce the index cost of our dynamic segment graph in practice. Extensive experimental results show that our approach outperforms existing methods for static RFANNS and is scalable in handling dynamic RFANNS.

## PVLDB Reference Format:

Zhencan Peng, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng.  
Dynamic Range-Filtering Approximate Nearest Neighbor Search. PVLDB, 18(10): 3256 - 3268, 2025.  
doi:10.14778/3748191.3748193

\*This author takes part in this work as a research intern at Alibaba Cloud.

†Corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.  
doi:10.14778/3748191.3748193

## PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at [https://github.com/rutgers-db/DynamicSegmentGraph/tree/release\\_version](https://github.com/rutgers-db/DynamicSegmentGraph/tree/release_version).

## 1 INTRODUCTION

Consider a collection  $\mathcal{D}$  of high-dimensional data vectors (or data points), each carrying an attribute with a total order. A range-filtering approximate nearest neighbor search (RFANNS) query consists of a query vector  $q$  and a query range. It finds the approximate nearest neighbors of  $q$  among all the data vectors whose attribute values fall in the query range. RFANNS has applications in vector databases [30], retrieval-augmented generation [19], document retrieval [20], and person or vehicle re-identification [38]. Two straightforward approaches for RFANNS, pre-filtering and post-filtering, do not work well when query range size shifts [40]. To address this issue, specialized index structures have been proposed recently, including SeRF [40], iRange [36], and WinFilter [5] on a static set  $\mathcal{D}$ . In other words, they require sorting vectors in  $\mathcal{D}$  in the order of their attribute values before constructing the index.

In many real-world scenarios, however, data vectors stream into the system in an arbitrary order of their attribute values. For example, on e-commerce platforms where products are represented as high-dimensional vectors, products are often searched with a price filter. In this case, new products with varying prices are constantly added, necessitating effective updates to the index for RFANNS. This paper studies the *dynamic range-filtering approximate nearest neighbor search* problem. Specifically, the data vectors and RFANNS queries mix in a stream where each RFANNS query is performed over all data vectors that have arrived before the query is posed.

Existing methods are designed for static datasets, i.e., they need the data vectors to be sorted by their attribute values before indexing. Thus, they cannot effectively handle new data vectors with an arbitrary attribute value. Specifically, both iRange and WinFilter build a segment tree over the attribute values of all data vectors. For each tree node in the segment tree, a graph-based approximate nearest neighbor search (ANNS) index (such as the de facto state-of-the-art hierarchical navigable small world (HNSW) graph [23]) is created. When a query arrives, WinFilter performs ANNS over a few segments (i.e., nodes) covered by or overlapped with the query range. The approximate nearest neighbors in each segment are merged to produce the final result. In contrast, iRange merges the indexes in these segments on-the-fly and performs a single ANNS over the merged index to find the results. They cannot handle new data vectors as they need to know all attribute values beforehand to build the segment tree and graph-based indexes. SegmentGraph is constructed incrementally, by inserting the data

vectors to the graph one by one, in the order of their attribute values. Thus SegmentGraph functions well when the attribute values of the data vectors are monotonically increasing/decreasing with their arriving time. Nevertheless, SeRF is unable to manage dynamic RFANNS when the attribute values of data vectors are unrelated to their arriving time.

A simple way to handle dynamic RFANNS is to build an HNSW graph for every possible range of attribute values pairs of  $\mathcal{D}$ . However, as new data vectors arrive, the number of HNSW graphs and the number of new edges grow quadratically to  $|\mathcal{D}|$ . We observe that the neighboring ranges often share similar edges in their HNSW graphs, allowing effective compression. To formalize this insight, we introduce *rectangle tree*, a structure that answers three key questions: How can we compactly represent all query ranges affected by a new data point? Is this representation canonical? And how complex is it to manage the representation? The rectangle tree provides a clear framework for efficiently organizing and merging overlapping ranges. Building on this, we propose the *dynamic segment graph*  $G$ , where each data vector is a node, and edges between nodes are labeled with rectangles which specify the query ranges for which the corresponding edges can be used for approximate nearest neighbor search. This design ensures efficient updates, i.e., the expected number of new edges upon the insertion of a new vector is linear to  $\log |\mathcal{D}|$  as opposed to  $|\mathcal{D}|^2$ . The design also largely maintains query performance under the compression. We also introduce optimizations to reduce storage costs while preserving search performance as new vector arrives.

In summary, we make the following contributions in this paper.

- To the best of our knowledge, this paper is the first study on dynamic range-filtering approximate nearest neighbor search.
- We design a dynamic segment graph structure to address the dynamic RFANNS problem. We prove the dynamic segment graph is a lossless compression of many HNSW graphs and analyze the time and space complexity of the dynamic segment graph.
- We design a few optimizations to significantly reduce the index cost of dynamic segment graph in practice.
- We conduct extensive experiments: our method significantly outperforms existing methods for both static and dynamic RFANNS.

## 2 PRELIMINARY

### 2.1 Problem Definition

Consider  $d$ -dimensional space of  $\mathbb{R}^d$  with a distance metric  $\delta$ , i.e., for any two points (vectors)  $u, v \in \mathbb{R}^d$ , their distance is  $\delta(u, v) \geq 0$ .

**DEFINITION 1 (NEAREST NEIGHBOR SEARCH).** Given a query vector  $q \in \mathbb{R}^d$ , an integer  $k > 0$ , and a set  $\mathcal{D}$  of vectors in  $\mathbb{R}^d$ , the  $k$ -nearest neighbors of  $q$ , denoted as  $\text{kNN}_\delta(q, \mathcal{D})$ , is the set of  $k$  vectors in  $\mathcal{D}$  with the smallest distances to  $q$  under metric  $\delta$ . Formally,  $\text{kNN}_\delta(q, \mathcal{D})$  is a set  $\mathcal{R} \subseteq \mathcal{D}$  of  $k$  vectors in  $\mathcal{D}$  such that  $\forall u \in \mathcal{R}$  and  $\forall v \in \mathcal{D} \setminus \mathcal{R}$ ,  $\delta(u, q) \leq \delta(v, q)$ .

We omit the subscription  $\delta$  when the context is clear.

Let  $A$  be an attribute whose domain  $\text{Dom}(A)$  has a *total order*, i.e., operator  $<$  exists between any pair of attribute values. A vector  $v \in \mathbb{R}^d$  associated with an  $A$ -attribute value  $\text{att}(v) \in \text{Dom}(A)$  is called an  $A$ -attributed vector, or *attributed vector* when the attribute  $A$  is clear in the context. For the simplicity of our discussion, we

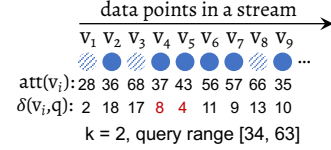


Figure 1: An example of DRFANNS.

introduce  $-\infty_A$  as a placeholder that is smaller than any attribute value in  $\text{Dom}(A)$  and  $+\infty_A$  a placeholder that is larger than any value in  $\text{Dom}(A)$ . Define  $-\infty_A < +\infty_A$  and use  $(-\infty_A, +\infty_A)$  to denote a range that contains all the attribute values in  $\text{Dom}(A)$ .

**DEFINITION 2 (RANGE-FILTERING NEAREST NEIGHBOR SEARCH [40]).** Let  $\mathcal{D}$  be a set of  $A$ -attributed vectors in  $\mathbb{R}^d$ . A range-filtering nearest neighbor search query  $Q = (q, [l, r], k)$  has  $q \in \mathbb{R}^d$ ,  $l, r \in \text{Dom}(A)$ , and  $k$  a positive integer. Define  $\mathcal{D}[l, r] \doteq \{v \in \mathcal{D}, \text{att}(v) \in [l, r]\}$ . The query returns  $\text{kNN}(q, \mathcal{D}[l, r])$ , a  $k$ -sized subset  $\mathcal{R}$  of  $\mathcal{D}[l, r]$  such that  $\forall u \in \mathcal{R}$  and  $\forall v \in \mathcal{D}[l, r] \setminus \mathcal{R}$ ,  $\delta(u, q) \leq \delta(v, q)$ .

For simplicity, assume there are always at least  $k$  vectors in  $\mathcal{D}[l, r]$ , i.e.,  $|\mathcal{D}[l, r]| \geq k$ . Due to the “curse of dimensionality” [13], a large body of existing research on nearest neighbor search focuses on approximate nearest neighbor search (ANNS), which reports a set  $\text{kANN}(q, \mathcal{D})$  of  $k$  vectors aiming at an optimized recall  $\frac{1}{k} |\text{kANN}(q, \mathcal{D}) \cap \text{kNN}(q, \mathcal{D})|$  for a vector  $q$ .

**DEFINITION 3 (RFANNS [40]).** Given a set  $\mathcal{D}$  of attributed vectors in  $\mathbb{R}^d$ , a range-filtering approximate nearest neighbor search query  $Q = (q, [l, r], k)$  aims at reporting  $\text{kANN}(q, \mathcal{D}[l, r])$ , a set of  $k$  vectors in  $\mathcal{D}[l, r]$ , with an optimized recall  $\frac{|\text{kANN}(q, \mathcal{D}[l, r]) \cap \text{kNN}(q, \mathcal{D}[l, r])|}{k}$ .

Consider RFANNS on a stream of  $A$ -attributed data vectors.

**PROBLEM 1 (DYNAMIC RANGE-FILTERING APPROXIMATE NEAREST NEIGHBOR SEARCH (DRFANNS)).** Let  $v_1, v_2, \dots$  be a sequence of attributed data vectors in  $\mathbb{R}^d$  arriving the system one at a time. For each integer  $t > 0$ ,  $v_t$  arrives at the system at time  $t$  and is associated with an  $A$ -attribute value  $\text{att}(v_t)$ ; denote by  $\mathcal{D}_t = \{v_1, v_2, \dots, v_t\}$  the set of vectors arrived the system by time  $t$ . Design a structure that can handle, at each time  $t$ , the insertion of  $v_t$ , and for any RFANNS query  $Q(q, [l, r], k)$  raised at time  $t$ , efficiently report  $\text{kANN}(q, \mathcal{D}_t[l, r])$ .

The main difficulty of DRFANNS is that the data vectors arriving the system have an arbitrary ordering of their attribute values.

**Example 1.** Figure 1 shows the snapshot of the system at time  $t = 9$  where a set  $\mathcal{D}_9 = \{v_1, v_2, \dots, v_9\}$  of 9 attributed data vectors arrived the system. Consider the RFANNS query  $Q = (q, [34, 63], k = 2)$ . The distances between the query vector  $q$  and the data vectors are shown in the figure. We have  $\mathcal{D}_9[34, 63] = \{v_2, v_4, v_5, v_6, v_7, v_9\}$  (i.e., the shadowed vectors are not in the query range). The query aims to report  $\text{kNN}(q, \mathcal{D}_9[34, 63]) = \{v_4, v_5\}$ . Note that although  $v_1$  is closer to  $q$  than  $v_4$  and  $v_5$ , they should not be reported as its attribute values  $\text{att}(v_1) = 28$  is outside of the query range  $[34, 63]$ .

Note that a special case of DRFANNS which we call Ordered-DRFANNS, assumes that the data vectors arriving the system are in the ascending order<sup>1</sup> of their  $A$ -attribute values. Ordered-DRFANNS can be addressed by an existing technique SeRF [40]. In its settings, for any two positive integers  $i$  and  $j$  with  $i < j$ ,  $\text{att}(v_i) < \text{att}(v_j)$ .

<sup>1</sup>or descending order. We restrict our discussion to the ascending order for simplicity.

---

**Algorithm 1:** 2DSEGMENTANNSearch( $\mathbb{G}$ ,  $q$ , range,  $ep$ ,  $K$ )

---

**Input:**  $\mathbb{G}$ : 2D segment graph; range: a query range  $(x, y)$  or  $[x, y]$ ;  $ep$ : entry vector,  $K$ : the parameter  $efsearch/efconstruction$  in HNSW.  
**Output:**  $ann$ :  $K$  approximate nearest neighbors of  $q$  in range.

```
1 if range is  $(x, y)$  then open = true; else open = false;
2 mark  $ep$  as visited;
3 push  $ep$  to the min-heap  $pool$  in the order of distance to  $q$ ;
4 push  $ep$  to the max-heap  $ann$  in the order of distance to  $q$ ;
5 while  $pool$  is not empty do
6    $v \leftarrow$  the vector nearest to  $q$  in  $pool$ , pop  $pool$ ;
7    $u \leftarrow$  the vector farthest to  $q$  in  $ann$ ;
8   if  $\delta(q, v) > \delta(q, u)$  then continue;
9   foreach unvisited  $o$  with  $(l, r, o, b, e) \in \mathbb{G}[v]$  do
10    if  $x \in (l, r]$  or (open and  $x = l$ ) then
11     if  $y \in [b, e)$  or (open and  $y = e$ ) then
12      mark  $o$  as visited;
13       $u \leftarrow$  the vector farthest to  $q$  in  $ann$ ;
14      if  $|ann| < K$  or  $\delta(q, o) < \delta(q, u)$  then
15       push  $o$  to  $pool$  and  $ann$ ;
16       if  $|ann| > K$  then pop  $ann$ ;
17 return  $ann$ ;
```

---

**Algorithm 2:** PRUNE( $o$ ,  $ann$ ,  $M$ )

---

**Input:**  $o$ : a vector;  $ann$ : 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  $\leftarrow \emptyset$ ;
2 foreach  $v \in ann$  in the ascending order of  $\delta(o, v)$  do
3   not_dominated  $\leftarrow$  true;
4   foreach  $u \in neighbors$  do
5     if  $u$  dominates  $v$  as  $o$ 's neighbors then
6       not_dominated  $\leftarrow$  false and break;
7   if not_dominated then add  $v$  to neighbors;
8   if  $|neighbors| \geq M$  then break;
9 return neighbors;
```

---

**Algorithm 3:** 2DSEGMENTGRAPHCONSTRUCTION

---

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

```
1 foreach  $1 < j \leq n$  do
2   For dummy vector  $v_0$ , let  $att(v_0) \leftarrow -\infty_A$ ;  $i = 0$ ;
3   while  $i < j - 1$  do
4      $ann \leftarrow$  2DSEGMENTANNSearch( $\mathbb{G}$ ,  $v_j$ ,  $(att(v_i), +\infty_A)$ ,  $v_{j-1}, K$ );
5      $i' = \min\{x | v_x \in ann\}$ ;
6     foreach  $v \in PRUNE(v_j, ann, M)$  do
7       add  $(att(v_i), att(v_{i'}), v, att(v_j), +\infty_A)$  to  $\mathbb{G}[v_j]$ ;
8       add  $(att(v_i), att(v_{i'}), v_j, att(v_j), +\infty_A)$  to  $\mathbb{G}[v]$ ;
9      $i = i'$ ;
10 return  $\mathbb{G}$ ;
```

---

For the simplicity of our discussion, assume that for a vector  $v_t$  in the stream, all the other stream vectors have different distances to  $v_t$ . In fact, we break ties using the arrival time of the vectors.

## 2.2 Graph-based RFANNS Structure SeRF

We introduce SeRF [40], the state-of-the-art RFANNS method on static datasets, which also serves as a solution to Ordered-DRFANNS.

Given an attributed vector set  $\mathcal{D}$ , SeRF constructs a graph  $\mathbb{G}$ , called **2DSegmentGraph**.  $\mathbb{G}$ 's nodes are the vectors in  $\mathcal{D}$ ; for each node  $u$  in  $\mathbb{G}$ , its neighbor list consists of tuples in the form of  $(l, r, v, b, e)$ . These tuples capture a key observation: in the HNSW graph built on each subset  $\mathcal{D}[x, y]$ , where  $x \in (l, r]$  and  $y \in [b, e)$ ,  $v$  always appears in  $u$ 's neighbor list. For example, suppose a node  $u$  in  $\mathbb{G}$  has a neighbor tuple  $(3, 6, v, 8, 9)$ . Then  $v$  is  $u$ 's neighbor in all the HNSW graphs built on  $\mathcal{D}[x, y]$  for all  $x \in (3, 6]$  and  $y \in [8, 9)$ . Given a RFANNS query  $(q, [x, y], k)$ , the greedy search of [40] traverses a neighbor  $(l, r, v, b, e)$  in  $\mathbb{G}$  if and only if  $x \in (l, r]$  and  $y \in [b, e)$ . This is equivalent to performing the greedy search in the HNSW graph built on the subset  $\mathcal{D}[x, y]$ .

Algorithm 1 details the search process. Specifically, the search starts from an entry vector  $ep$  and keeps two initially empty heaps, a min-heap  $pool$  recording all the visited yet explored nodes (Line 3) and a max-heap  $ann$  keeping  $K$  visited nodes that are closest to  $q$ . The search is prioritized by the distance to  $q$  (Line 6) and ends when depleting the nodes in  $pool$  (Line 5)<sup>2</sup>. For node  $v$  that is being explored, select the neighbors whose tuples fit the query range (Lines 9-10) for visiting. If a newly visited node has distance to  $q$  smaller than the  $K$ -th node in  $ann$ , update the heaps (Lines 14-16).

What makes SeRF suitable for Ordered-DRFANNS is its construction process (Algorithm 3). It inserts the nodes to an initially empty graph  $\mathbb{G}$  in the ascending order of their attribute values. The insertion of every node  $v_j$  (Lines 1-2) triggers a number of ANNS for  $v_j$  with different ranges (Lines 3-4) on the partially constructed graph  $\mathbb{G}$ . The aim is to identify the range  $(att(v_i), att(v_{i'}))$  that for all query ranges  $[x, y]$  with  $x \in (att(v_i), att(v_{i'}))$  and  $y \geq att(v_j)$ , the neighbor list of  $v_j$  would be the same; we call  $(att(v_i), att(v_{i'}))$  an interval for sharing neighbor lists. Specifically, to identify these intervals for  $v_j$ , SeRF first calls Algorithm 1 to find a set  $ann$  of  $K$  approximate nearest neighbors of  $v_j$  for the range  $(att(v_i), +\infty_A)$  (Line 4), here  $i$  is initially 0 and  $v_0$  is a dummy vector (Line 2). For  $i'$  being the index of the vector with the smallest attribute value in  $ann$ ,  $(att(v_i), att(v_{i'}))$  is an interval for sharing neighbor lists. Line 6 prunes the  $ann$  (same as in HNSW [23]) to prepare the neighbour list, and then Lines 7-8 add the edges and reverse edges with intervals to  $\mathbb{G}$ . In the next iteration,  $i$  jumps to  $i'$  (Line 9) and the process is repeated. It terminates when  $i$  meets  $j - 1$  (Line 3).

**Remark.** If assuming that 2DSEGMENTANNSearch in Line 4, Algorithm 3 returns exact nearest neighbors, and disabling reverse edges in HNSW to trigger pruning, SeRF proves that  $\mathbb{G}$  is a lossless compression of all the  $O(n^2)$  HNSW graphs, one for each set of data vectors  $\mathcal{D}[att(v_i), att(v_j)]$ , where  $1 \leq i < j \leq n$ . Both the correctness of SeRF and the compression technique are heavily based on the strict ordering on the attribute values of the inserted nodes.

We updated Algorithm 1 to accommodate an open query range  $(x, y)$  in Lines 10-12. In other words, an edge with label  $(l, r, o, b, e)$  is active under  $(x, y)$  if  $(x \in (l, r] \text{ or } x = l)$  and  $(y \in [b, e) \text{ or } y = e)$ .

<sup>2</sup> $K$  is essentially the  $efsearch/efconstruction$  in the original HNSW algorithm [23]. Same as in HNSW search, one can return the  $k$  vectors in  $ann$  closest to  $q$  as the final results.

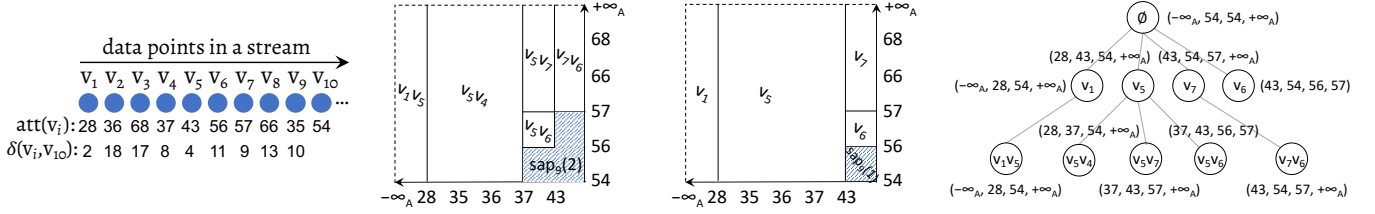


Figure 2: A running example of the rectangle tree.

### 3 RECTANGLE TREE AND DYNAMIC SEGMENT GRAPH

To deal with DRFANNS, we design a structure called the dynamic segment graph. Similar to the 2DSegmentGraph, each data vector is a node in the graph and each edge has a “rectangle” label  $(l, r) \times [b, e]$ . We aim to build the dynamic segment graph such that for any query range  $[x, y]$ , the subgraph induced by the edges whose labels  $(l, r) \times [b, e]$  satisfy  $x \in (l, r)$  and  $y \in [b, e]$  is exactly the HNSW graph on  $\mathcal{D}_{t-1}[x, y]$  where  $\mathcal{D}_{t-1} = \{v_1, \dots, v_{t-1}\}$  includes data vectors arrived when the query is issued.

Consider the next vector  $v_t$  in the stream. To insert  $v_t$  into the dynamic segment graph (by creating edges from/to it), one way is to find all the tuples  $(l, r, b, e, \text{KNNlist})$  such that for any query range  $[x, y]$  there is one and only one tuple  $(l, r, b, e, \text{KNNlist})$  such that  $x \in (l, r)$  and  $y \in [b, e]$ . Moreover, KNNlist is the K-nearest neighbors of  $v_t$  in  $\mathcal{D}_{t-1}[x, y]$ . We can then create edges for  $v_t$  in the dynamic segment graph by visiting every tuple  $(l, r, b, e, \text{KNNlist})$  in the structure, applying the pruning strategy in HNSW to KNNlist to get a neighbor list, and create an edge  $(l, r, v_t, b, e)$  from  $v_t$  and another edge  $(l, r, v_t, b, e)$  to  $v_t$  for every data vector  $v$  in the neighbor list. We design a structure called the “rectangle tree” in this section, whose leaf nodes are all the tuples  $(l, r, b, e, \text{KNNlist})$ .

The rectangle tree is defined based on the exact nearest neighbors of each vector  $v_t$  among its predecessors  $\mathcal{D}_{t-1} = \{v_1, v_2, \dots, v_{t-1}\}$  on the stream of  $v_1, v_2, \dots, v_t, \dots$ . We define the structure and show its properties in Section 3.1. Section 3.2.1 uses the rectangle tree to redesign the solution to Ordered-DRFANNS. Such a redesign leads us to a solution to DRFANNS (Problem 1), as shall be seen in Section 3.2.2. To distinguish from the graph constructed by SeRF, we use  $G$  to denote the dynamic segment graph we build for DRFANNS.

For the readers who are not interested in the complexity analysis, Section 3.1.2 can be skipped.

#### 3.1 Rectangle Tree Structure

A rectangle tree is built for a newly arrived vector  $v_t$ . Let  $K$  be an integer parameter. The tree has  $K + 1$  levels.

**3.1.1 Definitions. xNN sequence.** Given a vector  $v_t$ , an integer  $x$  and an attribute pair  $(l, r)$  with  $\text{att}(v_t) \in [l, r]$ , the xNNlist sequence includes the  $x$  nearest neighbors of  $v_t$  in  $\mathcal{D}_{t-1}[l, r]$  – we define xNNlist sequence only on closed ranges for simplicity. The  $x$  nearest neighbors are ordered by their distances to  $v_t$  ascendingly. Formally, the data vectors sequence  $v_{i_1}, v_{i_2}, \dots, v_{i_x}$  is the xNNlist sequence of  $[l, r]$  if

- (1) They all arrive before  $v_t$ , i.e., time  $i_1, i_2, \dots, i_x < t$ ;
- (2) Their attribute values  $\text{att}(v_{i_1}), \text{att}(v_{i_2}), \dots, \text{att}(v_{i_x}) \in [l, r]$ ;
- (3) Their distances to  $v_t$  ascend, i.e.,  $\delta(v_{i_1}, v_t) < \dots < \delta(v_{i_x}, v_t)$ ;
- (4) There does not exist a vector  $v_i$  in the stream that satisfies both (1) and (2) and has  $\delta(v_i, v_t) < \delta(v_{i_x}, v_t)$ .

We denote by  $+$  the **concatenation** of a sequence and a vector where the vector would be the last vector in the sequence.

**Short Attribute Pair.** Given an integer  $x$ , an attribute pair  $l, r$  may not even have a xNNlist sequence before inserting  $v_t$  – if there are less than  $x$  vectors in  $\mathcal{D}_{t-1}[l, r]$  – we call such a pair  $(l, r)$  a short attribute pair to  $x$ . On the other hand, as long as  $|\mathcal{D}_{t-1}[l, r]| \geq x$ ,  $l, r$  has an xNNlist sequence. Define  $\text{sap}_{t-1}(x)$  as the set of all the short attribute pairs to  $x$ . Formally,

$$\text{sap}_{t-1}(x) = \{(l, r) | l, r \in \text{Dom}(A) \text{ and } |\mathcal{D}_{t-1}[l, r]| < x\}.$$

**Nodes.** A node  $v^T$  of a rectangle tree is a tuple in the form of  $v^T = (L, L', R, R', x, N_x)$ , where  $L \leq L' \leq \text{att}(v_t) \leq R \leq R'$  are attribute values;  $x$  is an integer in  $[0, K]$ ;  $N_x$  is the xNNlist sequence of  $v_t, x$ , and all the possible  $[l, r]$  with  $l \in (L, L')$  and  $r \in [R, R']$ . In other words, for all possible attribute ranges  $[l, r]$  with  $(l, r)$  in the rectangle of  $(L, L') \times [R, R']$ , they share the same xNNlist sequence  $N_x$  under  $v_t$  and  $x$ . We call  $\text{rect}(v^T) = (L, L') \times [R, R']$  the rectangle of  $v^T$ . The tree ensures that any child  $u^T$  of  $v^T$  has  $\text{rect}(u^T) \subseteq \text{rect}(v^T)$ .

**Levels.** The nodes in the rectangle tree is leveled by the  $x$  values. The root is the only node with  $x = 0$ . The children are one level deeper than the father. All nodes with  $x = K$  are leaves.

A rectangle tree must satisfy two conditions, disjoint condition and covering condition, defined as below.

**Disjoint condition.** For any two nodes at the same level  $x \in [0, K]$ , their xNNlist sequences must be different.

**Covering condition.** For a node  $v^T = (L, L', R, R', x, N_x)$  in the rectangle tree with  $p$  children at level  $x + 1$ . Denote the set of its children’s rectangles as  $\mathcal{S} = \{S_1, S_2, \dots, S_p\}$ .  $\mathcal{S}$  covers all the attribute pairs in  $\text{rect}(v^T)$  that have an  $(x + 1)$ NNlist sequence, i.e.,

$$\text{rect}(v^T) \setminus \text{sap}_{t-1}(x + 1) \subseteq \bigcup_{i \in [1, p]} S_i.$$

**Example 2.** Figure 2 above shows a rectangle tree for the newly arrived vector  $v_{10}$  in the stream of vectors on the left side of the figure. The xNNlist sequences are inside the nodes. The levels  $x$  are 0 for the root, 1 for the four children of the root, and 2 for the grandchildren of the root. The rectangles of the nodes are adjacent to the nodes. To be more clear, we also plot the rectangles of the nodes in the figure. The second subfigure shows the rectangles of all the nodes at level 2, while the third subfigure shows the rectangles of all the nodes at level 1. The root node has the entire plane as its rectangle. The bottom-right corners in shadow represent the sets of all short attribute pairs. As we can see, the rectangle tree satisfies both the disjoint condition and the covering condition.

#### 3.1.2 Properties.

**LEMMA 1.** The rectangles of all the nodes at the same level are disjoint. In other words, there is no attribute pair that is in the rectangles of two distinct tree nodes at the same level at the same time.

PROOF. Consider two distinct nodes  $v^T$  and  $u^T$  at the same level  $x$ .  $v^T$  has rectangle  $S_v$  and list  $N_x$ .  $u^T$  has  $S_u$  and list  $N'_x$ . Let  $(l, r)$  be an attribute pair in both  $S_v$  and  $S_u$ . Thus, the  $x$ NNlist sequence of range  $[l, r] = N_x = N'_x$  since the  $x$ NNlist sequence of  $[l, r]$  is unique, conflicting the distinct condition that  $N_x \neq N'_x$ . Thus attribute pair  $(l, r)$  does not exist. Thus  $S_v$  and  $S_u$  are disjoint.  $\square$

LEMMA 2. For a non-leaf rectangle-tree node with a rectangle  $S_0$  at level  $x - 1$ , the rectangles  $\mathcal{S} = \{S_1, S_2, \dots, S_p\}$  of all its children (at level  $x$ ) form a disjoint partitioning of  $S_0 \setminus \text{sap}_{t-1}(x)$ . In other words, for every attribute pair  $(l, r) \in S_0$  that is not a short attribute pair to  $x$ , there is one and only one rectangle  $S$  in  $\mathcal{S}$  such that  $(l, r) \in S$ .

PROOF. From the definition of the rectangle tree node, we have  $S_i \subseteq S_0$  for each  $i \in [1, p]$ . For every attribute pair  $(l, r) \in S_0 \setminus \text{sap}_{t-1}(x)$ , there is one (covering condition) and only one (Lemma 1) rectangle  $S$  in  $\mathcal{S}$  such that  $(l, r) \in S$ .  $\square$

THEOREM 3 (CANONICAL PARTITIONING). Given a vector  $v_t$  and a rectangle  $S$  on  $\text{Dom}(A)$ . For all the rectangle trees for  $v_t$  whose root has the rectangle of  $S$ , for any level  $x \in [0, K]$ , the rectangles of all the tree nodes on level  $x$  form a canonical partitioning of the attribute pairs in  $S \setminus \text{sap}_{t-1}(x)$ . Specifically, let  $\mathcal{S} = \{S_1, S_2, \dots, S_p\}$  be the rectangles of all the tree nodes at level  $x$ . Let the corresponding collections of sets of the attribute pairs be

$$\mathcal{S}' = \{\{(l, r) | l, r \in \text{Dom}(A) \text{ and } (l, r) \in S_i\} | i \in [1, p]\}.$$

Then we have  $\mathcal{S}' = \mathcal{S}''$  where  $\mathcal{S}''$  is the grouping of all attribute pairs  $(l, r) \in S \setminus \text{sap}_{t-1}(x)$  by their corresponding  $x$ NNlist sequences.

PROOF. Firstly, for any two integers  $0 \leq x'' < x' \leq K$ , we have  $\text{sap}_{t-1}(x'') \subseteq \text{sap}_{t-1}(x')$  based on the definition of short attribute pair. Apply Lemma 2 top down from the root to the tree nodes at level  $x - 1$  level by level, thus the rectangles of all the rectangle tree nodes at level  $x$  is a disjoint partitioning of  $S \setminus \text{sap}_{t-1}(x)$ . Secondly, we prove that the partitioning is canonical. Consider all the attribute pairs in  $S \setminus \text{sap}_{t-1}(x)$  that have  $x$ NNlist sequences. Since each rectangle  $S_i \in \mathcal{S}$  ensures that all pairs  $(l, r) \in S_i$  have the same  $x$ NNlist sequence; Distinct condition ensures that pairs from different rectangles in  $\mathcal{S}$  have different  $x$ NNlist sequences. Therefore,  $\mathcal{S}'$  must be a grouping of all the attribute pairs  $(l, r)$  in  $S$  based on their  $x$ NNlist sequences which is  $\mathcal{S}''$  and is thus canonical.  $\square$

Our following average case analysis assumes the **independence** between the attribute values and distances, i.e., the attribute values of vectors in  $\{v_1, v_2, \dots, v_t\}$  are distinct and if fixing the ordering of vectors in  $\mathcal{D}_{t-1}$  based on their distances to  $v_t$ , and then reorder all the vectors based on their attribute values, then each of  $t!$  permutations has an equal probability to appear.

LEMMA 3. Consider a rectangle tree for a vector  $v_t$  with  $\text{att}(v_t) > \text{att}(v_1), \text{att}(v_2), \dots, \text{att}(v_{t-1})$  and let the rectangle of the root be  $(-\infty, \text{att}(v_t)] \times [\text{att}(v_t), +\infty)$ . The number of leaf nodes of the rectangle tree at level  $K$  is in the worst case  $O(t)$  and  $O(K \ln t)$  expected.

THEOREM 4. When inserting  $v_t$  and the root node has rectangle  $(-\infty_A, \text{att}(v_t)] \times [\text{att}(v_t), +\infty_A)$ , the number of leaf nodes at level  $K$  is in the worst case  $O(Kt)$ , and  $O(K^2 \ln t)$  in expectation.

---

#### Algorithm 4: ORDEREDINSERTION( $G, v_t, M, K$ )

---

**Input:**  $G$ : the dynamic segment graph constructed for  $\mathcal{D}_{t-1} = \{v_1, v_2, \dots, v_{t-1}\}$ ;  $v_t$ : a vector arriving at time  $t$ ;  $M$  and  $K$ : the parameters in HNSW construction

**Output:**  $G$ : the dynamic segment graph for  $\{v_1, v_2, \dots, v_t\}$ .

// *queue* is a min-heap of tuples in the form of  $(L, L', R, R', x, x\text{NNlist})$  in the order of  $x$ ; the tuple means for all the attribute ranges  $(x, y)$  with  $(x = L$  or  $x \in (L, L'])$  and  $(y = R' \text{ or } y \in [R, R'))$ ,  $v_t$  has the same set of the  $x$  nearest neighbors on  $\mathcal{D}_{t-1}$ , which is  $x\text{NNlist}$ .

```

1  queue.push( $-\infty_A, \text{att}(v_t), \text{att}(v_t), +\infty_A, 0, \emptyset$ );
2  while queue is not empty do
3       $(L, L', R, R', x, N_x) \leftarrow \text{queue.pop}()$ ;
4      if  $x = K$  then
5          foreach  $v \in \text{PRUNE}(N_x, M, v_t)$  do
6              add  $(L, L', v, R, R')$  to  $G[v_t]$ ;
7              add  $(L, L', v_t, R, R')$  to  $G[v]$ ;
8          continue;
9      while  $L < L'$  and  $R < R'$  do
10          $\text{ann} \leftarrow 2\text{DSEGMENTANNSEARCH}(G, v_t, (L, R'), v_1, K)$ ;
11         if  $\text{ann} \subseteq N_x$  then goto Line 5;
12         // When the # of vectors in attribute range  $(L, R')$ 
13         // is  $\leq x$ , call this node a sap node.
14          $v_c \leftarrow \arg \min_{v \in \text{ann} \setminus N_x} \delta(v, v_t)$ ;
15         if  $\text{att}(v_c) < L'$  then
16             queue.push( $L, \text{att}(v_c), R, R', x + 1, N_x + v_c$ );
17             L  $\leftarrow \text{att}(v_c)$ ;
18         else
19             queue.push( $L, L', R, R', x + 1, N_x + v_c$ );
20             break;
21 return G;
```

---

Please find the proofs of Lemma 3 and Theorem 4 in our technical report: [https://miaojiao.github.io/paper/VLDB25\\_TR.pdf](https://miaojiao.github.io/paper/VLDB25_TR.pdf)

## 3.2 Dynamic Segment Graph

**3.2.1 Ordered Insertion.** Consider the problem of Ordered-DRFANNS where the attributes of data vectors have  $\text{att}(v_1) < \text{att}(v_2) < \dots < \text{att}(v_t)$ . We maintain an initially empty graph called dynamic segment graph  $G$  for nearest neighbor search. Call ORDEREDINSERTION( $G, v_t, M, K$ ) (Algorithm 4) for every newly arrived vector  $v_t$  at time  $t$ , from  $t = 1$ .

We analyze our algorithms under the Accurate Search Assumption (ASA), i.e., the nearest neighbors returned by Algorithm 1 are exact. We make this assumption because if otherwise, we could not accurately assess the impact of the approximation of the nearest neighbor search to the index structure. Note that for the problem of Ordered-DRFANNS, the lossless compression of [40] is also achieved under the ASA assumption.

LEMMA 4. Algorithm 4 builds a rectangle tree with root rectangle  $(-\infty_A, \text{att}(v_t)] \times [\text{att}(v_t), +\infty_A)$  when inserting  $v_t$  under ASA, i.e., each tuple  $(L, L', R, R', x, N_x)$  of the queue in the algorithm corresponds to a rectangle tree node at level  $x$  with  $x\text{NNlist}$  sequence  $N_x$ .

**Explanations to Algorithm 4 and Proof Sketch to Lemma 4.** The root has level  $x = 0$ ,  $\emptyset$  is the  $0\text{NNlist}$  sequence of the rectangle



of  $(-\infty_A, \text{att}(v_t)) \times [\text{att}(v_t), +\infty_A)$ . The rectangle tree is generated level by level because the queue is a min-heap based on  $x$ . Each iteration (Lines 2-18) pops a tuple  $v^T = (L, L', R, R', x, N_x)$  with the smallest  $x$  from the queue (Line 3). If  $v^T$  is a leaf node with  $x = K$ , lodge the rectangle with the edge  $(v_t, v)$  to the graph  $G$  for each pruned vector  $v$  in the  $N_x$  sequence (Lines 4-8). Otherwise, generate all the children (Lines 9-18) of  $v^T$  and enqueue them.

Next we show that if  $v^T$  ensures that for all the attribute pairs  $(l, r)$  with  $l \in (L, L']$ ,  $r \in [R, R')$ , the  $x$ NNlist sequence of interval  $[l, r]$  is  $N_x$ , then under ASA, the properties below hold for all the  $v^T$ 's children  $(L_c, L'_c, R_c, R'_c, x+1, N_{x+1})$  generated in Lines 9-18.

- $N_{x+1}$  is the  $(x+1)$ NNlist sequence for all the attribute intervals  $[l, r]$  with  $l \in (L_c, L'_c]$  and  $r \in [R_c, R'_c)$ .
- The rectangles of the children of  $v^T$  are a partitioning of  $\text{rect}(v^T) \setminus \text{sap}_{t-1}(x+1)$ .
- The  $(x+1)$ NNlist sequences of the children of  $v^T$  are different, but they have a common prefix of  $N_x$ .

The children are generated in a sequence of jumps of  $L$  values (Line 15) until  $L$  reaches/exceeds  $L'$  (Line 9). We first find  $v_c$ , the  $(x+1)$ -th nearest neighbor of  $v_t$  on attribute range  $(L, R')$  using nearest neighbor search (Lines 11-12). If  $v_c$  does not exist (Line 11), then all the intervals  $[l, r]$  with  $l, r \in (L, R')$  are short to  $x+1$ , we shall add edges to  $G$  and proceed to the next iteration (Line 8).

If  $v_c$  has attribute value in  $[L', R]$  (Lines 16-18), then all the ranges  $[l, r]$  with  $(l, r)$  in the rectangle  $(L, L') \times [R, R')$  share not only the  $x$ NNlist sequence but also the  $(x+1)$ -th nearest neighbor. Thus they share the same sequence  $N_{x+1} = N_x + v_c$ . We can safely break the search (Line 18) after enqueue the child tuple (Line 17).

If  $v_c$  has attribute value in  $(L, L')$  (Lines 13-15), the two attribute ranges  $[\text{att}(v_c), L']$  and  $[r, L']$ ,  $\text{att}(v_c) < r$ , will not share their  $(x+1)$ -th nearest neighbor, as that of  $[\text{att}(v_c), L']$  will be  $v_c$  which is missing from range  $[r, L']$ . Therefore, we partition the rectangle into two on  $\text{att}(v_c)$ , the left one  $(L, \text{att}(v_c)) \times [R, R')$  which shares the  $(x+1)$ NNlist sequence  $N_{x+1} = N_x + v_c$  (enqueued in Line 14) while the remaining rectangle  $(\text{att}(v_c), L') \times [R, R')$  will be processed in the next loop (Line 15). The loop terminates when the remaining rectangle is enqueued entirely (Line 17). Therefore, the rectangles of the children of  $v^T$  form a partitioning of  $\text{rect}(v^T) \setminus \text{sap}_{t-1}(x+1)$ .

The above two cases are sufficient since by assumption,  $\text{att}(v_t) > \text{att}(v_i)$  for all  $i < t$ , so  $\text{att}(v_c)$  can never be larger than  $R = \text{att}(v_t)$ . Besides,  $\text{att}(v_c)$  cannot go equal or below  $L$  since it was generated by the nearest neighbor search in the attribute range  $(L, R')$ .

Therefore, each child  $(L_c, L'_c, R_c, R'_c, x+1, N_{x+1})$  of  $v^T$  ensures that for any  $(l, r) \in \text{rect}(v^T) \setminus \text{sap}_{t-1}(x+1)$ , the interval  $[l, r]$  has  $(x+1)$ NNlist sequence equal to  $N_{x+1}$ . The  $(x+1)$ NNlist sequences of all the children are different (distinct) and all the rectangles of the children form a partitioning of  $\text{rect}(v^T) \setminus \text{sap}_{t-1}(x+1)$  (covering).

Apply the above results level-by-level to the tuples popped from the queue, we verify that these tuples form a rectangle tree.  $\square$

**LEMMA 5.** *When Line 11, Algorithm 4 tests true, take a snapshot of  $L, L', R, R', x$ . Denote by  $y$  the number of vectors in the range  $(L, R')$  on  $\mathcal{D}_{t-1}$ , then  $x = y$ .*

---

**Algorithm 5:** UNORDEREDINSERTION( $G, v_t, M, K$ )

---

// Replace Lines 13-18 of Algorithm 4 with code:

```

1 if  $\text{att}(v_c) < L'$  then
2    $\text{queue.push}(L, \text{att}(v_c), R, R', x+1, N_x + v_c);$ 
3    $L \leftarrow \text{att}(v_c);$ 
4 else if  $R < \text{att}(v_c)$  then
5    $\text{queue.push}(L, L', \text{att}(v_c), R', x+1, N_x + v_c);$ 
6    $R' \leftarrow \text{att}(v_c);$ 
7 else
8    $\text{queue.push}(L, L', R, R', x+1, N_x + v_c);$ 
9   break;
```

---

**PROOF.** Let  $v^T$  be the tree node popped in the corresponding iteration. As all attribute pairs in  $\text{rect}(v^T)$  share the same  $x$ NNlist sequence,  $y \geq x$ ; as Line 11 tests true,  $y \leq x$ . Thus  $y = x$ .  $\square$

**THEOREM 5 (COMPLEXITY OF ALGORITHM 4).** *Under ASA, when the  $t$ -th vector is inserted,  $t \geq K$ , Algorithm 4 has the worst case space complexity  $O(Mt)$  and the average case space complexity  $O(KM \ln t)$ ; the number of calls of ANN search, i.e., Algorithm 1, is in the worst case  $O(Kt)$  and in the average case  $O(K^2 \ln t)$ .*

**PROOF.** Algorithm 4 writes tuples to  $G$  either on leaves at level  $K$  (Lines 4-8) or on tree nodes on any level  $x < K$  such that Line 11 tests true— we call these nodes sap nodes. Each time, we write at most  $2M$  edges to  $G$ . From Lemma 3, the total number of leaves at level  $K$  is  $O(t)$  in the worst case for  $v_t$ , and  $O(K \ln t)$  in expectation. Next, we show that the total number of sap nodes on each level  $x < K$  is at most 1 and thus the total number of sap node is  $O(K)$ .

When Line 11, Algorithm 4 tests true, take a snapshot  $L, L', R, R', x$ . Lemma 5 proves that range  $(L, R')$  has exactly  $x$  vectors in  $\mathcal{D}_{t-1}$ . Note  $R' = +\infty_A$  and  $R = \text{att}(v_t)$  are larger than the attribute values of all vectors,  $L$  must be the  $(x+1)$ -th largest attribute value on  $\mathcal{D}_{t-1}$  and  $L'$  the  $x$ -th. As all nodes on level  $x$  have disjoint rectangles, no node other than  $v^T$  has rectangle intersecting  $(L, L') \times [R, R')$  and thus Line 11 is tested true at most once at level  $x$ .

Thus, the space complexity is  $O(M(t+K)) = O(Mt)$  in the worst case and  $O(KM \ln t)$  in expectation. Besides, since each ANN search either labels a node as sap node or generates a node, the total number of ANN search is at most  $K$  + the total number of nodes. Thus, the worst case number of calls of ANN search is  $O(Kt)$  and the expected number of ANN calls is  $O(K^2 \ln t)$ .  $\square$

**3.2.2 Unordered Insertion.** The benefit of the rectangle tree is that adapting Algorithm 4 to unordered insertion, i.e., removing the assumption that all the vectors inserted are in ascending order of their attribute values, is easy.

Algorithm 5 shows the algorithm of unordered insertion. Compared to ordered insertion, when a vector  $v_c$  is found, in addition to cope with the case when  $L < \text{att}(v_c) < L'$ , and  $L' \leq \text{att}(v_c) \leq \text{att}(v_t)$ , Lines 4-6 cope with an additional case of  $R < \text{att}(v_c) < R'$  in a way symmetric to that of the case of  $L < \text{att}(v_c) < L'$ .

**Example 6.** Figure 2 on the left shows a stream of attributed data vectors and their distances to  $v_{10}$ . Consider inserting  $v_{10}$  to the dynamic segment graph  $G$  using Algorithm 5 with  $K = 2$ . The algorithm first processes the tuple  $(L = -\infty_A, L' = 54, R = 54, R' = +\infty_A, x = 0, N_0 = \emptyset)$  in the queue. For this purpose, it

first finds the 2-nearest neighbors of  $v_{10}$  in  $(-\infty_A, +\infty_A)$ , which is  $ann = \{v_1, v_5\}$ , and has  $v_c = v_1$ . Since  $\text{att}(v_1) = 28 < L' = 54$ , a tuple  $(-\infty_A, 28, 54, +\infty_A, 1, v_1)$  is added to the queue and  $L$  becomes 28. Next, it finds  $ann$  in  $(28, +\infty_A)$ , which is  $\{v_5, v_4\}$ , and has  $v_c = v_5$ . Since  $\text{att}(v_5) = 43 < L' = 54$ , another tuple  $(28, 43, 54, +\infty_A, 1, v_5)$  is added to the queue and  $L$  becomes 43. Then, it finds  $ann$  in  $(43, +\infty_A)$ , which is  $\{v_7, v_6\}$ , has  $v_c = v_7$ , adds  $(43, 54, 57, +\infty_A, 1, v_7)$  to the queue as  $R = 54 < \text{att}(v_7) = 57$ , and sets  $R'$  as 57. After that, it finds  $ann$  in  $(43, 57)$ , which is  $\{v_6\}$  (note that this is the only vector in  $\mathcal{D}_9$  whose attribute value is within  $(43, 57)$ ), has  $v_c = v_6$ , adds a tuple  $(43, 54, 56, 57, 1, v_6)$  to the queue as  $R = 54 < \text{att}(v_6) = 56$ , and sets  $R' = 56$ . Finally, it finds  $ann$  in  $(43, 56)$ , which is  $\emptyset$  as there is no vector in  $\mathcal{D}_9$  whose attribute value is within  $(43, 56)$ . Thus it goes to the edge generation steps, which results in no edges as  $N_0 = \emptyset$ . The above process essentially builds a level below the root node in the rectangle tree as illustrated in Figure 2 on the right.

Next, tuple  $(L = -\infty_A, L' = 28, R = 54, R' = +\infty_A, x = 1, N_1 = v_1)$  is popped from the queue. It finds  $ann$  in  $(-\infty_A, +\infty_A)$ , which is  $\{v_1, v_5\}$ . Since  $v_1 \in N_1$ , it has  $v_c = v_5$ . As  $L' = 28 \leq \text{att}(v_5) = 43 \leq R = 54$ ,  $(-\infty_A, 28, 54, +\infty_A, 2, v_1 v_5)$  is added to the queue and the while loop breaks. The process stops when the queue depletes.

**LEMMA 6.** *Algorithm 5 builds a rectangle tree  $T$  with root rectangle  $(-\infty_A, \text{att}(v_t)) \times [\text{att}(v_t), +\infty_A)$  when inserting  $v_t$  under ASA. Specifically, each tuple  $(L, L', R, R', x, N_x)$  in queue of the algorithm corresponds to a node at level  $x$  on  $T$  whose  $x$ NNlist sequence is  $N_x$ .*

**Proof Sketch.** The proof adds additional discussions on the case of  $R < \text{att}(v_c) < R'$  compared to the proof sketch of Lemma 4. It means that all the attribute ranges  $[l, r]$  with  $(l, r) \in (L, L') \times [R, R']$  share the same  $x$ NNlist sequence. However, when  $r \geq \text{att}(v_c)$  the  $(x+1)$ NNlist should be  $N_x + v_c$  (Line 4), while when  $r < \text{att}(v_c)$ , some other vectors in  $(L, r)$ , not including  $v_c$ , will be the  $(x+1)$ -th nearest neighbor of  $v_t$ . In this case, we generate a tuple with rectangle  $(L, L') \times [\text{att}(v_c), R']$  for  $(x+1)$ NNlist sequence  $N_x + v_c$  (Line 5), and the remaining rectangle will be left for the next round of the while loop (Line 9 of Algorithm 4). Therefore, the three properties listed in the second paragraph of the proof of Lemma 4 hold. Note that removing the assumption that the attribute values of  $v_t$  is larger than the vectors in  $\mathcal{D}_{t-1}$  only adds this additional case while the other discussions in the proof of Lemma 4 hold here. Therefore, Algorithm 5 constructs a rectangle tree.  $\square$

**THEOREM 7 (COMPLEXITY OF ALGORITHM 5).** *When the  $t$ -th vector is inserted,  $t \geq K$ , the worst-case index size of Algorithm 5 is  $O(KMt)$ . The average-case index size of Algorithm 5 is  $O(K^2 M \ln t)$ . The worst number of calls of ANN search is  $O(K^2 t)$  and the expected number of ANN search calls is  $O(K^3 \ln t)$ .*

Please find the proof of Theorem 7 in our technical report [https://miaoqiao.github.io/paper/VLDB25\\_TR.pdf](https://miaoqiao.github.io/paper/VLDB25_TR.pdf)

**THEOREM 8.** *Denote by  $a_1 \leq a_2 \leq \dots \leq a_{t-1}$  the attribute values of the vectors inserted by time  $t$ . When inserting  $v_t$  under ASA, the new edges of  $G$  created by Algorithm 5 is a lossless compression of the edges from/to  $v_t$  on the  $O(t^2)$  HNSW graphs, one for each attribute range  $[a_i, a_j]$ , i.e., on  $\mathcal{D}_t[a_i, a_j]$  with  $i \leq j$ .*

**PROOF.** Define  $a_0$  be  $-\infty_A$  and  $a_t$  be  $+\infty_A$ . Consider an attribute range of  $[l, r]$ . We only consider  $l \leq \text{att}(v_t) \leq r$  as otherwise  $v_t$  is

not in the corresponding HNSW graph and there will be no working edges among the newly added edges to  $G$  on  $v_t$ . Let  $i, j$  be such that  $a_{i-1} < l \leq a_i \leq a_j \leq r < a_{j+1}$ . Let  $S$  be the  $K$  nearest neighbors of  $v_t$  in  $\mathcal{D}_{t-1}[a_i, a_j] = \mathcal{D}_{t-1}[l, r]$ . The HNSW edges from/to  $v_t$  under search range  $[l, r]$  are the  $v_t$  edges on the HNSW graph built on attribute range  $[a_i, a_j]$ . Under ASA, these edges are between  $v_t$  and the pruned (w.r.t.  $v_t$ ) vectors  $S_p$  of  $S$ . Consider the rectangle tree  $T$  constructed for  $v_t$  by Algorithm 5. It suffices to show that all the working edges under search range  $[l, r]$  from  $v_t$  that are added by  $T$  to the graph of  $G$  are exclusively between  $v_t$  and  $S_p$ . Our proof has two cases,  $|S| < K$  and  $|S| \geq K$ .

When  $|S| < K$ , let  $x \doteq |S|$ . According to Theorem 3, there is exactly one node  $v^T$  on  $T$  at level  $x$  such that  $(l, r) \in \text{rect}(v^T)$ . Furthermore, since  $(l, r)$  belongs to  $\text{sap}_{t-1}(x+1)$ , it will not appear in any rectangle at level higher than  $x$  and thus there must be a time when processing  $v^T$ , Line 11 Algorithm 4 tests true: snapshot the values of  $(L, L', R, R')$  and thus  $\mathcal{D}_{t-1}$  on both  $(L, R')$  and  $[L', R]$  are  $S$  (as all attribute pairs in  $\text{rect}(v^T)$  share the same  $x$ NNlist sequence which is  $S$ ), thus  $L = a_{i-1}$ ,  $L' = a_i$ ,  $R = a_j$ , and  $R' = a_{j+1}$  (as they all align to attribute values of  $\mathcal{D}_{t-1}$ ). The edges between  $v_t$  and  $S_p$  are thus added to  $G$  under rectangle  $U = (a_{i-1}, a_i] \times [a_j, a_{j+1})$ . Thus the working edges of  $v_t$  under  $[l, r] \in U$  are exclusively with  $S_p$ . Moreover, as this rectangle  $U$  will not join with any rectangle in higher levels, the edges between  $v_t$  to  $S_p$  will not work under an interval which has more than  $x$  vectors in  $\mathcal{D}_{i-1}$ .

When  $|S| \geq K$ , there is exactly one node  $v^T(L, L', R, R', K, N_K)$  on  $T$  among level  $K$  nodes such that  $(l, r) \in \text{rect}(v^T)$  (Theorem 3). That is, for  $v_t$ , only edges added by  $v^T$  can work under  $[l, r]$ . Also, we have  $N_K = S$  due to the definition of rectangle tree. Since edges between  $v_t$  and  $S_p$  are added to  $G$  with rectangle  $\text{rect}(v^T)$  by this node, they are the exclusive working edges from  $v_t$  under  $[l, r]$ .

Therefore, the newly added edges to  $G$  form a lossless compression of the edges from/to  $v_t$  on the  $O(t^2)$  HNSW graphs.  $\square$

### 3.3 Early Pruning

Realizing that we perform pruning of the  $K$ NNlist sequence on leaf nodes before adding edges to the dynamic segment graph, we would like to explore if pruning the  $K$ NNlist early can reduce both the index time and index size.

Algorithm 6 revises the random insertion process in two aspects. Consider  $v^T = (L, L', R, R', x, x\text{NNlist})$  popped from the queue. As opposed to either generating a child/terminate the children generation based on  $v_c$  in Lines 1-9 of Algorithm 5, we keep generating  $v_c$  until either  $v_c$  is not be pruned by the existing  $x\text{NNlist}$  sequence (Lines 15), or a total of  $K$  points are accumulated (together with the points in the sequence) for the rectangle (Line 10). If  $v_c$  is not pruned, split the rectangle as usual (Line 17). For each tuple in queue (except for the root), the last point  $v_c$  of the  $x\text{NNlist}$  list must remain after pruning, we lodge edges between  $v_t$  and  $v_c$  as after-prune edge (Lines 5-7).

**Pruned nearest neighbors sequence of attribute interval  $[l, r]$ .** Define on set  $\mathcal{D}_{t-1}[l, r]$  the points that arrived before  $v_t$  whose attribute values falling in  $[l, r]$ , the pruned sequence below.

- (1) Sort all the points in  $\mathcal{D}_{t-1}[l, r]$  in ascending order of their distances to  $v_t$ . The resulting sequence is denoted as  $ann$ .

**Algorithm 6:** PRUNEDINSERTION( $G, v_t, M, K$ )

---

**Input:**  $G$ : the dynamic segment graph constructed for  $\mathcal{D}_{t-1} = \{v_1, v_2, \dots, v_{t-1}\}$ ;  $v_t$ : a point arriving at time  $t$ ;  $M$  and  $K$ : the parameters in HNSW construction

**Output:**  $G$ : the dynamic segment graph for  $\{v_1, v_2, \dots, v_t\}$ .

```

1  $queue.push(-\infty_A, att(v_t), att(v_t), +\infty_A, 0, \emptyset);$ 
2 while  $queue$  is not empty do
3    $(L, L', R, R', x, xNNlist) \leftarrow queue.pop();$ 
4   if  $x \neq 0$  then
5      $v_e \leftarrow$  the last point in  $xNNlist$ ;
6     add  $(L, L', v_e, R, R')$  to  $G[v_t]$ ;
7     add  $(L, L', v_t, R, R')$  to  $G[v_e]$ ;
8   if  $x = K$  then continue;
9   while  $L \leq L'$  and  $R \leq R'$  do
10    while  $x < K$  do
11       $ann \leftarrow 2DSEGMENTANNSearch(G, v_t, (L, L'), v_1, K);$ 
12       $v_c \leftarrow \arg \min_{v \in ann \setminus xNNlist} \delta(v, v_t);$ 
13      if  $v_c$  is dominated by any point in  $xNNlist$  then
14         $(x+1)NNlist \leftarrow xNNlist + v_c; x++;$ 
15      else break;
16      if  $v_c$  is in  $xNNlist$  then break;
17      Lines 1-9 of Algorithm 5;
18 return  $G$ ;
```

---

- (2) Prune, using Algorithm 2, by calling PRUNE( $o, ann, M$ ), and call the resulting sequence the Pruned Nearest Neighbors Sequence (PNNS) of  $[l, r]$ .

LEMMA 7. For each tuple  $v^T = (L, L', R, R', x, xNNlist)$ , if  $x > 0$ , then the last point  $v_c$  of  $xNNlist$  cannot be pruned by any point in  $xNNlist \setminus \{v_c\}$ .

LEMMA 8. For each tuple  $v^T = (L, L', R, R', x, xNNlist)$  in the queue of Algorithm 6, let  $S$  be the rectangle  $(L, L') \times [R, R']$ , let sequence  $P = PRUNE(xNNlist, M, v_t)$ , let  $m = |P|$ . We show that for all attribute pair  $(l, r) \in S$ ,  $P$  is the  $m$ -prefix of the PNNS of  $[l, r]$ .

Please find the proofs of Lemma 7 and Lemma 8 in our technical report: [https://miaoqiao.github.io/paper/VLDB25\\_TR.pdf](https://miaoqiao.github.io/paper/VLDB25_TR.pdf)

THEOREM 9. Algorithm 6 constructs a tree of aggregated rectangles where each aggregated rectangle  $S$  ensures that all attribute intervals  $[l, r]$  with  $(l, r) \in S$  share the same PNNS prefix. In other words, the tree is a prefix tree of the PNNS of different rectangles.

## 4 OPTIMIZATIONS FOR DYNAMIC SEGMENT GRAPH

Although the dynamic segment graph introduced earlier losslessly compresses many HNSW graphs, one for each possible query range, the index cost (i.e., index time and index size) is rather high in practice. In this section, we present a few optimizations to improve the practical performance of dynamic segment graph maintenance.

**O1: One ANN Search for All.** We observe that the procedure 2DSEGMENTANNSearch is invoked an excessive number of times (one search for each node in the rectangle tree). To reduce the index time, when a new data point  $v_t$  arrives, we propose to perform a single search using 2DSEGMENTANNSearch( $G, v_t, (-\infty_A, +\infty_A), v_1, Z$ )

where  $Z$  is a parameter to find a set  $ann$  of  $Z$  approximate nearest neighbors of  $v_t$ . Then, instead of invoking 2DSEGMENTANNSearch( $G, v_t, (L, R'), v_1, K$ ) in the algorithms to find the  $v_c$ , we visit the data points in  $ann$  in the ascending order of their distance to  $v_t$  and  $v_c$  is the first one in  $ann$  that (1) is not in  $xNNlist$  and (2) has  $att(v_c) \in (L, R')$ . If no such data point exists in  $ann$ , we simply break the while condition and process the next tuple in the queue.

**O2: Removing Dominated Neighbors.** Realizing that the  $KNNlist$  sequence on leaf nodes is pruned before added edges to the dynamic segment graph, we propose to prune the  $KNNlist$  early to reduce both the index time and index size. Specifically, instead of maintaining  $xNNlist$  in the rectangle tree node, we maintain the neighbor list after pruning  $xNNlist$ , which we denote it as  $xPNN$ . Then, consider  $v^T = (L, L', R, R', x, xPNN)$  popped from the queue. We visit the set  $ann$  of  $Z$  approximate nearest neighbors and use the first  $v_c$  that (1) is not dominated by any data point in  $xPNN$ , (2) is not in  $xPNN$  and (3) has  $att(v_c) \in (L, R')$ . If no such data point exists in  $ann$ , we move on to process the next tuple in the queue.

**O3: Merge Rectangles using MBR.** We observe that, between the same two endpoints in the dynamic segment graph, there might be multiple edges, each with a distinct rectangle label. To reduce the index size, we propose to merge them using minimum bounding rectangles (MBRs) [10]. There are different strategies in merging the rectangles similar to the construction of the R-tree [10]. For simplicity, this paper proposes to merge all these rectangles to a single MBR. Specifically, for each edge  $G[u][v]$  from  $u$  to  $v$  in the dynamic segment graph, a single MBR  $(l, r) \times [b, e]$  is maintained. Upon the arrival of a new data point, a new edge from  $u$  to  $v$  with label  $(l', r', v, b', e')$  may be created in our algorithm. We merge the edge with the existing one by updating the MBR in  $G[u][v]$  as  $(\min(l, l'), \max(r, r')) \times [\min(b, b'), \max(e, e')]$ . When a query with range  $[x, y]$  arrives, we use the subgraph induced by the set of edges whose MBRs containing  $[x, y]$  to process the query.

**Optimized Dynamic Segment Graph Algorithm.** Algorithm 7 shows the pseudo-code of our optimized algorithm for incremental dynamic segment graph construction. It revises the unordered insertion process in several aspects. Firstly, it replaces the repetitive ANNS with a single ANNS that finds a set  $ann$  of  $Z$  approximate nearest neighbors of  $v_t$  among all existing data points at the beginning (Line 1). Secondly, in Lines 7-8, instead of adding the neighbor  $(L, L', v_e, R, R')$  to  $G[v_t]$  and  $(L, L', v_t, R, R')$  to  $G[v_e]$ , it merges the rectangle  $(L, L') \times [R, R']$  with the MBRs  $G[v_t][v_e]$  and  $G[v_e][v_t]$ . Thirdly, it removes the dominated neighbors to prevent them from generating children in the rectangle trees (Lines 11-14). Fourth, instead of adding edges to the dynamic graph only at the leaf nodes “in batches”, when visiting a tuple  $(L, L', R, R', x, xPNN)$  in the queue (except for the root), as the last point  $v_e$  in the pruned neighbor list  $xPNN$  must remain after pruning, we lodge edges between  $v_t$  and  $v_e$  (Lines 5-8). Lastly, we stop splitting the rectangles when there are  $M$  neighbors in the pruned neighbor list (Lines 9).

Upon the arrival of a query  $(q, [x, y], k)$ , we call Algorithm 1, 2DSEGMENTANNSearch( $G, q, [x, y], v_1, efsearch$ ). Among the returned neighbors, we report the  $k$  neighbors closest to  $q$ .



**Algorithm 7: DYNAMICSEGMENTGRAPHINSERTION( $G, v_t, M, Z$ )**


---

**Input:**  $G, v_t, M$  are the same as Algorithm 4;  $Z$ : an integer.  
**Output:**  $G$ : the dynamic segment graph for  $\{v_1, v_2, \dots, v_t\}$ .

```

1  $ann \leftarrow \text{2DSEGMENTANNSEARCH}(G, v_t, (-\infty_A, +\infty_A), v_1, Z);$ 
2  $queue.push(-\infty_A, att(v_t), att(v_t), +\infty_A, 0, \emptyset);$ 
3 while  $queue$  is not empty do
4    $(L, L', R, R', x, xPNN) \leftarrow queue.pop();$ 
5   if  $x \neq 0$  then
6      $v_e \leftarrow$  the last point in  $xNNlist$ ;
7     merge  $(L, L') \times [R, R']$  with  $G[v_t][v_e]$ ;
8     merge  $(L, L') \times [R, R']$  with  $G[v_e][v_t]$ ;
9   if  $x = M$  then continue;
10  while  $L \leq L'$  and  $L' \leq R'$  do
11     $v_c = \text{null}$ ;
12    foreach  $v \in ann$  in ascending order of  $\delta(v, v_t)$  do
13      if  $v \notin xPNN$  and  $att(v) \in (L, R')$  and  $v$  is not dominated
14        by any point in  $xPNN$  then
15           $v_c \leftarrow v$ ;
16      if  $v_c$  is not null then
17        Lines 1-9 of Algorithm 5, replace  $xNNlist$  with  $xPNN$ ;
18      else break;
19 return  $G$ ;
```

---

## 5 EXPERIMENT

**Environment.** We implement our methods and baselines in C++ and compiled them using GCC 9.2.0 with -O3 optimization. We ran all our experiments on a server with an Intel(R) Xeon(R) Platinum 8358 CPU@2.60GHz with 64 cores and 256GB of RAM.

**Datasets.** We used three real-world datasets. (1) YouTube: each vector is a 1024-dimensional RGB feature vector of a YouTube video. This dataset came from YouTube8M<sup>3</sup>. The attribute value of each vector is the release time of the corresponding video. (2) WIT<sup>4</sup>: each vector is a 2048-dimensional ResNet-50 embedding of an image from Wikipedia. We used the size of the image as the attribute value. (3) DEEP<sup>5</sup>: each vector is a 96-dimensional feature vector of an image, which is acquired from the last fully-connected layer of the GoogLeNet model [4]. Each vector is assigned a random number as the synthetic attribute value.

**Workloads and Baselines.** We design three RFANNS workloads to evaluate our optimized algorithm DSG (Algorithm 7) against 6 baselines. (a) *Unordered Insertion* (i.e., Problem 1). Only three baselines, Prefiltering, Postfiltering, and Acorn support this workload. Specifically, (1) Prefiltering builds a self-balanced binary search tree over the attribute values. When a query arrives, it scans all the vectors whose attribute values fall in the query range. (2) Postfiltering builds a HNSW graph for all the data vectors. To process a query, it performs ANNS and keeps a returned vector only if its attribute value is within the query range. It terminates when enough vectors are collected. (3) Acorn [26] is a graph index for predicate-agnostic approximate nearest neighbor search. It explores multi-hop neighbors that satisfy the query predicate during greedy search. (b) *Ordered Insertion* (i.e., Ordered-DRFANNS). (4) Except for the above

methods, the only baseline that supports this workload is SeRF [40] (as introduced in Section 2.2). (c) *Static* (i.e., Definition 3). WinFilter and iRange are designed for static datasets. They cannot support ordered/unordered insertion. Specifically, (5) WinFilter [5] builds a segment tree based on the attribute values of all data vectors. A graph-based index is created for each tree node. When a query arrives, it performs ANNS over a few segments (i.e., nodes) covered by or overlapping with the query range and merges the results. (6) iRange [36] also builds a segment tree. However, it merges the indexes on the tree nodes on the fly and performs ANNS only once.

**Query Scenarios.** All query vectors were selected uniformly at random from the vectors that were not in the stream of data vectors. For each query vector, the left boundary of its query range was selected from the attribute values of all the inserted vectors uniformly at random, while the right boundary was determined by the query range size. We evaluated small/medium/large (SR/MR/LR) query range sizes, which contain 1%/4%/16% of inserted vectors, respectively. In addition, we used the blended query range size (BR) that includes  $x\%$  of the inserted vectors where  $x$  was drawn from  $\{1\%, 2\%, 4\%, 8\%, 16\%, 32\%\}$  with equal probability. We do not test ranges that are too small – Prefiltering could be an efficient solution. We can build a simple cost model. The cost model calculates the number of vectors in the query range. If it is smaller than a threshold, use Prefiltering; otherwise, use indexes.

**Parameters.** For DSG, the parameters  $M$  and  $Z$  were set to 16/32/32 and 500/1000/1000, respectively, for DEEP/YouTube/WIT, unless stated otherwise. For SeRF, we employed the maxleap strategy with  $M = 16/32/64$  for DEEP/YouTube/WIT. Notably, the value of  $M$  in DSG is smaller than that in SeRF on YouTube because the maxleap strategy in SeRF, designed to reduce graph size, discards many neighbors. This results in sparse connectivity for small query ranges, requiring SeRF to use a higher  $M$  to achieve a recall above 0.95 in our experiments. Besides, we set  $K = 100$  for all datasets. Postfiltering used the same  $M$  and  $K$  as SeRF. Acorn also used the same  $M$  as SeRF, with  $\gamma = 10$  and  $M_\beta$  set equal to  $M$  for all datasets. For WinFilter, we used superpostfiltering with the parameters  $\beta = 2$ ,  $K = 500$ , and  $M = 64$ . We set  $efsearch = 80$  and the final multiply factor to 1. For iRange, we set  $M = 64$  and  $K = 100$  for WIT,  $M = 64$  and  $K = 400$  for YouTube, and  $M = 32$  and  $K = 100$  for DEEP. All baseline parameter settings were based on their papers or the optimal results from a grid search.

### 5.1 Comparison with Existing Methods

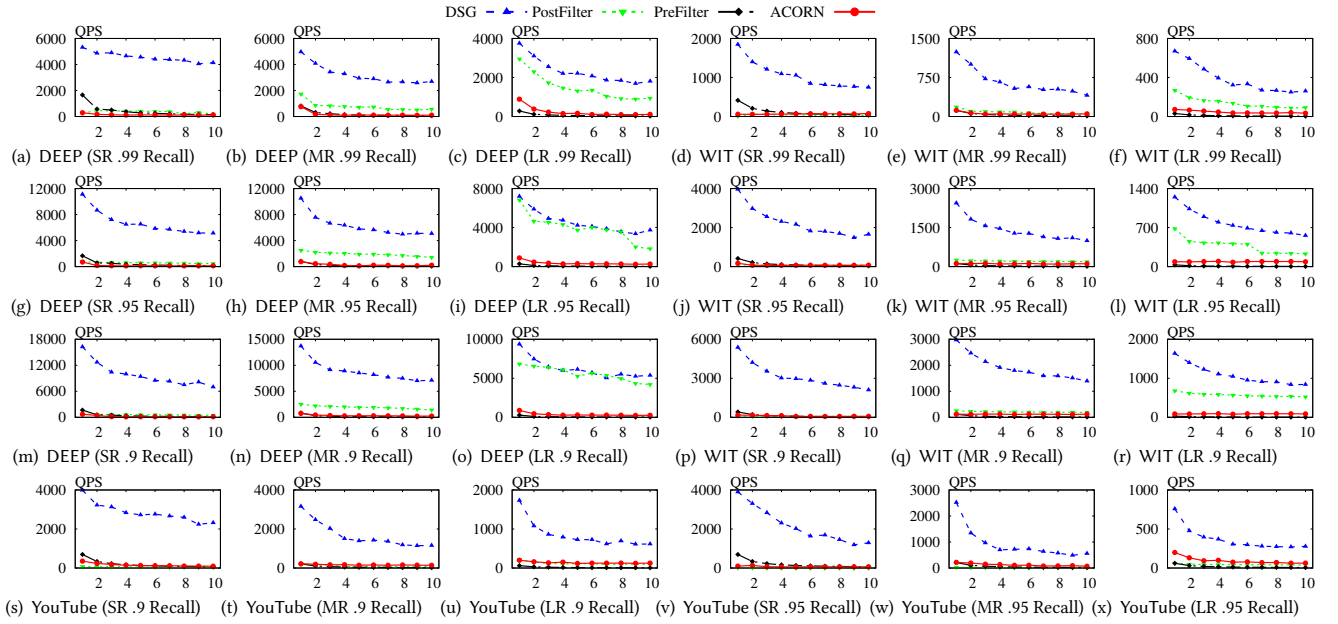
**Exp-1: Unordered Insertion.** We compare our method DSG with the only three baselines that support unordered insertion: Prefiltering, Postfiltering, and Acorn. We evaluated the query performance on SR/MR/LR after every 100,000 data vectors were inserted. We tuned the query parameters of these methods (except for Prefiltering, whose recall is always 1.0) such that their recall reached 0.99/0.95/0.9 on three datasets for all query scenarios and reported the QPS (query per second). Note that comparison at recall 0.99 on YouTube is omitted since all methods except Prefiltering cannot reach 0.99 recall. The results were averaged over 1,000 queries.

Figure 3 shows the results (the  $x$ -axis is the number of data vectors inserted in the unit of 100K). As expected, the QPS of all methods decreased almost logarithmically as more data vectors

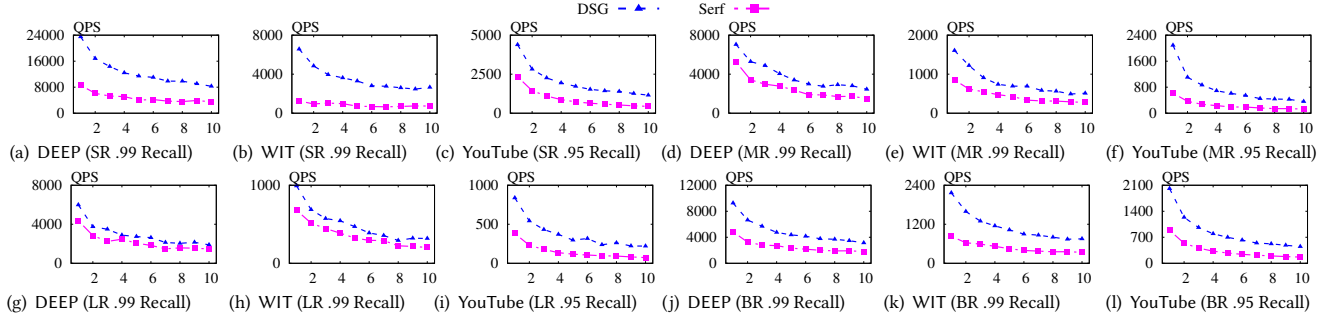
<sup>3</sup><https://research.google.com/youtube8m/download.html>

<sup>4</sup><https://github.com/google-research-datasets/wit>

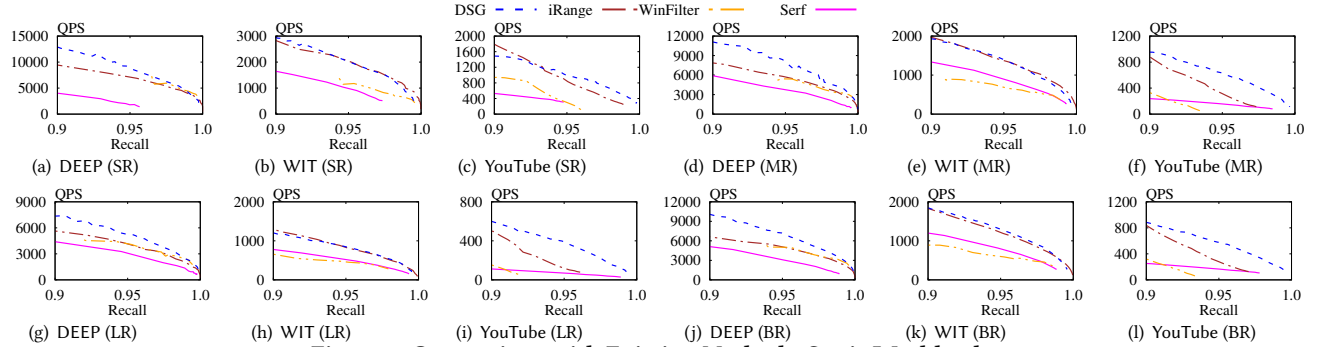
<sup>5</sup><https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search>



**Figure 3: Comparison with Existing Methods: Unordered Insertion (evaluated after every 100,000 data vectors inserted).**



**Figure 4: Comparison with Existing Methods: Ordered Insertion (evaluated after every 100,000 data vectors inserted).**



**Figure 5: Comparison with Existing Methods: Static Workload.**

were inserted. Nevertheless, DSG consistently and significantly outperformed the baselines. For example, on WIT at all recall levels, DSG achieved 1.6-2.5 $\times$  the QPS of the best-performing baseline Postfiltering throughout the process. The advantage was more obvious for SR, where DSG achieved 3-15 $\times$  the QPS of the best baselines at recall 0.99. This is because Postfiltering and Acorn only work well when the query range size is very large. For small query ranges, both Postfiltering and Acorn struggle to find enough neighbors that satisfy the range predicate, resulting in a longer search time to achieve the same recall as DSG. Although Prefiltering's recall was always 1.0, its QPS was extremely low. For example, on SR and LR in WIT, the QPS of DSG was 18 $\times$  and 87 $\times$  that of Prefiltering at 1M vectors.

**Exp-2: Ordered Insertion.** Next, we compare DSG with Serf for the ordered insertion workload. We omit Prefiltering, Postfiltering, and Acorn hereinafter as they were not competitive with DSG as illustrated in Exp-1. Note that the data vectors in the stream arrive in the ascending order of their attribute values in this workload. The settings are the same as the unordered insertion workload except that we require the recall of all methods to achieve at least only 0.9 for SR. This is because Serf cannot achieve a higher recall with its “max-leap” heuristic [40], which trades off index cost for reduced query performance.

Figure 4 shows the results. As we can see, DSG consistently outperformed Serf in all query scenarios throughout the process. For example, on DEEP and SR, with 100,000 vectors inserted, the QPS of DSG was 3 $\times$  that of Serf. Besides, it is worth to mention that DSG

**Table 1: Comparison of Index Cost.**

Dataset	Metric	WinF	iRange	DSG	Acorn	SeRF	PostF
DEEP	time (s)	27968	1052	1489	947	887	212.6
	size (GB)	7.28	3.23	3.08	0.69	0.64	0.53
WIT	time (s)	48471	16253	19449	25349	8532	1624
	size (GB)	30.15	13.48	10.65	8.66	8.18	8.20
YouTube	time (s)	71113	6775	31963	30201	6452	1046
	size (GB)	18.15	9.48	7.85	5.33	4.15	4.22

is more capable than SeRF as DSG supports unordered insertion whereas SeRF does not. The reason that DSG outperformed SeRF is that SeRF used the max leap heuristic which resulted in fewer edges and consequently poorer query performance.

*Exp-3: Static.* We compare DSG against iRange, SeRF, and WinFilter under the static workload. To build the index, iRange and WinFilter process the entire dataset at once, while SeRF and DSG insert vectors one by one in the order of their attribute values.

Figure 5 shows the recall (0.9 to 1.0) and QPS tradeoffs of all methods for various query scenarios and datasets. It can be observed that DSG almost always achieved the best QPS-recall tradeoff. For example, for YouTube and LR, the QPS of DSG was more than  $3\times$  than that of iRange when their recall was around 0.96. This is because DSG considers all the  $O(|\mathcal{D}|^2)$  attribute ranges, while iRange and WinFilter covers only up to  $O(|\mathcal{D}|\log(|\mathcal{D}|))$  ranges in the segment tree built on the vectors in  $\mathcal{D}$ . It is worth to mention that DSG is more capable than these baselines as DSG supports dynamic RFANNS whereas iRange and WinFilter do not.

*Exp-4: Index Cost.* Table 1 shows the index time and index size (memory footprint) of all methods for 1M vectors. Note that the index cost of Prefiltering was negligible and was omitted in the table. A single thread was employed. As we can see, the index sizes of WinFilter and iRange were significantly larger than DSG, though iRange had a lower index time. This is because WinFilter and iRange build multiple HNSW graphs for each segment in the segment tree, while DSG builds a lossless compression of many HNSW graphs directly. SeRF had a lower index cost than them as it used the max-leap heuristic. Although the index cost of Postfiltering was the lowest, its query performance (as well as SeRF’s) was extremely low for small query range sizes. Note our index construction can be accelerated through parallelization (similar to HNSW). Specifically, we can insert vectors concurrently into our index while using a spin-lock to manage reverse neighbor updates.

## 5.2 Sensitivity and Scalability Tests

*Exp-5: Optimization Sensitivity Test.* We evaluate the effectiveness of the three optimizations in Section 4 by comparing the following four combinations  $O_1$ ,  $O_{12}$ ,  $O_{13}$ , and  $O_{123}$  on DEEP with 10,000 data vectors. Here  $O_{xy}$  means using the combination of optimizations  $O_x$  and  $O_y$  as described in Section 4. The optimization  $O_1$  (one ANNS for all) is applied universally as the experiments would take too long to finish without it. Specifically, the index sizes (edges only) of  $O_1$ ,  $O_{12}$ ,  $O_{13}$ ,  $O_{123}$  were 45700MB, 220MB, 27MB, 27MB, while the index time was 508s, 8.5s, 288s, 8.1s. As we can see  $O_3$  (merge rectangles using MBR) significantly reduces the number of edges by merging edges, while  $O_2$  (removing pruned vectors) significantly reduces the indexing time. Figure 6 shows the query performance. As we can see, without  $O_2$  and  $O_3$ , the QPS was rather low. This is because the number of edges is huge in the graph. However, with

only  $O_2$ , the recall was low as the edges are sparse for small ranges without edge merging.

*Exp-6: Sensitivity Test on M.* We tested the sensitivity of DSG on the parameter of  $M$  (from 4 to 32) on three datasets. Figures 7(a), 7(d), and 7(g) show the QPS and recall. In general, as  $M$  increases, recall improves while QPS decreases. However, the effect of  $M$  on both recall and QPS saturates at approximately 16 for WIT and DEEP, and 32 for YouTube. This occurs because  $M$  defines the maximum degree of nodes, directly influencing recall and QPS. However, when  $M$  is sufficiently large, most nodes do not reach this limit due to the neighborhood pruning strategy, leading to a saturation effect in recall and QPS. The index size (edges only) for DEEP/WIT/YouTube was 6/8/6MB, 14/13/10MB, 20/17/15MB, 21/18/22MB respectively when  $M$  was 4, 8, 16, 32, while the index time was 3/46/29s, 9/85/44s, 21/136/78s, 31/173/140s. The index size did not increase proportionally with  $M$  as  $M$  is only the maximum degree (not the actual degree).

*Exp-7: Sensitivity Test on Z.* We tested the sensitivity of the parameter  $Z$  on three datasets by varying it from 400 to 1000. Figures 7(b), 7(e), and 7(h) show the query performance. For MR and LR, 400 was enough for all datasets. For SR (small query ranges), a large  $Z$  helps improve the recall. For example, as shown in Figure 7(e), on WIT, when  $Z$  increased from 400 to 1000, the recall improved from 0.87 to 0.96 for small query ranges. This is because the neighbors of a node in DSG are all from the  $Z$  approximate nearest neighbors of the node. A large  $Z$  helps the nodes in small query ranges to be connected in DSG, which benefits the recall. The index time for DEEP/WIT/YouTube was 21/61/31s, 30/86/46s, 40/111/59s, 51/136/71s respectively when  $Z$  was 400, 600, 800, 1000, while the index size (edges only) was 20/12/10MB, 23/13/12MB, 25/15/13MB, 27/17/15MB. This is because, a large  $Z$  adds more edges to the dynamic segment graph, making the index cost higher.

*Exp-8: Scalability Test.* We used 10K, 100K, 1M, 10M vectors in DEEP, and 10K, 100K, 1M vectors in WIT and YouTube to test the scalability of our method DSG. Figures 7(c), 7(f), and 7(i) show the query performance. For all query scenarios, the QPS only decreased sublinearly with the increase of the number of data vectors, while the recall remained very high (above 0.98). The index size scaled linearly while the index time grew sublinearly until 1M vectors. For example, in DEEP, the index sizes (edges only) were 0.03GB, 0.33GB, 3.73GB, 47.3GB respectively for 10K, 100K, 1M, 10M vectors, while the average insertion latencies were 2.8ms, 6.4ms, 8.7ms, 8.7ms.

## 6 RELATED WORK

**Approximate Nearest Neighbor Search (ANNS).** LSH (Locality-Sensitive Hashing) [1, 7, 12, 13, 21, 27], product quantization [2, 8, 16, 17, 25, 32], and proximity graph [3, 6, 11, 14, 22, 23] are three classes of indexes for ANNS. Each of them has a rich line of research. At a high level, LSH provides strong theoretical guarantees but does not perform well in practice. Product quantization effectively compresses the high-dimensional vectors into tiny codes that are suitable for linear scans, though its query accuracy is often not high enough. Many graph-based methods, such as HNSW [22, 23], NSG [6], and DiskANN [15], are approximations of the relative neighborhood graph (RNG), which bears favorable properties but is expensive to construct [14]. They typically offer low query latency

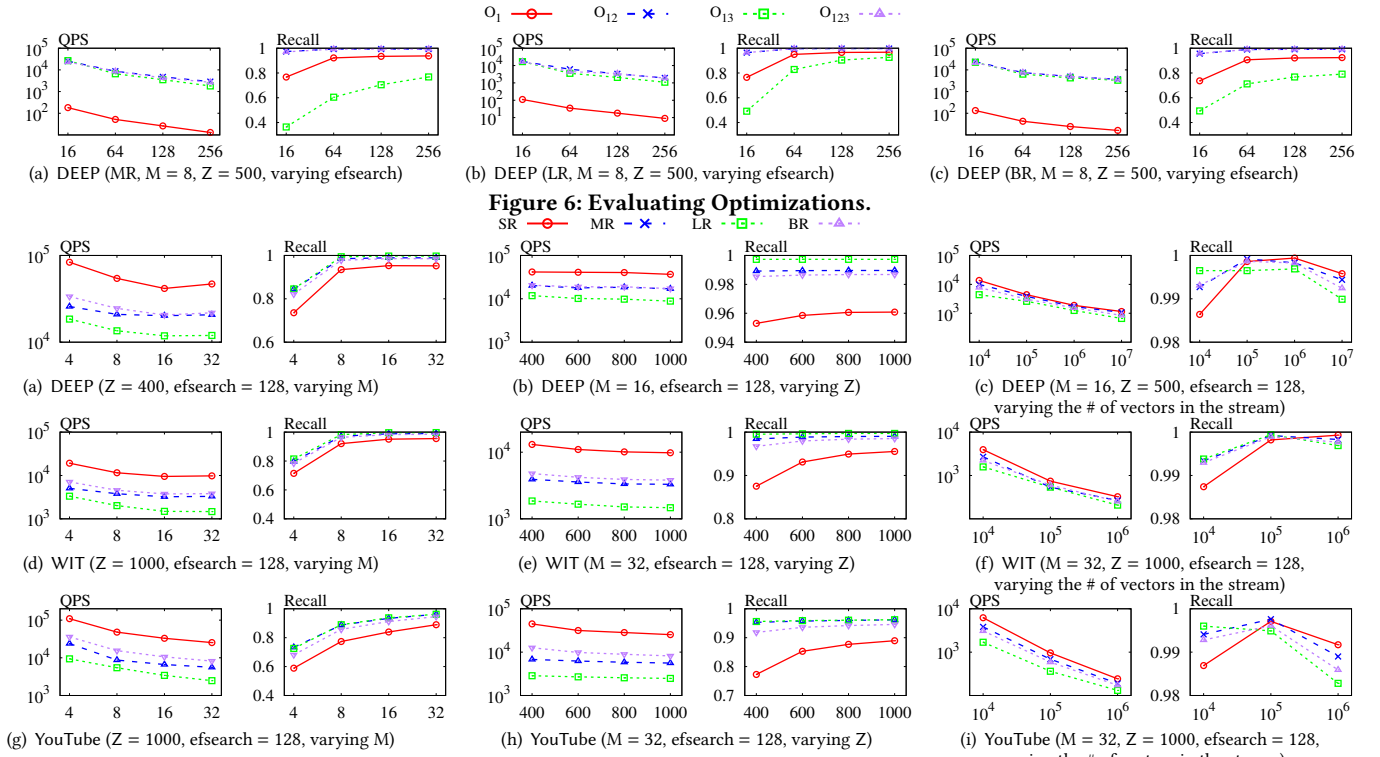


Figure 7: Sensitivity and Scalability Tests.

and high query accuracy, but their index sizes are often large since the vectors are not compressed.

**Attribute-Filtering Approximate Nearest Neighbor Search.** SeRF, iRange, and WinFilter are three recent works for range-filtering ANNS. SeRF introduces the segment graph, which is a compression of multiple HNSW graph, one for each possible query range. For half-bounded query range, it losslessly compresses  $n$  HNSW graphs using nearly the same index cost as building a single HNSW graph for  $n$  data points [40]. WinFilter proposes to build a segment tree over the attribute values of all data points [5]. For each segment containing a sufficient number of data points, a graph-based ANNS index is created. When a query arrives, it performs ANNS over a few segments overlapping with the query range and merges the results. Instead of performing multiple ANNS, iRange proposes to build an index based on the segments overlapping with the query range on the fly and search that index only [36]. FilteredDiskANN [9] is designed to process tag-filtered ANNS, where the tags of the returned approximate nearest neighbors must contain a few query tags. It proposes to incorporate the tag information in edge pruning. A node can only dominate other nodes sharing the same tags with it. Mesh [29] is designed for spatial-range-constrained ANNS, which retrieves the approximate  $k$ -nearest neighbors within a specified 2D rectangular spatial range for a given query vector. ACORN is designed for predicate-agnostic ANNS [26], where the predicate is arbitrary (e.g., regex match, keyword match, etc). It proposes to explore two-hop neighbors during the greedy search in case there are not enough one-hop neighbors satisfying the query predicate. It further compresses the graph by removing the two-hop neighbors from the neighbor list. A few studies, including AnalyticDB-V [33] and reconfigurable inverted index (Rii) [24] propose to design cost models to choose from pre-filtering and post-filtering for attribute-filter ANNS. Milvus further proposes to partition the dataset and apply different approaches for different partitions [30]. NHQ [31] and HQANN [34] propose to fuse the

attribute values into the vectors for attribute-filtering ANNS. ARK-Graph studies how to compress the approximate  $k$ -nearest neighbor graphs of all ranges [39]. Note that it does not discuss the impact of HNSW pruning. Zhao et al. [37] design a few optimizations for attribute-filtering ANNS, including entry point selection, biased priority queue selection, and multi-direction search.

**Dynamic Approximate Nearest Neighbor Search.** Insertion can be naturally supported by the HNSW graph as it is constructed by repeatedly inserting nodes to the graph. FreshDiskAnn [28] designs update rules for the Vamana graph, a variant of the HNSW graph. The deletion rule can be generalized to maintain the HNSW graph. Xu et al. propose online product quantization which incrementally updates the quantization codebook to accommodate incoming streaming data [35]. Leng et al. study online sketching hashing to handle new data points in data-dependent hashing-based methods for approximate nearest neighbor search [18].

## 7 CONCLUSIONS

Range-filtering approximate nearest neighbor search (RFANNS) identifies approximate nearest neighbors for a query vector among data vectors whose attributes fall within a specified range. Existing RFANNS methods are designed for static datasets and struggle with dynamic scenarios where data vectors arrive continuously. To address this, we propose the dynamic segment graph, which losslessly compresses multiple hierarchical navigable small-world (HNSW) graphs, each corresponding to a query range. This structure supports efficient insertion of incoming data vectors. We analyze its time and space complexity and introduce optimizations to reduce index costs in practice.

## ACKNOWLEDGMENTS

Dong Deng\* was supported by the National Science Foundation grant #2212629 and a gift from Adobe Research.

## REFERENCES

- [1] Alexandr Andoni and Piotr Indyk. 2006. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *FOCS*. 459–468.
- [2] 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.
- [3] Sunil Arya and David M. Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *ACM/SIGACT-SIAM*. 271–280.
- [4] 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>
- [5] Joshua Engels, Benjamin Landrum, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2024. Approximate Nearest Neighbor Search with Window Filters. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21–27, 2024*. OpenReview.net. <https://openreview.net/forum?id=8t8zBaGFar>
- [6] 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.
- [7] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*. 541–552.
- [8] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. 2946–2953.
- [9] 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.
- [10] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.
- [11] Ben Harwood and Tom Drummond. 2016. FANN: Fast Approximate Nearest Neighbour Graphs. In *CVPR*. 5713–5722.
- [12] 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.
- [13] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. 604–613.
- [14] 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>
- [15] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NeurIPS*, Vol. 32.
- [16] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *TPAMI* 33, 1 (2011), 117–128.
- [17] Yannis Kalantidis and Yannis Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. 2329–2336.
- [18] Cong Leng, Jiaxiang Wu, Jian Cheng, Xiao Bai, and Hanqing Lu. 2015. Online sketching hashing. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7–12, 2015*. IEEE Computer Society, 2503–2511. <https://doi.org/10.1109/CVPR.2015.7298865>
- [19] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
- [20] Jimmy Lin. 2024. Operational Advice for Dense and Sparse Retrievers: HNSW, Flat, or Inverted Indexes? [arXiv:2409.06464](https://arxiv.org/abs/2409.06464) [cs.LG] <https://arxiv.org/abs/2409.06464>
- [21] 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.
- [22] Yuri 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.
- [23] 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.
- [24] Yusuke Matsui, Ryota Hinami, and Shin’ichi Satoh. 2018. Reconfigurable Inverted Index. In *MM*. 1715–1723. <https://doi.org/10.1145/3240508.3240630>
- [25] 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.
- [26] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proc. ACM Manag. Data* 2, 3 (2024), 120. <https://doi.org/10.1145/3654923>
- [27] Anshumali Shrivastava and Ping Li. 2014. Asymmetric LSH (ALSH) for Sublinear Time Maximum Inner Product Search (MIPS). In *NeurIPS*. 2321–2329.
- [28] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. [arXiv:2105.09613](https://arxiv.org/abs/2105.09613) [cs.LG] <https://arxiv.org/abs/2105.09613>
- [29] Yitong Song, Bin Yao, Zhida Chen, Xin Yang, Jiong Xie, Feifei Li, and Mengshi Chen. 2025. Efficient top-k spatial-range-constrained approximate nearest neighbor search on geo-tagged high-dimensional vectors. *The VLDB Journal* 34, 1 (2025), 1–21.
- [30] 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>
- [31] 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](https://arxiv.org/abs/2203.13601) [cs.DB]
- [32] 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>
- [33] 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>
- [34] 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>
- [35] Donna Xu, Ivor W. Tsang, and Ying Zhang. 2018. Online Product Quantization. *IEEE Trans. Knowl. Data Eng.* 30, 11 (2018), 2185–2198. <https://doi.org/10.1109/TKDE.2018.2817526>
- [36] Yuxuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. *CoRR* abs/2409.02571 (2024). <https://doi.org/10.48550/ARXIV.2409.02571> [arXiv:2409.02571](https://arxiv.org/abs/2409.02571)
- [37] 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](https://arxiv.org/abs/2210.14958)
- [38] Liang Zheng, Liyue Shen, Lu Tian, Shengjin Wang, Jingdong Wang, and Qi Tian. 2015. Scalable person re-identification: A benchmark. In *Proceedings of the IEEE international conference on computer vision*. 1116–1124.
- [39] 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>
- [40] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 1 (2024), 69:1–69:26. <https://doi.org/10.1145/3639324>