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 systems 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 arrives 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, proved 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 retaining 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, 14(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

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

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

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.

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 [29], retrieval-augmented generation [19], document retrieval [20], and person or vehicle re-identification [37]. Two straightforward approaches for RFANNS, pre-filtering and post-filtering, do not work well when query range size shifts [39]. To address this issue, specialized index structures have been proposed recently, including SeRF [39], iRange [35], 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 *naive approach* to dynamic RFANNS maintains $O(t^2)$ HNSW graphs where $t = |\mathcal{D}|$, one for each *basic range* – a range $[l, r]$ is basic if both l and r are attribute values of vectors in $\mathcal{D}$. When a new vector v with attribute a arrives at the system, there are expectedly $O(t^2)$ HNSW graphs whose basic ranges $[l, r]$ satisfy $\mathrm{a} \in [l, r]$. Thus, one needs to add $O(\mathrm{M}t^2)$ new edges to the HNSW graphs where M is the maximum degree parameter of HNSW. This becomes prohibitive as t grows in a data stream. To compress these HNSW graphs for an efficient update, we make the following observation.

Consider a time when $\mathcal{D}$ has t vectors $v_1, v_2, \cdots, v_t$ whose attribute values are (after sorting) $\mathrm{a}_1 \le \mathrm{a}_2 \le \cdots \le \mathrm{a}_t$. When a new vector v with value a arrives, let i and j be two integers such that $\mathrm{a}_i \le \mathrm{a} \le \mathrm{a}_j$. In this case, the naive solution would add M new edges to v in the HNSW graph for range $[\mathrm{a}_i, \mathrm{a}_j]$. Consider two "adjacent" basic ranges $[\mathrm{a}_i, \mathrm{a}_j]$ and $[\mathrm{a}_i, \mathrm{a}_{j+1}]$ where $j < t$. It is likely that v has the same list of K nearest neighbors under the two query ranges because $\mathcal{D}$ has only one vector difference on the two ranges; if that happens, we can merge the edges of v in the two corresponding HNSW graphs in the naive solution by adding a "rectangle" label of $(\mathrm{a}_{i-1}, \mathrm{a}_i] \times [\mathrm{a}_j, \mathrm{a}_{j+2})$ to the corresponding edges for the compression. A query range $[x, y]$ activates the edges with "rectangles" $(l, r] \times [b, e)$ where $x \in (l, r]$ and $y \in [b, e)$ for nearest neighbor search. More importantly, we prove that by recursively merging adjacent ranges, we only need to insert $O(\mathrm{K}^2\mathrm{M}\log t)$ edges (with rectangles) on v, which is dramatically smaller than $O(\mathrm{M}t^2)$ with an increasing t. K is the "efconstruct" parameter of HNSW graph.

To support the above observation, we devise a conceptual structure called *rectangle tree* which provides insights to dynamic RFANNS in answering the following questions upon the arrival of a new data vector v with a random attribute value a. i) How to represent the aggregation of all possible query ranges based on the K-nearest neighbors of v in the query ranges, i.e., how to compactly represent the new "edges" from/to v? ii) Is the representation canonical? iii) What is the complexity of the representation and its identification?

To this end, we design a structure called the *dynamic segment graph* G where each data vector is a node in it and each edge is labeled with a "rectangle" $(l, r] \times [b, e)$. For the new vector v, we find a way of generating edges with "rectangle" labels in the dynamic segment graph such that 1) under certain constraints, the subgraph induced by the activated edges is exactly the HNSW graph over data vectors with attribute values in $[x, y]$ to ensure the search quality; and 2) the expected number of new edges is $O(\mathrm{K}^2\mathrm{M}\log t)$ when inserting the t-th vector. We develop optimizations to significantly reduce the index cost of the dynamic segment graph; the query performance of dynamic segment graph remains robust under the optimizations empirically.

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

- To the best of our knowledge, this paper is the first to study the dynamic range-filtering approximate nearest neighbor search problem.
- 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 rigorously 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 and show our method outperforms existing methods significantly for both static and dynamic RFANNS.

2 Preliminary

2.1 Problem Definition

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

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 $\mathsf{kNN}_\delta(q, \mathcal{D})$, is the set of k vectors in $\mathcal{D}$ with the smallest distances to q under metric δ. Formally, $\mathsf{kNN}_\delta(q, \mathcal{D})$ is a set $\mathcal{R} \subseteq \mathcal{D}$ of k vectors in $\mathcal{D}$ such that $\forall u \in \mathcal{R}$ and $\forall v \in \mathcal{D} \setminus \mathcal{R}$, $\delta(u, q) \le \delta(v, q)$.*

We omit the subscription δ when the context is clear.

Let A be an attribute whose domain 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 $\mathsf{att}(v) \in \mathsf{Dom}(\mathrm{A})$ 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 introduce $-\infty_\mathrm{A}$ as a placeholder that is smaller than any attribute value in Dom(A) and $+\infty_\mathrm{A}$ a placeholder that is larger than any value in Dom(A). Define $-\infty_\mathrm{A} < +\infty_\mathrm{A}$ and use $(-\infty_\mathrm{A}, +\infty_\mathrm{A})$ to denote a range that contains all the attribute values in Dom(A).

Definition 2 (Range-Filtering Nearest Neighbor Search [39]). *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 $q \in \mathbb{R}^d$, $l, r \in \mathsf{Dom}(A)$, and k a positive integer. Define $\mathcal{D}[l, r] \doteq \{v | v \in \mathcal{D}, \mathsf{att}(v) \in [l, r]\}$. The query returns $\mathsf{kNN}(q, \mathcal{D}[l, r])$, a k-sized subset $\mathcal{R}$ of $\mathcal{D}[l, r]$ such that $\forall\ u \in \mathcal{R}$ and $\forall\ v \in \mathcal{D}[l, r] \setminus \mathcal{R}$, $\delta(u, q) \le \delta(v, q)$.*

For simplicity, assume there are always at least k vectors in $\mathcal{D}[l, r]$, i.e., $|\mathcal{D}[l, r]| \ge k$. 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 $\mathsf{kANN}(q, \mathcal{D})$ of k vectors aiming at an optimized recall $\frac{1}{k}|\mathsf{kANN}(q, \mathcal{D}) \cap \mathsf{kNN}(q, \mathcal{D})|$ for a vector q.

Definition 3 (RFANNS [39]). *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 $\mathsf{kANN}(q, \mathcal{D}[l, r])$, a set of k vectors in $\mathcal{D}[l, r]$, with an optimized recall $\frac{|\mathsf{kANN}(q, \mathcal{D}[l,r]) \cap \mathsf{kNN}(q, \mathcal{D}[l,r])|}{k}$.*

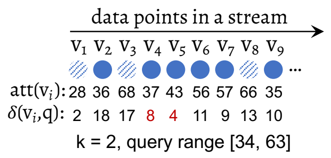

data points in a stream
v1 v2 v3 v4 v5 v6 v7 v8 v9 ...
att(vi): 28 36 68 37 43 56 57 66 35
δ(vi,q): 2 18 17 8 4 11 9 13 10
k = 2, query range [34, 63]

Figure 1: An example of DRFANNS.

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

Problem 1 (Dynamic Range-Filtering Approximate Nearest Neighbor Search (DRFANNS)). *Let $v_1, v_2, \cdots$ 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 $\mathrm{att}(v_t)$; denote by $\mathcal{D}_t = \{v_1, v_2, \cdots, v_t\}$ 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* $\mathrm{kANN}(q, \mathcal{D}_t[l, r])$.

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 $\mathcal{D}_9 = \{v_1, v_2, \cdots, v_9\}$ 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 $\mathrm{kNN}(q, \mathcal{D}_9[34, 63]) = \{v_4, v_5\}$. Note that although v_1 is closer to q than v_4 and v_5, they should not be reported as its attribute values $\mathrm{att}(v_1) = 28$ 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 [1] of their A-attribute values. Ordered-DRFANNS can be addressed by an existing technique SeRF [39]. In its settings, for any two positive integers i and j with $i < j$, $\mathrm{att}(v_i) < \mathrm{att}(v_j)$.

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 [39], 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). Given a RFANNS query $(q, [x, y], k)$, v is a neighbor of u only if there is a tuple (l, r, v, b, e) in the neighbor list of u such that the query attribute range $[x, y]$ has $x \in (l, r]$ and $y \in [b, e)$. Note that it is always true that $r \le \mathrm{att}(v) \le b$. The working edges of $\mathbb{G}$ for search are subjected to the query range.

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)[2]. 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).

[1] or descending order. We restrict our discussion to the ascending order for simplicity.
[2] 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.

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

```
Input: 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;
```

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 $(\mathrm{att}(v_i), \mathrm{att}(v_{i'})]$ that for all query ranges $[x, y]$ with $x \in (\mathrm{att}(v_i), \mathrm{att}(v_{i'})]$ and $y \ge \mathrm{att}(v_j)$, the neighbor list of v_j would be the same; we call $(\mathrm{att}(v_i), \mathrm{att}(v_{i'})]$ 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 $(\mathrm{att}(v_i), +\infty_{\mathrm{A}})$ (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*, $(\mathrm{att}(v_i), \mathrm{att}(v_{i'})]$ 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 $\mathcal{D}[\mathrm{att}(v_i), \mathrm{att}(v_j)]$, where $1 \le i \le j \le n$. 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 ($x \in (l, r]$ or $x = l$) and ($y \in [b, e)$ or $y = e$).

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: $\mathcal{D} = \{v_1, v_2, \cdots, v_n\}$; K: an integer; M: the max degree.
Output: $\mathbb{G}$: 2D segment graph for $\mathcal{D}$.
1 foreach $1 < j \leq n$ do
2     For dummy vector $v_0$, let $\mathrm{att}(v_0) \leftarrow -\infty_{\mathrm{A}}$; $i = 0$;
3     while $i < j - 1$ do
4         $ann \leftarrow$2DSEGMENTANNSEARCH($\mathbb{G}, v_j, (\mathrm{att}(v_i), +\infty_{\mathrm{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 $(\mathrm{att}(v_i), \mathrm{att}(v_{i'}), v, \mathrm{att}(v_j), +\infty_{\mathrm{A}})$ to $\mathbb{G}[v_j]$;
8             add $(\mathrm{att}(v_i), \mathrm{att}(v_{i'}), v_j, \mathrm{att}(v_j), +\infty_{\mathrm{A}})$ to $\mathbb{G}[v]$;
9         $i = i'$;
10 return $\mathbb{G}$;
```

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 $\mathcal{D}_{t-1} = \{v_1, \cdots, v_{t-1}\}$ 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, 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 $\mathcal{D}_{t-1} = \{v_1, v_2, \cdots, v_{t-1}\}$ on the stream of $v_1, v_2, \cdots, v_t, \cdots$. We define the structure and show its fine properties in Section 3.1. Section 3.2.1 uses the rectangle tree structure 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, skipping Sections 3.1.2 and 3.1.3 directly works.

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 $\mathrm{att}(v_t) \in [l, r]$, 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 $v_{i_1}, v_{i_2}, \cdots, v_{i_x}$ is the xNNlist sequence of $[l, r]$ if

(1) They all arrive before v_t, i.e., time $i_1, i_2, \cdots, i_x < t$;
(2) Their attribute values $\mathrm{att}(v_{i_1}), \mathrm{att}(v_{i_2}), \cdots, \mathrm{att}(v_{i,x}) \in [l, r]$;
(3) Their distances to v_t ascend, i.e., $\delta(v_{i_1}, v_t) < \cdots < \delta(v_{i_x}, v_t)$;
(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 $\mathrm{sap}_{t-1}(x)$ as the set of all the short attribute pairs to x. Formally,

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

Nodes. A node v^T of a rectangle tree is a tuple in the form of

$$v^T = (\mathrm{L}, \mathrm{L}', \mathrm{R}, \mathrm{R}', x, \mathrm{N}_x)$$

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

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 $x = \mathrm{K}$ 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 $x \in [0, \mathrm{K}]$, their xNNlist sequences must be different.

Covering condition. For a node $v^T = (\mathrm{L}, \mathrm{L}', \mathrm{R}, \mathrm{R}', x, \mathrm{N}_x)$ in the rectangle tree with p children at level $x + 1$. Denote the set of its

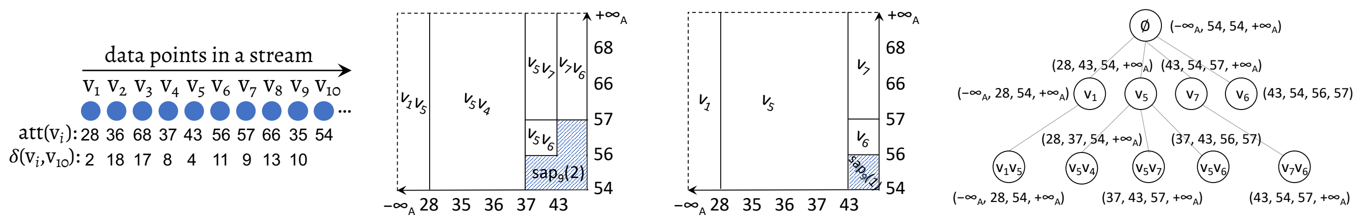


Figure 2: A running example of the rectangle tree.

children's rectangles as $\mathcal{S} = \{S_1, S_2, \cdots, S_p\}$. $\mathcal{S}$ covers all the attribute pairs in $\mathrm{rect}(v^T)$ that have an $(x+1)$NNlist sequence, i.e.,

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

Example 2. Figure 2 on the right 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 xNNlist sequence of range $[l, r] = \mathsf{N}_x = \mathsf{N}'_x$ since the xNNlist sequence of $[l, r]$ is unique, conflicting the distinct condition that $\mathsf{N}_x \neq \mathsf{N}'_x$. Thus attribute pair (l, r) does not exist. Thus S_v and S_u are disjoint. □

Lemma 2. *For a non-leaf rectangle-tree node with a rectangle S_0 at level $x-1$, the rectangles $\mathcal{S} = \{S_1, S_2, \cdots, S_p\}$ of all its children (at level x) form a disjoint partitioning of $S_0 \setminus \mathrm{sap}_{t-1}(x)$. 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 $(l, r) \in S_0 \setminus \mathrm{sap}_{t-1}(x)$, there is one (covering condition) and only one (Lemma 1) rectangle S in $\mathcal{S}$ such that $(l, r) \in S$. □

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

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

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

Proof. Firstly, for any two integers $0 \leq x'' < x' \leq \mathrm{K}$, we have $\mathrm{sap}_{t-1}(x'') \subseteq \mathrm{sap}_{t-1}(x')$ 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 $S \setminus \mathrm{sap}_{t-1}(x)$. Secondly, we prove that the partitioning is canonical. Consider all the attribute pairs in $S \setminus \mathrm{sap}_{t-1}(x)$ that have xNNlist sequences. Since each rectangle $S_i \in \mathcal{S}$ ensures that all pairs $(l, r) \in S_i$ have the same xNNlist sequence; Distinct condition ensures that pairs from different rectangles in $\mathcal{S}$ have different xNNlist sequences. Therefore, $\mathcal{S}'$ must be a grouping of all the attribute pairs (l, r) in S based on their xNNlist sequences which is $\mathcal{S}''$ and is thus canonical. □

Our following average case analysis assumes the **independence** between the attribute values and distances, i.e., the attribute values of vectors in $\{v_1, v_2, \cdots, v_t\}$ 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 $\mathrm{att}(v_t) > \mathrm{att}(v_1), \mathrm{att}(v_2), \cdots, \mathrm{att}(v_{t-1})$ and let the rectangle of the root be $(-\infty, \mathrm{att}(v_t)) \times [\mathrm{att}(v_t), +\infty)$. The number of leaf nodes of the rectangle tree at level K is in the worst case $O(t)$ and $O(\mathrm{K} \ln t)$ expected.*

Proof. According to Theorem 3, the number of nodes at level K (the leaf nodes) is the same as the number of different KNNlist sequences. As the indexes of vectors $v_1, \cdots, v_{t-1}$ do not affect the construction of the rectangle tree when inserting v_t, assume in this proof that $\delta(v_1, v_t) < \delta(v_2, v_t) < \cdots < \delta(v_{t-1}, v_t)$. For each $i \in [1, t]$, denote by $S_i = \{v_1, \cdots, v_i\}$ the set of the first i vectors. Let S'_i be the set of K vectors in S_i with the largest attribute values. $S'_{\mathrm{K}} = S_{\mathrm{K}}$ is a KNNlist sequence of all ranges (l, r) with $l \leq \min_{v \in S'_{\mathrm{K}}} \mathrm{att}(v) \leq \mathrm{att}(v_t) \leq r$, i.e., in rectangle $(-\infty_{\mathrm{A}}, \min_{v \in S'_{\mathrm{K}}} \mathrm{att}(v)] \times [\mathrm{att}(v_t), +\infty_{\mathrm{A}})$. For each v_i, $i > \mathrm{K}$, v_i can enter S'_i with probability $\frac{i}{\mathrm{K}}$ based on the independence assumption. S'_i is the KNNlist sequence of ranges (l, r) with $l \in (\min_{v \in S'_{i-1}} \mathrm{att}(v), \min_{v \in S'_i} \mathrm{att}(v)]$ and $r \in [\mathrm{att}(v_t), +\infty_{\mathrm{A}})$. Therefore, different sets among S'_i, $i \in [\mathrm{K}, t]$, cover the rectangle of $(-\infty_{\mathrm{A}}, \mathrm{att}(t)] \times [\mathrm{att}(t), +\infty_{\mathrm{A}})$ jointly. The total number of different KNNlist sequences is thus $O(t)$ in the worst case, and $O(\sum_{j \in [\mathrm{K}, t]} \frac{\mathrm{K}}{t}) = O(\mathrm{K} \ln t)$ in expectation. This concludes the proof. □

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

3.1.3 Proof of Theorem 4

According to Theorem 3, the number of nodes at level K (the leaf nodes) is the same as the number of different KNNlist sequences of all attribute pairs in the rectangle $(-\infty_\mathrm{A}, \mathrm{att}(v_t)] \times [\mathrm{att}(v_t), +\infty_\mathrm{A})$. As the indexes of vectors $v_1, \cdots, v_{t-1}$ do not affect the construction of the rectangle tree when inserting v_t, we rename the vectors in $\mathcal{D}_{t-1}$ such that $\delta(v_1, v_t) < \delta(v_2, v_t) < \cdots < \delta(v_{t-1}, v_t)$. For $\forall i \in [1, t-1]$, denote by $S_i = \{v_1, \cdots, v_i\}$ the set of i vectors closest to v_t.

For an attribute value val, define its predecessor in S_i as $\mathrm{pred}_i(val) =$

$$\begin{cases} -\infty_\mathrm{A}, & \text{if } val \leq \min_{v \in S_i} \mathrm{att}(v) \\ \max_{v \in S_i, \mathrm{att}(v) < val} \mathrm{att}(v), & \text{if } \min_{v \in S_i} \mathrm{att}(v) < val \leq \mathrm{att}(v_t) \\ \mathrm{att}(v_t) & \text{if } val > \mathrm{att}(v_t). \end{cases}$$

Define the successor of val in S_i as $\mathrm{succ}_i(val) =$

$$\begin{cases} +\infty_\mathrm{A}, & \text{if } val \geq \max_{v \in S_i} \mathrm{att}(v) \\ \min_{v \in S_i, \mathrm{att}(v) > val} \mathrm{att}(v), & \text{if } \max_{v \in S_i} \mathrm{att}(v) > val \geq \mathrm{att}(v_t) \\ \mathrm{att}(v_t) & \text{if } val < \mathrm{att}(v_t). \end{cases}$$

Let S_i^L be the set of at most (depends on the availability) K vectors in S_i with the largest attribute values that are smaller than $\mathrm{att}(v_t)$; S_i^R that in S_i with the smallest attribute values that are larger than $\mathrm{att}(v_t)$. Denote by w the size $|S_i^L|$ and m the size $|S_i^R|$, i.e., $w, m \leq \mathrm{K}$.

For each $i \geq \mathrm{K}$, we sort all nodes in $S_i^L \cup S_i^R$ based on their attribute values as the following sequence called the *LR sequence*.

$$v_1^l, v_2^l, \cdots, \underline{v_z^l}, \cdots, v_w^l, \cdots, v_w^l, v_1^r, \cdots, v_m^r.$$

Report Process. For each $i \geq \mathrm{K}$, we report rectangles with their KNNlist sequences when v_i enters the LR sequence. v_i can appear in any position, for example, v_i could be v_z^l. Consider each window $W$of size K in this sequence that contains a consecutive K vectors including v_i – note that the window may include both vectors from S_i^L and S_i^R, and there are at most $w + m - \mathrm{K} \leq \mathrm{K}$ such windows. Let v^l be the leftmost vector in the window and v^r the rightmost vector in the window. W is the KNNlist sequence (after sorted based on distances to v_t) exclusively to the rectangle below

$$\mathrm{rect} = (\mathrm{pred}_i(\mathrm{att}(v^l)), \mathrm{att}(v^l)] \times [\mathrm{att}(v^r), \mathrm{succ}_i(\mathrm{att}(v^r))).$$

To see the reason, v_i is the node in the window W with the longest distance to v_t and there are exactly $\mathrm{K} - 1$ vectors in the window whose distances are smaller than that of v_i. To include the nodes in W in $[l, r]$, we must have $l \leq \mathrm{att}(v^l)$ and $\mathrm{att}(v^r) \leq r$. Since $r \geq \mathrm{att}(v^r)$, once l goes to the left of $\mathrm{pred}_i(\mathrm{att}(v^l))$, $[l, r]$ includes the vector that holds the predecessor attribute of v^l in S_i, then there will be at least K vectors in $[l, r]$ whose distance to v_t is smaller than that of v_i, then v_i will not appear in the KNNlist sequence, contradiction. Thus, $l > \mathrm{pred}_i(\mathrm{att}(v^l))$. Similarly, we have $r < \mathrm{succ}_i(\mathrm{att}(v^r))$.

Completeness. Next we show that for any attribute interval $[l, r]$ with $\mathrm{att}(v_t) \in [l, r]$, their KNNlist sequence and the corresponding rectangle are reported in the above process. Find the KNNlist sequence S of the interval and let v_i be the last node in the sequence (farthest to v_t), and let v^l be the vector in the sequence with the smallest attribute value and v^r the vector the largest. Note that KNNlist should be a subset of S_i (based on the distance ordering). Note that all vectors in S_i with attribute values in $[\mathrm{att}(v^l), \mathrm{att}(v^r)]$ should be in S because if otherwise, v_i will not be the K-th vector in the KNNlist sequence – there are more than $\mathrm{K} - 1$ vectors in $[l, r]$ whose distance to v_t is smaller than v_i. Therefore, S corresponds to a window on the sorted sequence of $S_i^L \cup S_i^R$, and thus has been reported in the above reporting process.

Complexity. In the worst case, a total of $O(\mathrm{K}t)$ windows will be reported each corresponding to a KNNlist sequence. In expectation, for each v_i, $i \geq \mathrm{K}$, v_i enters S_i^L with probability $\frac{\mathrm{K}}{i}$; so does in S_i^R. Once v_i gets in $S_i^L \cup S_i^R$, there will be at most K windows to be reported. Therefore, the total number of windows reported in expectation is $O(\mathrm{K}^2 \ln t)$. This concludes the proof.

Algorithm 4: OrderedInsertion(G, v_t, M, K)

```
Input: G: the dynamic segment graph constructed for
    $\mathcal{D}_{t-1} = \{v_1, v_2, \cdots, v_{t-1}\}$; $v_t$: a vector arriving at time $t$; M
    and K: the parameters in HNSW construction
Output: G: the dynamic segment graph for $\{v_1, v_2, \cdots, v_t\}$.
// queue is a min-heap of tuples in the form of
    (L, L′, R, R′, $x$, $x$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′ or $\in$ [R, R′)), $v_t$ has the same set of
    the $x$ nearest neighbors on $\mathcal{D}_{t-1}$, which is $x$NNlist.
1  queue.push($-\infty_\mathrm{A}$, att($v_t$), att($v_t$), $+\infty_\mathrm{A}$, 0, $\emptyset$);
2  while queue is not empty do
3      (L, L′, R, R′, $x$, $\mathrm{N}_x$) ← queue.pop();
4      if $x$ = K then
5          foreach $v \in$ Prune($\mathrm{N}_x$, 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         ann ← 2DSegmentANNSearch(G, $v_t$, (L, R′), $v_1$, K);
11         if ann ⊆ $\mathrm{N}_x$ then goto Line 5;
           // When the # of vectors in attribute range (L, R′) is
               ≤ $x$, call this node a sap node.
12         $v_c$ ← arg $\min_{v \in ann \setminus \mathrm{N}_x} \delta(v, v_t)$;
13         if att($v_c$) < L′ then
14             queue.push(L, att($v_c$), R, R′, $x$ + 1, $\mathrm{N}_x + v_c$);
15             L ← att($v_c$);
16         else
17             queue.push(L, L′, R, R′, $x$ + 1, $\mathrm{N}_x + v_c$);
18             break;
19 return G;
```

3.2 Dynamic Segment Graph

3.2.1 Ordered Insertion

Consider the problem of Ordered-DRFANNS where the attributes of data vectors have $\mathrm{att}(v_1) < \mathrm{att}(v_2) < \cdots < \mathrm{att}(v_t)$. 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 find a method in assessing the impact of the approximation of the nearest neighbor search to the index complexity.

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* xNNlist *sequence* N_x*.*

Explanations to Algorithm 4 and Proof Sketch to Lemma 4. The root has level $x = 0$, $\emptyset$ is the 0NNlist 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 xNNlist 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 xNNlist 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. □

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* $x = y$*.*

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

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 expection. 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 $O(K^2 \ln t)$. □

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 $v_c = v_5$.

Algorithm 5: UnorderedInsertion(G, v_t, M, K)

```
// Replace Lines 13-18 of Algorithm 4 with code:
1 if $att(v_c) < L'$ then
2     $queue$.push($L, att(v_c), R, R', x+1, N_x + v_c$);
3     $L \leftarrow att(v_c)$;
4 else if $R < att(v_c)$ then
5     $queue$.push($L, L', att(v_c), R', x+1, N_x + v_c$);
6     $R' \leftarrow att(v_c)$;
7 else
8     $queue$.push($L, L', R, R', x+1, N_x + v_c$);
9     break;
```

Since $\mathrm{att}(v_5) = 43 < \mathrm{L}' = 54$, another tuple $(28, 43, 54, +\infty_{\mathrm{A}}, 1, v_5)$ is added to the queue and L becomes 43. Then, it finds *ann* in $(43, +\infty_{\mathrm{A}})$, which is $\{v_7, v_3\}$, has $v_c = v_7$, adds $(43, 54, 57, +\infty_{\mathrm{A}}, 1, v_7)$ to the queue as $\mathrm{R} = 54 < \mathrm{att}(v_7) = 57$, 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 $\mathrm{R} = 54 < \mathrm{att}(v_6) = 56$, and sets $\mathrm{R}' = 56$. 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, whic results in no edges as $\mathrm{N}_0 = \emptyset$. 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 $(\mathrm{L} = -\infty_{\mathrm{A}}, \mathrm{L}' = 28, \mathrm{R} = 54, \mathrm{R}' = +\infty_{\mathrm{A}}, x = 1, \mathrm{N}_1 = v_1)$ is popped from the queue. It finds *ann* in $(-\infty_{\mathrm{A}}, +\infty_{\mathrm{A}})$, which is $\{v_1, v_5\}$. Since $v_1 \in \mathrm{N}_1$, it has $v_c = v_5$. As $\mathrm{L}' = 28 \le \mathrm{att}(v_5) = 43 \le \mathrm{R} = 54$, $(-\infty_{\mathrm{A}}, 28, 54, +\infty_{\mathrm{A}}, 2, v_1 v_5)$ 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 $(-\infty_{\mathrm{A}}, \mathrm{att}(v_t)] \times [\mathrm{att}(v_t), +\infty_{\mathrm{A}})$ when inserting v_t under* ASA. *Specifically, each tuple* $(\mathrm{L}, \mathrm{L}', \mathrm{R}, \mathrm{R}', x, \mathrm{N}_x)$ *in queue of the algorithm corresponds to a node at level x on T whose xNNlist sequence is N_x.*

Proof Sketch. The proof adds additional discussions on the case of $\mathrm{R} < \mathrm{att}(v_c) < \mathrm{R}'$ compared to the proof sketch of Lemma 4. It means that all the attribute ranges $[l, r]$ with $(l, r) \in (\mathrm{L}, \mathrm{L}'] \times [\mathrm{R}, \mathrm{R}')$ share the same xNNlist sequence. However, when $r \ge \mathrm{att}(v_c)$ the $(x+1)$NNlist should be $\mathrm{N}_x + v_c$ (Line 4), while when $r < \mathrm{att}(v_c)$, 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 $(\mathrm{L}, \mathrm{L}'] \times [\mathrm{att}(v_c), \mathrm{R}')$ for $(x+1)$NNlist sequence $\mathrm{N}_x + v_c$ (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. □

Theorem 7 (Complexity of Algorithm 5). *When the t-th vector is inserted, $t \ge \mathrm{K}$, the worst-case index size of Algorithm 5 is $O(\mathrm{KM}t)$. The average-case index size of Algorithm 5 is $O(\mathrm{K}^2\mathrm{M} \ln t)$. The worst number of calls of ANN search is $O(\mathrm{K}^2 t)$ and the expected number of ANN search calls is $O(\mathrm{K}^3 \ln t)$.*

Proof. Algorithm 5 only writes tuples to G either on leaves at level K (Lines 4-8, Algorithm 4) or on tree nodes on any level $x < \mathrm{K}$ such that Line 11 tests true – we call these nodes sap nodes. Each time we add 2M edges to the graph. From Theorem 4, the total number of leaves at level K is $O(\mathrm{K}t)$ in the worst case for v_t, and $O(\mathrm{K}^2 \ln t)$ in expectation. Next, we show that the total number of sap nodes on each level $x < \mathrm{K}$ is at most K and thus the total number of sap nodes is $O(\mathrm{K}^2)$.

When Line 11, Algorithm 4 tests true, take snapshot of $\mathrm{L}, \mathrm{L}', \mathrm{R}, \mathrm{R}', x$. Lemma 5 proves that range $(\mathrm{L}, \mathrm{R}')$ has exactly x vectors in $\mathcal{D}_{t-1}$. Note that $\mathrm{att}(v_t) \in (\mathrm{L}, \mathrm{R}')$, L, R' are attribute values of $\mathcal{D}_{t-1}$ if they are not $-\infty_{\mathrm{A}}$ or $+\infty_{\mathrm{A}}$, so the total number of possible attribute values that L could take is no more than $x + 1 \le \mathrm{K}$ and each value of L uniquely determines R' as the interval has x vectors.

Thus, the space complexity is $O(\mathrm{KM}t)$ in the worst case and $O(\mathrm{K}^2\mathrm{M} \ln t)$ in expection. 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^2 + the total number of nodes. Therefore, the worst case number of calls of ANN search is $O(\mathrm{K}^2 t)$ and the expected number of ANN calls is $O(\mathrm{K}^3 \ln t)$. □

Theorem 8. *Denote by $\mathrm{a}_1 \le \mathrm{a}_2 \le \cdots \le \mathrm{a}_{t-1}$ 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 $[\mathrm{a}_i, \mathrm{a}_j]$, i.e., on $\mathcal{D}_t[\mathrm{a}_i, \mathrm{a}_j]$ with $i \le j$.*

Proof. Define a_0 be $-\infty_{\mathrm{A}}$ and a_t be $+\infty_{\mathrm{A}}$. Consider an attribute range of $[l, r]$. We only consider $l \le \mathrm{att}(v_t) \le r$ 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 $\mathrm{a}_{i-1} < l \le \mathrm{a}_i \le \mathrm{a}_j \le r < \mathrm{a}_{j+1}$. Let S be the K nearest neighbors of v_t in $\mathcal{D}_{t-1}[\mathrm{a}_i, \mathrm{a}_j] = \mathcal{D}_{t-1}[l, r]$. 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 $[\mathrm{a}_i, \mathrm{a}_j]$. 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, $|S| < \mathrm{K}$ and $|S| \ge \mathrm{K}$.

When $|S| < \mathrm{K}$, let $x \doteq |S|$. According to Theorem 3, there is exactly one node v^T on T at level x such that $(l, r) \in \mathrm{rect}(v^T)$. Furthermore, since (l, r) belongs to $\mathrm{sap}_{t-1}(x+1)$, 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 $(\mathrm{L}, \mathrm{L}', \mathrm{R}, \mathrm{R}')$ and thus $\mathcal{D}_{t-1}$ on both $(\mathrm{L}, \mathrm{R}')$ and $[\mathrm{L}', \mathrm{R}]$ are S (as all attribute pairs in $\mathrm{rect}(v^T)$ share the same xNNlist sequence which is S), thus $\mathrm{L} = \mathrm{a}_{i-1}$, $\mathrm{L}' = \mathrm{a}_i$, $\mathrm{R} = \mathrm{a}_j$, and $\mathrm{R}' = \mathrm{a}_{j+1}$ (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 $U = (\mathrm{a}_{i-1}, \mathrm{a}_i] \times [\mathrm{a}_j, \mathrm{a}_{j+1})$. 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 $|S| \ge \mathrm{K}$, there is exactly one node $v^T(\mathrm{L}, \mathrm{L}', \mathrm{R}, \mathrm{R}', \mathrm{K}, \mathrm{N}_{\mathrm{K}})$ on T among level K nodes such that $(l, r) \in \mathrm{rect}(v^T)$ (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 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. □

3.3 Early Prunning

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

Algorithm 6: PrunedInsertion(G, v_t, M, K)

```
Input: G: the dynamic segment graph constructed for
    $\mathcal{D}_{t-1} = \{v_1, v_2, \cdots, v_{t-1}\}$; $v_t$: a point arriving at time $t$; M
    and K: the parameters in HNSW construction
Output: G: the dynamic segment graph for $\{v_1, v_2, \cdots, v_t\}$.
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′, $x$, $x$NNlist) ← queue.pop();
4      if $x \neq 0$ then
5          $v_e$ ← the last point in $x$NNlist;
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 ← 2DSegmentANNSearch(G, $v_t$, (L, R′), $v_1$, K);
12             $v_c$ ← $\arg\min_{v \in ann \setminus x\text{NNlist}} \delta(v, v_t)$;
13             if $v_c$ is dominated by any point in $x$NNlist then
14                 $(x+1)$NNlist ← $x$NNlist + $v_c$; $x$++;
15             else break;
16         if $v_c$ is in $x$NNlist then break;
17         Lines 1-9 of Algorithm 5;
18 return G;
```

Algorithm 6 revises the random insertion process in two aspects. Consider v^T = (L, L′, R, R′, x, xNNlist) 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 xNNlist 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 xNNlist 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.*
(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 $\setminus \{v_c\}$.

Proof. Because Line 16 indicates that if v_c is in xNNlist, i.e., v_c is dominated by any other point in xNNlist, then break. In other words, if a tuple is generated in Line 17, then x_c cannot be pruned by any other point in attribute range (L, R′) and x_c will be the end of the $(x+1)$NNlist lists enqueued. □

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′] × [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* $[l, r]$.

Proof. As the base case, it is trivial to verify that the lemma holds on the root tuple with $x = 0$. We next show that if the lemma holds on a tuple v^T = (L, L′, R, R′, x, xNNlist) popped in Line 3, then it holds on all the tuples generated in Line 17.

Lines 10-15 carry out the following steps:

(1) Remove all points in xNNlist from the underlying point set.
(2) Get the nearest neighbor v_c of v_t in attribute range (L, R′).
(3) If v_c can be pruned by xNNlist, remove v_c from the underlying dataset. Extend sequence xNNlist with v_c, its length x is increased by 1. Go to Step (1).
(4) Terminate otherwise.

This process ensures that at Line 16, by removing xNNlist from the underlying dataset, v_c is the nearest neighbor of v_t for range (L, R′) and it cannot be pruned by xNNlist.

If att$(v_c) \leq$ L′, then for $l \in$ (L, att(v_c)], $r \in$ [R, R′), there is no other point with attribute value in $[l, r]$ and distance to v_t smaller than $\delta(v_c, v_t)$, except for the points in xNNlist. Thus $P + v_c$ is the prefix of the PNNS of $[l, r]$. Besides, P is the prefix of the PNNS of $[l, r]$ with $l \in$ (att(v_c), L′], $r \in$ [R, R′), while the PNNS will not include v_c.

Symmetrically, if att$(v_c) \geq$ R, then for $l \in$ (L, L′], $r \in$ [att(v_c), R′) then $P + v_c$ is the prefix of the PNNS of $[l, r]$. Besides, P is the pruned sequence of the nearest neighbour list on all $[l, r]$ with $l \in$ (L, L′], $r \in$ [R, att(v_c)) and v_c will not appear in this PNNS.

If att$(v_c) \in$ [L′, R], then for $l \in$ (L, L′], $r \in$ [R, R′), $P \cup \{v_c\}$ will be the prefix of the PNNS of $[l, r]$.

By induction, we can prove that the lemma holds for all the tuples in the queue. □

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$(\mathrm{G}, v_t, (-\infty_\mathrm{A}, +\infty_\mathrm{A}), v_1, \mathrm{Z})$ where Z is a parameter to find a set *ann* of Z approximate nearest neighbors of v_t. Then, instead of invoking 2DSegmentANNSearch $(\mathrm{G}, v_t, (\mathrm{L}, \mathrm{R}'), v_1, \mathrm{K})$ 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 $\mathrm{att}(v_c) \in (\mathrm{L}, \mathrm{R}')$. 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 $v^T = (\mathrm{L}, \mathrm{L}', \mathrm{R}, \mathrm{R}', x, x\mathrm{PNN})$ 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 $\mathrm{att}(v_c) \in (\mathrm{L}, \mathrm{R}')$. 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 $\mathrm{G}[u][v]$ 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 $\mathrm{G}[u][v]$ 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 $(\mathrm{L}, \mathrm{L}', v_e, \mathrm{R}, \mathrm{R}')$ to $\mathrm{G}[v_t]$ and $(\mathrm{L}, \mathrm{L}', v_t, \mathrm{R}, \mathrm{R}')$ to $\mathrm{G}[v_e]$, it merges the rectangle $(\mathrm{L}, \mathrm{L}'] \times [\mathrm{R}, \mathrm{R}')$ with the MBRs $\mathrm{G}[v_t][v_e]$ and $\mathrm{G}[v_e][v_t]$. 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 $(\mathrm{L}, \mathrm{L}', \mathrm{R}, \mathrm{R}', x, x\mathrm{PNN})$ 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).

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 $\{v_1, v_2, \cdots, v_t\}$.
1  ann ← 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) ← queue.pop();
5      if $x \neq 0$ then
6          $v_e$ ← the last point in xNNlist;
7          merge (L, L'] × [R, R') with G[$v_t$][$v_e$];
8          merge (L, L'] × [R, R') with G[$v_e$][$v_t$];
9      If $x$ = M then continue;
10     while L ≤ L' and L' ≤ R' do
11         $v_c$ = null;
12         foreach $v \in$ ann in ascending order of $\delta(v, v_t)$ do
13             if $v \notin$ xPNN and att($v$) ∈ (L, R') and $v$ is not
                 dominated by any point in xPNN then
14                 $v_c$ ← $v$;
15         if $v_c$ is not null then
16             Lines 1-9 of Algorithm 5, replace xNNlist with xPNN;
17         else break;
18 return G;
```

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.

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[3]. The attribute value of each vector is the release time of the corresponding video. (2) WIT[4]: 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[5]: 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

[3]https://research.google.com/youtube8m/download.html
[4]https://github.com/google-research-datasets/wit
[5]https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search

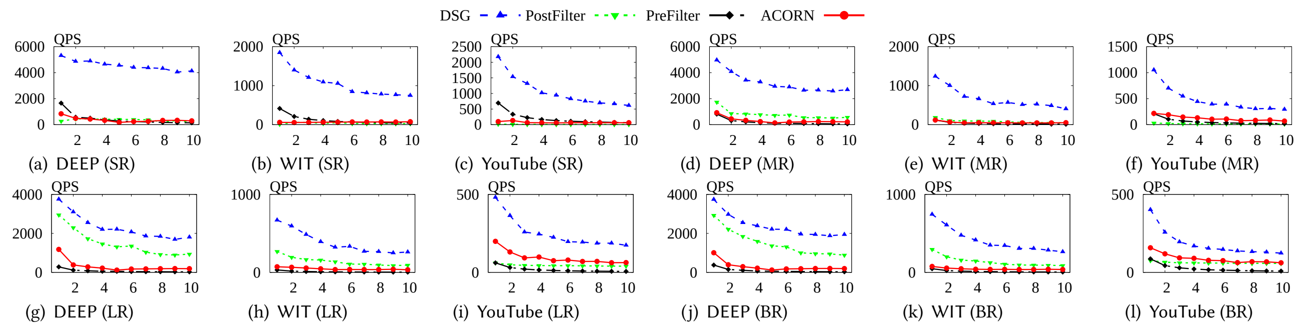


Figure 3: Comparison with Existing Methods: Unordered Insertion (evaluated after every 100,000 data vectors inserted).

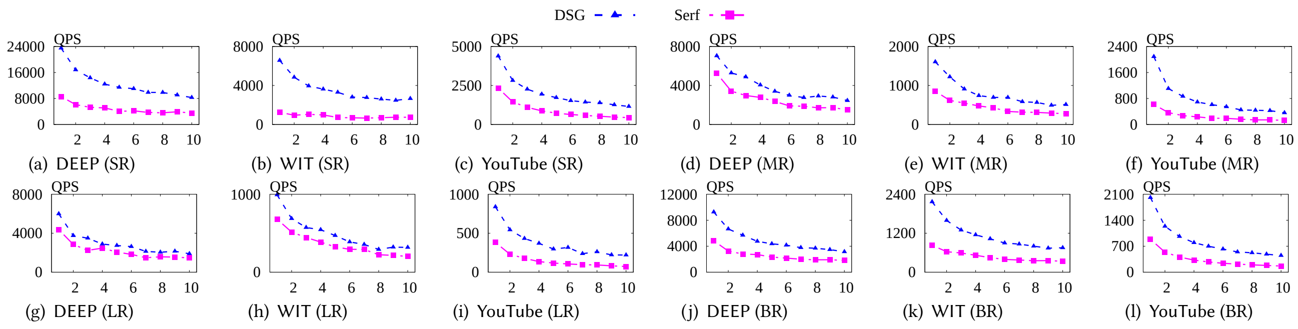


Figure 4: Comparison with Existing Methods: Ordered Insertion (evaluated after every 100,000 data vectors inserted).

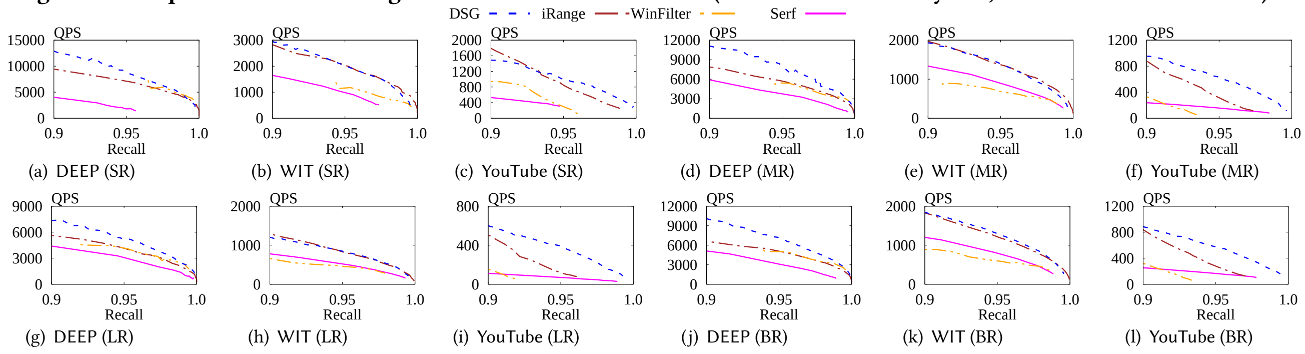


Figure 5: Comparison with Existing Methods: Static Workload.

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 [39] (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 [35] 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 query range size (SR), medium query range size (MR), and large query range size (LR), which include 1%, 4%, and 16% of the inserted vectors, respectively. In addition, we evaluated the blended query range size (BR) that includes x% of the inserted vectors where x was draw from $\{1\%, 2\%, 4\%, 8\%, 16\%, 32\%\}$ with equal probability. Note that we do not test ranges that are too small – Prefiltering could be an efficient solution. We can build a simple cost model based on the query range size. 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 16 and 500, respectively, on DEEP, and 32 and 1000 on both WIT and YouTube, unless otherwise specified. For SeRF, we used maxleap with M = 16 for DEEP, M = 32 for YouTube and M = 64 for wiki. Besides, 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_β 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 paper or the optimal result 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 supports unordered insertion: Prefiltering, Postfiltering, and Acorn. We evaluated the query performance 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 on DEEP and WIT, and 0.95 on YouTube for all query scenarios and reported the QPS (query per second). 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 were inserted. Nevertheless, DSG consistently and significantly outperformed the baselines. For example, on WIT and BR (Figure 3(k)), DSG achieved 2.5× the QPS of the best-performing baseline Postfiltering at 100K vectors and 3× at 1M vectors. The advantage was more obvious for SR, where DSG achieved 3-15× the QPS of the best baselines throughout the process. This is because Postfiltering and Acorn only work well when the query range size is very large (i.e., when the predicate selectivity is very high). 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× and 87 × 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 [39], 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× that of SeRF. Besides, it is worth to mention that DSG 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× than that of iRange when their recall was around 0.96. It is worth to mention that DSG is more capable than these baselines as DSG supports dynamic RFANNS whereas iRange and WinFilter do not. This is attribute to the lossless guarantee of DSG and the effectiveness of the proposed optimizations.

Table 1: Comparison of Index Cost.

Dataset	Metric	WinF	iRange	DSG	Acorn	SeRF	PostF
DEEP	time (s)	27968	1052	1489	2504	887	212.6
	size (GB)	7.28	3.23	3.08	1.79	0.64	0.53
WIT	time (s)	48471	16253	19449	40988	8532	1624
	size (GB)	30.15	13.48	10.65	24.70	8.18	8.20
YouTube	time (s)	71113	6775	31963	18948	6452	1046
	size (GB)	18.15	9.48	7.85	12.99	4.15	4.22

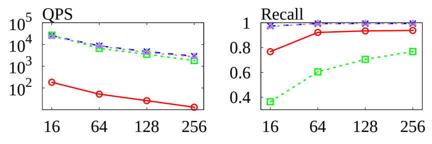
Exp-4: Index Cost. Table 1 shows the index time and index size (memory footprint) of all methods for 1M vectors. Note that the index cost of Prefiltering was negligible and was omitted in the table. A single thread was employed. As we can see, the index size of WinFilter and iRange were significantly 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.

5.2 Sensitivity and Scalability Tests

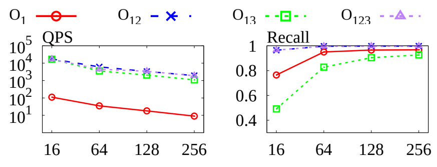
Exp-5: Optimization Sensitivity Test. We evaluate the effectiveness of the three optimizations in Section 4 by comparing the following four combinations O_1, O_{12}, O_{13}, and O_{123} on DEEP with $10,000$ data vectors. Here O_{xy} means using the combination of optimizations O_x and O_y as described 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 DEEP. Figure 7(a) shows the QPS and recall. With the increase of M, the recall steadily increased. This is because the dynamic segment graph is more connected with a large M. The index size (edges only) was 15MB, 26MB, 36MB, 37MB respectively when M was 4, 8, 16, 32, while the index time was 2.7s, 6.9s, 25s, 34s. The index size did not increase proportionally with M as M is only the max degree (not the actual degree).

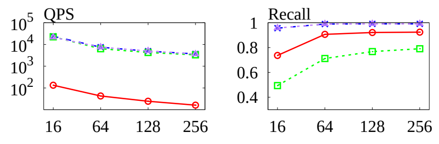
Exp-7: Sensitivity Test on Z. We tested the sensitivity of the parameter Z on DEEP by varying it from 400 to 1000. Figure 7(b) shows the query performance. For all query scenarios, the QPS only slightly decreased with the increase of Z, while the recall remained unchanged. The index time was 25s, 37s, 47s, 59s respectively when Z was 400, 600, 800, 1000, while the index size (edges only) was 36MB, 41MB, 45M, 47M. This is because, a large Z adds more edges to the dynamic segment graph, making the index cost higher, QPS lower, and potentially recall higher. However, the recall was already very high (above 0.99) and was hard to be further improved.



(a) DEEP (MR, M = 8, Z = 500, varying efsearch)

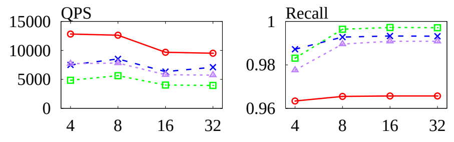


(b) DEEP (LR, M = 8, Z = 500, varying efsearch)

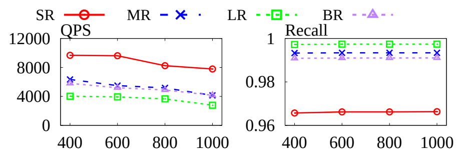


(c) DEEP (BR, M = 8, Z = 500, varying efsearch)

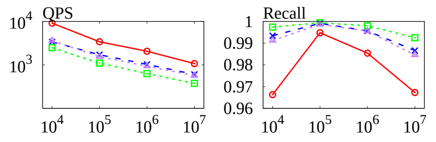
Figure 6: Evaluating Optimizations.



(a) DEEP (Z = 400, efsearch = 64, varying M)



(b) DEEP (M = 16, efsearch = 64, varying Z)



(c) DEEP (M = 16, Z = 500, efsearch = 128, varying the # of vectors in the stream)

Figure 7: Sensitivity and Scalability Tests.

Exp-8: Scalability Test. We used 10K, 100K, 1M, 10M vectors in DEEP to test the scalability of our method DSG. Figure 7(c) shows 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 sizes (edges only) were 0.04GB, 0.44GB, 4.58GB, 46.8GB respectively for 10K, 100K, 1M, 10M vectors, while the average insertion latencies were 3.6ms, 4.1ms, 4.8ms, 6.7ms. Clearly, the index size scaled linearly while the index time remained almost constant.

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, 31], 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 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 [39]. 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 [35]. Filtered-DiskANN [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. ACORN is designed for predicate-agnostic ANNS [26], where the predicate is arbitrary (e.g., regex match, keyword match, etc). It proposes to explore multi-hop neighbors that satisfying the query predicate during greedy search. A few studies, including AnalyticDB-V [32] 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 [29]. NHQ [30] and HQANN [33] propose to fuse the attribute values into the vectors for attribute-filtering ANNS. ARKGraph studies how to compress the approximate k-nearest neighbor graphs of all ranges [38]. Note that it does not discuss the impact of HNSW pruning. Zhao et al. [36] design a few optimizations for attribute-filtering ANNS, including entry point selection, biased priority queue selection, and multi-direction search.

Dynamic Approxiamte 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 [34]. 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) finds approximate nearest neighbors for a query vector among data vectors whose attribute values fall within a query range. Existing methods for RFANNS are designed for static datasets and cannot efficiently handle dynamic RFANNS, where data vectors arrive continuously in a stream. To address this, this paper designs the dynamic segment graph, a structure that losslessly compresses multiple hierarchical navigable small-world (HNSW) graphs, one for each possible query range. This structure allows for efficient insertion of new data vectors as they arrive. We rigorously analyze the time and space complexity of the dynamic segment graph and propose several optimizations to reduce the index cost in practice.

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. FANNG: 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 Krishnawamy, 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 [cs.IR] 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] 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.

[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 [cs.IR] https://arxiv.org/abs/2105.09613

[29] 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

[30] 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]

[31] 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

[32] 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

[33] 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

[34] 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

[35] Yuexuan 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

[36] 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

[37] 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.

[38] 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

[39] 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