# WoW: A Window-to-Window Incremental Index for Range-Filtering Approximate Nearest Neighbor Search

Ziqi Wang State Key Laboratory for Novel Software Technology Nanjing University, China ziqiw.nju@gmail.com Jingzhe Zhang State Key Laboratory for Novel Software Technology Nanjing University, China jzzhang.nju@gmail.com Wei Hu\*
State Key Laboratory for Novel
Software Technology
National Institute of Healthcare
Data Science
Nanjing University, China
whu@nju.edu.cn

### **Abstract**

Given a hybrid dataset where every data object consists of a vector and an attribute value, for each query with a target vector and a range filter, range-filtering approximate nearest neighbor search (RFANNS) aims to retrieve the most similar vectors from the dataset and the corresponding attribute values fall in the query range. It is a fundamental function in vector database management systems and intelligent systems with embedding abilities. Dedicated indices for RFANNS accelerate query speed with an acceptable accuracy loss on nearest neighbors. However, they are still facing the challenges to be constructed incrementally and generalized to achieve superior query performance for arbitrary range filters. In this paper, we introduce a window graph-based RFANNS index. For incremental construction, we propose an insertion algorithm to add new vector-attribute pairs into hierarchical window graphs with varying window size. To handle arbitrary range filters, we optimize relevant window search for attribute filter checks and vector distance computations by range selectivity. Extensive experiments on real-world datasets show that for index construction, the indexing time is on par with the most building-efficient index, and 4.9× faster than the most query-efficient index with 0.4-0.5× smaller size; For RFANNS query, it is 4× faster than the most efficient incremental index, and matches the performance of the best statically-built index.

### **CCS** Concepts

 $\bullet$  Information systems  $\to$  Database query processing; Nearest-neighbor search.

# **Keywords**

approximate nearest neighbor search, graph-based index, range search, vector database

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '26, May 31-June 5, 2026, Bengaluru, India

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-XXXX-X/18/06

https://doi.org/10.1145/3769843

#### **ACM Reference Format:**

Ziqi Wang, Jingzhe Zhang, and Wei Hu. 2026. WoW: A Window-to-Window Incremental Index for Range-Filtering Approximate Nearest Neighbor Search. In *Proceedings of the ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD '26)*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3769843

#### 1 Introduction

Nearest neighbor search (NNS) [55] is widely used as a fundamental process in various applications. On e-commerce platforms, people use an image to search for similar commodities [36, 50, 63, 66, 69]. In the retrieval-augmented generation (RAG) pipeline [24, 68], people retrieve paragraphs relevant to a question and provide them as context for large language models. These functions can be realized by encoding the original data into vectors with modern embedding models [8, 46, 53, 54] and persisting with a vector database management system (VDBMS) [28, 50]. With a guery also embedded into the same vector space, VDBMS uses k-NNS to find similar results in the database. However, finding the exact k nearest neighbors is time-consuming because the time complexity is O(nd), where n is the number of vectors and d is the dimension of vectors [12, 22]. On real-world datasets, the query latency can be even unacceptable for online scenarios, as n varies from millions to billions while d is in the order of hundreds or thousands.

Recently, a family of algorithms are proposed without having to retrieve all exact nearest neighbors in trade for sublinear query time complexity [13, 14, 30, 37, 50]. It is referred to as *approximate nearest neighbor search* (ANNS) and there exist mainly three categories: hashing-based [35, 60–62, 70], partition-based [4, 31, 34], and graph-based algorithms [11, 20, 21, 42, 43, 52, 65]. With the rapid change in the demand for various query scenarios, it is inadequate to retrieve data only by vector similarity [50, 63]. A popular scenario is called *filtering search* [26, 51, 64, 71], which aims to find the most similar vectors whose attribute payloads can pass a certain predicate filter. In this work, we focus on the case where the dataset has only one attribute and the predicate is a range (a.k.a. window) filter, which is named *range-filtering ANNS* (RFANNS) [19, 38, 50, 72, 81, 82].

For instance, instead of searching for similar commodities solely by an image, we can provide an additional price range to retrieve affordable ones. Another example resides in a medical question answering system, a typical RAG application [2, 56]. When a query comes with "What symptoms are common for people with hypertension aged from 50 to 60?", the retriever may generate an RFANNS

 $<sup>^{\</sup>star}$ Corresponding author

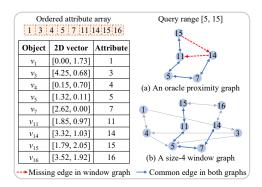


Figure 1: Example of (a) an oracle proximity graph and (b) a size-4 window graph. Dataset is on the left.

query to find medical records about "hypertension" tagged with ages of patients between 50 and 60.

Generally, there are three methods to address the RFANNS problem [19, 72, 82]. Pre-filtering first selects in-range vectors from the dataset. The term "in-range" indicates the attribute value alongside the vector can pass the range filter while the term "out-of-range" is opposite, hereafter. As there is no index built for in-range vectors, it can be inefficient to search by linear scan. Post-filtering builds an ANNS index over the entire dataset, and retrieves some intermediate vectors before eliminating the out-of-range vectors. It may cause the number of query results less than k, and thus another trial to retrieve more intermediate vectors is needed. Dedicated RFANNS indices only visit in-range vectors during searching, called in-filtering. For partition-based indices [79], filtering happens in cluster discovery and posting list scan. Graph-based indices [19, 32, 38, 72, 81, 82] aim to simulate the proximity graph exclusively built over in-range vectors, referred to as oracle proximity graph [51, 72]. Figure 1a shows an oracle proximity graph with outdegree = 2 built over vectors in the range [5, 15]. It has been acknowledged that graph-based RFANNS indices are superior in query efficiency over non-graph-based solutions [50, 79, 82].

In analyzing applications where dedicated RFANNS indices are used, we find two key challenges to index building and searching:

Challenge 1. Can RFANNS indices be built incrementally? Dedicated graph-based RFANNS indices are slower to build compared with hashing/partition-based ones [37, 74, 79]. Also, as materializing all proximity graphs for arbitrary range filters is unrealistic, they attempt to build a denser graph that covers edges of all oracle graphs, at the cost of longer indexing time and larger size than a vanilla graph built on the vector set. If the index is static [19, 72] or suffers query performance degradation due to limited increment support[32, 79, 82], it must be rebuilt upon new data arrivals or maintained via a small auxiliary index before periodic merging. Considering the high reconstruction cost, an incremental index is preferable to skip frequent rebuilding or merging [38, 57]. However, most RFANNS indices are either static or only provide limited incremental support [19, 32, 72, 79, 82], presenting an opportunity to devise an index with unrestricted incremental capabilities.

**Challenge 2.** Can RFANNS indices handle varying query correlations and selectivity? Another challenge is the no or negative

correlation [51] between the query vector neighborhood and the filtering subset [9, 51, 72]. For example, the most similar commodities to an image may not satisfy the requested price range. The performance on such workloads may largely affect the overall efficiency. Queries of high selectivity are also challenging as most of nearest neighbors are out of range. Many RFANNS indices [19, 38, 79, 82] fail to obtain the optimal balance between query speed and accuracy for all levels of query correlations and selectivity. As far as we know, only iRangeGraph [72] and DIGRA [32] can achieve close performance to the oracle graph. However, iRangeGraph is a static index and lacks support for insertion and DIGRA encounters severe accuracy loss after frequent insertions. There is still room for improvement in index building and searching by more advanced structures and optimized query algorithms.

In this paper, we design a *window graph*-based RFANNS index. Intuitively, if two vertices are connected in the oracle proximity graph of a query range, they should be similar in both attribute values and vectors. Through an ordered array, the attribute similarity for a given value can be modeled using an attribute window that is centered at the value and extended symmetrically in two directions. Considering attribute and vector similarity, a vertex in a window graph only connects to in-window nearest vertices. When the range cardinality matches the window size, the window graph achieves optimal in both similarity metrics. Figure 1b shows a size-4 window graph on the ordered attribute array. Each vertex connects to its two nearest neighbors, in accord with the oracle graph's outdegree constraint. For example,  $v_7$  connects to  $v_5$  and  $v_{14}$  as they are more similar in its window [4, 5, 11, 14] than  $v_4$  and  $v_{11}$ .

However, only one window graph cannot generalize to varying query cardinality, leading to candidate missing due to out-of-range vertices. As shown in Figure 1, the window graph misses 2 of 10 directed edges in the oracle graph. To recover the missing connections, smaller window graphs can be built as complements. For example, in a size-2 window graph, the missing edges can be retrieved from the neighborhoods of  $v_{14}$  and  $v_{15}$ , whose windows are [11, 15] and [14, 16], respectively. Large and small window graphs together form a hierarchy for RFANNS indexing and searching.

Based on the above intuition, we propose **WoW**. Its basic idea is to map multiple **W**indow graphs to **W**indow filters. To address Challenge 1, we design an efficient algorithm that supports inserting vector-attribute pairs into hierarchical window graphs with varying window size. To resolve Challenge 2, we optimize traversal on window graphs of different size to tightly cover arbitrary range filters with varying query correlations. We conduct extensive experiments on benchmark datasets to evaluate WoW and its key components. The experimental results show the superiority of WoW in index building and query efficiency against state-of-the-art dedicated RFANNS indices. The code and experimental data can be visited at https://github.com/nju-websoft/WoW.

The main contributions of this paper are outlined as follows:

- For index building, WoW is fully incremental from an empty index without data presorting or partial indexing. The worst-case time complexity of insertion scales to  $O(\log^2 n)$ , which can be highly parallelized to further boost building speed.
- For RFANNS query, we employ query selectivity to choose the most relevant window graphs. As far as we know, we are

**Table 1: Frequently-used notations** 

Notation	Description
$\mathcal{D} = \{\mathcal{V}, \mathcal{A}\}$	Hybrid dataset $\mathcal{D}$ consisting of vector dataset $\mathcal{V}$ and
	attribute set $\mathcal{A}$ , s.t. $ \mathcal{V}  =  \mathcal{A}  = n$
$v_a$	Vector $v$ with attribute value $a$
R = [x, y]	Query range R with left and right boundaries
$L = [l_{\min}, l_{\max}]$	Layer range L with lower and upper limits
$N_v^l$	Neighbor set of $v$ at layer $l$
$W_a^l$	Window of $a$ at layer $l$
o	Window boosting base
$\omega_c$	Beam search width in index construction
m	Graph maximum outdegree

the first to reduce filter tests of attribute values and distance computations of vectors at the same time. We prove that the query time complexity is  $O(\log n')$ , equivalent to that on the oracle relative neighbor graph (RNG) with n' in-range vectors. We also conduct a theoretical analysis to estimate the optimal parameter setting for best query performance.

• Extensive experiments show the efficiency of WoW: (1) The indexing time of WoW competes with the most building-efficient baseline [82], and is 4.9× faster with 0.4–0.5× smaller size than the most query-efficient index [32]. (2) WoW delivers 4× faster queries than the best incremental index [38] across all workloads. (3) WoW is 1.5× faster than the best statically-built index [32] on workloads of high selectivity, with equal or better performance on others.

### 2 Preliminaries

In this section, we formulate the studied problem, followed by a survey of related work. Table 1 lists the frequently used notations.

#### 2.1 Problem Formulation

Definition 1 (Approximate Nearest Neighbor Search). A vector dataset V is defined in vector space with a distance metric  $\delta$ , where each vector has dimension d. Given a query vector q, ANNS aims to find a subset  $S \subseteq V$  with k vectors to minimize  $\sum_{v \in S} \delta(v, q)$ .

Often times, the result contains non-closest vectors. The search accuracy can be defined by the fraction of the exact nearest neighbors, i.e.  $\frac{|\mathcal{S} \cap \mathcal{G}|}{k}, \text{ where } \mathcal{G} \text{ is the set of the exact nearest neighbors (i.e. gold standard)}.$ 

RFANNS aims to find vector-attribute pairs such that vectors are the closest to the query vector and attribute values are in range.

DEFINITION 2 (RANGE-FILTERING ANNS). A hybrid dataset  $\mathcal{D} = \{\mathcal{V}, \mathcal{A}\}$  consists of a vector set  $\mathcal{V}$  and an attribute set  $\mathcal{A}$  with equal size n. Given a range-filtering query (q,R), where q is the query vector and R = [x,y] is the range (window) filter over  $\mathcal{A}$ , with x and y as its left and right boundaries, RFANNS aims to find a subset  $\mathcal{S} \subseteq \mathcal{D}$  with k vector-attribute pairs to minimize  $\sum_{(v,a) \in \mathcal{S}} \delta(v,q), x \leq a \leq y$ . We call a pair (v,a) in-range only when  $x \leq a \leq y$  (i.e.  $a \in R$ ).

The selectivity of a query range indicates how many attribute values are in range. High selectivity means that only a few in the attribute set can pass the range filter.

**Table 2: Comparison of RFANNS indices** 

Algorithm	ANNS	Indexing	OOR	Key structure
Pre-filtering	-	Increment	×	-
Post-filtering	Any	Increment	✓	-
SeRF	HNSW	Ordered inc.	×	Segment graph
WST	Vamana	Static	$\checkmark$	Segment tree
iRangeGraph	NSW	Static	×	Segment tree
RanePQ	PQ	Post-increment	×	WBT
DIGRA	NSW	Post-increment	×	Multi-way tree
HSIG	HNSW	Increment	$\checkmark$	Unified graph
WoW (ours)	NSW	Increment	×	Window graph

DEFINITION 3 (QUERY SELECTIVITY). For a range filter R = [x, y], let n' denote the number of attributes in  $\mathcal A$  that are in range R, the fraction of in-range attributes over the dataset cardinality is  $f = \frac{n'}{n}$ , and the selectivity of R is defined by  $s = \frac{1}{f}$ .

Note that if n' is less than k in the entire dataset, recall should be calculated by  $Recall = \frac{|S \cap \mathcal{G}|}{n'}$ .

### 2.2 Related Work

*ANNS indices.* Existing works [37, 50] can be categorized into hashing-based, partition-based, and graph-based. Hashing-based indices adopt static [60, 70] or learnable [61, 62] hash functions to map similar vectors to identical or nearby hash buckets. Partition-based indices [4, 23, 25, 31, 47] group similar vectors in the same cluster. Graph-based ones [11, 20, 21, 27, 42, 43, 52] represent vector similarity by edges, aiming to approximate the RNG: vertices  $r, s \in V$  are connected if and only if  $\forall t \in V, \delta(r, s) < \delta(r, t)$  or  $\delta(r, s) < \delta(s, t)$  [7, 65]. Hierarchical navigable small world (HNSW) [43] is widely used for fast indexing and superior query performance.

Graph-based indices are further divided into refinement-based and increment-based by construction strategies [74]. Increment-based ones (e.g., NSW [42] and HNSW [43]) offer greater flexibility than static, refinement-based ones (e.g., NSG [21] and NSSG [20]). We focus on the challenge of incremental support, which remains insufficiently addressed yet. Deletion for graph-based indices is addressed by a separate line of works [41, 57, 77]. They exploit the RNG property to reconstruct the neighborhood around a deleted vertex, making them applicable to any underlying graph structure.

Filtering ANNS and RFANNS. There are mainly three types of filters: label filter, range filter, and generic predicate filter. Some indices [1, 9, 26, 64, 71] attempt to resolve label-filtering ANNS. For generic query predicate, VDBMS and ANNS libraries [10, 17, 33, 36, 40, 44, 63, 69, 75, 80] estimate query selectivity to pick pre-filtering or post-filtering. As the inherent defects of pre/post-filtering, dedicated indices [18, 45, 51] are designed to only visit those vectors whose corresponding attributes can satisfy the predicate.

Graph-based RFANNS indices aim to search on proximity graphs built solely over in-range vectors. SeRF [82] attempts to compress oracle HNSWs for all ranges. But the compression is not lossless, and some less proximate edges are retained, which may harm query speed. WST [19] and iRangeGraph [72] are based on segment tree [16]. A single graph-based index is built for each tree node. WST runs several ANNS on tree nodes whose range intersects with the

query filter and merges separate results. It inherits the weakness of post-filtering that the distance of out-of-range (OOR) vectors may be calculated. iRangeGraph [72] acquires candidates by tree traversal at each hop. However, tree traversal consumes  $O(\log n)$ time overhead that may diminish query speed, especially for filters of high selectivity and low-dimensional datasets [72]. All RFANNS indices above are static indices [19, 72] or only support attributeordered insertions [82]. They need to presort the vectors based on attribute values before construction. RangePQ [79] and DIGRA [32] support insertion after static construction on a subset, namely postincremental indices. RangePQ is the only partition-based RFANNS index. It uses product quantization (PO) to partition the subset and a weighted balanced tree (WBT) [49] to group coarse cluster centers within the same range. However, clusters have to be retrained to handle distribution shift due to insertions. DIGRA presorts the subset like static indices and recursively builds a multi-way tree structure, before building separate HNSW inside each tree node. For insertion, it makes B-tree-like splitting on tree nodes but may break graph interconnections. HSIG [38] supports unordered insertion. It designs a learnable estimator to navigate queries with different selectivity to pre-filtering, post-filtering, and a dedicated index.

Table 2 compares the features of representative RFANNS indices: (1) *ANNS* indicates the ANNS algorithm that an index is based on. (2) *Indexing* compares how to construct the indices: static, post-incremental, ordered incremental, or unordered incremental. (3) *OOR* marks whether the distance of out-of-range vectors needs be calculated. (4) *Key structure* lists the structure employed to arrange vectors and attribute values. This table suggests that WoW is the only index to support fully incremental construction (cf. Challenge 1) without OOR-vector visit (cf. Challenge 2).

# 3 Window-to-Window Incremental Index

In this section, we first give the definition of *hierarchical window graphs* and the structure of WoW. Then, we describe how to incrementally build the index and use it to answer RFANNS queries. For illustration simplicity, we assume attribute values in  $\mathcal A$  are all *unique*. However, WoW is *not* limited by this assumption, and we show how to handle duplicate values at the end of this section.

#### 3.1 WBT and Hierarchical Window Graphs

Definition 4 (Window Graph). A window graph is a directed graph  $G = \{\mathcal{P}, \mathcal{E}, w\}$ , where  $\mathcal{P}$  is the vertex set,  $\mathcal{E}$  is the edge set, and w is the half window size. Each vertex  $v_i$  denotes a vector-attribute pair, where v is the vector and i is the attribute value.  $\mathcal{E}$  satisfies:

- (1) RNG property:  $\forall (v_i, v_j') \in \mathcal{E}, v_k'' \in \mathcal{P} \setminus \{v_i, v_j'\}, \text{ then } \delta(v_i, v_j') < \delta(v_i, v_k'') \lor \delta(v_i, v_j') < \delta(v_j', v_k'');$
- (2) Window property:  $\forall (v_i, v'_j) \in \mathcal{E}$ , then |rank(i) rank(j)| < w, where rank(i) counts the number of unique values less than i.

Definition 5 (Hierarchical Window Graphs). Hierarchical window graphs are a set of window graphs  $\mathcal{H} = \bigcup_{l \in [0,top]} G_l$ . Define a window boosting base  $o \geq 2$ , then  $G_l.w = o^l$  and  $top = \lceil \log_o \frac{|\mathcal{A}|_u}{2} \rceil$ , where  $|\mathcal{A}|_u$  is the total number of unique values.

Figure 2 depicts an example of WoW over a hybrid 2D dataset, with maximum outdegree m=4 and window boosting base o=4. Numbers on tree nodes and graph vertices represent the attribute

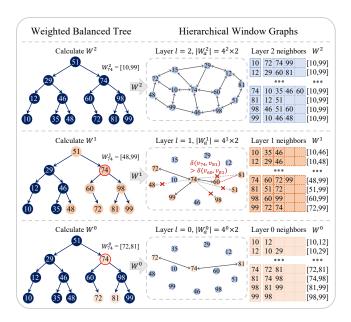


Figure 2: Index structure of WoW

values. The distance of vertices in the graph corresponds to the distance of vectors, and edges are all directed.

On the left side, a *weighted balanced tree* (WBT) [49] is used to calculate the windows of an attribute in different layers. Every node in WBT records the rooted tree size. WoW can accurately maintain windows of vertices and easily retrieve in-window neighbors. An ordered array (Figure 1) can also be used to compute windows. But it suffers a degradation to linear insertion complexity, whereas WBT achieves logarithmic time. Unlike RangePQ [79] tightly coupling the WBT and PQ structures into an integrated framework, WoW adopts WBT as a lightweight plug-in to accelerate attribute insertion, replacing the ordered array.

On the right side, there are three window graphs from Layer 0 to Layer 2 (the top layer). In a window graph, each vertex is only aware of the proximate vertices in a fixed-size window of the attribute. It can handle RFANNS queries where the range filter is compatible with the window size. To handle arbitrary range filters, we build hierarchical window graphs with varying size. For visual clarity, in our example, only edges of  $v_{74}$  in Layer 0 and Layer 1 are shown, and it has two out-edges and two in-edges in Layer 0. Neighbors and windows of some vertices are also provided on the right. Compared with the multi-layer structure of HNSW [43] for faster vector neighborhood approaching, WoW deploys hierarchy to store neighbors in varying-sized windows, designed to serve the attribute instead of the vector.

The window of an attribute value a contains an equal number of ordered values halved by a, where the number is calculated by  $o^l$ . If there are inadequate values inside the window, which may happen at the boundaries of the ordered dataset, the window boundaries would be limited to the dataset boundaries. For example in Figure 2,  $W_{74}^1 = [48, 99]$  as  $o^l = 4$  in Layer l = 1, and the 4th smaller value of 74 is 48, which is used as the left window boundary. As there are only three values greater than 74, the right window boundary is

# Algorithm 1: Insert **Input:** $v_a$ : vector v with attribute value a to insert **Output:** index I with $v_a$ inserted **Hyperparameter**: m, o, $\omega_c$ : defined in Table 1 1 *top* ← top layer of I; $_{2}$ if $|\mathcal{A}| + 1 > 20^{top}$ then ightharpoonup top window cannot cover ${\mathcal A}$ Clone graph at top layer to top + 1 layer; $top \leftarrow top + 1$ ; 4 5 for $l \leftarrow top$ to 0 do $W_a^l \leftarrow \text{window of size } 20^l \text{ halved by } a;$ $ep \leftarrow$ a random vertex with attribute value in $W_a^l$ ; $L \leftarrow [l, top], U \leftarrow \{v_i \mid v_i \in U^{l+1} \land i \in W_a^l\}; \triangleright U^{top+1} = \emptyset$ if |U| > m then $U^l \leftarrow U$ ; else $U^l \leftarrow U \cup \text{SearchCandidates}(\textit{ep}, v_i, W_a^l, L, \omega_c);$ 10 $N_{v_a}^l \leftarrow \mathtt{RNGPrune}(v_a, U^l, \frac{m}{2}); \qquad \triangleright \; \mathtt{select} \; \frac{m}{2} \; \mathtt{neighbors}$ 11 $\begin{array}{c|c} \sigma_a & \text{for each } v_b \in N^l_{v_a} \text{ do} \\ & \text{if } |N^l_{v_b}| < m \text{ then} \\ & & L^l_{v_b} \leftarrow N^l_{v_b} \cup \{v_a\}, \text{ continue}; \end{array}$ 12 13 14 $W_b^l \leftarrow \text{window of size } 2o^l \text{ halved by } b;$ $U' \leftarrow \{v_a\} \cup \{v_i \mid v_i \in N_{v_b}^l \land i \in W_b^l\};$ $N_{v_b}^l \leftarrow \text{RNGPrune}(v_b, U', m);$ 15 16 18 Insert a into WBT and connect $\bigcup_{l=0}^{top} N_{v_a}^l$ from $v_a$ ;

set to the right boundary of the dataset, 99. The size of top-layer windows (i.e.  $|W^2|$ ) is greater than the number of inserted attribute values ( $|\mathcal{A}|$ ), yielding a proximity graph only aware of vectors.

WoW leverages the RNG pruning strategy (abbr. RNGPrune) inside windows to diversify the neighbor distribution and strengthen the navigation ability. It has been widely studied as one of the most effective way to accelerate query speed [7, 21, 43, 76]. For example, in Layer 1,  $v_{81}$  is not selected as the neighbor of  $v_{74}$ , because edge  $v_{74} \rightarrow v_{81}$  violates the RNG property in Definition 4 (i.e.  $\delta(v_{74},v_{81})>\delta(v_{74},v_{60})\wedge\delta(v_{74},v_{81})>\delta(v_{60},v_{81})$ ). As a result, it becomes the longest edge in triangle  $\Delta v_{74}v_{60}v_{81}$  and thus deleted. The final neighbors of  $v_{74}$  are settled as  $v_{60},v_{72}$ , and  $v_{99}$ .

#### 3.2 Top-down Insertion

19 return I;

The hierarchical window graphs are constructed layer by layer, from an empty graph at the beginning.

In Algorithm 1, when the top-layer window fails to cover the inserted attribute set (Line 2), the top layer would be raised by cloning the entire old-top to the newly allocated layer (Lines 3–4). Next, starting from the top layer, we compute the window of  $v_a$  in Layer l,  $W_a^l$ , which covers  $2o^l$  vertices whose attribute values are halved by a (Line 6). This can be determined by WBT with logarithmic time complexity (see Appendix A). In Lines 8–10, the nearest candidates can be found from two sources: (1) In-window candidates of the previous layer,  $U^{l+1}$ , if the number of them is more than m. (2) Newly retrieved candidates by the beam search procedure on the existing graph, merged with inadequate previous-layer in-window candidates. We prove in Theorem 3.1 that in

higher layers of WoW, candidates are averagely closer to  $v_a$  than those in lower layers. So, if some candidates in the previously processed layers are reserved after filtering for the current layer in Line 8, they should exist in the result given by SearchCandidates (Algorithm 2) for the current layer. Thereby, SearchCandidates can be skipped for faster indexing.

In Line 11, as suggested in [43],  $\frac{m}{2}$  nearest neighbors without dominance relation are selected from the candidates. The rest  $\frac{m}{2}$  empty slots are reserved for latecomers. For each neighbor  $v_b$ , we append  $v_a$  into the neighbor list if the list is not full (Line 13). Otherwise, a two-stage pruning procedure is triggered. It first recalculates the window of  $v_b$  and discards neighbors outside the window (Line 16). Then for the rest of them, RNGPrune works as the second-stage pruning to get the final neighbor list of  $v_b$  (Line 17). Notice that some neighbors may fall out of window after insertion of  $v_a$  in Lines 13–14. They are not pruned immediately for two reasons: (1) Although they are out of the window, they may be in the query range if later insertions do not prune them. (2) After insertion of more vertices, they may get back into the window and become valid edges again. At last, the attribute value a is inserted into WBT and relevant vertices are connected with each other (Line 18).

Theorem 3.1. Without pruning dominated vertices in Line 11 and Line 17, for vertex  $v_a$ , we have  $\sum_{s \in N_{p_a}^{l+1}} \delta(s, v_a) \leq \sum_{t \in N_{p_a}^{l}} \delta(t, v_a)$ .

PROOF. Without RNGPrune,  $N_{v_a}^l$  and  $N_{v_a}^{l+1}$  contain the nearest vertices to  $v_a$  covered by  $W_a^l$  and  $W_a^{l+1}$ , respectively. As window  $W_a^l$  is a subset of window  $W_a^{l+1}$ , there exists a subset  $W'\subseteq W_a^{l+1}\backslash W_a^l$  s.t.  $\forall j\in W', \delta(v_j,v_a)<\min_{i\in W_a^l}\delta(v_i,v_a)$ .

In addition, SearchCandidates traverses on the graph in a greedy manner, thus for layer l+1 it first expands the candidate list from W' and later from  $W_a^l$  before the list is full. The inequality holds.  $\sum_{u \in N_{v_a}^l} \delta(v_a, u) = \sum_{u \in N_{v_a}^{l+1}} \delta(v_a, u), \text{ if and only if } W' = \emptyset.$ 

The conclusion also holds with a high probability when RNG-Prune is not omitted, as the pruning algorithm runs in a greedy manner and the nearest neighbors are often reserved [7, 43, 64]. □

Figure 3 depicts an example of inserting  $v_{73}$  into the index built in Figure 2. First, we calculate the windows of  $v_{73}$  in all layers using WBT (Figure 3a), where  $W_{73}^2 = [10,99], W_{73}^1 = [48,99], W_{73}^0 = [72,74]$ . With these windows, as shown in Figure 3b, we can acquire candidates for  $v_{73}$  in all window graphs using Algorithm 2 (which will be presented shortly). In Figure 3c, once we have acquired the candidates,  $\frac{m}{2} = 2$  neighbors of  $v_{73}$  are picked for each layer, and the neighbors of neighbors are adjusted if necessary. In Layer 1,  $v_{74}$  and  $v_{98}$  are chosen, and  $v_{60}$  and  $v_{99}$  are discarded for adequate out-edges.  $v_{99}$  is also dominated by  $v_{74}$  as  $v_{73} \rightarrow v_{99}$  is the longest edge in triangle  $\Delta v_{73}v_{74}v_{99}$ . In the perspective of  $v_{98}$ ,  $v_{60}$  is outside the window after the insertion, but pruning it can be postponed because there is empty space in the neighbor list.

#### 3.3 Multi-layer Candidate Acquisition

As shown in Algorithm 2, WoW employs beam search to approach the neighborhood of the target vector following many graph-based indices [21, 39, 43, 59, 78]. It starts with an entry ep, whose corresponding attribute value satisfies the range filter R. Beam search only traverses in layers within a range L with a fixed beam width

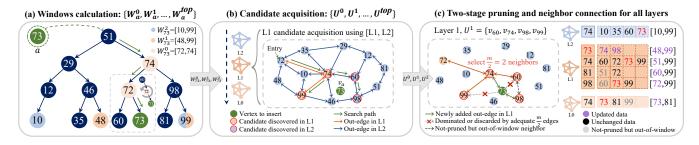


Figure 3: Single insertion of WoW. (a) The windows of  $v_{73}$  in different layers, and insertion triggers the self-balancing of WBT. (b) The procedure of acquiring candidates for Layer 0 to Layer *top*, taking Layer 1 for instance. (c) The procedure of selecting neighbors from the candidates and connecting the new vertex into existing graphs, taking Layer 1 for instance.

```
Algorithm 2: SearchCandidates
   Input: ep: graph entry; v: target vector; R: range filter;
            L = [l_{\min}, l_{\max}]: layer range; \omega: beam search width
   Output: U: nearest candidates
   Hyperparameter: m: defined in Table 1
1 C \leftarrow ep;
                               ▶ candidate min-heap during beam search
_{2} U \leftarrow ep;
                                ▶ result max-heap of nearest neighbors
3 while |C| ≠ 0 do
        l \leftarrow l_{\max}, c_n \leftarrow 0, next \leftarrow \mathbf{true}; \quad \triangleright \text{ next: early-stop flag}
        s \leftarrow the nearest vertex to v in C;
5
        if \delta(s, v) > \max_{u \in U} \delta(u, v) then break;
 6
        while l \ge l_{\min} and next do
                                                      ▶ top-down traversal
7
             next \leftarrow false;
                                       ▶ shall we check the next layer?
 8
             foreach unvisited neighbor v_i \in N_s^l do
                 if j \notin R then next \leftarrow true;
10
                 else if c_n \leq m then
                                                    ▶ discover candidates
11
                      Mark v_i as visited, c_n \leftarrow c_n + 1;
12
                      t \leftarrow the farthest vertex to v in U;
13
                      if |U| < \omega or \delta(v_i, v) < \delta(t, v) then
14
                           Add v_i into C, U;
15
                           if |U| > \omega then pop from U;
            l \leftarrow l - 1;
17
18 return U;
```

 $\omega$  (a.k.a. ef in HNSW [43]). Lastly, the nearest neighbors to target v with attribute values all in range R are returned.

At each hop, the nearest candidate s is selected from C (Line 5). If the potential neighbors of v are not fully examined (Line 6), the neighbors of s should be checked in a top-down manner (Line 7). As suggested in Theorem 3.1, we give priority to high-layer unvisited neighbors (Line 9), and push them into C, U (Line 15) if their distance is smaller than the greatest in U (Line 14). In Line 4,  $c_n$  is used to record how many distance computations are made, similar to [51, 72]. It informs WoW to jump to the next hop if m neighbors are checked (Line 11). The flag next is used as an efficient early-stop strategy to avoid unnecessary low-layer visit. We notice that if all neighbors are in range, despite whether they have been visited, we do not need to check layers below because neighbors are adequate

#### Algorithm 3: SearchKNN

```
Input: (q, R): query vector with range; k: number of nearest vectors; \omega_s: beam search width for query

Output: k nearest neighbors

Hyperparameter: o: defined in Table 1

> Step 1: decide landing layer

1 n' \leftarrow number of vertices in R;

2 l_h \leftarrow \lfloor \log_o \frac{n'}{2} \rfloor;

3 l_d \leftarrow \arg\max_{l \in \{l_h, l_h + 1\}} \frac{\min(2o^l, n')}{\max(2o^l, n')};

> landing layer l_d

4 ep \leftarrow vertex with attribute value closest to the median of R;

> Step 2: acquire multi-layer candidates

5 U \leftarrow SearchCandidates (ep, q, R, [0, l_d], \omega_s);

6 return k nearest vectors to q in U;
```

at the current hop (Line 10). If the early-stop flag fails to prevent low-layer checks, which means that some neighbors are filtered and the search is not sufficient, we go deeper to find more candidates (Line 17). The search terminates when *C* is exhausted (Line 3).

Figure 3b gives a candidate acquisition example with layer range L=[1,2], range filter R=[48,99] (i.e.  $W^1_{73}$ ), target vector  $v_{73}$ , and beam search width  $\omega=4$ . The random entry ep is set to  $v_{72}$ . The neighbors of  $v_{72}$  in Layer 2 are  $v_{10}, v_{35}$ . As they are filtered out by R, the algorithm then checks Layer 1 and appends the only in-range neighbor  $v_{74}$  into the candidate list. At the 2-hop  $v_{74}$ , it finds an in-range neighbor  $v_{60}$  in Layer 2 and  $v_{99}$  in Layer 1. Since  $v_{60}$  is closer to  $v_a$ , it is picked as the 3-hop whose neighbor  $v_{98}$  in Layer 1 is added into C, U at this hop. The last hop is  $v_{99}$  and there are no more candidates in C because other vertices are far away from  $v_a$ . The search converges and the algorithm returns  $v_{60}, v_{74}, v_{98}, v_{99}$  as the candidates, with attribute values all in range [48,99].

#### 3.4 Selectivity-aware Range Search

To answer an RFANNS query (q,R), Algorithm 3 employs two steps. (1) The above candidate acquisition procedure requires to decide a feasible layer range  $L = [l_{\min}, l_{\max}]$ . For  $l_{\min}$ , we can simply set it to Layer 0 and let the early-stop strategy in Algorithm 2 to decide the actual lowest layer at each hop. For  $l_{\max}$ , it is acceptable to set it to top, as Theorem 3.1 suggests that neighbor lists in high layers may have closer candidates than those in lower layers. However,

windows in these layers are large and most neighbors are out of range, which may increase the overhead of many unnecessary filter checks. Thus, we want to choose a landing layer  $l_d$  that contains most in-range proximate vertices to reduce filter checks and distance computations simultaneously. Such a layer resides in either the one with the largest window whose size is less than n' (the number of in-range vertices), namely  $l_h$  (Line 2), or  $l_h+1$  (Line 3). So, n' becomes essential, which also reflects the selectivity of range filter in Definition 3. Here, WBT functions as an order statistic tree [15] to give an accurate n' in logarithmic time (see Appendix B). (2) We retrieve the most nearest  $\omega_s$  in-range vectors with Algorithm 2 and return the top-k final results.

# 3.5 Influence of Hyperparameters

Like other graph-based indices [7, 21, 43], WoW has two important hyperparameters in construction: maximum outdegree m and construction beam search width  $\omega_c$ . m determines the density of the graph and distance computations at one hop. The optimal m varies with the vector distribution according to [9, 43, 59, 64, 65]. Higher  $\omega_c$  yields more candidates for neighbor selection, leading to more diversified connections at the cost of longer indexing time [43, 65].

The window boosting base o controls the number of layers. This, in turn, affects indexing speed. Furthermore, Theorem 3.2 proves that o can also impact the proportion of in-range neighbors of a certain vertex on the landing-layer search path. Recall that it is the optimal layer in terms of the balance between filter checks and distance computations as described in Section 3.4. More in-range neighbors in the landing layer would benefit query with faster navigation speed [20, 21, 43]. Therefore, Theorem 3.2 motivates us to select a suitable o for fast indexing and optimal query performance.

Theorem 3.2. In the landing layer  $l_d$ , the expected fraction  $f_R$  of in-range neighbors at a single hop on the path is bounded by

$$f_{R} = \begin{cases} \left(\frac{1}{\sqrt{o}}, \frac{1}{2}\right) & (a) \ l \in (l'-1, l'-\frac{1}{2}), o > 4, \\ \left[\frac{\sqrt{2}}{2} - \frac{1}{4o^{l+1}}, \frac{3}{4} - \frac{1}{4o^{l+1}}\right) & (b) \ l \in (l'-1, l'-\frac{1}{2}), o \leq 4, \\ \left[\frac{3}{4} - \frac{1}{4o^{l}}, 1 - \frac{o^{l}+1}{4o^{l+\frac{1}{2}}}\right] & (c) \ l \in [l'-\frac{1}{2}, l'], \end{cases}$$

where n' is the number of in-range vectors for filter R = [x,y],  $l' = \log_o \frac{n'}{2}$ , and  $l = l_h = \lfloor \log_o \frac{n'}{2} \rfloor$ , which is the highest layer defined in Line 2 of Algorithm 3, whose window size is less than R.

The proof is provided in Appendix C. For example, when o = 2, n' = 2,048, we have l = l' = 10, which satisfies Case (c). In this setting, the lower bound is 74.97% and the upper is 82.30%, leading to 12–13 in-range neighbors at a single hop when m = 16. If o is a large value, the construction would be fast, but the lower-bound in Case (a) would be small. To achieve the optimal fraction lower bound in all cases, while maintaining good indexing performance, setting o = 4 is recommended, as verified in Section 4.4.

#### 3.6 Complexity Analysis

Index size. The index size is twofold. (1) WBT costs O(cn) space, where c is a constant overhead of the tree storage. (2) Neighbor lists in the hierarchy of window graphs occupy most of the space. A vertex in Layer l has an average outdegree of  $\min(2o^l, m)$ . As the top layer is  $top = \lceil \log_o \frac{n}{2} \rceil$ , the total space for all link lists is

 $n\sum_{l=0}^{top}\min(2o^l,m)$ . Considering the two parts, the overall asymptotic space complexity is  $O(cn+mn\lceil\log_o\frac{n}{2}+1\rceil)$ .

Insertion time. In Lines 1–4 of Algorithm 1, the amortized complexity to raise the top layer is  $O(\lceil \log_o \frac{n}{2} \rceil)$ . In Lines 5–18, the vector would be inserted into  $\lceil \log_o \frac{n}{2} + 1 \rceil$  layers. In Layer l, it costs  $O(\log_{\alpha'} n)$  to locate the window boundaries (by searching in WST, a.k.a. BB[ $\alpha$ ] tree), where  $\alpha' = \frac{1}{1-\alpha}$ . Then in Line 10, candidates are retrieved from the window graph inside  $W_a^l$  with size  $2o^l$ . In high-dimensional space, the complexity of beam search scales to  $O(\log 2o^l)$  on RNG-based indices [7, 20, 21, 43, 52, 65]. The worst-case time complexity of candidate acquisition occurs in the situation when the SearchCandidates procedure is invoked for all layers, and the result is  $\sum_{l=0}^{top} O(\log 2o^l) = O(\lceil \log_o n \rceil^2 \log 2o)$ . RNG neighbor selection for v and two-stage pruning for the neighbors of v in Lines 11–17 have sublinear time complexity,  $O(\frac{m}{2}\log \omega_c + m^2\log m + m\log_{\alpha'} n)$ . The insertion of WBT scales to  $O(\log_{\alpha'} n)$  with amortized constant number of self-balancing operations. In summary, the worst-case insertion time complexity scales to  $O(\log^2 n)$ .

Query time. For landing layer selection, the cardinality of the filtering subset by the range filter [x, y] can be calculated by the rank i, j of the upper bound of x and the lower bound of y from WBT, which requires  $O(\log n)$  time separately. Then, n' can be calculated by j - i + 1. The complexity of candidate acquisition is  $O(\log n')$ . The overall query time complexity scales to  $O(\log n')$ .

#### 3.7 Further Extensions

Duplicate attribute values. Algorithm 2 is agnostic to value redundancy and relies solely on the query range. Duplicate values have the same rank in Definition 4, and thus only vectors are considered.

Specifically, duplicate values are not inserted into WBT. Only their corresponding vectors are inserted into the window graphs. The condition of Line 2 in Algorithm 1 is changed to  $|\mathcal{A}|_u + 1 > 20^{top}$ , where  $|\mathcal{A}|_u$  denotes the number of unique values. We define *unique selectivity* as the ratio of distinct values in the filtered set to those in  $\mathcal{A}$ . In Step 1 of Algorithm 3, the layer with window size closest to the number of in-range unique values is selected, aligning with the filter's unique selectivity. With duplicate values, the number of layers reduces from  $\lceil \log_o \frac{n}{2} + 1 \rceil$  to  $\lceil \log_o \frac{|\mathcal{A}|_u}{2} + 1 \rceil$ . This leads to proportionally lower space and insertion time complexity.

Deletion and in-place update. For a single deletion, we can mark the deleted vertex and normally traverse it without pushing it into the result max-heap. It will be removed from the neighbor list if the two-stage pruning procedure is triggered. The query complexity stays unchanged if the amount of deletions is not significant. For frequent deletions and in-place updates, existing methods [41, 57, 77] can be integrated into all graph-based indices, including WoW.

Multi-attribute RFANNS. It is still an open problem to design a dedicated multi-attribute index. Existing works discuss several rudimentary solutions [38, 72, 82]: (1) Building a composite index and querying on it via lexicographic order; (2) Decomposing one query into sub-queries on corresponding single-attribute indices before merging the separate results; (3) Querying on a single-attribute index for one predicate and applying in-filtering or post-filtering on others. There are two future directions to support multi-attribute

RFANNS in WoW: (1) For the second existing solution, sub-queries can be conducted in one round by composing relevant layers from multiple single-attribute indices. Our 1D window can also be extended to high-dimensional rectangles for disjunctive range queries [58]; (2) If the filter has complex predicates on several attributes, a learned cardinality estimator [29, 48] for these predicates can be employed to assist WBT to guide the layer selection.

# 4 Experiments and Results

In this section, we evaluate WoW in index construction and query by answering the research questions below:

- **RQ1.** Is the parallel incremental construction of WoW efficient? Are its indexing time and index size competitive?
- **RQ2.** Does WoW surpass existing methods in RFANNS querying? How is the performance under different query selectivity?
- **RQ3.** Can our optimizations, e.g., RNG pruning, early-stop and landing layer selection strategies, improve performance? How are the index extensibility and parameter sensitivity?

# 4.1 Experiment Setting

*Environment.* All experiments are conducted on a Ubuntu 20.04 server with an Intel Xeon E5-1607 v3 (3.10GHz) CPU and 240GB RAM. The CPU has 128K L1 cache, 10M L3 cache, and SIMD support. All methods are implemented in C++ and compiled by GCC 9.4.0.

*Datasets.* We pick five real-world datasets with different characteristics. Table 3 lists their statistical data.

- **Sift** and **Gist**<sup>1</sup> are two datasets embedded from images to assess the query performance of ANN indices. Following the practice of prior works [19, 51, 64, 72, 82], we generate a random integer for each vector as its attribute value.
- ArXiv<sup>2</sup> is a recent hybrid dataset released by QDrant. It
  involves vectors embedded from over two million paper
  titles using the all-MiniLM-L6 encoder, along with release
  time as the attribute.
- Wikidata4M<sup>3</sup> is embedded from Wikipedia entity descriptions by the Cohere Embed v3 model. We use the first four million vectors, with entity title as the attribute.
- Deep10M<sup>4</sup> is obtained from the last fully-connected layer of the GoogleNet model [73]. We use the first 10 million vectors, with vector ID as the attribute.

For simplicity, we follow previous works [19, 72, 82] to first sort the vectors by attribute values and then use vector IDs as the new attribute values. The new vector-attribute pairs are reshuffled to simulate the *unordered insertion* scenario.

Each query vector is assigned a query range, which is randomly generated by the fraction of in-range vectors over the dataset size f (cf. Definition 3). For example, the query range fraction  $f = 2^{-6}$  on a dataset with 1,000 vectors indicates that there exist  $\lfloor 1000 \times 2^{-6} \rfloor = 15$  in-range vectors, and a corresponding query range can be [1, 15], [3, 17], etc. We refer to ranges with fractions in  $\lfloor 2^{-10}, 2^{-9} \rfloor$  as exhibiting extreme selectivity,  $\lfloor 2^{-8}, 2^{-6} \rfloor$  as high

selectivity,  $[2^{-5}, 2^{-3}]$  as moderate selectivity, and  $[2^{-2}, 2^0]$  as low selectivity. We further assemble a *mixed* range workload comprising query ranges with equal number of fractions in  $[2^{-10}, 2^0]$ .

In our experiments, each query workload contains 1,000 rangefiltering queries. We estimate the hardness of a query workload by Local Intrinsic Dimensionality (LID) [3, 6, 37, 67], as defined below:

DEFINITION 6 (LOCAL INTRINSIC DIMENSIONALITY, LID). Given a hybrid dataset  $\mathcal{D} = \{V, \mathcal{A}\}$  and a range-filtering query workload  $Q = \{V_Q, \mathcal{R}\}$ , the LID of Q is estimated by each query (v, R)'s top-k in-range nearest neighbors  $(u_i, a_i) \in \{u_i \in N(v) \land a_i \in R\}$ :

$$LID@k = \mathbb{E}_{(v,R) \in Q} \left[ -\left( \frac{1}{k} \sum_{i=1}^{k} \log \left( \frac{\delta(v,u_i)}{\delta(v,u_k)} \right) \right)^{-1} \right]. \tag{2}$$

In Table 3, we show LID@10 of the fractions  $2^0$ ,  $2^{-1}$ ,  $2^{-4}$ ,  $2^{-7}$ , and  $2^{-10}$ . A larger LID indicates a more challenging dataset. For the datasets defined under the same distance metric (e.g., Sift, Gist, and Deep10M), the hardest one is marked with " $\star$ " (e.g., Gist).

Competitors and parameters. We choose six RFANNS indices, WST [19], iRangeGraph [72], SeRF[82], HSIG [38], RangePQ [79], and DIGRA [32], two generic predicate-filtering indices, Milvus-HNSW [63] and ACORN [51], as well as two baselines, pre-filtering and post-filtering. All indices except pre-filtering and RangePQ are RNG-based, so they share two important hyperparameters in index building: the construction beam search width  $\omega_c$  and the maximum outdegree m (for HNSW, m is the bottom-layer maximum degree). Considering dataset size and hardness (LID@10), we prepare a set of default values for fair comparison:  $\omega_c = 128$ , m = 16 for Sift, and  $\omega_c = 256$ , m = 16 for the rest. Following [19, 26, 38, 51, 64, 72, 82], all indices find k = 10 nearest neighbors for each of 1,000 queries.

- **WoW** is our index. We set the window boosting base to *o* = 4 as analyzed in Section 3.5.
- **SeRF** employs *2DSegmentGraph* to handle arbitrary ranges. We use  $leap = MAX\_POS$  as suggested in the paper. Since SeRF cannot gain satisfactory recall (0.85) under the default hyperparameters, we increase to  $\omega_c = 512$ , m = 64.
- **WST**. We pick the most performant indexing strategy in it, *SuperPostFiltering*. The branching factor is set to  $\beta = 2$ .
- iRangeGraph does not have additional parameters.
- **HSIG** splits a dataset into slots according to the attribute. We set *slot\_num* = 8 as studied in its paper.
- RangePQ. We set L<sub>base</sub> to 1,000 for Sift, 3,000 for Gist, and 10,000 for the others, as studied in its paper.
- DIGRA has no extra parameters. Due to query performance degradation (recall < 0.85) caused by insertions, RangePQ and DIGRA are evaluated only under static construction.
- **Milvus and ACORN**. We use Milvus v2.5 and build HNSW indices, where  $\omega_c$ , m are set to default. For ACORN, we set m = 64,  $m_\beta = 2m$  to get the best performance, and  $\gamma = 2^{10}$  to support the highest selectivity as it suggests.
- **Pre-filtering and Post-filtering.** Pre-filtering is used to generate the ground truth. For post-filtering, we set the size of intermediate result to  $s \times k$  (s is defined in Definition 3).

<sup>1</sup>http://corpus-texmex.irisa.fr

<sup>&</sup>lt;sup>2</sup>https://github.com/qdrant/ann-filtering-benchmark-datasets

<sup>&</sup>lt;sup>3</sup>https://huggingface.co/datasets/Cohere/wikipedia-2023-11-embed-multilingual-v3

 $<sup>^4</sup> https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search\\$ 

Dataset	Modality	Size	Dimension	Distance metric	Attribute type	LID@10 $(2^0 / 2^{-1} / 2^{-4} / 2^{-7} / 2^{-10})$
Sift	Image	1,000,000	128	Euclidean	Random integer	28.3 / 31.7 / 28.8 / 26.0 / 22.3
Gist	Image	1,000,000	960	Euclidean	Random integer	★ 66.6 / 64.9 / 57.3 / 48.5 / 37.6
ArXiv	Text	2,138,591	384	Cosine	Time	12.6 / 13.0 / 15.5 / 17.7 / 17.9
Wikidata4M	Text	4,000,000	1,024	Cosine	Title	<b>★</b> 25.3 / 26.7 / 27.9 / 29.4 / 29.6
Deep10M	Image	10,000,000	96	Euclidean	Vector ID	32.8 / 31.6 / 30.1 / 27.5 / 24.2

Table 3: Statistical data of the used datasets

Table 4: Index size and indexing time

Size (MB)	Sift	Gist	ArXiv	Wikidata4M	Deep10M
HNSW-L0	76	72	163	305	762
Milvus	137	137	294	550	1,375
ACORN	4,832	4,832	9,821	18,371	45,967
SeRF	430	457	1,133	2,385	5,486
WST	1,002	806	3,043	5,763	16,353
iRangeGraph	869	793	2,247	4,715	11,598
HSIG	1,101	1,101	2,355	4,405	11,014
RangePQ	791	3,857	3,495	16,259	4,293
DIGRA	1,533	1,533	3,547	9,501	21,477
WoW	713	713	1,664	3,112	8,430
Time (s)	Sift	Gist	ArXiv	Wikidata4M	Deep10M
Time (s) HNSW-L0	Sift 17	Gist	ArXiv 111	Wikidata4M 351	Deep10M 319
					-
HNSW-L0	17	113	111	351	319
HNSW-L0 Milvus-HNSW	17 25	113 176	111 252	351 918	319 529
HNSW-L0 Milvus-HNSW ACORN	17 25 313	113 176 2,343	111 252 2,201	351 918 8,542	319 529 6,089
HNSW-L0 Milvus-HNSW ACORN	17 25 313 <b>112</b>	113 176 2,343 <u>591</u>	111 252 2,201 <b>592</b>	351 918 8,542 3,053	319 529 6,089 <b>1,574</b>
HNSW-L0 Milvus-HNSW ACORN SeRF WST	17 25 313 <b>112</b> 257	113 176 2,343 591 1,688	111 252 2,201 <b>592</b> 6,418	351 918 8,542 3,053 35,394	319 529 6,089 <b>1,574</b> 11,602
HNSW-L0 Milvus-HNSW ACORN SeRF WST iRangeGraph	17 25 313 <b>112</b> 257 463	113 176 2,343 591 1,688 2,401	111 252 2,201 <b>592</b> 6,418 3,211	351 918 8,542 3,053 35,394 8,770	319 529 6,089 <b>1,574</b> 11,602 9,729
HNSW-L0 Milvus-HNSW ACORN SeRF WST iRangeGraph HSIG	17 25 313 <b>112</b> 257 463 960	113 176 2,343 <u>591</u> 1,688 2,401 2,305	111 252 2,201 <b>592</b> 6,418 3,211 4,383	351 918 8,542 3,053 35,394 8,770 20,507	319 529 6,089 <b>1,574</b> 11,602 9,729 12,976
HNSW-L0 Milvus-HNSW ACORN SeRF WST iRangeGraph HSIG RangePQ	17 25 313 <b>112</b> 257 463 960 566	113 176 2,343 <u>591</u> 1,688 2,401 2,305 4,508	111 252 2,201 <b>592</b> 6,418 3,211 4,383 4,105	351 918 8,542 3,053 35,394 8,770 20,507 20,494	319 529 6,089 <b>1,574</b> 11,602 9,729 12,976 4,493

# 4.2 RQ1: Index Construction Efficiency

We build DIGRA and RangePQ in one thread as they do not implement parallelism, and other indices with 16 threads, including a single-layer HNSW built by hnswlib (HNSW-L0). We also build WoW in a single thread, denoted by WoW (1 thd.), for comparison. Table 4 presents the index size and indexing time of the evaluation datasets. The size of raw vectors and attribute values are excluded.

- (1) For the indices designed for generic predicate, the index size and indexing time of Milvus-HNSW are better than those of ACORN, since Milvus-HNSW is a vanilla HNSW over all vectors, which is fast to build and the index size is on par with HNSW-L0. ACORN builds a dense HNSW without RNG pruning to ensure query accuracy, leading to the largest index size among all indices.
- (2) SeRF outperforms all competitors in terms of both index size and indexing time on the easy datasets (i.e. Sift, ArXiv, and Deep10M). Despite its high indexing efficiency relying on ordered insertion, SeRF fails to achieve stable recall on most query workloads, which is shown shortly in Section 4.3.

- (3) WoW achieves the second best in indexing time on the easy datasets, and outperforms all indices on the hard datasets, Gist and Wikidata4M. The acceleration over graph-based indices is from (i) Its worst-case time complexity is on par with the average time complexity of others [19, 32, 38, 72, 82]; (ii) Like SeRF, WoW acquires candidates on the incomplete graph. The candidate acquisition speed relies heavily on the graph quality. SeRF compresses oracle HNSWs and some proximity relations may be lost. Differently, WoW has almost identical quality to the oracle graph, as evaluated in Section 4.4. On the hard datasets, dense neighborhoods around targets make higher-quality graphs beneficial for faster indexing. The size and indexing speed of RangePQ are sensitive to dimensions. It is faster than WoW (1 thd.) on low-dimensional datasets (e.g., Sift and Deep10M) but less competitive on high-dimensional datasets.
- (4) For a given dataset, the size of WoW is roughly the product of the layer number and the size of HNSW-L0.

If the layer number is consistent, the index size also grows linearly with the number of vectors. Comparing the indices built on ArXiv and Wikidata4M, their layers are the same, which can be calculated by  $\lceil \log_o \frac{n}{2} + 1 \rceil$ . The index size on Wikidata4M is about  $2\times$  of that on ArXiv.

(5) WoW can achieve 5.0–6.9× multi-thread acceleration by fine-grained locks at the vertex level [43]. The parallelism ability enables WoW to scale to datasets with larger size and higher dimensions.

# 4.3 RQ2: RFANNS Query Performance

This evaluation is carried out using one single thread. As shown in Figure 4, we assess query performance using the Queries Per Second (QPS)-Recall@10 curves [5, 37, 43]. According to Table 2, we categorize the competing indices into static, post-incremental, ordered incremental, and unordered incremental. We also verify WoW with ordered insertion, named WoW (ordered). Since post-filtering, ACORN, and Milvus-HNSW are not aware of the attribute order, they are regarded as unordered incremental in this experiment.

(1) WoW surpasses other indices under the mixed range fraction workloads of all datasets. Compared to the best static indices, WoW obtains about 1.4× acceleration over iRangeGraph and 6× over WST at 90% recall. Compared to the post-incremental index DIGRA, which is built statically in our experiments, WoW achieves identical or better performance. It is nontrivial for a dynamic index to maintain competitive performance against statically-built ones, because it has to correctly reorganize the neighbor relations at every new insertion. Compared to the unordered incremental index HSIG, WoW gains about 4× query speed-up at 90% recall. Compared to the ordered incremental index SeRF, WoW is 6× speedup on the Wikidata4M dataset that SeRF can achieve 90% recall.

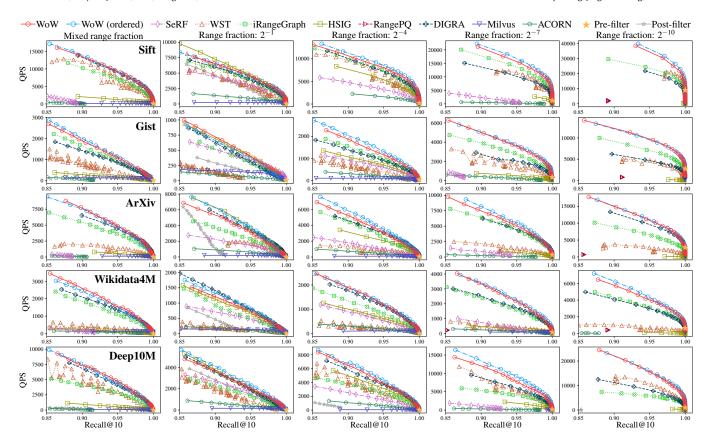


Figure 4: QPS-Recall@10 curves. For each dataset, five workloads with varying range fractions are shown, including the mixed range fraction workloads. Dotted line  $(\cdot \cdot \cdot)$  stands for static index, dashed line (- - -) for post-incremental, dash-dotted line  $(- \cdot -)$  for ordered incremental, and solid line (-) for unordered incremental. Curves below 85% recall are omitted.

(2) WoW (ordered) slightly exceeds WoW under a few workloads (e.g., on Sift). This is because for ordered insertion, the window of a certain vertex would stay unchanged after the data located at the right window boundary is inserted. For unordered insertion of a data object, however, windows of vertices that have close attribute values to the inserted one are changing all the time. Therefore, the first pruning stage is triggered continuously to maintain the window property and some high-quality neighbors may be pruned out. It may lead to the loss of some neighbors that are supposed to exist in the final window, and a slight decline in query performance compared to WoW (ordered).

(3) Across workloads of varying selectivity, WoW and WoW (ordered) consistently perform the best, while some competitors (e.g., SeRF, HSIG, DIGRA, and iRangeGraph) face performance degradation as the selectivity increases. As most vectors can pass the range filter under workloads of low selectivity (e.g.,  $f=2^{-1}$ ), it is easy for all RFANNS indices to discover neighbors with high recall. When the selectivity increases, some indices like WST and HSIG need to do more distance computations to achieve high recalls due to less correlation between vectors and attribute values. SeRF fails to achieve 95% recall because some in-range neighbors are missing during traversal and falls into local optima, due to the lossy graph compression as reported in [72, 82]. iRangeGraph has to make a

thorough tree traversal from the root to the low-level nodes at each hop on the search path, leading to diminished query performance under workloads of high selectivity. DIGRA uses several tree nodes to cover the query range but suffers redundant distance computations if the range is less compatible with tree nodes, causing unstable performance on workloads  $f=2^{-7}$  or  $2^{-10}$  (e.g., good performance on ArXiv but decreased on others). WoW benefits from the early-stop strategy in Algorithm 2 and the selectivity-aware layer selection in Algorithm 3, to resist redundant filter checks and distance computations for workloads of high selectivity.

(4) On the hard datasets Gist and Wikidata4M, WoW, WoW (ordered), and iRangeGraph achieve superior QPS than other indices in terms of the same range fraction. Vectors in these datasets tend to crowd around the query vector, making the graph-based algorithms harder to determine the true neighbors [37, 65]. iRangeGraph can smartly decide from which tree node to choose neighbors at each hop. WoW has similar or even better graph quality than iRange-Graph, revealed in Section 4.4. In these two indices, the selected edges together compromise a high-quality RNG to withstand the hardness of the datasets. Due to the lack of navigation edges and less accurate proximity relations, other indices are struggling to achieve high query speed at high recalls (e.g., > 95%). For the easy

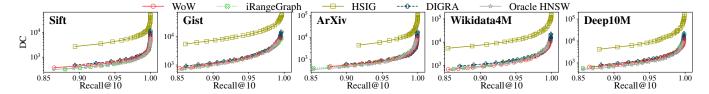


Figure 5: DC-Recall@10 curves compared to oracle HNSW for mixed range fraction. Down and to the right is better.

Table 5: QPS-Recall@10 and DC-Recall@10 of WoW and WoW without early-stop

	Recall@10		Sift Gist		ArXiv			Wikidata4M			Deep10M					
	-10-14-26 -1	90%	95%	99%	90%	95%	99%	90%	95%	99%	90%	95%	99%	90%	95%	99%
QPS	WoW	13,993	8,740	4,563	1,815	1,043	503	7,772	4,801	1,883	2,518	1,486	523	6,952	4,521	1,455
	w/o early-stop	10,491	7,400	3,538	1,538	842	484	6,702	4,472	1,628	2,265	1,171	388	6,632	4,101	1,336
DC	WoW	432	665	1,216	1,139	1,942	3,857	534	849	2,074	918	1,539	4,200	883	1,350	3,919
	w/o early-stop	520	726	1,517	1,272	2,277	3,890	617	926	2,540	1,015	1,946	5,676	943	1,569	4,855

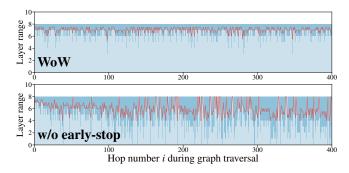


Figure 6: Layer range footprints of WoW and w/o early-stop on a single query of Gist under the range fraction  $f=2^{-4}$ . The results of the first 400 hops are shown. Dark blue bars represent the highest visited layer  $l_{max}$  and light ones are the lowest  $l_{min}$ . The gap between two bars is the exploring depth. The red curve is the trend of the median as  $\frac{1}{2}(l_{\min} + l_{\max})$ .

datasets, all indices can perform well because vectors are distributed sparsely, making it easier to determine the nearest neighbors.

- (5) Under all workloads, graph-based RFANNS indices generally outperforms those for generic predicates. Milvus can achieve stable accuracy due to its effective query optimizer to select between pre-filtering and post-filtering. ACORN is good at queries of moderate selectivity compared to Milvus, as it only calculates distance for in-range vectors, better than the post-filtering strategy of Milvus. However, its performance drops when the selectivity increases. This is because out-of-range vertices are the majority, causing beam search to end prematurely before obtaining enough candidates.
- (6) It is worth noting that SeRF cannot achieve 85% recall rate under some mixed workloads, due to its lossy compression of multiple HNSWs. RangePQ has reasonable recall under workloads with high and extreme selectivity (e.g., 88% and 91%), but struggles to obtain high recall under low and moderate ones, which is common to PQ-based indices and in accord with the conclusion in its paper. As post-incremental indices, both RangePQ and DIGRA encounter

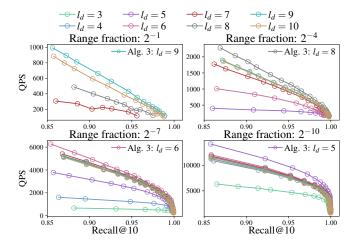


Figure 7: Effectiveness of landing layer selection. The layers selected by Algorithm 3 are given on the top right corner. Some curves are absent for failing to achieve 85% recall.

recall loss after bulk insertions. For example, after building on 50% of vectors by static construction and inserting the rest half, the highest recall that DIGRA can achieve on Sift with range fraction  $f=2^{-1}$  falls from 99% to 27%, due to the broken interconnections of graphs by tree node splitting. The performance of WST is not stable and the curves fluctuate with recall. In the post-filtering phase of WST, the query performance is affected by an extra parameter beyond  $\omega_s$ . It is used to adjust the intermediate result size, and with a fixed  $\omega_s$ , a larger size can enhance accuracy. Thus, WST brings in an additional challenge to parameter tuning. iRangeGraph encounters a prominent performance drop on the Deep10M dataset. Due to  $O(\log n + m)$  neighbor-selection overhead per hop, it cannot perform well on low-dimensional and large-scale datasets, where the distance computations are fast or the search paths are long, consistent with the observation in its paper [72].

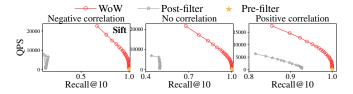


Figure 8: QPS-Recall@10 under workloads with different query correlations on the Sift dataset

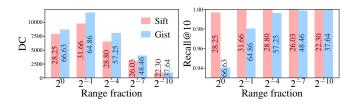


Figure 9: DC and Recall@10 for different LID@10 (marked in the middle of the bars)

# 4.4 RQ3: Detailed Analysis

Approximation to oracle RNG. The performance of graph-based RFANNS indices is bounded by the oracle proximity graph [51, 72]. The approximation quality of an RFANNS index to the oracle graph can be evaluated by comparing the Distance Computations per Query (DC) between the two. We consider indices using NSW or HNSW as their underlying graph, and construct oracle HNSWs for each query range with identical  $\omega_c$  and m. The number of distance computations performed on these oracle HNSWs represents the lower bound achievable by the most accurate RFANNS indices.

As shown in Figure 5, the DC-Recall@10 curves of WoW coincide closely with those of oracle HNSW, especially on the harder datasets and for recall above 95%. iRangeGraph and DIGRA also achieve optimal approximation to oracle HNSW. However, DIGRA encounters accuracy loss due to insertions, while iRangeGraph fails to achieve close QPS to that of WoW, due to its neighbor selection overhead as reported in Section 4.3. Moreover, they have full horizon of the entire dataset during static construction to finetune the neighbor proximity, while WoW is only aware of the inserted vectors with limited global information. HSIG supports incremental construction, but it cannot obtain a comparable approximation.

Early-stop strategy. Table 5 depicts the query performance of WoW and WoW without early-stop in Algorithm 2. WoW outperforms WoW w/o early-stop in terms of QPS and DC. This is because without early-stop, WoW has to search more on lower layers. As demonstrated in Theorem 3.1, vectors in lower layers are less accurate than those in higher ones in terms of vector proximity. It results in longer search paths and more unnecessary distance computations that may diminish query speed. The early-stop strategy can also avoid some low-layer filter checks to further improve QPS.

To further demonstrate the effectiveness of the early-stop strategy, Figure 6 shows the layer range footprints of WoW for a single query with  $f=2^{-4}$  on the hard Gist dataset. The layer range at a single hop is represented as  $[l_{\min}, l_{\max}]$ . According to the landing layer selection strategy in Algorithm 3,  $l_{\max}$  is set to 8 instead of

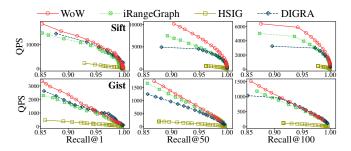


Figure 10: QPS-Recall@k (k = 1,50,100) on the mixed range fraction workloads of the Sift and Gist datasets

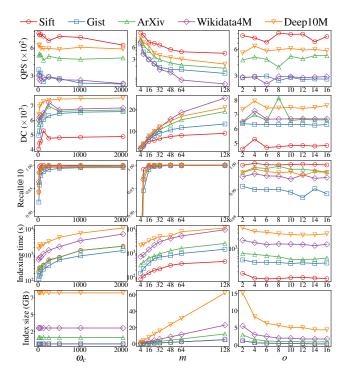


Figure 11: Parameter sensitivity on all datasets

the top layer 10.  $l_{\rm min}$  is the lowest layer that Algorithm 2 reaches. The early-stop strategy can bound the layer exploration within 1–2 layers deeper at each hop, while allowing lower layer checks if current exploration is not adequate. Without early-stop,  $l_{\rm min}$  is not bounded, therefore 4–5 lower layers are checked for most hops. In summary, the query speed acceleration is twofold: (1) Avoiding more distance computations for less accurate vectors in low layers. (2) Avoiding unnecessary filter checks in low layers.

Landing layer selection. Apart from low-layer checks optimized by the early-stop strategy, the selectivity-aware landing layer selection in Algorithm 3 can also avoid unnecessary high-layer checks. Figure 7 compares QPS of WoW landing on different layers from the *top* layer down to layer 0, on the Gist dataset. Under all studied workloads, WoW can select the optimal landing layer to achieve the best query performance. Notably, for vertices that are reachable during graph traversal, a few of them can still pass the filter above

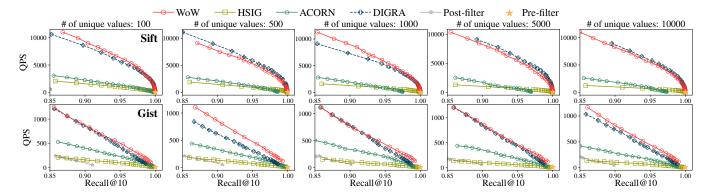


Figure 12: QPS-Recall@10 with varying numbers of unique values. Range filters are generated randomly with mixed selectivity.

the layer selected by Algorithm 3. However, out-of-range vertices constitute the majority, and are filtered out by a multitude of unnecessary filter checks on the path. Redundant filter checks may offset the improvement by the reduced distance computations. For workloads of extreme selectivity ( $f=2^{-10}$ ), the query performance improvement by the layer selection strategy is the most prominent, and the curves of  $l_d \in [6,10]$  generally follows the descending order. WoW can skip most unnecessary filter checks in high layers and directly search in the most relevant layer with the optimal expectation of in-range vertices proved in Theorem 3.2, obtaining the optimal balance between filter checks and distance computations.

Robustness for varying query correlations. We evaluate WoW on three workloads with different query correlations. The correlations are classified by the highest recall of post-filtering: low correlation ( $\leq$  20%), no correlation (40%–60%), and high correlation ( $\geq$  80%). High correlation means most of nearest vectors can satisfy the filter, while low correlation means most of them cannot. Figure 8 shows that WoW has steady performance on different workloads, demonstrating robustness for different query correlations.

Impact of dataset hardness. As shown in Table 3, LID@10 of Gist is 1.6-2.7 times of Sift. Figure 9 reveals that (1) For a large LID (e.g., 64.86 on Gist range fraction  $2^{-1}$ ), WoW requires over 11,730 DC to achieve 98.03% recall. (2) In comparison, for the same fraction  $2^{-1}$ , Sift has a small LID (31.66), leading to 9,782 DC (16% fewer than Gist) to achieve 99.91% recall. In short, higher LID can make graph-based indices harder to find results.

Performance for different Recall@k. Figure 10 shows two findings: (1) As k grows, maintaining the same recall incurs increased query latency. (2) WoW consistently attains superior query performance across different k, validating robustness to retrieval size.

*Parameter sensitivity.* Figure 11 shows the impact of the hyperparameters,  $\omega_c$ , m, and o, on building and searching performance. Overall, our parameter settings are robust across all datasets in both indexing and query efficiency.

(1) A greater beam width  $\omega_c$  results in more accurate candidates and a more accurate RNG, at the cost of longer indexing time that grows linearly with  $\omega_c$ . However, the query speed and accuracy can benefit from the high-quality RNG, as there are more long-distance

Table 6: Performance of WoW on larger datasets

	Inde	king	QPS	QPS-Recall@10				
	Size (MB)	Time (s)	90%	95%	99%			
Wikidata4M	3,112	1,557	2,518	1,486	523			
Wikidata41M	37,666	32,183	1,212	713	197			
Deep10M	8,430	3,360	6,952	4,521	1,455			
Deep100M	90,789	146,465	2,078	1,264	293			

edges to navigate search to the target neighborhood with fewer DC. Setting  $\omega_c$  from 128 to 256 is sufficient for most datasets.

- (2) A greater maximum outdegree m leads to a denser RNG and more DC per hop, while a smaller m causes a sparse graph with possibly unreachable vertices. Setting m=16 is sufficient to achieve a high recall for all datasets.
- (3) As the window boosting base o increases, the layer number, calculated by  $\lceil \log_o \frac{n}{2} \rceil + 1$ , decreases. Fewer layers reduce recall due to a lower fraction of in-range vertices in the landing layer. E.g., achieving 80% recall on Wikidata4M needs 521 distance computations when o=2, compared to 609 when o=16, representing 16% increase in redundancy. This result aligns with Theorem 3.2, Case (a). At o=4 and o=2, QPS remains competitive. o=4 yields higher recall (e.g., 99.83% vs 99.66% on ArXiv), slightly more distance computations, and reduced indexing time and index size.

Duplicate attribute values. Section 3.7 discusses the mechanism to handle duplicate attribute values. In this experiment, we evaluate indices with native support for duplicate attribute values. Each vector is randomly assigned an attribute value in range  $[1, n_c]$ , where  $n_c$  is the number of unique values. Figure 12 shows that WoW and DIGRA have clear advantage on all datasets. WoW is still superior to DIGRA for its faster indexing speed, smaller size, parallelism support, and stable performance after insertions.

Scalability to larger datasets. Table 6 demonstrates the scalability of WoW to larger datasets. Wikidata41M is the complete vector set embedded from 41,488,110 English entities. Deep100M consists of 100 million vectors from the original Deep1B dataset. As the size of Wikidata41M and Deep100M is multiplied by about 10, the index size is multiplied by 10.7 and 12.0, while the indexing time

is multiplied by 20.6 and 43.5, resp. These results are in accord with the index building analysis in Section 3.6. Due to memory limitation, the most competitive indices, iRangeGraph and DIGRA, fail to build on the Deep100M dataset. Besides, the query speed of WoW decreases sublinearly, consistent with the query time analysis.

#### 5 Conclusion

In this paper, we propose a dedicated RFANNS index WoW. The index addresses two main challenges about incremental construction and efficiency for varying range-filtering query correlations and selectivity. For incremental construction, WoW leverages a WBT to build hierarchical window graphs without dataset preprocessing. To handle varying query workloads, WoW employs query selectivity and combines several window graphs to retrieve in-range nearest neighbors. We prove the time complexity of insertion and query and also suggest optimal parameter settings based on theoretical analysis. Extensive experiments demonstrate the superiority of WoW in terms of indexing and searching efficiency. In future work, we plan to study multi-attribute range filtering. We also want to extend WoW with in-place update and deletion.

# Acknowledgments

This work was supported by the Noncommunicable Chronic Diseases - National Science and Technology Major Project (Grant No.: 2023ZD0503600/2023ZD0503604). We thank Mr. Shuo Shen for his contributions to deploying baseline methods.

# References

- Anas Ait Aomar, Karima Echihabi, Marco Arnaboldi, Ioannis Alagiannis, Damien Hilloulin, and Manal Cherkaoui. 2025. RWalks: Random Walks as Attribute Diffusers for Filtered Vector Search. Proc. ACM Manag. Data 3, 3 (2025), 26 pages.
- [2] Mohammad Alkhalaf, Ping Yu, Mengyang Yin, and Chao Deng. 2024. Applying generative AI with retrieval augmented generation to summarize and extract key clinical information from electronic health records. J. Biomed. Inform. 156 (2024), 104662.
- [3] Laurent Amsaleg, Oussama Chelly, Teddy Furon, Stéphane Girard, Michael E. Houle, Ken-ichi Kawarabayashi, and Michael Nett. 2015. Estimating Local Intrinsic Dimensionality. In SIGKDD. ACM, Sydney, NSW, Australia, 29–38.
- [4] 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. Proc. VLDB Endow. 9, 4 (2015), 288–299.
- [5] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. Inf. Syst. 87 (2020), 101374.
- [6] Martin Aumüller and Matteo Ceccarello. 2021. The role of local dimensionality measures in benchmarking nearest neighbor search. Inf. Syst. 101 (2021), 101807.
- [7] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2025. Graph-Based Vector Search: An Experimental Evaluation of the State-of-the-Art. Proc. ACM Manag. Data 3, 1 (2025), 31 pages.
- [8] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. In *NeurIPS*, Vol. 26. Curran Associates, Inc., Lake Tahoe, NV, USA, 2787–2705.
- [9] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. 2024. Navigating Labels and Vectors: A Unified Approach to Filtered Approximate Nearest Neighbor Search. Proc. ACM Manag. Data 2, 6 (2024), 27 pages.
- [10] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proc. VLDB Endow.* 17, 12 (2024), 3772–3785.
- [11] Jie Chen, Haw-ren Fang, and Yousef Saad. 2009. Fast Approximate kNN Graph Construction for High Dimensional Data via Recursive Lanczos Bisection. J. Mach. Learn. Res. 10 (2009), 1989–2012.
- [12] Patrick Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. 2023. FINGER: Fast Inference for Graph-based Approximate Nearest Neighbor Search. In WWW. ACM, Austin, TX, USA, 3225–3235.

- [13] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: highly-efficient billion-scale approximate nearest neighbor search. In *NeurIPS*. Curran Associates Inc., Red Hook, NY, USA, 14 pages.
- [14] Rongxin Cheng, Yifan Peng, Xingda Wei, Hongrui Xie, Rong Chen, Sijie Shen, and Haibo Chen. 2024. Characterizing the Dilemma of Performance and Index Size in Billion-Scale Vector Search and Breaking It with Second-Tier Memory. arXiv:2405.03267 [cs.DC]
- [15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. Introduction to Algorithms (4th ed.). MIT Press, Cambridge, MA, USA, 481–485
- [16] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. Computational Geometry: Algorithms and Applications (3rd ed.). Springer, Berlin, Germany. 231–237.
- [17] Etienne Dilocker, Bob van Luijt, Byron Voorbach, Mohd Shukri Hasan, Abdel Rodriguez, Dirk Alexander Kulawiak, Marcin Antas, and Parker Duckworth. 2025. Weaviate. https://github.com/weaviate/weaviate.
- [18] Simeon Emanuilov and Aleksandar Dimov. 2024. Billion-Scale Similarity Search Using a Hybrid Indexing Approach with Advanced Filtering. Cybern. Inf. Technol. 24, 4 (2024), 45–58.
- [19] Joshua Engels, Benjamin Landrum, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2024. Approximate Nearest Neighbor Search with Window Filters. In ICML. JMLR.org, Vienna, Austria, 12469–12490.
- [20] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. IEEE Trans. Pattern Anal. Mach. Intell. 44, 8 (2022), 4139–4150.
- [21] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. Proc. VLDB Endow. 12, 5 (2019), 461–474.
- [22] Jianyang Gao and Cheng Long. 2023. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. Proc. ACM Manag. Data 1, 2 (2023), 27 pages.
- [23] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. Proc. ACM Manag. Data 2, 3 (2024), 27 pages.
- [24] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Ji-awei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997 [cs.CL]
- [25] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. IEEE Trans. Pattern Anal. Mach. Intell. 36, 4 (2014), 744-755.
- [26] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In WWW. ACM, Austin, TX, USA, 3406–3416.
- [27] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. 2024. SymphonyQG: Towards Symphonious Integration of Quantization and Graph for Approximate Nearest Neighbor Search. arXiv:2411.12229 [cs.DB]
- [28] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: a cloud native vector database management system. Proc. VLDB Endow. 15, 12 (2022), 3548–3561.
- [29] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. Proc. VLDB Endow. 15, 4 (2021), 752–765.
- [30] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In STOC. ACM, Dallas, TX, USA, 604–613.
- [31] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. IEEE Trans. Pattern Anal. Mach. Intell. 33, 1 (2011), 117–128.
- [32] Mengxu Jiang, Zhi Yang, Fangyuan Zhang, Guanhao Hou, Jieming Shi, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter. Proc. ACM Manag. Data 3, 3 (2025), 26 pages.
- [33] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. IEEE Trans. Big Data 7, 3 (2021), 535–547.
- [34] Yannis Kalantidis and Yannis Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In CVPR. IEEE, Columbus, OH, USA, 2329–2336.
- [35] Yifan Lei, Qiang Huang, Mohan Kankanhalli, and Anthony K. H. Tung. 2020. Locality-Sensitive Hashing Scheme based on Longest Circular Co-Substring. In SIGMOD. ACM, Portland, OR, USA, 2589–2599.
- [36] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. 2018. The Design and Implementation of a Real Time Visual Search System on JD E-commerce Platform. In Middleware. ACM, Rennes, France, 9–16.
- [37] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional

- Data Experiments, Analyses, and Improvement. *IEEE Trans. Knowl. Data Eng.* 32, 8 (2020), 1475–1488.
- [38] Anqi Liang, Pengcheng Zhang, Bin Yao, Zhongpu Chen, Yitong Song, and Guangxu Cheng. 2025. UNIFY: Unified Index for Range Filtered Approximate Nearest Neighbors Search. Proc. VLDB Endow. 18, 4 (2025), 1118–1130.
- [39] Kejing Lu, Chuan Xiao, and Yoshiharu Ishikawa. 2024. Probabilistic routing for graph-based approximate nearest neighbor search. In *ICML*. JMLR.org, Vienna, Austria, 19 pages.
- [40] Rui Ma, Kai Zhang, Zhenying He, Yinan Jing, X. Sean Wang, and Zhenqiang Chen. 2025. CHASE: A Native Relational Database for Hybrid Queries on Structured and Unstructured Data. arXiv:2501.05006 [cs.DB]
- [41] Ruiyao Ma, Yifan Zhu, Baihua Zheng, Lu Chen, Congcong Ge, and Yunjun Gao. 2025. GTI: Graph-Based Tree Index with Logarithm Updates for Nearest Neighbor Search in High-Dimensional Spaces. Proc. VLDB Endow. 18, 4 (2025), 986–999.
- [42] 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.
- [43] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. IEEE Trans. Pattern Anal. Mach. Intell. 42, 4 (2020), 824–836.
- [44] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In PPoPP. ACM, Edinburgh, UK, 270–285.
- [45] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F. Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-Throughput Vector Similarity Search in Knowledge Graphs. Proc. ACM Manag. Data 1, 2 (2023), 25 pages.
- [46] Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. 2023. MTEB: Massive Text Embedding Benchmark. In EACL. ACM, Dubrovnik, Croatia, 2014–2037
- [47] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 11 (2014), 2227–2240.
- [48] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (2023), 1520–1533.
- [49] Jürg Nievergelt and Edward Martin Reingold. 1973. Binary Search Trees of Bounded Balance. SIAM J. Comput. 2, 1 (1973), 33–43.
- [50] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of Vector Database Management Systems. VLDB J. 33 (2024), 1591–1615.
- [51] 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), 27 pages.
- [52] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. Proc. ACM Manag. Data 1, 1 (2023), 27 pages.
- [53] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In EMNLP. ACM, Doha, Qatar, 1532–1542.
- [54] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. arXiv:2103.00020 [cs.CV]
- [55] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. 1995. Nearest neighbor queries. In SIGMOD. ACM, San Jose, CA, USA, 71–79.
- [56] Deborah L. McGuinness Sabbir M. Rashid. 2025. Designing and Evaluating an Ensemble Reasoning-based Clinical Decision Support System. *Data Intelligence* 7, 1 (2025), 1–39.
- [57] 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]
- [58] Yitong Song, Bin Yao, Zhida Chen, Xin Yang, Jiong Xie, Feifei Li, and Mengshi Chen. 2025. Efficient top-k spatial-range-constrained approximate nearest neighbor search on geo-tagged high-dimensional vectors. *The VLDB Journal* 34, 1 (2025), 21 pages.
- [59] 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. Curran Associates, Inc., Vancouver, Canada, 13766–13776.
- [60] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and Efficiency in High Dimensional Nearest Neighbor Search. In SIGMOD. ACM, Providence, RI, USA, 563–576.
- [61] Yao Tian, Xi Zhao, and Xiaofang Zhou. 2024. DB-LSH 2.0: Locality-Sensitive Hashing With Query-Based Dynamic Bucketing. IEEE Trans. Knowl. Data Eng. 36, 3 (2024), 1000–1015.

- [62] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. 2015. Learning to Hash for Indexing Big Data — A Survey. Proc. IEEE 104, 1 (2015), 34–57.
- [63] 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, Xi'an, China, 2614–2627.
- [64] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2023. An Efficient and Robust Framework for Approximate Nearest Neighbor Search with Attribute Constraint. In NeurIPS, Vol. 36. Curran Associates, Inc., New Orleans, LA, USA, 15738–15751.
- [65] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. Proc. VLDB Endow. 14, 11 (2021), 1964–1978.
- [66] Xin Wang, Zirui Chen, Haofen Wang, Leong Hou U, Zhao Li, and Wenbin Guo. 2025. Large language model enhanced knowledge representation learning: A survey. Data Science and Engineering 10, 3 (2025), 315–338.
- [67] Zeyu Wang, Qitong Wang, Xiaoxing Cheng, Peng Wang, Themis Palpanas, and Wei Wang. 2025. Steiner-Hardness: A Query Hardness Measure for Graph-Based ANN Indexes. Proc. VLDB Endow. 17, 13 (2025), 4668–4682.
- [68] Ziting Wang, Haitao Yuan, Wei Dong, Gao Cong, and Feifei Li. 2024. CORAG: A Cost-Constrained Retrieval Optimization System for Retrieval-Augmented Generation. arXiv:2411.00744 [cs.DB]
- [69] 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. Proc. VLDB Endow. 13, 12 (2020), 3152–3165.
- [70] Jiuqi Wei, Botao Peng, Xiaodong Lee, and Themis Palpanas. 2024. DET-LSH: A Locality-Sensitive Hashing Scheme with Dynamic Encoding Tree for Approximate Nearest Neighbor Search. Proc. VLDB Endow. 17, 9 (2024), 2241–2254.
- [71] 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. ACM, Atlanta, GA, USA, 4580–4584.
- [72] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. Proc. ACM Manag. Data 2, 6 (2024), 26 pages.
- [73] Artem Babenko Yandex and Victor Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In CVPR. IEEE, Las Vegas, NV, USA, 2055– 2063
- [74] Shuo Yang, Jiadong Xie, Yingfan Liu, Jeffrey Xu Yu, Xiyue Gao, Qianru Wang, Yanguo Peng, and Jiangtao Cui. 2024. Revisiting the Index Construction of Proximity Graph-Based Approximate Nearest Neighbor Search. arXiv:2410.01231 [cs.DB]
- [75] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In SIGMOD. ACM, Portland, OR, USA, 2241–2253.
- [76] Ziqi Yin, Jianyang Gao, Pasquale Balsebre, Gao Cong, and Cheng Long. 2025. DEG: Efficient Hybrid Vector Search Using the Dynamic Edge Navigation Graph. Proc. ACM Manag. Data 3, 1 (2025), 28 pages.
- [77] Song Yu, Shengyuan Lin, Shufeng Gong, Yongqing Xie, Ruicheng Liu, Yijie Zhou, Ji Sun, Yanfeng Zhang, Guoliang Li, and Ge Yu. 2025. A Topology-Aware Localized Update Strategy for Graph-Based ANN Index. arXiv:2503.00402 [cs.DB]
- [78] Qiang Yue, Xiaoliang Xu, Yuxiang Wang, Yikun Tao, and Xuliyuan Luo. 2024. Routing-Guided Learned Product Quantization for Graph-Based Approximate Nearest Neighbor Search. In ICDE. IEEE, Utrecht, Netherlands, 4870–4883.
- [79] Fangyuan Zhang, Mengxu Jiang, Guanhao Hou, Jieming Shi, Hua Fan, Wenchao Zhou, Feifei Li, and Sibo Wang. 2025. Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search. Proc. ACM Manag. Data 3, 3 (2025), 26 pages.
- [80] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In OSDI. USENIX Association, Boston, MA, USA, 377–395.
- [81] Chaoji Zuo and Dong Deng. 2023. ARKGraph: All-Range Approximate K-Nearest-Neighbor Graph. Proc. VLDB Endow. 16, 10 (2023), 2645–2658.
- [82] 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), 26 pages.

#### **Window Calculation** Α

Algorithm 4 illustrates the window calculation procedure used in Lines 6 and 15 of Algorithm 1, where a window  $W_a^l$  with length  $2o^l$ halved by *a* is requested.

The algorithm first calculates the left window boundary  $w_{\min}$ . For clarity, trees at the children of leaves and the parent of the root are defined as empty. Two variables are defined in Line 1: c records the currently under-processed node during tree traversal, and r works as a budget to record the number of values that should be checked and is greater than the left boundary. After locating at the node with value a (Line 1), it has to decide which branch to traverse using the size of its left subtree,  $|\mathcal{T}_{left}|$ . Remember that the tree size rooted at each WBT node is recorded by the node itself. If  $r \leq |\mathcal{T}_{left}|$ (Line 2), the answer is in the left subtree, and we set  $w_{min}$  to the r-th closest value to a. For example, in an ordered array [1, 2, 3, 4, 5], the second closest value to 5 is 3 and the forth is 1. Otherwise, the boundary is less than the smallest value in  $\mathcal{T}_{left}$ , and we need "climb up" to find the potential subtree that may contain the left boundary value in Lines 5-15. In Lines 9-13, we have "climbed" from the right child, which means the left subtree of the parent may contain the answer. If the budget is only one (Line 11), the value of the parent is what we want, otherwise we get the (r-1)-th closest value to a in the left subtree (Line 12). If the left subtree cannot cover the budget (Line 13,  $|p.\mathcal{T}_{left}| + 1 < r$ ), we reduce the budget by  $|p.\mathcal{T}_{left}| + 1$  and keep looking for the potential tree node. In Line 14, we have "climbed" from the left child, and it means that the value of the current node is smaller than the expected left window boundary, thus we keep climbing. If we have checked all ancestors

```
Algorithm 4: GetWindow
   Input: a: attribute value; l: window graph layer
   Output: W_a^l: window of attribute value a in layer l
   Hyperparameter: o: window boosting base
1 c \leftarrow \text{WBT} node with value a, r \leftarrow o^l;
2 if r \leq |c.\mathcal{T}_{left}| then
    w_{\min} \leftarrow \text{the } r\text{-th closest value to } a \text{ in } c.\mathcal{T}_{left};
4 else
         r \leftarrow r - |c.\mathcal{T}_{left}|;
5
         while c.\mathcal{T} \neq \emptyset do
 6
              p \leftarrow \text{parent of } c;
 7
              if p.\mathcal{T} = \emptyset then w_{\min} \leftarrow \min value in p.\mathcal{T};
 8
               else if c is the root of p.\mathcal{T}_{right} then
 9
                    if |p.\mathcal{T}_{left}| + 1 \ge r then
10
                          if r = 1 then w_{\min} \leftarrow \text{value of } p;
11
                          else w_{\min} \leftarrow \text{the } (r-1)\text{-th closest value to}
12
                         a in p.\mathcal{T}_{left};
                    else r \leftarrow r - (|p.\mathcal{T}_{left}| + 1);
13
               else c \leftarrow p;
14
               if w_{\min} is determined then break;
w_{max} is calculated by a dual version of the above procedure:
```

replacing  $\mathcal{T}_{left} \leftrightarrow \mathcal{T}_{right}$ , min  $\rightarrow$  max,  $w_{min} \rightarrow w_{max}$ ; 17 **return**  $W_a^l = [w_{\min}, w_{\max}];$ 

including the tree root but the window boundary cannot be found (Line 8),  $w_{\min}$  is set to the dataset boundary, i.e. the minimum value.

After the determination of  $w_{\min}$ ,  $w_{\max}$  can be found with a dual version of the above procedure, which can be generated by replacing  $\mathcal{T}_{left} \leftrightarrow \mathcal{T}_{right}$ , min  $\rightarrow$  max,  $w_{min} \rightarrow w_{max}$ . For example, if we have checked all ancestors in Line 8,  $w_{\text{max}}$  should be set to the maximum (replacing minimum) value in  $p.\mathcal{T}$ . The single-branch tree traversal (i.e. only one node is visited on each level of the tree on the search path) is conducted at most three times for one boundary: locating value a in Line 1; "climbing" towards root for the potential ancestor; and finding the closest value to a in the potential subtree.

The time complexity of Algorithm 4 scales to  $O(\log n)$ , where n denotes the number of tree nodes.

# Range Filter Selectivity Calculation

Algorithm 5 shows how to count the size of the filtered dataset n' used in Line 1 of Algorithm 3, which reflects the selectivity of range filter *R* in Definition 3. The algorithm first calculates the true ranges  $[x_u, y_l]$ , where the boundaries exist in the dataset and are covered by R. Then, it calculates their ordered ranks and uses the subtraction as the final cardinality.

The rank calculation procedure starts from the root of WBT and traverses to the node with the target value a. If the target value is less than the value of the visited node, we jump to the left child. If the target is greater, which means that the rank is also greater, we increase rank by  $|c.\mathcal{T}_{left}| + 1$  and jump to the right child. Otherwise, the node of the target value is found. After increasing rank by the size of its left subtree where the values are all less than a, we return the accumulated rank as the result. The algorithm requires four rounds of single-branch tree traversals. Thus, the time complexity is also logarithmic. In the real-world implementation, the complexity can be further improved by combining the boundary determination in Lines 1-2 and the rank calculation in Line 3 together to obtain two round of traversals.

# Algorithm 5: FilteredSetCardinality

```
Input: R = [x, y]: range filter
   Output: n': cardinality of the filtered dataset
1 x_u ← upper bound of x;
y_l ← lower bound of y;
i \leftarrow \text{GetRank}(x_u), j \leftarrow \text{GetRank}(y_l);
4 return n' = i - i + 1;
5 procedure GetRank(a)
         c \leftarrow \text{root of WBT}, rank \leftarrow 0;
         while c.\mathcal{T} \neq \emptyset do
              if a < value of c then c \leftarrow \text{root of } c.\mathcal{T}_{left};
              else if a > value \ of c \ then
10
                    rank \leftarrow rank + |c.\mathcal{T}_{left}| + 1;
                    c \leftarrow \text{root of } c.\mathcal{T}_{right};
11
12
                                               \triangleright a = \text{value of } c, \text{ target found}.
                    rank \leftarrow rank + |c.\mathcal{T}_{left}|;
13
                    break;
14
         return rank;
```

#### C Proof of Theorem 3.2

Theorem 3.2. In the landing layer  $l_d$ , the expected fraction  $f_R$  of in-range neighbors at a single hop on the path is bounded by

$$f_{R} = \begin{cases} \left(\frac{1}{\sqrt{o}}, \frac{1}{2}\right) & (a) \ l \in (l'-1, l'-\frac{1}{2}), o > 4, \\ \left[\frac{\sqrt{2}}{2} - \frac{1}{4o^{l+1}}, \frac{3}{4} - \frac{1}{4o^{l+1}}\right) & (b) \ l \in (l'-1, l'-\frac{1}{2}), o \leq 4, \\ \left[\frac{3}{4} - \frac{1}{4o^{l}}, 1 - \frac{o^{l}+1}{4o^{l+\frac{1}{2}}}\right] & (c) \ l \in [l'-\frac{1}{2}, l'], \end{cases}$$

where n' is the number of in-range vectors for filter R = [x, y],  $l' = \log_o \frac{n'}{2}$ , and  $l = l_h = \lfloor \log_o \frac{n'}{2} \rfloor$ , which is the highest layer defined in Line 2 of Algorithm 3, whose window size is less than R.

PROOF. The landing layer  $l_d$  is determined by comparing  $\frac{2o^l}{n'}$  and  $\frac{n'}{2o^{l+1}}$ . Range filter [x,y] can be rewritten to [x,x+n'-1] by assuming all attribute values are sequential. Moreover, vectors have an equal probability to be selected as neighbors in a certain range. There are two situations:

• **Situation 1.**  $2o^{l+\frac{1}{2}} < n' < 2o^{l+1}$ , then  $\frac{2o^l}{n'} < \frac{n'}{2o^{l+1}}$  and  $l_d = l+1$ , the half window size is  $o^{l+1}$ . For a vector with attribute value x+i in range [x, x+n'-1] where  $i \in [0, n'-1]$ , the window of a vector with attribute value x+i is  $W_{x+i}^{l+1} = [x+i-o^{l+1}+1, x+i+o^{l+1}-1]$ .

<u>Case (a).</u> When  $n' < o^{l+1}$ , the range filter is always covered by the windows. In this case,  $2o^{l+\frac{1}{2}} < o^{l+1}$  and o > 4. The fraction can be calculated by  $f_R = \frac{n'}{2o^{l+1}} \in (\frac{1}{\sqrt{o}}, \frac{1}{2})$ .

<u>Case (b).</u> When  $n' \ge o^{l+1}$ , some windows fail to cover the whole range. If the window left boundary is less than the filter left boundary, i.e.  $x+i-o^{l+1} < x \Rightarrow i \le o^{l+1}-1$  The fraction of in-range vectors is  $f_1 = \frac{(x+i+o^{l+1}-1)-x+1}{2o^{l+1}}$ , and the expectation is

$$\bar{f}_1 = \mathbb{E}_{i \in [0, o^{l+1} - 1]} \left[ \frac{i + o^{l+1}}{2o^{l+1}} \right],$$
(4)

which is  $\bar{f_1}=\frac{1}{4}(3-o^{l+1})$ . On the other hand, if  $i\in[o^{l+1},n'-1]$ , the window right boundary is greater than the filter right boundary. The fraction is  $f_2=\frac{(x+n'-1)-(x+i-o^{l+1}+1)+1}{2o^{l+1}}$ , and

$$\bar{f}_2 = \mathbb{E}_{i \in [o^{l+1}, n'-1]} \left[ \frac{n' + o^{l+1} - 1 - i}{2o^{l+1}} \right], \tag{5}$$

which is  $\bar{f}_2 = \frac{n'+o^{l+1}-1}{4o^{l+1}}$ . Combining two parts,  $f_R = w_1\bar{f}_1 + w_2\bar{f}_2$ , where  $w_1 = \frac{o^{l+1}}{n'}$  and  $w_2 = \frac{n'-o^{l+1}}{n'}$ , the total fraction is

$$f_{R} = \frac{3o^{l+1} - 1}{4o^{l+1}} \cdot \frac{o^{l+1}}{n'} + \frac{n' + o^{l+1} - 1}{4o^{l+1}} \cdot \frac{n' - o^{l+1}}{n'}$$

$$= \frac{o^{l+1}}{2n'} + \frac{n' - 1}{4o^{l+1}} \in \left[\frac{1}{4}\left(2\sqrt{2} - o^{-(l+1)}\right), \frac{1}{4}\left(3 - o^{-(l+1)}\right)\right),$$
(6)

and the lower bound is at  $n' = \sqrt{2}o^{l+1} \in [2o^{l+\frac{1}{2}}, 2o^{l+1})$ , while the upper bound is determined by  $n' = 2o^{l+1}$ .

• **Situation 2.** Case (c).  $2o^l \le n' \le 2o^{l+\frac{1}{2}}$ , then  $\frac{2o^l}{n'} \ge \frac{n'}{2o^{l+1}}$  and  $l_d = l$ , the half window size is  $o^l$ . For a vector with attribute value x+i in range [x,x+n'-1] where  $i \in [0,n'-1]$ , its window in Layer l is  $W^l_{x+i} = [x+i-o^l+1,x+i+o^l-1]$ .

When  $i \le o^l - 1$ , we have  $x + i - o^l + 1 < x$  that the left boundary of the window is less than that of the filter. The fraction of in-range vectors over the window size is  $f_1 = \frac{(x+i+o^l-1)-x+1}{2o^l}$ . The expected fraction of this situation is

$$\bar{f}_1 = \mathbb{E}_{i \in [0, o^l - 1]} \left[ \frac{i + o^l}{2o^l} \right], \tag{7}$$

which is  $\bar{f}_1 = \frac{1}{4}(3 - o^{-l})$ . When  $i \ge n' - o^l$ , we have  $x + n' - 1 < x + i + o^l - 1$  that the right boundary of the filter is less than that of the window. The fraction  $f_2 = \frac{(x + n' - 1) - (x + i - o^l + 1) + 1}{2o^l}$ . We can calculate  $\bar{f}_2 = \frac{1}{4}(3 - o^{-l})$  in the same way as  $\bar{f}_1$ . When  $o^l - 1 < i < n' - o^l$ , the expected fraction is  $\bar{f}_3 = 1$  because all windows of these elements are covered by the filter.

Finally, the total fraction of in-range neighbors is  $w_1\bar{f}_1+w_2\bar{f}_2+w_3\bar{f}_3$  where  $w_1=w_2=\frac{o^l}{n'}$  and  $w_3=\frac{n'-2o^l}{n'}$ .

$$f_R = \frac{2o^l}{n'} \left( \frac{1}{4} \left( 3 - o^{-l} \right) \right) + \frac{n' - 2o^l}{n'} \cdot 1$$

$$= 1 - \frac{o^l + 1}{2n'} \in \left[ \frac{1}{4} \left( 3 - o^{-l} \right), 1 - \frac{o^l + 1}{4o^{l + \frac{1}{2}}} \right].$$
(8)