Data Science and Engineering

https://doi.org/10.1007/s41019-025-00343-5

Article in Press

MiniRFANN: Range-Filtered Approximate Vector Similarity Search on Edge Devices

Received: 29 August 2025

Accepted: 24 December 2025

Published online: 29 April 2026

Cite this article as: Li S., Liu Y., Wang Y. *et al.* MiniRFANN: Range-Filtered Approximate Vector Similarity Search on Edge Devices. *Data Sci. Eng.* (2026). https://doi.org/10.1007/s41019-025-00343-5

Shuyuan Li, Yuheng Liu, Yuxiang Wang, Zimu Zhou, Kaining Zhang & Yongxin Tong

We are providing an unedited version of this manuscript to give early access to its findings. Before final publication, the manuscript will undergo further editing. Please note there may be errors present which affect the content, and all legal disclaimers apply.

If this paper is publishing under a Transparent Peer Review model then Peer Review reports will publish with the final article.

ARTICLE IN PRESS

© The Author(s) 2026. **Open Access** This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

ARTICLE IN PRESS

MiniRFANN: Range-Filtered Approximate Vector Similarity Search on Edge Devices

Anonymous Author(s)

Abstract

Approximate vector similarity search (ANN) is fundamental to applications such as recommendation, multimedia retrieval, and retrieval-augmented generation. In many scenarios, search results must also satisfy constraints on auxiliary attributes such as timestamp or location, giving rise to the range-filtered approximate ***k***-nearest neighbor (RFANN) problem. Existing RFANN approaches, including pre-filtering, post-filtering, and in-filtering, perform well on servers but are poorly suited to edge devices due to limited memory and high I/O costs. We present MiniRFANN, a disk-resident RFANN scheme for edge environments. MiniRFANN builds dual B+trees over product-quantized vectors, keyed respectively by attributes and cluster identifiers, enabling range scans with early pruning under tight memory budgets. An adaptive index selection strategy models range filtering and similarity pruning as hard and soft selectivity, choosing the index that minimizes I/O for each query. A Z-order data layout over attribute and cluster identifier space further improves locality, reducing random access during exact re-ranking. Experiments on standard benchmarks show that MiniRFANN achieves high recall with substantially lower memory usage and latency compared with state-of-the-art RFANN methods adapted for edge deployment.

Keywords: vector similarity search, range filtering, product quantization, edge devices

ARTICLE IN PRESS

1 Introduction

Approximate vector similarity search (ANN) [1] is a fundamental building block in modern data-intensive applications, enabling fast retrieval of high-dimensional data in recommender systems, multimedia search, question answering, and retrieval-augmented generation (RAG) [2–4]. Given a query vector, ANN aims to efficiently find the top-k most similar vectors from a large dataset, trading accuracy for sublinear query time while maintaining high recall. State-of-the-art ANN solutions [5, 6] achieve impressive performance on *server* hardware (*i.e., in-memory* setting) by combining powerful index structures (*e.g.,* graphs [5–7], trees [8, 9]) with compression techniques such as product quantization (PQ) [10, 11].

ARTICLE IN PRESS

In many practical applications, similarity alone is insufficient. Results must also satisfy constraints on auxiliary *attributes* such as timestamp, location, or price. This leads to the range-filtered approximate k-nearest neighbor (RFANN) problem [12–17]: retrieving the top-k vectors that are both close to the query vector in feature space and whose attribute values fall within a specified range. RFANN enables expressive queries like "find images visually similar to this one, taken in the past month" or "retrieve sensor readings matching a target pattern, from a specific location range."

Existing RFANN methods extend ANN via three main filtering strategies.

- Pre-filtering [12, 18]: Apply attribute constraints first, then perform ANN search on the reduced dataset. It works well for narrow ranges, but when ranges are wide, it must scan and load many scattered vectors, incurring high random I/O cost on disk.
- Post-filtering [10, 19]: Perform ANN search over the full dataset, then discard results that violate attribute bounds. It is a common baseline to first retrieve candidates and then filter them by attributes. However, this approach suffers from redundant I/O when handling narrow-range queries.
- In-filtering [13, 15, 20]: Integrate range checks into index traversal for early pruning. It is highly efficient on in-memory indices but becomes unsuitable for disk-resident search, as it requires the entire index in memory.

While these designs work well on servers with abundant memory, a growing number of applications must run RFANN on *edge devices* with tight resource budgets, *i.e.,* often sub-GB memory [21–24]. On such hardware, in-filtering exhausts memory, and pre/post-filtering suffers severe I/O amplification, both of which significantly degrade query latency.

In this paper, we present MiniRFANN, a disk-friendly RFANN framework designed for edge deployment. MiniRFANN addresses the limitations of existing RFANN methods through a *hybrid filtering* and *adaptive indexing* design that is both memory-efficient and I/O-aware. Our design is guided by three key insights: *(i)* range filtering and similarity pruning can be jointly modeled as *hard* and *soft* selectivity; *(ii)* the relative pruning power of these filters varies across queries and can be estimated at runtime; and *(iii)* disk I/O cost can be greatly reduced by co-designing the index structure with the on-disk data layout.

Our main technical novelties are as follows.

- Dual B+Tree Indexing. We assign semi-ordered cluster IDs via hierarchical clustering, compress vectors using product quantization (PQ), and build two B+ trees over PQ codes, *i.e.,* one keyed by attributes, the other by cluster IDs, thereby enabling efficient range scans with early pruning while keeping the main index structure disk-resident.
- Adaptive Index Selection. We formalize attribute selectivity (deterministic) and distance selectivity (estimated from recall requirements) and adaptively choose the index that yields fewer I/Os for the given query, ensuring robust performance across diverse range widths.

ARTICLE IN PRESS

- Data Layout Optimization. We store raw vectors on disk following a Z-order curve [25, 26] over (attribute, cluster ID) space, ensuring that relevant vectors are spatially correlated in the disk and allowing batch loading to reduce I/Os during re-ranking.

We implement MiniRFANN and evaluate it against state-of-the-art RFANN baselines [12, 19, 27] on three benchmarks [10, 28]. Results show that MiniRFANN delivers high recall while significantly reducing memory usage and query latency, achieving 10.02%-87.13% lower latency and 5.88%-66.67% lower memory overhead than state-of-the-art RFANN baselines, with comparable index construction time.

2 Problem Definition

Definition 1 (**Attribute Vector Dataset**) Let $O = \{o_1, o_2, \ldots, o_N\}$ be a set of N objects. Each object o_i is represented as a pair $(\boldsymbol{v}_i, a_i)$, where $\boldsymbol{v}_i = (v_i^{(1)}, v_i^{(2)}, \ldots, v_i^{(d)}) \in \mathbb{R}^d$ is a d-dimensional feature vector in Euclidean space, and $a_i \in \mathbb{R}$ is a scalar attribute (*e.g.,* timestamp or location).

Consider an on-device media retrieval app that maintains a continuously updated archive of high-dimensional embeddings (such as image or text embeddings), each associated with attributes like timestamps or location tags. Users issue queries with attribute constraints, such as "photos from the last seven days near campus," restricting search results to certain attribute ranges before retrieving relevant embeddings. On typical edge devices (*e.g.,* smartphones with 2-8GB RAM), these datasets can reach multi-GB sizes, significantly exceeding available application memory. Consequently, the entire dataset resides primarily on device disk storage, with only a small fraction cached in RAM.

Definition 2 (**Range-Filtered k-Nearest Neighbor Query (RFNN)**) Given a query $Q = \{\boldsymbol{v}_Q, (a_L, a_R)\}$ over the dataset O, the *range-filtered kNN query* aims to retrieve a set of k objects $A^* = \{o_1^*, o_2^*, \ldots, o_k^*\} \subseteq O$ such that:

- Each $o_i^* = (\boldsymbol{v}_i^*, a_i^*,)$ satisfies $a_L \leq a_i^* \leq a_R$;
- The vectors $\{\boldsymbol{v}_i^*\}$ are the k closest to $\boldsymbol{v}_Q$ measured in Euclidean distance among all objects in O whose attributes are in $[a_L, a_R]$.

We refer to A^* as the *exact* RFNN answer set.

Unlike low-dimensional scenarios, exact RFNN computation over high-dimensional vectors incurs excessive computation. Thus, approximate approaches are often applied to provide practical query performance. We adopt recall-based approximation, widely used in practice, to quantify the accuracy-performance trade-off.

Definition 3 (**Range-Filtered Approximate Nearest Neighbor Query (RFANN)**) Given a query $Q = \{\boldsymbol{v}_Q, (a_L, a_R)\}$, integer k, a user-defined recall threshold $\tau \in (0, 1]$, the *range-filtered approximate kNN query* returns a set A of k objects satisfying:

- Each $o_i = (\boldsymbol{v}_i, a_i) \in A$ satisfies $a_L \leq a_i \leq a_R$;

ARTICLE IN PRESS

ARTICLE IN PRESS

Table 1: Major notations.

Symbol	Description
O	Attribute vector dataset
N	$\|O\|$, the number of objects in O
o_i	Basic objects in attribute vector dataset
$\boldsymbol{v}_i$	The d-dimensional vector contained in the basic objects
a_i	Numerical attribute contained in the basic objects
Q	RFANN query which contains a vector and a attribute range
$\boldsymbol{v}_Q$	Query vector contained in Q
(a_L, a_R)	Attribute range contained in Q
$c(\boldsymbol{v}_i)$	Cluster ID to which vector $\boldsymbol{v}_i$ belongs
$ID_{\boldsymbol{v}_i}$	Index ID of vector $\boldsymbol{v}_i$
$PQ(\boldsymbol{v}_i)$	Code of vector $\boldsymbol{v}_i$ after product quantization
C	Codebook of PQ codes $\{PQ(\boldsymbol{v}_i)\}$
$\mathcal{D}_{o_i}$	Insertion entry of o_i during index construction
B	The maximum number of entries each node can hold in the B+tree index
r	The number of candidates for exact re-ranking

- Let A^* be the exact RFNN answer set. Then the range-filtered recall@k satisfies Recall@$k(Q) \triangleq \frac{|A \cap A^*|}{k} \geq \tau$.

Our goal is to design RFANN query *indexing* and *processing* methods tailored for edge devices. Specifically, we aim to:

- Achieve high recall rates (accuracy) that meet user-specified thresholds.
- Minimize query latency, primarily governed by disk-based I/O operations due to limited available RAM.

Tab. 1 lists the major notations of this paper.

3 Method

3.1 MiniRFANN Overview

Limitations of Prior Arts. Existing methods for RFANN queries follow one of three strategies: *pre-filtering*, *post-filtering*, or *in-filtering*. However, each suffers from limitations in memory-constrained edge environments. In-filtering methods [13–17, 20] load both index structures and raw vectors entirely into memory, making them infeasible on devices with limited RAM. Pre-filtering [12, 18] applies attribute constraints before similarity search, reducing memory usage but incurring excessive disk I/O when attribute ranges are wide. Post-filtering [10, 19] retrieves candidate neighbors using disk-optimized vector indices and filters by attributes afterward, but often wastes computation on irrelevant candidates when attribute ranges are narrow.

Design Rationales. We propose MiniRFANN, a hybrid RFANN framework that *dynamically* balances attribute- and distance-based filtering for memory-constrained edge devices. The key insight is to coordinate two complementary operations: *(i)*

ARTICLE IN PRESS

ARTICLE IN PRESS

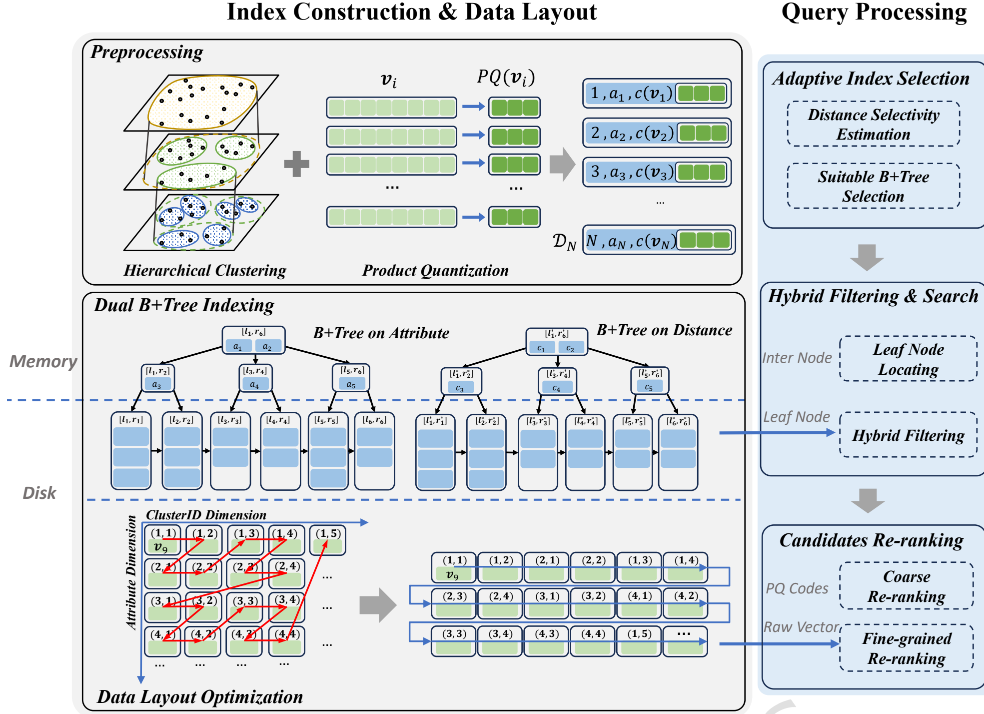


Fig. 1: MiniRFANN overview.

attribute-based hard filtering, which eliminates irrelevant objects early using attribute constraints, and *(ii) distance-based soft filtering*, which prunes candidates distant to query vector using quantization-based similarity approximations and hierarchical clustering. Furthermore, since disk I/O dominates latency on edge devices, MiniRFANN optimizes on-disk vector distribution to enhance spatial locality.

MiniRFANN Workflow. MiniRFANN consists of an offline indexing phase and an online query phase (see Fig. 1).

- **Indexing Phase (Sec. 3.2).** We first apply *hierarchical clustering* to partition the attribute vector dataset into semantically meaningful clusters, assigning each vector a *cluster ID* that reflects its similarity locality. Next, we perform *product quantization (PQ)* [10, 11, 15, 17] to encode each vector into a compact representation for distance estimation. Over the PQ codes, we build two *B+tree indices*, one ordered by *attribute* values and one by *cluster IDs*, to support flexible filtering during query time. Finally, we reorganize raw vectors on disk by a *Z-order space-filling curve* across both attribute and cluster dimensions, ensuring spatial locality for efficient block-level I/O.
- **Query Phase (Sec. 3.3).** Given a query with an attribute range and vector, MiniRFANN adaptively selects the appropriate B+tree index based on the width of the attribute interval. It then traverses the index while jointly applying attribute

ARTICLE IN PRESS

ARTICLE IN PRESS

constraints and PQ-based distance pruning to efficiently narrow the candidate set. Coarse ranking is performed using PQ codes, and only the top-ranked candidates are loaded from disk for exact distance computation and final result selection.

We elaborate on the two phases below.

3.2 Index Construction and Data Layout Optimization

This section describes the indexing phase of MiniRFANN, which prepares the data structures and on-disk layout for efficient RFANN query processing. We first apply hierarchical clustering to the raw vectors to assign each a semi-ordered cluster ID that preserves similarity locality. In parallel, we perform product quantization (PQ) to compactly encode the vectors for efficient approximate distance computation (Sec. 3.2.1). Based on the resulting PQ codes, we construct dual B+tree indices, one indexed by attribute values and the other by cluster IDs, to support adaptive query execution (Sec. 3.2.2). Finally, we reorganize the physical layout of raw vectors on disk to improve spatial locality and reduce I/O overhead during exact re-ranking (Sec. 3.2.3).

3.2.1 Preprocessing

In this section, we preprocess the attribute vector dataset O to prepare for index construction. Specifically, we apply hierarchical clustering to assign a semi-ordered cluster ID $c(\boldsymbol{v}_i)$ to each vector and perform product quantization (PQ) to generate compact codes $PQ(\boldsymbol{v}_i)$ for efficient similarity search. The result of preprocessing is a structured record for each object in the form $\{\boldsymbol{v}_i, a_i, c(\boldsymbol{v}_i), PQ(\boldsymbol{v}_i)\}$.

Hierarchical Clustering. In RFANN queries, objects with similar attribute values or vectors are often accessed together [13, 14, 20]. Co-locating such objects during index construction can improve disk access locality. While attributes can be naturally sorted to preserve locality, no such ordering exists for high-dimensional vectors due to the lack of a total ordering in Euclidean space. To approximate this, we perform hierarchical clustering to organize vectors by similarity and assign them semi-ordered cluster IDs.

We use recursive K-means clustering [29] to build a hierarchical tree over the vector set $\{\boldsymbol{v}_i\}$. At each level, clusters are partitioned into K subclusters until a desired granularity is reached. Intuitively, vectors within the same subtree share greater similarity. Once the hierarchy is constructed, we traverse it in depth-first order to assign consecutive cluster IDs to leaf nodes. Each vector inherits the ID of its corresponding leaf, which is later used to guide coarse distance filtering and layout optimization.

Product Quantization. To enable memory-efficient similarity search, we apply product quantization (PQ) [10, 11, 15, 17] to compress each high-dimensional vector into a compact code. PQ allows approximate distance computation with reduced memory and storage requirements. The process consists of three steps:

- *Subspace Partitioning.* Each vector $\boldsymbol{v}_i$ is evenly divided into M disjoint subvectors: $\boldsymbol{v}_i = [\boldsymbol{s}_{i,1} \parallel \boldsymbol{s}_{i,2} \parallel \ldots \parallel \boldsymbol{s}_{i,M}], \quad \boldsymbol{s}_{i,m} \in \mathbb{R}^{d/M}$, where $\parallel$ denotes vector concatenation.

ARTICLE IN PRESS

ARTICLE IN PRESS

- *Codebook Training.* For each subspace $m \in [1, M]$, a codebook $C_m = \{\boldsymbol{c}_{m,1}, \ldots, \boldsymbol{c}_{m,Z}\}$ is trained via Z-means clustering over $\{\boldsymbol{s}_{i,m}\}_{i=1}^{N}$, where Z is the number of centroids. Each centroid $\boldsymbol{c}_{m,z}$ represents a subvector cluster.
- *Vector Encoding.* Each subvector $\boldsymbol{s}_{i,m}$ is quantized to the index of its nearest centroid in C_m: $q_{i,m} = \arg\min_{z\in[1,Z]} \|\boldsymbol{s}_{i,m} - \boldsymbol{c}_{m,z}\|_2$. The final PQ code for vector $\boldsymbol{v}_i$ is $PQ(\boldsymbol{v}_i) = (q_{i,1}, \ldots, q_{i,M}) \in [Z]^M$.

Given two PQ-encoded vectors $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$, their approximate Euclidean distance is computed as $\sqrt{\sum_{m=1}^{M} \left\|\boldsymbol{c}_{m,q_{i,m}} - \boldsymbol{c}_{m,q_{j,m}}\right\|_2^2}$, which can be efficiently calculated using M precomputed lookup tables of centroid distances. Product quantization offers a practical trade-off between memory efficiency and retrieval accuracy, making it suited for approximate nearest neighbor search of high-dimension data on memory-constrained edge devices. For low-dimensional data, the clustering and quantization steps can be simplified, while the following structure remains effective for hybrid filtering.

3.2.2 Dual B+Tree Indexing

To support efficient hybrid filtering under memory and I/O constraints, MiniRFANN constructs two complementary B+trees indices over the encoded vector dataset. After preprocessing (Sec. 3.2.1), each object is represented as a tuple $\{\boldsymbol{v}_i, a_i, c(\boldsymbol{v}_i), PQ(\boldsymbol{v}_i)\}$, where $\boldsymbol{v}_i$ is the raw vector, a_i is the associated attribute, $c(\boldsymbol{v}_i)$ is the assigned cluster ID, and $PQ(\boldsymbol{v}_i)$ is its PQ code. We build two B+trees: *(i) an attribute B+tree*, indexed by a_i, to support efficient range filtering on query attributes; and *(ii) a cluster B+tree*, indexed by $c(\boldsymbol{v}_i)$, to enable distance-based pruning via cluster-locality.

Tree Structure and Search Semantics. B+ trees are chosen over graph-based indexing (*e.g.,* HNSW [5]) because they are inherently *range-aware*, *disk-friendly*, and *memory-efficient*. Unlike graph-based indices that assume memory residency and optimize for point-wise similarity, B+trees enable efficient range-restricted scans, an essential requirement for attribute-constrained queries on edge devices.

Both trees share the same structure and payload design. Leaf nodes can store up to B_{leaf} records, the maximum that fits into a single disk block, and hold arrays of tuples in the form $\mathcal{D}_{o_i} = \{ID_{\boldsymbol{v}_i}, a_i, c(\boldsymbol{v}_i), PQ(vi)\}$, which are later used for both attribute-based hard filtering and PQ-based soft filtering. Internal nodes, in turn, store at most $Binter$ keys and act as routing tables, each pointing to child nodes.

Each node $\mathcal{N}$ (internal or leaf) maintains a key range $\mathcal{N}.[l, r]$ indicating the minimum and maximum values across its subtree. In the attribute tree, this range reflects attribute values; in the cluster tree, it reflects cluster IDs. Given a query range $[a_L, a_R]$, the tree traverses to locate two boundary leaves $\mathcal{N}_{left}$ and $\mathcal{N}_{right}$ such that $a_L \in \mathcal{N}_{left}.[l, r]$ and $a_R \in \mathcal{N}_{right}.[l, r]$. All leaf nodes in between (excluding the boundaries) are guaranteed to contain only records within the specified range. And we simply require subsequent scanning and validation of records within boundary leaves. This enables efficient, sequential scanning without additional filtering.

Index Construction. MiniRFANN incrementally builds each B+tree by inserting records one at a time. We describe the process for the attribute tree. The cluster tree is constructed identically, using cluster IDs as keys.

ARTICLE IN PRESS

Insertion starts by locating the target leaf node whose key range $[l, r]$ covers the record's attribute (or cluster ID) value, using a top-down traversal that caches the path for potential structural updates. Once the leaf is loaded into memory, the new tuple $\mathcal{D}_{o_i}$ is inserted in key order, and the node's $[l, r]$ range is immediately updated to reflect the revised minimum and maximum key values. If the leaf remains within capacity ($\leq B_{leaf}$ entries), it is written back to disk directly. Otherwise, the node is split at the median key, producing a new leaf node with its own $[l, r]$ range. The separator key is propagated upward to the parent along the cached traversal path. If the parent's key count then exceeds B_{inter}, it is likewise split, and the process of key promotion and range update continues recursively toward the root until the B+tree's balance is restored. Throughout this process, the $[l, r]$ ranges are maintained in a consistent and up-to-date manner, ensuring that both the attribute and cluster trees preserve their range-aware semantics, thereby enabling efficient sequential scans during subsequent queries.

We now present the overall MiniRFANN index construction algorithm (Algorithm 1) which takes an attribute vector dataset O as input and outputs MiniRFANN index $\mathcal{I} = \{\mathcal{T}_{attribute}, \mathcal{T}_{distance}\}$, along with product quantization codebook C.

Algorithm 1 MiniRFANN Index Construction

Require: Attribtue vector dataset O
Ensure: Index $\{\mathcal{T}_{attribute}, \mathcal{T}_{distance}\}$, PQ codebook C
1: **procedure** IndexConstruction(O)
2: $V \leftarrow \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_N\}$
3: $H \leftarrow$ HierarchicalClustering(V)
4: $\{PQ(\boldsymbol{v}_i)\}, C \leftarrow$ ProductQuantization(V)
5: **for** $i \leftarrow 1$ **to** N **do**
6: $c(\boldsymbol{v}_i) \leftarrow$ AssignClusterID($\boldsymbol{v}_i, H$)
7: $\mathcal{D}_{o_i} \leftarrow \{ID_{\boldsymbol{v}_i}, a_i, c(\boldsymbol{v}_i), PQ(\boldsymbol{v}_i)\}$
8: **end for**
9: $\mathcal{T}_{attribute} \leftarrow$ BuildAttrTree($\{\mathcal{D}_{o_i}\}$)
10: $\mathcal{T}_{distance} \leftarrow$ BuildDistTree($\{\mathcal{D}_{o_i}\}$)
11: **return** $\{\mathcal{T}_{attribute}, \mathcal{T}_{distance}\}$, C
12: **end procedure**

ARTICLE IN PRESS

- First, the dataset vectors V are extracted from O (Line 2). Hierarchical clustering is performed on V to partition vectors into semantic clusters, generating cluster structure H (Line 3). Product quantization is applied to V to obtain compact PQ codes for each vector, producing and the PQ codebook C (Line 4).
- For each vector $\boldsymbol{v}_i$, the corresponding cluster ID $c(\boldsymbol{v}_i)$ is assigned according to H, and the record D_{o_i} is constructed (Lines 5–8).
- Using records set $\{\mathcal{D}_{o_i}\}$ of O, two complementary B+ trees are built: the attribute tree $\mathcal{T}_{attribute}$ is indexed by attribute values $\{a_i\}$, and the distance tree $\mathcal{T}_{distance}$ is indexed by cluster IDs $\{c(\boldsymbol{v}_i)\}$ (Lines 9–10).

ARTICLE IN PRESS

- Finally, the constructed MiniRFANN index $\{\mathcal{T}_{attribute}, \mathcal{T}_{distance}\}$ along with the PQ codebook C are returned for query processing(Line 11).

Complexity of Index Construction. We analyze the I/O complexity to construct the B+tree index by incrementally inserting N records, each in the form of $\mathcal{D}_{o_i}$.

Each insertion performs a top-down traversal from the root to the boundary leaf node. In a B+tree with fanout B_{inter} for internal nodes and B_{leaf} entries per leaf, the tree height is $O(\log_{B_{inter}}(\frac{N}{B_{leaf}}))$. Hence, the traversal time per insertion is $O(\log N)$, assuming bounded fanout. Besides, an insertion triggers $O(\log N)$ node splits due to recursive propagation up to the root, yielding a total complexity of $O(N \log N)$ for index construction.

In practice, MiniRFANN keeps internal nodes in memory during index construction which eliminates disk I/Os for traversal and internal splits. With this optimization, the practical I/O cost becomes $O(N)$.

3.2.3 On-Disk Data Layout Optimization

Achieving high recall in RFANN queries requires retrieving the original raw vectors after PQ-based approximate filtering to perform precise distance computations and final re-ranking [10]. However, accessing raw vectors from disk on edge devices can incur significant I/O overhead. Therefore, optimizing the physical layout of raw vector data is critical for reducing overall query latency.

Since vectors with similar attribute values or feature similarity are often accessed together, MiniRFANN reorganizes their on-disk layout to enhance spatial locality. It leverages a Z-order space-filling curve [25, 26] to reorder vectors based on two key dimensions relevant to query access patterns: *(i) Attribute dimension (a_i)*, which governs attribute-based filtering. Vectors with similar attributes tend to be retrieved together in narrow-range queries. *(ii) Cluster dimension ($c(\boldsymbol{v}_i)$)*, which reflects vector similarity. Vectors in the same or nearby clusters are likely candidates after coarse distance-based pruning.

Each vector's 2D coordinate $(a_i, c(\boldsymbol{v}_i))$ is mapped to a 1D Z-order value, which determines its physical position on disk. Due to the neighbor-preserving nature of the Z-order curve, vectors that are close in attribute-cluster space are likely to be stored contiguously. This optimized layout reduces I/O overhead during re-ranking by increasing the density of relevant data per access. When raw vectors are loaded, a single sequential disk read often retrieves multiple useful candidates. This reduces random seeks and boosts the efficiency of re-ranking, thereby lowering query latency without compromising recall.

Updating MiniRFANN. To support vector updates, MiniRFANN employs a two-tier pipeline that combines in-memory buffering with batch synchronization on disk[30]. Newly arriving data are first preprocessed and appended to an in-memory update buffer, which temporarily holds inserted, modified, or deleted records. When the buffer reaches a predefined threshold or during periodic maintenance, MiniRFANN performs batch updates to both the attribute-based and cluster-based B+ trees [30, 31]. In this process, both the internal nodes and the corresponding leaf nodes are updated synchronously. To enable efficient updates to the Z-order layout on disk, the

ARTICLE IN PRESS

ARTICLE IN PRESS

Z-order key space over the attribute–cluster pair $(a_i, c(\boldsymbol{v}_i))$ is divided into fixed-size Z-partitions [32]. During batch synchronization, only the partitions that overlap with buffered entries are loaded, merged with the buffered data, re-sorted according to Z-order, and rewritten back to disk. This localized reorganization ensures that updates remain confined to small contiguous regions, maintaining both the logical–physical alignment of the index and efficient I/O, while avoiding the need for global reconstruction. This design allows MiniRFANN to efficiently handle append-dominant or moderately dynamic workloads while maintaining query accuracy and low latency. Future work will explore concurrent update handling and incremental re-clustering strategies to further enhance scalability under highly dynamic data streams.

3.3 Query Processing

This section details how MiniRFANN efficiently executes RFANN queries upon the indexing structures introduced in Sec. 3.2. Upon receiving a query, MiniRFANN first selects the most suitable B+ tree index based on the width of the attribute range. It then traverses the index while jointly applying attribute constraints and PQ-based distance pruning to efficiently narrow down the candidate set. Coarse ranking is performed using PQ codes, and only the top-ranked candidates are loaded from disk for exact distance computation and final result selection. We present the overall MiniRFANN query processing as Algorithm 2.

Algorithm 2 MiniRFANN Query Processing

Require: $\mathcal{I} = \{\mathcal{T}_{attribute}, \mathcal{T}_{distance}\}$, $Q = \{\boldsymbol{v}_Q, (a_L, a_R)\}$, $L_{candidates}$, r
Ensure: Answer set $\mathcal{R}_{exact} = \{(ID_{\boldsymbol{v}_i}, PQ(\boldsymbol{v}_i))\}$

```
1: procedure QueryProcessing(I, Q, L_candidates, r)
2:     ρ_attribute ← |f_r(O)| / N
3:     ρ_distance ← min{N × ρ_attribute, L_candidates} / (N × ρ_attribute)
4:     if ρ_attribute < ρ_distance then
5:         T_sel ← T_attribute
6:     else
7:         T_sel ← T_distance
8:     end if
9:     C_Q ← SelectClusters(v_Q, ρ_distance)
10:    R ← RangeSearch(T_sel, C_Q, [a_L, a_R])
11:    R_coarse ← CoarseRank(v_Q, R)
12:    R_exact ← ExactRank(v_Q, R_coarse, r)
13:    return R_exact
14: end procedure
```

Adaptive Index Selection. A key novelty of MiniRFANN is its ability to adaptively choose between the two B+ tree indices based on *selectivity*, *i.e.*, the fraction of dataset objects expected to satisfy a given filtering condition.

ARTICLE IN PRESS

ARTICLE IN PRESS

For attribute filtering, let f_r be the filtering function and N the dataset size. The *attribute selectivity* is estimated as:

$$\rho_{attribute} = \frac{|f_r(O)|}{N} \tag{1}$$

Given an RFANN query, the attribute range is specified, allowing us to directly evaluate the filtering function f_r over the dataset O. Since both the filtering condition and dataset size N are known at query time, $\rho_{attribute}$ can be computed at query time.

In contrast, the *distance selectivity* $\rho_{distance}$, *i.e.,* the fraction of data accessed via cluster-based filtering, cannot be directly measured. Instead, it is estimated to ensure that the combined effect of attribute and distance filtering yields enough candidates to meet the target recall rate.

Let $L_{candidates}$ be the minimum number of valid candidates (after both filters) needed for the target recall, which is empirically configured beforehand. Assuming uniform data distribution, the expected number of candidates is:

$$L_{candidates} = N \times \rho_{attribute} \times \rho_{distance} \tag{2}$$

Then we can estimate the distance selectivity as

$$\rho_{distance} = \frac{\min\{N \times \rho_{attribute}, L_{candidates}\}}{N \times \rho_{attribute}} \tag{3}$$

Finally, MiniRFANN compares $\rho_{attribute}$ and $\rho_{distance}$ to select the index with greater pruning power. If $\rho_{attribute} < \rho_{distance}$, it uses the attribute-based B+ tree; otherwise, it uses the cluster-based B+ tree.

Hybrid Filtering and Search. After MiniRFANN selects the appropriate B+ tree index, it first determines the set of candidate cluster IDs by computing distances between the query vector and all cluster centroids, thereby identifying the *cluster ID range* to be probed. The search then proceeds as a range query on the chosen B+ tree.

When using the attribute-based B+ tree, the query operates over a specified attribute interval. Internal nodes guide the traversal to the leaf node containing the starting key of the attribute range, from which consecutive leaf nodes are batch-loaded and scanned. If a leaf node's key range lies completely outside the attribute interval, the search terminates early; if fully contained, the attribute check can be skipped and only the cluster ID condition is applied; partial overlaps require record-wise evaluation of both constraints.

Conversely, when the cluster-based B+ tree is used, the search is conducted over the candidate cluster ID interval instead, with the traversal and batch scanning focused on the cluster ID range while applying attribute constraints as filters per record. In both cases, PQ codes of records meeting the combined filtering criteria are collected for subsequent re-ranking.

Candidates Re-ranking. Once MiniRFANN retrieves the candidate vector IDs and their PQ codes that satisfy both attribute and cluster constraints, it applies a two-stage re-ranking process.

ARTICLE IN PRESS

ARTICLE IN PRESS

- In the *coarse stage*, asymmetric distance computation over PQ codes is used to estimate query-candidate distances efficiently. This step avoids accessing raw vectors and produces a preliminary ranking. Only the top-r candidates from this stage are selected for the next step.
- In the *fine-grained stage*, exact distances are computed using the original vectors. To reduce the I/O cost of loading these vectors, MiniRFANN exploits the Z-order disk layout (Sec. 3.2.3) and an in-memory cache. If a required vector is absent from the cache, a contiguous block of vectors centered around it is pre-fetched from disk, leveraging spatial locality to amortize I/O overhead.

Table 2: Example objects of attribute vector dataset O.

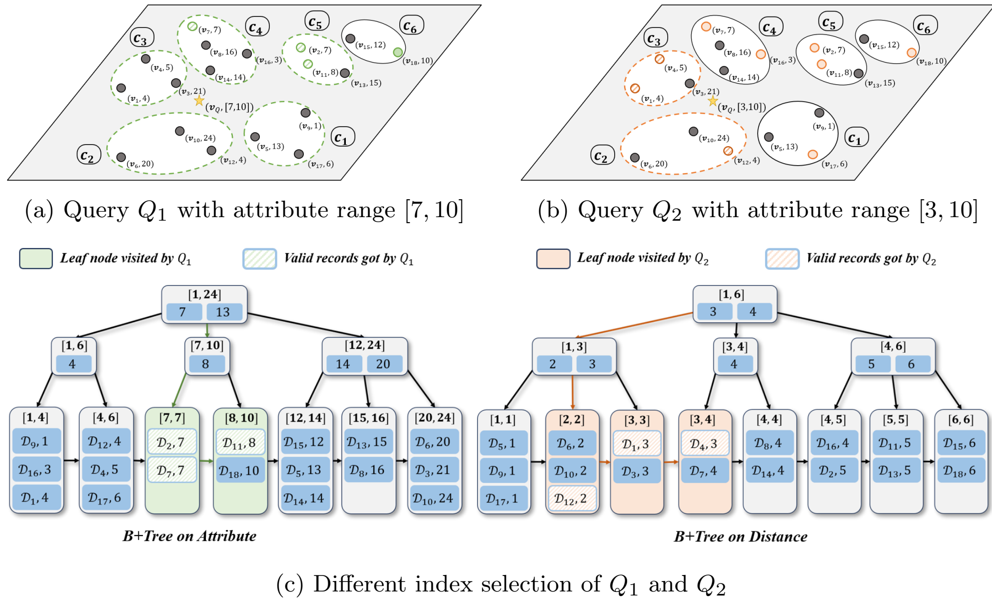
Object	a_i	$c(\boldsymbol{v}_i)$	PQ_{dist}	E_{dist}	Object	a_i	$c(\boldsymbol{v}_i)$	PQ_{dist}	E_{dist}
o_1	4	3	9.8	10.9	o_{10}	24	2	5.5	5.4
o_2	7	5	20.6	20.7	o_{11}	8	5	19.2	19.8
o_3	21	3	2.7	3.6	o_{12}	4	2	6.8	7.3
o_4	5	3	10.0	10.2	o_{13}	15	5	24.5	23.9
o_5	13	1	11.6	11.1	o_{14}	14	4	4.2	4.7
o_6	20	2	16.2	16.9	o_{15}	12	6	28.3	28.6
o_7	7	4	16.4	15.8	o_{16}	3	4	9.7	9.4
o_8	16	4	7.3	8.3	o_{17}	6	1	17.7	17.5
o_9	1	1	13.5	14.2	o_{18}	10	6	32.4	31.7

ARTICLE IN PRESS

Example 1 Here we present examples to illustrate the complete query processing workflow. We first provide 18 sample objects from the attribute vector dataset O (Tab. 2), including each object's attribute value, cluster ID, PQ-approximated distance and exact distance to the query vector $\boldsymbol{v}_Q$. Subsequently, we introduce two queries, Q_1 and Q_2, which share the same query vector $\boldsymbol{v}_Q$ but differ in their attribute ranges: Q_1 specifies $[7, 10]$, while Q_2 specifies $[3, 10]$. Query processing of Q_1 and Q_2 are summarized as follow:

- Firstly, $L_{candidates}$ is set to be 3. For Q_1, $\rho_{attribute} = \frac{|\{o_2,o_7,o_{11},o_{18}\}|}{18} = 0.22$, thus $\rho_{distance} = \frac{\min\{4,3\}}{4} = 0.75$. With $\rho_{attribute} < \rho_{distance}$, attribute-based B+tree is chosen to do hybrid filtering and PQ codes retrieve.
  For Q_2, $\rho_{attribute} = \frac{|\{o_1,o_2,o_4,o_7,o_{11},o_{12},o_{13},o_{17},o_{18}\}|}{18} = 0.5$, thus $\rho_{distance} = \frac{\min\{9,3\}}{9} = 0.33$. With $\rho_{attribute} > \rho_{distance}$, distance-based B+tree is chosen.
  After that, we determine the cluster range to be probed. By computing the distances from each cluster centroid to the query vector $\boldsymbol{v}_Q$, we obtain the centroid order from nearest to farthest as $\{3, 4, 2, 1, 5, 6\}$. For Q_1, we need to search the top $\rho_{distance} \times 6 = 5$ clusters, *i.e.,* $\{3, 4, 2, 1, 5\}$. For Q_2, we search the top $\rho_{distance} \times 6 = 2$ clusters, *i.e.,* $\{3, 4\}$. Thus, Q_2 cluster range can be expressed as $[3, 4]$.
- For each query, the search is conducted on the selected B+tree. The process begins by traversing the internal nodes to reach the relevant leaf nodes, which are then loaded from disk into memory for hybrid filtering. Fig. 2c illustrates the hybrid filtering process for Q_1 and Q_2 on different trees. Specifically, Q_1 performs a range

ARTICLE IN PRESS



(a) Query Q_1 with attribute range $[7, 10]$

(b) Query Q_2 with attribute range $[3, 10]$

(c) Different index selection of Q_1 and Q_2

Fig. 2: MiniRFANN query processing example

search $[7, 10]$ on the attribute B+tree, applying clustering-based filtering during the scan. Since the target cluster set for Q_1 is $\{3, 4, 2, 1, 5\}$, record $\mathcal{D}_{18}$ is excluded from the results.
Similarly, Q_2 executes a range search $[3, 4]$ on the distance B+tree, applying attribute-based filtering. Given that the attribute range for Q_2 is $[7, 10]$, only records $\mathcal{D}_{12}, \mathcal{D}_1$, and $\mathcal{D}_4$ within this range are retained as candidates, while $\mathcal{D}_6, \mathcal{D}_{10}, \mathcal{D}_3$, and $\mathcal{D}_7$ are pruned.

- In the final stage, the candidates obtained from the previous step are re-ranked, with the parameter r set to 3. For query Q_1, the candidates are first reordered based on the approximate distances computed from their PQ codes. Subsequently, the raw vectors are retrieved from disk to perform exact distance computation. The final results for Q_1 are $\{7, 11, 2\}$. Similarly, for query Q_2, the coarse ranking yields the candidate set $\{12, 1, 4\}$, which, after exact re-ranking, produces the final results $\{12, 4, 1\}$.

ARTICLE IN PRESS

Complexity of Query Processing. During the query processing phase, the overall latency can be decomposed into three components:

- The time spent to identify relevant cluster centroids and compute distance lookup tables via the codebook.
- The time spent to search the B+ tree to retrieve the PQ codes of candidate vectors that satisfy the query attribute range.

ARTICLE IN PRESS

- The time spent to access the raw vectors from disk for exact distance computation and subsequent final re-ranking.

Since both the cluster centroid information and the codebook are fully stored in main memory, this step incurs purely *in-memory computation* without any disk I/O cost. Its computational complexity depends on the number of probed clusters, but it is typically negligible compared to the subsequent I/O-bound steps.

Given the query's attribute filter $[a_L, a_R]$, we traverse the B+ tree index to locate all PQ codes of vectors whose attribute values satisfy the range constraint. Let $\rho_{\text{attribute}}$ and ρ_{distance} respectively denote the fractions of vectors filtered by the attribute constraint and the cluster proximity filter. Only $\min\{\rho_{\text{attribute}}, \rho_{\text{distance}}\} \cdot N$ candidate vectors remain for PQ code retrieval, where N is the dataset size. Given a B+tree leaf node capacity B and a sequential leaf-node loading factor β_{tree}, the I/O complexity for this step is:

$$O\left(\frac{\min\{\rho_{\text{attribute}}, \rho_{\text{distance}}\} \cdot N}{B \cdot \beta_{\text{tree}}}\right).$$

This term captures the number of block reads from disk when retrieving candidate PQ codes.

The final step retrieves the raw vectors for exact distance computation on the top-r candidates returned by the PQ-based filtering stage. Since raw vectors are stored on disk, β_{raw} is the number of disk blocks each I/O loaded, λ_Z represents the proportion of actually retrieved candidate vectors relative to the total vectors loaded per I/O operation. The I/O complexity for this stage is:

$$O\left(\frac{r}{B \cdot \beta_{\text{raw}} \cdot \lambda_Z}\right).$$

Based on the preceding analysis, the I/O complexity during query processing is:

$$O\left(\frac{\min\{\rho_{\text{attribute}}, \rho_{\text{distance}}\} \cdot N}{B \cdot \beta_{\text{tree}}} + \frac{r}{B \cdot \beta_{\text{raw}} \cdot \lambda_Z}\right).$$

ARTICLE IN PRESS

4 Experiments

4.1 Experimental Setup

Datasets. We experiment on the following datasets:

- **SIFT** [10]: It contains 1,000,000 vectors of 128-dimensional image patch descriptors generated using the SIFT algorithm, with 10,000 query vectors for evaluation.
- **GIST** [10]: It contains 1,000,000 vectors of 960-dimensional global image descriptors computed via the GIST algorithm, with 1,000 query vectors for evaluation.
- **Paper** [28]: It consists of 2,029,997 vectors of 200-dimensional dense embeddings extracted from an academic paper, with 10,000 query vectors for evaluation.

ARTICLE IN PRESS

Since the three datasets contain no numerical attributes, we assign each vector a random numerical value within specified ranges: ([0, 10000] for SIFT and GIST, and [0, 20000] for Paper). Tab. 3 summarizes the statistics of the three datasets.

Table 3: Statistics of datasets.

Dataset	Data Size	Dimension	Query Num	Attribute
SIFT	1000000	128	10000	random int.
GIST	1000000	960	1000	random int.
Paper	2029997	200	10000	random int.

Baselines. We compare our method with the following baselines:

- **Pre-Filtering** [12, 18]: It first filters vectors whose attribute values fall within the query attribute range, and then scans the qualifying vectors to find the closest one. This method always guarantees exact results through scanning all candidate vectors.
- **DiskIVF-PostFiltering** [10]: It first retrieves a large candidate vector set using an on-disk IVFPQ index without considering vector attribute values, and then filters vectors in this candidate set by their attribute values.
- **SPANN-PostFiltering** [19]: It adopts a post-filtering strategy but utilizes the SPANN index for candidate vector retrieval. This method tailors the conventional IVFPQ index for on-disk retrieval.

Implementation. All methods are implemented in C++ and compiled using GCC 7.5.0 with -O3 optimization. For our MiniRFANN, we construct internal and leaf nodes with a default size of 4KB during index building, and set the default re-ranking object count $r = 500$. All experiments are conducted in a single-threaded environment, with each experiment repeated three times.

Environment. All the experiments are carried on an Ubuntu18.04 virtual machine running on a host machine with an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz. The virtual machine is configured with 4GB RAM, 40GB disk storage, and allocated two vCPUs with each vCPU assigned 1 core.

Metrics. We assess the performance of different methods via four metrics:

- **Latency**: It measures the average time from submitting the query to obtaining the results. The metric reflects the research efficiency.
- **Recall rate**: It measures the accuracy of RFANN. It is calculated as $\frac{|A \cap A^*|}{K}$, where A^* is the ground truth result set for the RFNN query, A is the results returned by the RFANN algorithm, and K is the required size of the result set.
- **Memory overhead**: It measures the memory consumption of the RFANN algorithm during query processing.
- **Construction time**: It measures the total time elapsed from scratch to completion when building the RFANN index.

ARTICLE IN PRESS

ARTICLE IN PRESS

4.2 Experimental Results

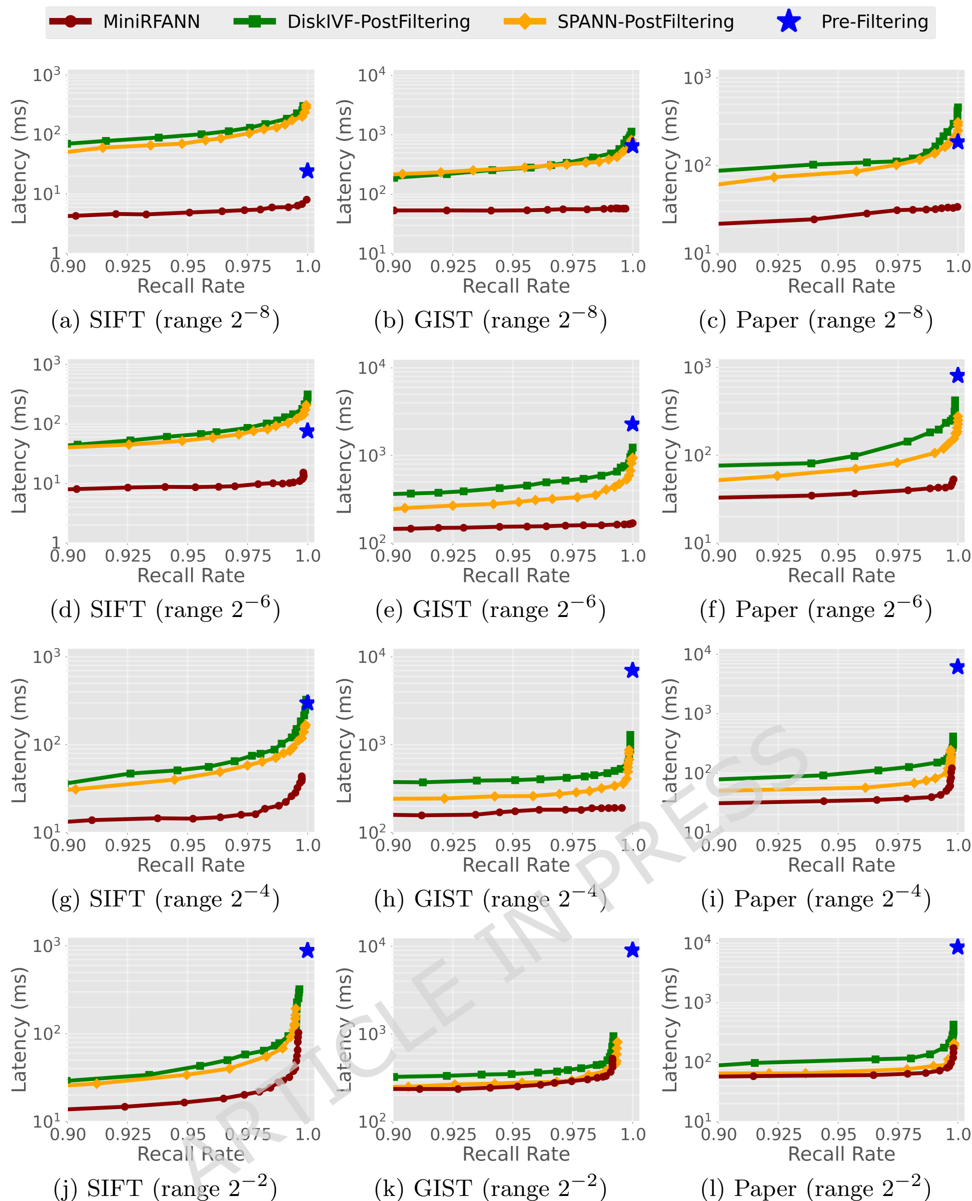
4.2.1 Results of Searching Performance

We first evaluate the query performance of all methods. Specifically, four distinct attribute range ratios, including 2^{-8}, 2^{-6}, 2^{-2}, 2^{-2}, are configured to simulate real-world scenarios where query intervals expand from narrow to broad. Experiments are conducted across all three datasets under varying query range ratios. We plot the **latency** and **recall@10** curves to show the trade-off between efficiency and accuracy. Different points in these plots are generated by changing $L_{candidates}$

Comparison between Method. Fig. 3 shows the relationship between latency and recall@10 of all methods (Pre-Filtering method are denoted by single pentagram marker in figures) across three different datasets. For all evaluated methods, latency and recall rate exhibit a positive correlation, implying that query latency inevitably increases as higher recall rates are pursued. Across all query workloads, MiniRFANN consistently reduces query latency by 10.02% to 87.13% while maintaining the same recall levels. This latency advantage becomes especially significant when the queried attribute range is narrow. For example, on the Paper dataset with 2^{-6} query range ratio, MiniRFANN achieves a latency of 39.88 ms, offering an 51.38% reduction compared to the best baseline latency of 82.02 ms.

ARTICLE IN PRESS

ARTICLE IN PRESS



(a) SIFT (range 2^{-8}) (b) GIST (range 2^{-8}) (c) Paper (range 2^{-8})

(d) SIFT (range 2^{-6}) (e) GIST (range 2^{-6}) (f) Paper (range 2^{-6})

(g) SIFT (range 2^{-4}) (h) GIST (range 2^{-4}) (i) Paper (range 2^{-4})

(j) SIFT (range 2^{-2}) (k) GIST (range 2^{-2}) (l) Paper (range 2^{-2})

Fig. 3: Query performance on three datasets under different attribute range.

ARTICLE IN PRESS

Varying Range Ratios. The query range ratios affect query performance by influencing the proportion of objects satisfying the attribute condition relative to the overall dataset cardinality. The impact of query range ratios differs between pre-filtering and post-filtering approaches. Fig. 3 shows that pre-filtering method is more suitable for

ARTICLE IN PRESS

cases where the query range ratio is small. As the query range ratio increases, the number of objects meeting the interval condition grows, and the latency of the pre-filtering method increases sharply. On the contrary, the latency of the Post-filtering method decreases as the range ratio increases. This is because a larger range ratio allows searching fewer clusters to achieve the same recall rate. For our method, an increase in the query range ratio results in higher latency. On the SIFT dataset, query latencies are observed to be 5.48 ms, 9.78 ms, 16.18 ms, and 24.24 ms as the query range expands progressively. This is because as the query range increases, although the proportion of clusters that need to be searched decreases, the number of qualified objects grows substantially. The need to load PQ codes for these valid objects and compute their distances consequently contributes to increased query latency. Under varying query range settings, our search strategy dynamically selects index trees based on the query range ratio, achieving a superior latency-recall trade-off compared to the baseline methods.

4.2.2 Results of Memory Overhead

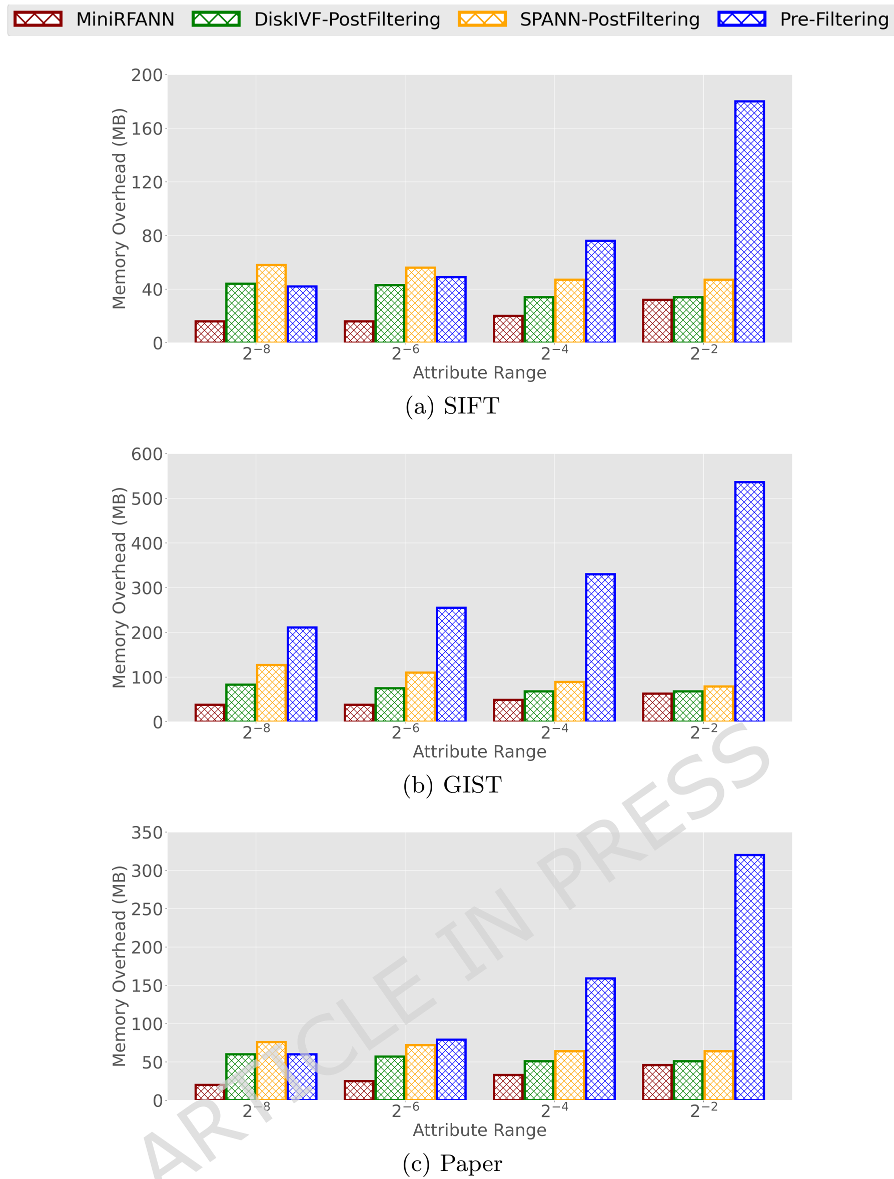
In this part, we evaluate the memory overhead of different RFANNS methods, which is a critical metric for assessing RFANNS performance in memory-constrained scenarios. Experiments are conducted across three distinct datasets under four query workloads, with results visualized through histogram plotting.

Comparison between Method. Fig. 4 shows the search memory overhead across different methods, and four query workloads. Crucially, MiniRFANN achieves the lowest memory overhead in all workload scenarios and datasets, reducing overhead by 5.88%–66.7% compared to the best baselines. This advantage is particularly pronounced when the queried attribute range is narrow. For instance, on the Paper dataset at query range ratio 2^{-6}, MiniRFANN only consumes 25 MB memory, offering a 56.14% reduction from DiskIVF-PostFiltering which is the most memory-efficient baseline at 57 MB.

Varying Range Ratios. The impact of query range ratios on memory overhead is similar to latency trends for both pre-Filtering and post-Filtering methods. As the range ratio increases, pre-Filtering method requires loading more valid objects into memory. In contrast, Post-Filtering only need to load a relatively smaller number of clusters into memory. MiniRFANN experiences escalating memory overhead with higher ratios, as evidenced on the GIST dataset where consumption progressively increases from 38 MB to 38 MB, 58 MB, and 75 MB. This growth stems directly from expanding pools of valid objects, forcing more PQ codes to reside in memory.

ARTICLE IN PRESS

ARTICLE IN PRESS



(a) SIFT

(b) GIST

(c) Paper

Fig. 4: Query memory overhead on three datasets under different attribute range.

ARTICLE IN PRESS

4.2.3 Results of Index Construction

In this part, we evaluate the time required for constructing indexes from scratch across three datasets, with results shown in Tab. 4. Pre-Filtering achieves the shortest index

ARTICLE IN PRESS

construction time since it only requires indexing based on attribute values. MiniRFANN delivers comparable construction efficiency to PostFiltering methods, despite employing hierarchical clustering adapted from the SPANN framework. Crucially, versus SPANN-PostFiltering, MiniRFANN incurs merely 7.4%, 2.1%, and 9.1% additional index construction time overhead on the three datasets respectively. This marginal increase primarily stems from MiniRFANN indexing overhead.

Table 4: Index construction time(s)

	SIFT1M	GIST1M	Paper
MiniRFANN	502	2,289	1,349
Pre-Filtering	< 10	< 10	< 10
DiskIVF-PostFiltering	442	2,692	2,148
SPANN-PostFiltering	465	2,241	1,236

4.3 Summary of Results

The previous experimental results can be summarized as follows:

- MiniRFANN demonstrates strong performance in terms of query latency and recall rate, consistently outperforming baseline methods across diverse datasets and query workloads. Under all tested query interval ratios, it achieves latency reductions ranging from 10.02% to 87.13% compared to baseline methods.
- Regarding memory overhead during query execution, MiniRFANN enables efficient queries with memory footprints below 50MB on the SIFT1M and Paper datasets, and under 80MB on GIST1M. Across all tested query interval ratios, it reduces memory overhead by 5.88%–66.67% relative to baseline methods.
- In terms of index construction, the build time of MiniRFANN is comparable to that of PostFiltering methods. Compared to the SPANN-PostFiltering method, it incurs only modest increases of 7.4%, 2.1%, and 9.1% in index construction time across the three datasets.

ARTICLE IN PRESS

5 Related Work

5.1 ANN for High-Dimensional Vectors

Approximate k-Nearest Neighbor (ANN) search is a key building block in high-dimensional retrieval tasks such as recommendation, vision, and natural language systems. It aims to retrieve data points that are close to a query vector, with sublinear query time while maintaining acceptable accuracy.

Index Structures. A wide range of ANN methods have been developed based on different indexing strategies. Hash-based methods, such as LSH [33] and E2LSH [34], use data-dependent or data-independent hashing to map similar vectors into same buckets for efficient candidate retrieval. Tree-based methods, including FLANN [35], RPTree

ARTICLE IN PRESS

[36], and ANNOY [9], partition the vector space hierarchically facilitate efficient pruning during search. Graph-based methods, such as HNSW [5], NSG [6], and SPTAG [37], represent the state-of-the-art for in-memory performance. They construct navigable small-world or monotonic search graphs to capture local proximity in the vector space and perform fast greedy search from selected entry points.

Quantization for Memory and Speed Efficiency. Orthogonal to indexing, quantization techniques are used to reduce memory footprint and accelerate distance computations. Product Quantization (PQ) [10] divides each vector into subspaces and encodes them using the indices of nearest centroids from the learned codebook, allowing for fast approximate distance calculations via lookup tables. Quantization is often integrated with indexing structures. For instance, IVFPQ [10] and IVFOPQ [11] combine inverted file (IVF) coarse partitioning with fine-grained PQ for compressing vector residuals, enabling efficient two-level retrieval with high space efficiency.

ANN for Edge Devices. Recent studies have extended ANN to low-resource environments such as edge devices. DiskANN [38] supports SSD-resident search by optimizing the traversal order and reducing random disk access. SPANN [19] focuses on memory and disk efficiency via hierarchical partitioning and quantization. Starling [39] enhances streaming access by optimizing the storage layout for block-level sequential reads.

Discussion. The vector indexing and quantization techniques in ANN provides the foundations for RFANN. Our work draws inspiration from ANN tailored for edge environments, and extends these principles to support RFANN queries under similar constraints. We also incorporate representative edge-optimized ANN systems as baselines in our experimental evaluation to benchmark the performance of MiniRFANN in a fair manner.

5.2 RFANN for High-Dimensional Vectors

Range-Filtered Approximate k-Nearest Neighbor (RFANN) search extends ANN by enforcing an additional constraint on an auxiliary attribute. Given a query vector and an attribute range, RFANN retrieves the top-k nearest vectors that also satisfy the attribute condition.

Filtering Strategies. To integrate attribute constraints into ANN, three filtering strategies are commonly used. *(i)* In-filtering [13–17, 20] integrates attribute filtering directly into the ANN index traversal, allowing early pruning of candidates during similarity search. These methods are efficient in reducing unnecessary computations but require loading both index and data entirely into memory, which limits their applicability on edge devices. *(ii)* Pre-filtering [12, 18] first applies the attribute constraint to reduce the candidate set before performing ANN search. This reduces memory usage but can trigger costly disk I/O operations when the attribute range is broad. *(iii)* Post-filtering [12, 18] executes ANN search without attribute constraints and filters the retrieved candidates afterward. But it may waste computation and I/O on irrelevant candidates, especially for narrow attribute ranges.

ARTICLE IN PRESS

ARTICLE IN PRESS

Indexing and Quantization for RFANN. Indexing and quantization techniques designed for ANN can be adapted to RFANN. PQ-based methods like RII [15] incorporate scalar range predicates into inverted indices, pruning candidates during search via early range checks. Graph-based approaches such as SeRF [13], SuperPostFilter [16], ACORN [14], and iRangeGraph [20] embed range-awareness into graph traversal using techniques like neighbor pruning, binary masking. These methods reduce unnecessary distance computations and avoid costly post-verification, achieving improved performance especially under diverse query range conditions. However, their reliance on in-memory traversal makes them less suitable for edge deployment scenarios, where memory and computational resources are limited.

Discussion. To our knowledge, no prior work systematically addresses RFANN on edge devices. We aim to fill this gap by designing a hybrid indexing and layout strategy that enables accurate, low-latency RFANN under memory and I/O constraints. citejegou2011PQ, IVFOPQ[11] aim to reduce memory footprint and accelerate distance calculations. They combine coarse vector quantization using inverted file (IVF) indexing for coarse partitioning with finer-grained product quantization for compressing vector residuals, allowing distances to be computed using efficient lookup tables.

6 Conclusion

In this paper, we present MiniRFANN, a disk-friendly framework for range-filtered approximate vector similarity search on memory-constrained edge devices. By combining dual B+ tree indexing, selectivity-driven adaptive index selection, and a Z-order data layout, MiniRFANN jointly addresses the memory and I/O bottlenecks of existing RFANN methods. By modeling range filtering and similarity pruning as complementary hard and soft selectivity, MiniRFANN adaptively chooses the most efficient index per query, achieving balanced performance across varying attribute range widths. Experiments on various benchmarks show that MiniRFANN outperforms state-of-the-art baselines in both query latency and recall. Future work includes extending MiniRFANN to support richer filter types and integrating dynamic index maintenance for evolving data. Considering more complex application scenarios, future research will focus on RFANN queries with multi-attribute constraints, where filters on time, location, and other contextual information must be jointly handled.

ARTICLE IN PRESS

References

[1] Li, W., Zhang, Y., Sun, Y., Wang, W., Li, M., Zhang, W., Lin, X.: Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. IEEE Transactions on Knowledge and Data Engineering **32**(8), 1475–1488 (2019)

[2] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., *et al.*: Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in neural information processing systems **33**, 9459–9474 (2020)

ARTICLE IN PRESS

[3] Izacard, G., Lewis, P., Lomeli, M., Hosseini, L., Petroni, F., Schick, T., Dwivedi-Yu, J., Joulin, A., Riedel, S., Grave, E.: Few-shot learning with retrieval augmented language models. arXiv preprint arXiv:2208.03299 **1**(2), 4 (2022)

[4] Wei, S., Tong, Y., Zhou, Z., Xu, Y., Gao, J., Wei, T., He, T., Lv, W.: Federated reasoning llms: a survey. Frontiers of Computer Science **19**(12), 1912613 (2025)

[5] Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Trans. Pattern Anal. Mach. Intell. **42**(4), 824–836 (2020) https://doi.org/10.1109/TPAMI.2018.2889473

[6] Fu, C., Xiang, C., Wang, C., Cai, D.: Fast approximate nearest neighbor search with the navigating spreading-out graph. Proc. VLDB Endow. **12**(5), 461–474 (2019) https://doi.org/10.14778/3303753.3303754

[7] Malkov, Y., Ponomarenko, A., Logvinov, A., Krylov, V.: Approximate nearest neighbor algorithm based on navigable small world graphs. Inf. Syst. **45**, 61–68 (2014) https://doi.org/10.1016/J.IS.2013.10.006

[8] Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975) https://doi.org/10.1145/361002.361007

[9] Bernhardsson, E.: Annoy: Approximate Nearest Neighbors in C++/Python. (2018). https://pypi.org/project/annoy/Pythonpackageversion1.13.0

[10] Jegou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. IEEE transactions on pattern analysis and machine intelligence **33**(1), 117–128 (2010)

[11] Ge, T., He, K., Ke, Q., Sun, J.: Optimized product quantization for approximate nearest neighbor search. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2946–2953 (2013)

[12] Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., Wang, X., Guo, X., Li, C., Xu, X., *et al.*: Milvus: A purpose-built vector data management system. In: Proceedings of the 2021 International Conference on Management of Data, pp. 2614–2627 (2021)

[13] Zuo, C., Qiao, M., Zhou, W., Li, F., Deng, D.: Serf: segment graph for range-filtering approximate nearest neighbor search. Proceedings of the ACM on Management of Data **2**(1), 1–26 (2024)

[14] Patel, L., Kraft, P., Guestrin, C., Zaharia, M.: Acorn: Performant and predicate-agnostic search over vector embeddings and structured data. Proceedings of the ACM on Management of Data **2**(3), 1–27 (2024)

ARTICLE IN PRESS

ARTICLE IN PRESS

[15] Matsui, Y., Hinami, R., Satoh, S.: Reconfigurable inverted index. In: Proceedings of the 26th ACM International Conference on Multimedia, pp. 1715–1723 (2018)

[16] Engels, J., Landrum, B., Yu, S., Dhulipala, L., Shun, J.: Approximate nearest neighbor search with window filters. In: Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024. OpenReview.net, ??? (2024). https://openreview.net/forum?id=8t8zBaGFar

[17] Zhang, Q., Xu, S., Chen, Q., Sui, G., Xie, J., Cai, Z., Chen, Y., He, Y., Yang, Y., Yang, F., *et al.*: {VBASE}: Unifying online vector similarity search and relational queries via relaxed monotonicity. In: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pp. 377–395 (2023)

[18] Wei, C., Wu, B., Wang, S., Lou, R., Zhan, C., Li, F., Cai, Y.: Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. Proceedings of the VLDB Endowment **13**(12), 3152–3165 (2020)

[19] Chen, Q., Zhao, B., Wang, H., Li, M., Liu, C., Li, Z., Yang, M., Wang, J.: Spann: Highly-efficient billion-scale approximate nearest neighborhood search. Advances in Neural Information Processing Systems **34**, 5199–5212 (2021)

[20] Xu, Y., Gao, J., Gou, Y., Long, C., Jensen, C.S.: irangegraph: Improvising range-dedicated graphs for range-filtering nearest neighbor search. Proceedings of the ACM on Management of Data **2**(6), 1–26 (2024)

[21] Tong, Y., She, J., Ding, B., Wang, L., Chen, L.: Online mobile micro-task allocation in spatial crowdsourcing. In: 2016 IEEE 32Nd International Conference on Data Engineering (ICDE), pp. 49–60 (2016). IEEE

[22] Meruje Ferreira, L.M., Coelho, F., Pereira, J.: Databases in edge and fog environments: A survey. ACM Computing Surveys **56**(11), 1–40 (2024)

[23] Seemakhupt, K., Liu, S., Khan, S.: Edgerag: Online-indexed rag for edge devices. arXiv preprint arXiv:2412.21023 (2024)

[24] Zhang, J., Su, K., Zhang, H.: Making in-memory learned indexes efficient on disk. Proceedings of the ACM on Management of Data **2**(3), 1–26 (2024)

[25] Morton, G.M.: A computer oriented geodetic data base and a new technique in file sequencing (1966)

[26] Mokbel, M.F., Aref, W.G., Kamel, I.: Analysis of multi-dimensional space-filling curves. GeoInformatica **7**, 179–209 (2003)

[27] Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., Jégou, H.: The faiss library (2024) arXiv:2401.08281 [cs.LG]

[28] Wang, M., Lv, L., Xu, X., Wang, Y., Yue, Q., Ni, J.: Navigable proximity

ARTICLE IN PRESS

graph-driven native hybrid queries with structured and unstructured constraints. CoRR **abs/2203.13601** (2022) https://doi.org/10.48550/ARXIV.2203.13601 2203.13601

[29] Liu, H., Huang, Z., Chen, Q., Li, M., Fu, Y., Zhang, L.: Fast clustering with flexible balance constraints. In: 2018 IEEE International Conference on Big Data (Big Data), pp. 743–750 (2018). IEEE

[30] On, S.T., Hu, H., Li, Y., Xu, J.: Lazy-update b+-tree for flash devices. In: 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware, pp. 323–328 (2009). IEEE

[31] Du, M., Scott, M.L.: Buffered persistence in b+ trees. Proceedings of the ACM on Management of Data **2**(6), 1–24 (2024)

[32] Gong, S., Sun, H., Fang, Z., Liu, L., Chen, L., Gao, Y.: Vstream: A distributed streaming vector search system. Proceedings of the VLDB Endowment **18**(6), 1593–1606 (2025)

[33] Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: Vitter, J.S. (ed.) Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998, pp. 604–613. ACM, ??? (1998). https://doi.org/10.1145/276698.276876 . https://doi.org/10.1145/276698.276876

[34] Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Snoeyink, J., Boissonnat, J. (eds.) Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004, pp. 253–262. ACM, ??? (2004). https://doi.org/10.1145/997817.997857 . https://doi.org/10.1145/997817.997857

[35] Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. VISAPP (1) **2**(331-340), 2 (2009)

[36] Dasgupta, S., Freund, Y.: Random projection trees and low dimensional manifolds. In: Dwork, C. (ed.) Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008, pp. 537–546. ACM, ??? (2008). https://doi.org/10.1145/1374376.1374452 . https://doi.org/10.1145/1374376.1374452

[37] Chen, Q., Wang, H., Li, M., Ren, G., Li, S., Zhu, J., Li, J., Liu, C., Zhang, L., Wang, J.: SPTAG: A Library for Fast Approximate Nearest Neighbor Search. (2018). https://github.com/Microsoft/SPTAG

[38] Jayaram Subramanya, S., Devvrit, F., Simhadri, H.V., Krishnawamy, R., Kadekodi, R.: Diskann: Fast accurate billion-point nearest neighbor search on a single node. Advances in neural information processing Systems **32** (2019)

ARTICLE IN PRESS

ARTICLE IN PRESS

[39] Wang, M., Xu, W., Yi, X., Wu, S., Peng, Z., Ke, X., Gao, Y., Xu, X., Guo, R., Xie, C.: Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. Proceedings of the ACM on Management of Data **2**(1), 1–27 (2024)

ARTICLE IN PRESS