Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search (Technical Report)

Fangyuan Zhang
fzhang@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR

Mengxu Jiang
mxjiang@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR

Guanhao Hou
ghhou@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR

Sibo Wang
swang@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR

ABSTRACT

Given a set $\mathcal{O}$ of objects consisting of n high-dimensional vectors, the problem of *approximate nearest neighbor (ANN)* search for a query vector $\boldsymbol{q}$ is crucial in many applications where objects are represented as feature vectors in high-dimensional spaces. Each object in $\mathcal{O}$ often has attributes like popularity or price, which influence the search. Practically, searching for the nearest neighbor to $\boldsymbol{q}$ might include a range filter specifying the desired attribute values, e.g., within a specific price range. Existing solutions for range filtered ANN search often face trade-offs among excessive storage, poor query performance, and limited support for updates. To address this challenge, we propose a novel indexing scheme that supports efficient range filtered ANN search and updates, requiring only linear space. Our scheme integrates seamlessly with existing PQ-based index—a widely recognized, scalable index type for ANN searches—to enhance range-filtered ANN queries and update capabilities. Our indexing method, supporting arbitrary range filters, has a space complexity of $O(n \log K)$, where K is a parameter of the PQ-based index and $\log K$ scales with $O(\log n)$. To reduce the space cost, we further present a hybrid two-layer structure to reduce space usage to $O(n)$, preserving query efficiency without additional update costs. Experimental results demonstrate that our indexing scheme significantly improves query performance while maintaining competitive update performance and space efficiency.

PVLDB Reference Format:
Fangyuan Zhang, Mengxu Jiang, Guanhao Hou, and Sibo Wang. Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search (Technical Report). PVLDB, 16(1): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/oldjang/RangePQ.

1 INTRODUCTION

In recent years, with the unprecedented rise of large language models [41] and the surge in machine learning [28], the management of high-dimensional data has attracted considerable demand [45]. High-dimensional data, also known as vectors, is an important data type in the current era. A large number of machine learning models are used to compress real-world objects into high-dimensional feature vectors. These models have transformed a wide variety of data types, such as images and graphs, into vectors that encapsulate information about real-world objects in a high-dimensional space [19, 20, 27, 35, 36, 39]. Consequently, vector analysis queries have become essential to numerous applications, including online search [13] and recommendation systems [29, 30, 49]. The varied applications and rapid growth of data volume have also spurred advancements in vector database management systems [45, 48, 51].

A vector database can be defined as a collection, $\mathcal{O}$, of n objects, each represented as a vector in d-dimensional space, $\mathbb{R}^d$. A fundamental challenge is how to find the nearest neighbor of a given query vector $\boldsymbol{q} \in \mathbb{R}^d$ [26]. Searching for the nearest neighbor in high-dimensional space is notoriously difficult due to the curse of dimensionality [33, 47]. As a result, the focus often shifts to identifying the approximate nearest neighbor (ANN) of $\boldsymbol{q}$ [2]. Currently, ANN search algorithms are essential tools for extracting relevant information from large vector databases in high-dimensional spaces. These algorithms are utilized in systems and applications to expedite vector analysis queries, exemplified by Apache Lucene [10] and Milvus [45]. ANN searches support top-k queries, returning the k vectors from the set $\mathcal{O}$ that are closest to the query vector $\boldsymbol{q}$ in approximation. In many cases, each object has associated attributes like popularity or price, which can influence the search outcome. During an ANN search using a query vector $\boldsymbol{q}$, a range filter based on these attributes can be applied. This effectively narrows the search to a subset of $\mathcal{O}$ that meets the specified criteria. For instance, Google Multisearch not only conducts Top-k searches based on image vector data but also integrates attribute filters during the querying process. For example, suppose there exists an items table of e-commerce, where each item corresponds to a feature vector and is associated with a price as the corresponding attribute value. Someone would like to find the top-k items in the table that are closest to the distance of the given item image vector, with the price of the items not less than a threshold value t. This is a common query in vector databases, and many research works have been conducted on this topic [38, 45, 48, 51, 52].

Limitations of existing solutions. In order to handle range-filtered ANN search, existing solutions are categorized into the following types: ANN-first methods, range-first methods, and range-index methods. Given a query, ANN-first methods search the well-constructed index for the objects closest to the query vector $\boldsymbol{q}$ and then check each accessed object to see if it meets the filter

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

criteria [51]. When the number of accessed objects that satisfy the filter reaches a configurable parameter, the nearest k objects are returned as the approximate answer. Range-first methods use a pre-established index on attributes to select objects that satisfy the filter criteria and then perform a linear scan on these objects [45]. These simple query strategies would scan many irrelevant objects, resulting in inferior query performance.

The existing range-index method SeRF [52] uses a graph-based ANN search algorithm HNSW [32], which involves compressing indexes built for different ranges. This approach avoids retrieving objects that do not meet the filter criteria during the search. However, using this method faces the following challenges. Due to the compression of graphs constructed within various attribute ranges, the worst-case space overhead can reach up to $O(n^2M)$, where M is the maximum out-degree of nodes in the HNSW graph. When the data volume is large, such a space cost is clearly unacceptable. When there are favorable assumptions, that is, when the distribution of attributes and vectors satisfies the independence assumption, the average space overhead can be reduced to $O(Mn \log n)$. However, real data may not match these assumptions, e.g., recipes with higher popularity may be closer in higher dimensional spaces. According to the experiments [52], the index SeRF consumes multiple times the space of the original data. However, an effective index should be lightweight. Moreover, it is difficult to support updates to a set of objects, such as inserting and deleting objects in $\mathcal{O}$. This is because the construction process requires objects to be inserted in order of attribute value. This demands that the attribute value of the newly inserted object be larger than the attribute values of all objects in the set $\mathcal{O}$ to incrementally maintain the index; otherwise, it needs to rebuild the index. For delete operations, it is deficient for SeRF to support updating the index for the updated set.

Product quantization (PQ) (detailed in Section 2.1) is an effective indexing method that compresses high-dimensional vectors into several low-dimensional subspaces. To accelerate the search process further, the data points are grouped into coarse clusters [23]. PQ-index methods are generally more scalable than graph-based indices. For example, they require only 30GB for a billion-entry dataset while delivering millisecond query latencies and a recall@100 rate above 0.9, indicating good accuracy [7]. Due to its impressive performance, PQ-indexes have been widely adopted in real-world systems like Faiss [24], ScaNN [21], Milvus [45], and AnalyticDB [48]. However, challenges persist with PQ-based indices in range-filtered ANN search. Traditional ANN-first and range-first approaches often yield suboptimal results for previously discussed reasons. A potential solution involving distinct PQ-based indices for each interval would incur significant space overheads. To the best of our knowledge, there are currently no PQ-based indices specially designed to handle range-filtered ANN searches.

Our solution. Addressing the shortcomings of existing techniques, we introduce RangePQ, an indexing scheme tailored for efficient range-filtered ANN searches, which also facilitates efficient updates within linear space. The core of our approach involves constructing a binary search tree (BST) $\mathcal{T}$ for the set $\mathcal{O}$, using the attribute of interest as the key, and then encoding the coarse clustering information of each object in PQ-index into each node of the BST. This encoding ensures that each node includes a set of coarse clusters containing objects that meet the range filter criteria. The encoded information at each node of the BST is the set of coarse clusters that include objects fulfilling the range filter.

The query process involves two main steps: (i) Identify and retrieve disjoint subtrees or singleton nodes from $\mathcal{T}$, covering all data points within the range filter. This step allows us to directly access the coarse clusters containing the relevant data points. (ii) From these coarse clusters, retrieve the L objects closest to the query vector $\mathbf{q}$, selected based on the distance from their cluster centers to the query vector $\boldsymbol{q}$. This approach also leverages the encoded information in the BST $\mathcal{T}$ without the need for additional indexing structures and facilitating efficient data retrieval within specified ranges. RangePQ offers significant advantages: unlike ANN-first methods, it avoids computing distances for vectors outside the specified range, and compared to Range-first methods, it does not retrieve all objects within the range but still uses the PQ-based index for efficient querying. This results in superior query performance.

We further propose efficient algorithms to update the index when the object set $\mathcal{O}$ changes. It should be noted that using some popular BSTs such as AVL trees and Red-Black trees makes it difficult to update efficiently since each node includes auxiliary structures. A rotation might cause the worst update cost of $O(n)$. In contrast, with our weight balanced strategy, we can reduce the update cost to $O(\log n)$. Encoding all information into the BST would result in a space cost of $O(n \log n)$. To address this, we further propose a hybrid two-layer index RangePQ+, which reduces space costs while maintaining efficient update and query processing. To summarize, our main contributions are as follows.

- We propose an effective index to efficiently answer range filtered ANN search, which can be seamlessly integrated with existing PQ-index schemes; We further prove that the query cost of the proposed index is only related to the number of objects that fall into the range, thus avoiding additional overhead.
- We further propose update algorithms to efficiently maintain indexes against dynamic object sets and prove that our update algorithms have an amortized update cost of $O(\log n)$ time.
- We further reduce the space cost from $O(n \log K)$ to $O(n)$ with a hybrid two-layer index RangePQ+, while achieving a comparable query cost and the same update cost.
- Extensive experiments on real high-dimensional datasets show that RangePQ+ improves query times tenfold while maintaining accuracy and update efficiency comparable to existing solutions.

2 BACKGROUND

2.1 Preliminaries

Let $\mathcal{O}$ be a set of n objects $\{\boldsymbol{o}_1, \boldsymbol{o}_2, \ldots, \boldsymbol{o}_n\}$, where each object is a vector in a d-dimensional Euclidean space $\mathbb{R}^d$. For any two objects $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^d$, we can measure their distance by using Euclidean distance $dis(\boldsymbol{p}, \boldsymbol{q})$, i.e., $dis(\boldsymbol{p}, \boldsymbol{q}) = \sqrt{\sum_{i=1}^{d} (\boldsymbol{p}^{(i)} - \boldsymbol{q}^{(i)})^2}$, where $\boldsymbol{p}^{(i)}$ is the i-th coordinate of vector $\boldsymbol{p}$. In high-dimensional spaces, many applications in information retrieval and database management are related to nearest neighbor search problems in high-dimensional spaces. The nearest neighbor search problem is defined as follows:

Definition 2.1 (NN Search). Given an object set $\mathcal{O}$, a query vector $\boldsymbol{q}$, and a positive integer k, the Nearest Neighbor (NN) search returns a set T with k objects so that $T = \arg\min_{T \subseteq \mathcal{O}, |T|=k} \sum_{\boldsymbol{e} \in T} dis(\boldsymbol{e}, \boldsymbol{q})$.

The exact search of NN is expensive since it takes too much computation [46]. Therefore, an approximate nearest neighbor (ANN) search is more popular in practice due to its good trade-off between accuracy and efficiency. The definition is given as follows:

Definition 2.2 (ANN search). Given an object set $\mathcal{O}$, a query vector $\boldsymbol{q}$, an approximation ratio $c > 1$ and a positive integer k, an approximate nearest neighbor (ANN) search returns k objects $\boldsymbol{o_1}, \ldots, \boldsymbol{o_k}$ sorted in ascending order of their distances to $\boldsymbol{q}$. If $\boldsymbol{o_i^*}$ is the i-th nearest neighbor of $\boldsymbol{q}$ in $\mathcal{O}$, it satisfies that $dis(\boldsymbol{q}, \boldsymbol{o_i}) \leq c \cdot dis(\boldsymbol{q}, \boldsymbol{o_i^*})$.

In practice, for ease of evaluation, we typically do not compute the exact approximation value c. Instead, recall is often employed as a proxy metric. The popular metric *Recall@k*, is used to measure the quality of the search [23]. It is defined as the fraction of query objects that the nearest neighbor is contained in the top k results. In many applications [38, 45, 48, 51], each object in the database $\mathcal{O}$ is associated with a specific attribute $attr$ of interest. For the user, the goal is to filter elements from the set $\mathcal{O}$ based on attributes within a given range. For example, in a scenario where each object is a product vector, the "price" attribute might be particularly relevant for queries that target products within a certain price range. We use $attr(\boldsymbol{o})$ to denote the value for attribute $attr$ of object $\boldsymbol{o}$, with each $attr(\boldsymbol{o})$ falling into $\mathbb{R}$. This is known as range filtered ANN search, a concept introduced earlier, formally defined as follows:

Definition 2.3 (Range Filtered ANN Search). Given a query vector $\boldsymbol{q}$, a range $Q = [x, y]$ and a positive integer k, a range filtered ANN search returns the ANN search with query vector $\boldsymbol{q}$ on the set $\mathcal{O}_Q = \{\boldsymbol{o} | attr(\boldsymbol{o}) \in Q \wedge \boldsymbol{o} \in \mathcal{O}\}$.

Example 2.4. Consider an object set $\mathcal{O} = \{\boldsymbol{o_1}, \boldsymbol{o_2}, \ldots, \boldsymbol{o_{15}}\}$ of $n = 15$ objects, as shown in Figure 1. Each pair in Figure 1 indicates an object and its attribute value. Given a query vector $\boldsymbol{q}$ which is depicted as a circular point, the NN search with $k = 1$ would return $\boldsymbol{o_2}$ as the nearest neighbor. Furthermore, for a query with the same input vector $\boldsymbol{q}$ with a range filter, where the query range is $[4, 5]$, the subset filtered by query range is $\mathcal{O}_Q = \{\boldsymbol{o_5}, \boldsymbol{o_6}, \boldsymbol{o_8}, \boldsymbol{o_{11}}, \boldsymbol{o_{12}}\}$. Therefore, $\boldsymbol{o_5}$ is the nearest neighbor satisfying the filter condition.

2.2 Product Quantization

Next, we provide a concise review of Product Quantization (PQ). PQ is a method that effectively compresses high-dimensional vectors into compact, memory-friendly codes. It enables the efficient approximation of the squared Euclidean distance between an input vector and its compressed counterpart. In PQ, a d-dimensional input row vector $\boldsymbol{x} \in \mathbb{R}^d$ is split into M-subvectors, where each sub-vector includes $d' = d/M$ dimensions, assuming that d can be divided by M. The vector $\boldsymbol{x}$ can be represented as:

$$\begin{aligned}\boldsymbol{x} &= [x_1, x_2, \ldots, x_{d'}, x_{d'+1}, x_{d'+2}, \ldots, x_{2d'}, \cdots, x_{d-d'+1}, \ldots, x_d] \\ &= [\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(M)}],\end{aligned}$$

where $\boldsymbol{x}^{(i)} \in \mathbb{R}^{d'}$ is the i-th sub-vector, for $i \in \{1, 2, \ldots, M\}$. Define $\mathcal{O}_i'$ as the set containing the i-th sub-vector $\boldsymbol{x}^{(i)}$ from each vector $\boldsymbol{x} \in \mathcal{O}$. In PQ-based solution, for each set $\mathcal{O}_i'$ ($1 \leq i \leq M$), it further identifies Z representative d'-dimensional data points as the surrogate for each of $\boldsymbol{x}^{(i)} \in \mathcal{O}_i'$. A classic solution to identify the representative surrogate is to apply a clustering algorithm, say k-Means, on set $\mathcal{O}_i'$ to derive Z clusters and use the Z centroids,

$$\{\boldsymbol{b}_1^{(i)}, \boldsymbol{b}_2^{(i)}, \cdots, \boldsymbol{b}_Z^{(i)}\},$$

as the representative data points. The set of centroids for $\mathcal{O}_i'$ is also called the i-th *sub-codebook*. The j-th centroid $\boldsymbol{b}_j^i$ for $\mathcal{O}_i'$ is also called the j-th *sub-codeword* from the i-th sub-codebook.

Then, each sub-vector $\boldsymbol{x}^{(i)}$ is assigned to the closest centroid $\boldsymbol{b}_j^{(i)}$ and use $\boldsymbol{b}_j^{(i)}$ as its surrogate. To reduce the space, PQ-based method further directly uses the identifier to represent its surrogate. For instance, in case $\boldsymbol{x}^{(i)}$ has a surrogate $\boldsymbol{b}_j^{(i)}$, then we directly use an ID $\bar{x}^{(i)} = j$ to represent sub-vector $\boldsymbol{x}^{(i)}$. After this mapping, each vector $\boldsymbol{x} = [\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(M)}]$ is encoded as:

$$\boldsymbol{x} \rightarrow \bar{\boldsymbol{x}} = [\bar{x}^{(1)}, \bar{x}^{(2)}, \cdots, \bar{x}^{(i)}, \cdots, \bar{x}^{(M)}] \in \{1, 2, \cdots, Z\}^M,$$

where $\bar{x}^{(i)}$ is the ID of the closest centroid of $\boldsymbol{x}^{(i)}$ from the i-th sub-codebook. We call $\bar{\boldsymbol{x}}$ the *PQ-code* of $\boldsymbol{x}$. Given a query vector $\boldsymbol{q}$, in the search phase, instead of using the original Euclidean distance, an *asymmetric distance* is used as an approximation. A distance table $A \in \mathbb{R}^{M \times Z}$ is computed on the fly by searching the query vector $\boldsymbol{q}$ to m sub-codebooks. Here, $A(m, z)$ is the squared Euclidean distance between the m-th sub-vector of $\boldsymbol{q}$ and the z-th sub-codeword from the m-th sub-codebook. The asymmetric distance is derived as:

$$dis(\boldsymbol{q}, \boldsymbol{x}) \approx d_A(\boldsymbol{q}, \boldsymbol{x}) = \sum_{i=1}^{M} A(i, \bar{x}^i).$$

It is an approximation of Euclidean distance between $\boldsymbol{q}$ and $\boldsymbol{x}$. The PQ index is typically constructed using *inverted file system (IVF)* [23], which employs clustering algorithms to partition all n objects into K coarse clusters to speed up the search efficiency. In particular, it first identifies K coarse centers $\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_K$ by using the clustering algorithm on the n objects. In existing solutions, K is generally set as $\Theta(\sqrt{n})$. After obtaining these coarse centers, we can divide the objects into K coarse clusters. Then, for each coarse cluster $i \in \{1, 2, \cdots, K\}$, we keep a set C_i for the IDs of objects falling into the cluster i. In the search phase, it first computes the distances between the query vector and the K coarse centers. Next, it identifies the closest n_{probe} centers and merges the object IDs in their clusters to obtain the set of candidate object IDs, where n_{probe} is a tunable parameter determined by the user. Then, it uses the distance table A to derive the approximate distance between each object and the query vector $\boldsymbol{q}$. Finally, it returns the nearest k vectors based on the approximate distance. This significantly reduces the query time.

Example 2.5. With the same object set $\mathcal{O}$ in Example 2.4, we divide the object set into 5 coarse clusters. The set C_5 of coarse cluster with ID 5 contains three objects $\boldsymbol{o_6}, \boldsymbol{o_8}, \boldsymbol{o_{12}}$. If we want to search the approximate nearest neighbor of query vector q, we can retrieve the nearest coarse center $\boldsymbol{c_2}$. Next, the set C_2 is scanned, and the asymmetric distance between each object and the query vector is computed one by one. The k nearest objects are then returned.

2.3 Exisiting solutions

Milvus. Milvus [45] employs several strategies for handling range-filtered ANN searches, each tailored for specific use cases: *Strategy (i)*, "Attribute-First-Vector-Full-Scan": Utilizes attribute range filters

to locate relevant objects via binary search or B-tree indices, followed by a full scan to produce the top-k results. This method is optimal under conditions of high selectivity, where only a limited number of objects meet the criteria of the range filter. *Strategy (ii)*, "Attribute-First-Vector-Search": This approach begins by filtering objects based on attribute range, creating a bitmap of object IDs. Subsequent vector query processing checks if each encountered object is included in the bitmap. *Strategy (iii)*, "Vector-First-Attribute-Full-Scan": Starts with vector queries without range filtering to collect objects, which are then evaluated against the attribute range filter. This strategy targets to fetch $\theta \cdot k$ objects initially, ensuring at least k objects satisfy the filter criteria with $\theta > 1$. Based on these strategies, they developed mixed strategies to achieve better performance for range filtered ANN search.

SeRF. To efficiently respond to ANN searches with arbitrary attribute range queries, one direct approach is to construct an ANN index for each subset corresponding to an interval. Then, during querying, the ANN index of the corresponding interval is utilized for the search. However, this would generate up to $O(n^2)$ ANN indexes, creating an $O(n^2)$ factor, which is clearly unacceptable when the data volume is large. To address this issue, SeRF [52] combines with the HNSW graph [32] to incrementally compress the ANN index of each interval into a single ANN index as much as possible. Each edge in the SeRF graph is attached with four values, l, r, b, e, which indicate that the edge exists in the HNSW graph constructed from a subset of the corresponding dataset when the query filter is $[x, y]$ and $x \in [l, r]$ and $y \in [b, e]$. SeRF first sorts objects by attribute value, and then sequentially inserts the objects into the index graph using the HNSW construction method in the ascending order of the attribute value of objects. Each time a new object is inserted, it creates M new edges, where M is the maximum out-degree of the HNSW node. When the out-degree of the previously inserted object exceeds M, SeRF performs a pruning operation to reduce the out-degree to M again and sets the valid intervals l, r, b, e of the pruned edges to the corresponding values. To execute the given query, it performs a standard ANN graph search, traversing only those edges that meet the range filter. The space overhead of this method can reach $O(Mn^2)$ in the worst case. When the attribute value is unrelated to the object set, its expected space overhead is $O(Mn \log n)$, and the construction time can reach $O(M^2n^2)$. Moreover, SeRF does not support arbitrary insertion and deletion of objects. Therefore, SeRF will not be able to support large and dynamically changing data volumes.

VBase. VBase [51] is designed based on the iterator model [18], which supports traversing objects in the index one by one via the `Next` interface. It processes a range filtered query as follows. First, it traverses the objects in the ANN index according to the normal ANN search process. When a new object is traversed, *relaxed monotonicity* will be checked, where the relaxed monotonicity is used to indicate whether the search process is steadily deviating from the query vector. A range filter is then performed on the traversed objects so that when an object meets the filter criteria, it is added to the result set. When the relaxed monotonicity is satisfied, it stops the query process and returns the top-k nearest objects as the final result. In contrast to Vbase, some vector database systems [45, 48, 50] answer queries with range filter by first obtaining k objects using top-k search. Then they use predicates to filter these objects. Yet, these methods have no way of knowing the exact number of objects that satisfy the filter predicate. This makes it possible for the number of objects passing through the filter to be much smaller than the k value of the query request. Setting the proper k' that produces the exact k result is challenging in practice. These methods typically employ either a conservative high k' or a trial-and-error approach using multiple k' values. Both approaches may lead to suboptimal query performance. Based on the theoretical analysis, VBase is equivalent to the conventional method that achieved the optimal k' setting. In practice, VBase also creates an index for attributes to expedite filtering in query processing. VBase also deploys cost-based query plan selection to achieve superior performance.

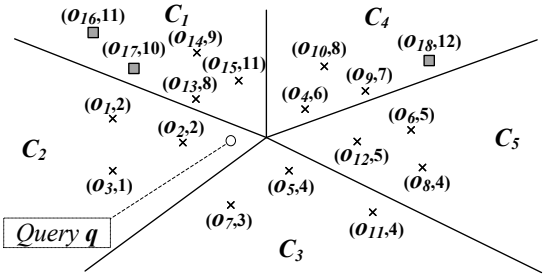


Figure 1: An example of an object set. Circular point: query vector q. Square points: new objects added in latter examples.

RII. The Reconfigurable Inverted Index (RII) [34] was initially designed to manage ANN searches with dynamically created subsets $\mathcal{O}_s$ of $\mathcal{O}$ by using the set S, of object IDs of each object in $\mathcal{O}_s$, as input. Unlike traditional methods that assume a static input set $\mathcal{O}$, RII adjusts to changes by managing and searching within these subsets. RII mainly utilizes a PQ-based index as its backbone. Recap that PQ-based index divides the n objects in $\mathcal{O}$ into K clusters. During a search, RII first derives the distance between the query vector $\boldsymbol{q}$ and each coarse center $\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_K$ of the PQ index. It then selects the top-$\lceil \frac{KL}{|\mathcal{O}_s|} \rceil$ nearest coarse centers, choosing their cluster as candidate clusters for subsequent searches. Here, L is a parameter to balance the trade-off between query time and accuracy.

After identifying the candidate clusters, RII obtains the elements in S that fall into these candidate clusters. Once all candidate lists are checked or L IDs from S are found, RII sorts these IDs and returns the top-k elements with the nearest approximate distances calculated using the distance table A. If fewer than θ elements are found in S, where θ is a predefined parameter, RII conducts a linear scan. This process involves directly comparing the distance between the query vector and each element, returning the k objects with the smallest distances. To maintain search efficiency despite the dynamic insertion of elements, RII will perform an index reconstruction when significant changes in size occur. For queries requiring a range filter, RII first retrieves the IDs of the subset of objects that meet the specified range, then applies the above query algorithm to this subset to generate the results.

3 OUR SOLUTION

As described in Sec. 2.2, PQ-index methods divide the object set into K coarse clusters, using a clustering algorithm, with coarse

centers $\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_K$. For ANN searches, these methods locate the nearest coarse centers and the associated clusters to retrieve potential matches. However, with a specified attribute range $Q = [\ell, r]$, the closest clusters might not contain any objects that meet this range, leading to unnecessary scanning and computational overhead. When only a few clusters contain objects that meet the range filter Q, it incurs a huge computational cost. In addition, existing approaches further generate unnecessary accesses when retrieving objects that satisfy the range filtering condition in the order of the distance from the cluster centers to the query vector $\boldsymbol{q}$, which also results in sub-optimal performance.

To address this issue, we propose an output-optimal solution that efficiently returns relevant coarse clusters and a specified number of objects, sorted by distance from these coarse cluster centers to the query vector $\boldsymbol{q}$, and with attribute values within $[\ell, r]$. This method reduces unnecessary computations and enhances query efficiency. We utilize a BST to encode the range information linked to cluster IDs and propose a two-step query method. The first step exclusively identifies coarse clusters containing objects within the range $[\ell, r]$. Subsequently, the index retrieves the top L objects from these clusters, ordered by the distance of query vector $\boldsymbol{q}$ to the coarse centers. Next, we introduce our indexing scheme RangePQ, which supports efficient range-filtered ANN searches and index updates.

3.1 Indexing Scheme

The main idea of the proposed index is to use the BST to encode the range information and the coarse clustering information so that given an arbitrary range $[\ell, r]$, we can easily retrieve the set of coarse clusters that include objects with $attr(\cdot)$ falling into $[\ell, r]$. To tackle this challenge, for each node u in the BST $\mathcal{T}$, we further map it as a range. In particular, let $u.lp$ (lowest point) and $u.rp$ (highest point) be the minimum and maximum attribute values of all nodes in the subtree rooted at u, respectively. We define the range of node u as $u.range = [u.lp, u.rp]$. Then, given an arbitrary range $[\ell, r]$ at query time, we have the following theorem.

Theorem 3.1 ([43, 44]). *For an arbitrary range $[\ell, r]$ in a BST, where each node u is associated with the range $[u.lp, u.rp]$, we can find: (i) A set O_1 of $O(\log n)$ singleton nodes, where each node $v \in O_1$ has $v.range$ not completely within $[\ell, r]$ but $attr(v.\boldsymbol{o}) \in [\ell, r]$. (ii) Another set O_2, also of $O(\log n)$ nodes, where each node $w \in O_2$ has $w.range \subseteq [\ell, r]$ and the subtrees rooted at nodes in O_2 are disjoint. Let O_w denote the set of objects in the subtree rooted at w. Define $O' = \bigcup_{w \in O_2} O_w$. The combined set $O = O_1 \cup O'$ exactly includes all objects whose attribute values fall within $[\ell, r]$.*

Using the theorem above, we can easily identify all objects in $\mathcal{O}$ with attribute values within $[\ell, r]$. However, this does not improve ANN query performance as it lacks the PQ-index information necessary for speeding up ANN query processing. To address this, we encode cluster IDs into the BST. Specifically, each node u includes the set of all cluster IDs for objects in its subtree. This allows us to effectively identify clusters containing data points within $[\ell, r]$. Maintaining additional cluster ID information at each node poses challenges. Traditional height-balanced trees use rotation and can result in $O(n)$ changes in the nodes of the rotated subtree, significantly altering cluster IDs and making updates prohibitively expensive. To efficiently handle dynamic insertions and deletions, we use weighted-balanced BST techniques. These trees amortize update costs across $O(s)$ nodes in a subtree of size s. We will show that the update costs for the tree index can be bounded in $O(\log n)$. Next, we explain the detailed index structure, how queries are processed with the index, and how to update the index efficiently.

RangePQ index structure. Suppose we have already performed PQ index processing for the entire object set $\mathcal{O}$ and have constructed K coarse clusters. The RangePQ index scheme integrates a binary search tree (BST) with cluster IDs to enable efficient retrieval of clusters containing objects that fall within a designated range.

Let $\mathcal{T}$ be the BST constructed from the set $\mathcal{O}$ of n objects, sorted ascendingly by the attribute values of $attr(\cdot)$, with unique object IDs to distinguish objects with identical attributes. Each node in $\mathcal{T}$ contains an object from $\mathcal{O}$ and maintains range information: $u.range = [u.lp, u.rp]$, covering the attribute ranges of its child nodes. Additionally, $u.left$ and $u.right$ represent the left and right children of u, which can be null if no child exists.

Next, we associate each node u in the tree with its object $u.\boldsymbol{o}$ (we only keep the object ID) and the corresponding cluster ID $u.P$. For each node u, we derive a union set of the cluster IDs of all nodes in the subtree of u and save it as $u.SP$. We use $u.num[i]$ to denote the number of objects in the subtree of u that belong to cluster $\boldsymbol{c_i}$. These two auxiliary structures can be implemented using a hash table. This will help subsequent algorithms quickly extract L objects correlated to the query range based on the distance of the candidate coarse centers to the query vector $\boldsymbol{q}$. The above structure can be built bottom-up for the set $\mathcal{O}$ by modifying the standard method of building a binary search tree using recursion. This is done by updating the auxiliary data structure of each node from the leaves upward, and the process can be completed in $O(n \log n)$ time. The pseudo-code for building the index is omitted due to simplicity.

We first revisit weight-balanced BST and its key properties.

Definition 3.2 (Weight-Balancing Condition). Given a balancing parameter $\alpha \in (0, 0.2]$, for any node u, either $min\{size(u.left), size(u.right)\} \geq \alpha \cdot size(u)$ or $size(u) \geq 4$ must hold.

If a node u does not satisfy the weight-balancing condition, we call it *imbalanced*. When all nodes are balanced, we can bound the height of index $\mathcal{T}$ with the following lemma:

Lemma 3.3. *The height of index $\mathcal{T}$ is $O(\log n)$.*

This is obvious: $size(u.left) \geq \alpha \cdot size(u)$ and $size(u.right) \geq \alpha \cdot size(u)$. The size of each subtree decreases exponentially. So the height of $\mathcal{T}$ is bounded by $O(\log n)$. The next lemma will be used in the design and cost analysis of subsequent update algorithms.

Lemma 3.4 ([12]). *Whenever a node u becomes imbalanced, we can fix it in constant time by performing constant rotations. After the fix, u can become imbalanced only after $\Omega(size(u))$ updates have taken place in the subtree of node u.*

Query with RangePQ index. Alg. 1 shows the pseudo-code of how to do a range filtered ANN search query with the index $\mathcal{T}$. First, the algorithm takes as input a range $[\ell, r]$ on $attr(\cdot)$, a query vector $\boldsymbol{q}$, and a positive integer k. Then we initialize three sets: candidate set C, which stores IDs of coarse clusters that contain objects in the query range $[\ell, r]$; result set R for storing the final query results;

Algorithm 1: RangePQ-Query

Input: RangePQ index $\mathcal{T}$, the query vector $\boldsymbol{q}$, the attribute query range $[\ell, r]$, the number of results k
Output: Top-k approximate nearest neighbors in the range

1 $C \leftarrow \emptyset, R \leftarrow \emptyset, NS \leftarrow \emptyset$;
2 $IndexSetUnion(root(\mathcal{T}), \ell, r, C, NS)$;
3 $SearchByCCenters(\boldsymbol{q}, C, \ell, r, R, NS)$;
4 **return** top-k nearest objects to $\boldsymbol{q}$ in R;
5 **procedure** *IndexSetUnion(u, ℓ, r, C, NS)*:
6 **if** $attr(u.o) \in [\ell, r]$ **then** $C \leftarrow C \cup \{u.P\}, NS \leftarrow NS \cup \{u\}$;
7 **if** $[u.lp, u.rp] \cap [\ell, r] = \emptyset$ **then return**;
8 **if** $[u.lp, u.rp] \subseteq [\ell, r]$ **then**
9 $C \leftarrow C \cup u.SP, NS \leftarrow NS \cup \{u\}$;
10 **return**;
11 **if** *node* $u.left \neq \emptyset$ **then**
12 $IndexSetUnion(u.left, \ell, r, NS)$;
13 **if** *node* $u.right \neq \emptyset$ **then**
14 $IndexSetUnion(u.right, \ell, r, NS)$;
15 **end**

node set $NS = O_1 \cup O_2$ to hold the set O_1 of roots of the $O(\log n)$ disjoint subtrees and the set O_2 of $O(\log n)$ singleton nodes according to Theorem 3.1, which is further needed for subsequent object retrieval (Line 1). Starting from the root node $root(\mathcal{T})$, we obtain the corresponding sets by calling *IndexSetUnion* (Line 2). Given the currently searched node u, if the attribute value $attr(u.\boldsymbol{o})$ of the object stored by u is within the query range, we would update set C to $C \cup \{u.P\}$ and NS to $NS \cup \{u\}$ (Line 6). If the subtree rooted at the current node u being searched has no intersection with the query range, the procedure returns directly (Line 7). If $u.range$ is fully contained by the query range, we update set C and NS and end the search process for nodes (Lines 8-10). Otherwise, we recursively search left and right children nodes of u to continue updating the maintained sets (Lines 11-15). Through the above process, we can obtain the candidate cluster set C and the node set NS.

Next, we can compute the final query result by invoking the *SearchByCCenters* procedure, whose corresponding pseudo-code is shown in Alg. 2. First, for each cluster ID i in the candidate set C, we compute the distance between its coarse center $\boldsymbol{c}_i$ and the query vector $\boldsymbol{q}$ (Lines 1-3). After calculating the distances, all the clusters are sorted in increasing order of distances (Line 4). Next, we traverse the clusters with ID in C based on distance order. For each cluster, we perform the find and distance calculations for the objects (Lines 5-13). In particular, for each traversed coarse cluster ID i, we first record the size of the current result set R as cnt_{old}. Then we extract a new object $\boldsymbol{np}$ from the set NS that satisfies the query range and belongs to the cluster C_i by calling *FetchNewObject* (Line 8). Notice that at each node u, we have encoded the cluster ID information $u.P$ for each object, and the set of coarse clusters $u.SP$ for all the subtree rooted at u. Then, we can take $O(\log n)$ time to go through the nodes in NS one by one and fetch the j-th object in cluster ID i (assuming that the objects are ordered based on nodes in NS) (Lines 16-25). As we have maintained the $u.num[i]$ and $u.P$, we can derive the j-th object by taking a prefix sum over the objects in NS using $O(\log n)$ time. In case the j-th object falls into a singleton node u, we can immediately return $u.\boldsymbol{o}$. In case u represents a subtree, then we can further map the j-th object to the cnt-th object in cluster ID i inside the subtree rooted at u (Line 20). Again, as we have recorded $u.num[i]$ and $u.P$, we can easily find the cnt-th object in cluster ID i using a recursive function *FindObjectFromNode* in Line 19 with $O(\log n)$ time. The recursion shares a similar idea as finding the k-th smallest element in BST with minor modification and hence the detailed pseudo-code is omitted due to simplicity. After fetching the j-th object $\boldsymbol{np}$ in cluster ID i, we then use the distance table to compute the distance between $\boldsymbol{np}$ and $\boldsymbol{q}$ and add it to the set R (Line 10). When the size of set R reaches the set parameter L, we return the set R directly (Line 11). Finally, the result set R is returned. We have the following theorem for the query time complexity for RangePQ.

Theorem 3.5. *The RangePQ index $\mathcal{T}$ can answer a range filtered ANN search query in $O(d \cdot Z + (C_Q + L) \cdot (M + \log n))$ time, where Z and M are defined in PQ-index (Sec. 2.2), C_Q is the number of clusters that includes objects in $\mathcal{O}_Q$. The space cost of RangePQ is $O(n \log K)$.*

Proof. The query cost is summarized as follows. Firstly, the distance table A can be computed in $O(d \cdot Z)$ time. With Theorem 3.1, we know that there are $O(\log n)$ nodes in NS. Thus, deriving the union incurs at most $O(\log n)$ times and each union takes $O(C_Q)$ time. Next, we search for the top-k nearest neighbors through the *SearchByCCenters* method. We spend $O(C_Q(M + \log C_Q))$ time to compute the distance from each candidate coarse center to the query vector $\boldsymbol{q}$ and sort all the centers by distance. Next, we traverse the objects for each coarse center $\boldsymbol{c}_i$. Since NS contains at most $O(\log n)$ nodes and the height of the tree is $O(\log n)$, it takes $O(\log n)$ time to get a new object $\boldsymbol{np}$. After all the candidate coarse clusters have been traversed or the number of retrieved objects reaches L, the search result set R is returned. This process can be completed in $O(min(|\mathcal{O}_Q|, L) \cdot (\log n + M)$ time, where $|\mathcal{O}_Q|$ is the size of objects that fall in the query range Q. In summary, the total query time spent is $O(d \cdot Z + (C_Q + L) \cdot (M + \log n))$.

Consider the space cost of index $\mathcal{T}$, it can be represented by

$$O(\sum_{i=0}^{\lceil \log_2 n \rceil} min(2^i, K) \cdot \frac{n}{2^i}). \tag{1}$$

Discussing sub-cases of the minimum value, it splits into two parts:

$$O(\sum_{i=0}^{\lceil \log_2 K \rceil} 2^i \cdot \frac{n}{2^i}) + O(\sum_{i=\lceil \log_2 K \rceil + 1}^{\lceil \log_2 n \rceil} K \cdot \frac{n}{2^i}). \tag{2}$$

The space cost of the first part is $O(n\lceil \log_2 K \rceil)$. Further, we can find that the space cost of the second part can be bounded by

$$O(2K \cdot \frac{n}{2^{\lceil \log_2 K \rceil + 1}}) = O(n). \tag{3}$$

Overall, the space cost of this part is $O(n \log K)$. □

Example 3.6. Continuing with the object set from Example 2.4, we now construct the proposed index for this set, with the completed index shown in Figure 2(a). Given a query vector $\boldsymbol{q}$ and a range filter of $[4, 7]$. We first quickly identify the nodes as shown in gray color, where we have two disjoint subtrees with nodes at $\boldsymbol{o}_8$

Algorithm 2: SearchByCCenters

Input: query vector $\boldsymbol{q}$, candidate set C of cluster IDs, query range $[\ell, r]$, result set R, the node set NS
Output: Search results in the range from C

for *each* $i \in C$ **do**
  Compute asymmetric distance between $\boldsymbol{q}$ and coarse center $\boldsymbol{c_i}$ by using distance table;
end
Sort cluster IDs in C in increasing order of distance to $\boldsymbol{q}$;
for *each* $i \in C$ *in increasing order of distance from* $\boldsymbol{c}_i$ *to* $\boldsymbol{q}$ **do**
  $cnt_{old} \leftarrow |R|$;
  while true do
    $\boldsymbol{np} \leftarrow FetchNewObject(NS, i, j = |R| - cnt_{old} + 1, \ell, r)$;
    if $\boldsymbol{np} == \emptyset$ **then break**;
    Append $\boldsymbol{np}$ into R and compute the asymmetric distance between $\boldsymbol{np}$ and $\boldsymbol{q}$ by using distance table;
    if $|R| == L$ **then return** R;
  end
end
return R;
procedure *FetchNewObject(NS, i, cnt, ℓ, r)*:
  for *node* $u \in NS$ **do**
    if $[u.lp, u.rp] \subseteq [\ell, r]$ **then**
      if $cnt \leq u.num[i]$ **then**
        return $FindObjectFromNode(u, i, cnt)$;
      $cnt \leftarrow cnt - u.num[i]$;
    else
      if $u.P == c$ **then**
        if $cnt == 1$ **then return** $u.\boldsymbol{o}$;
        $cnt \leftarrow cnt - 1$;
    end
  end
  return $\emptyset$;
procedure *FindObjectFromNode(u, i, cnt)*:
  Fetch the cnt-th object in cluster ID i from the subtree rooted at u using $u.num[i]$, at node u.

Algorithm 3: RangePQ-Insertion($\mathcal{T}, \boldsymbol{e}$)

1 $Insert(root(\mathcal{T}), \boldsymbol{e})$;
2 **procedure** *Insert(u, e)*:
3 **if** $u = \emptyset$ **then**
4 Replace node u with an node contains object $\boldsymbol{e}$;
5 **return**;
6 Update auxiliary structures maintained by internal node u with the coarse cluster ID of $\boldsymbol{e}$ and $attr(\boldsymbol{e})$;
7 **if** $attr(\boldsymbol{e}) < attr(u.\boldsymbol{o})$ **then** $Insert(u.left, \boldsymbol{e})$;
8 **else** $Insert(u.right, \boldsymbol{e})$;
9 $Maintain(u)$;
10 **procedure** *Maintain(u)*:
11 **if** u *is imbalanced* **then**
12 Make u balanced by rotation [12];
13 Update the auxiliary structures of u based on the auxiliary structures of $u.left$ and $u.right$;
14 **else return**;

Algorithm 4: RangePQ-Deletion($\mathcal{T}, \boldsymbol{e}$)

1 $u \leftarrow root(\mathcal{T})$;
2 **while** $u.\boldsymbol{o} \neq \boldsymbol{e}$ **do**
3 Update auxiliary structures maintained by internal node u with the coarse cluster ID of $\boldsymbol{e}$ and $attr(\boldsymbol{e})$;
4 **if** $attr(e) < attr(u.\boldsymbol{o})$ **then** $u = u.left$;
5 **else** $u = u.right$;
6 **end**
7 Mark u as invalid, inv += 1;
8 **if** $2 \cdot inv > size(root(\mathcal{T}))$ **then** Rebuild the entire index $\mathcal{T}$;

and $\boldsymbol{o}_4$ as the root. We further have a singleton node storing object $\boldsymbol{o}_6$. The corresponding range of each of these three nodes is shown in Figure 2. The *SP* sets maintained by the nodes containing $\boldsymbol{o}_4$ and $\boldsymbol{o}_8$ are $\{4, 5\}$ and $\{3, 5\}$, respectively. The coarse cluster ID P for in the node containing $\boldsymbol{o}_6$ is 5. Then we obtain the final candidate set of query coarse cluster IDs, $C = \{3, 4, 5\}$. With the obtained candidate set C and the query range, we can further obtain approximate nearest neighbor query results using the *searchByCCenters*.

The choice of *L*. The parameter L controls the number of objects accessed. A larger L increases recall but also raises query time, while a smaller L reduces both time and recall. Typically, L is set based on experimental results. Additionally, the ratio of L to the number of objects within the query range impacts recall. For implementation, we use an adaptive mechanism to select L, based on a predefined L_{base} and the query range coverage percentage r_Q. We define L as $max(L_{base} \cdot \frac{r_Q}{r_{base}}, L_{base})$ where r_{base} is the base parameter of coverage percentage. Experiments have validated the effectiveness of the proposed adaptive method.

3.2 Dynamic Updates

Insertion of RangePQ index. Alg. 3 shows the pseudo-code for dealing with insertions. Starting from the root node $root(\mathcal{T})$, we insert a new object $\boldsymbol{e}$. The insertion process is performed through recursive traversal. If the current node u is empty, it means we have found the position where the new node is to be inserted. Then, we create a new node at the corresponding position of u on the tree, which contains the newly inserted object $\boldsymbol{e}$ (Lines 3-5). We also update the link relationship between node u and its parent. When the current node u is not empty, we first update the auxiliary structures ($u.SP$, $u.num$, $u.lp$, and $u.rp$) maintained by u with the coarse cluster ID of the new object $\boldsymbol{e}$ and the attribute value of the new object $\boldsymbol{e}$ (Line 6). The reason is that the new node will be inserted into the subtree of u, so $u.SP$ and $u.num$ will change with the insertion of the new node. Then, we compare the attribute value of the node to be inserted with the attribute value of the current node u, to decide whether to search left or right for the position to insert the new node (Lines 7-8). After inserting the node, it backtracks up to the root node level by level, and a balance

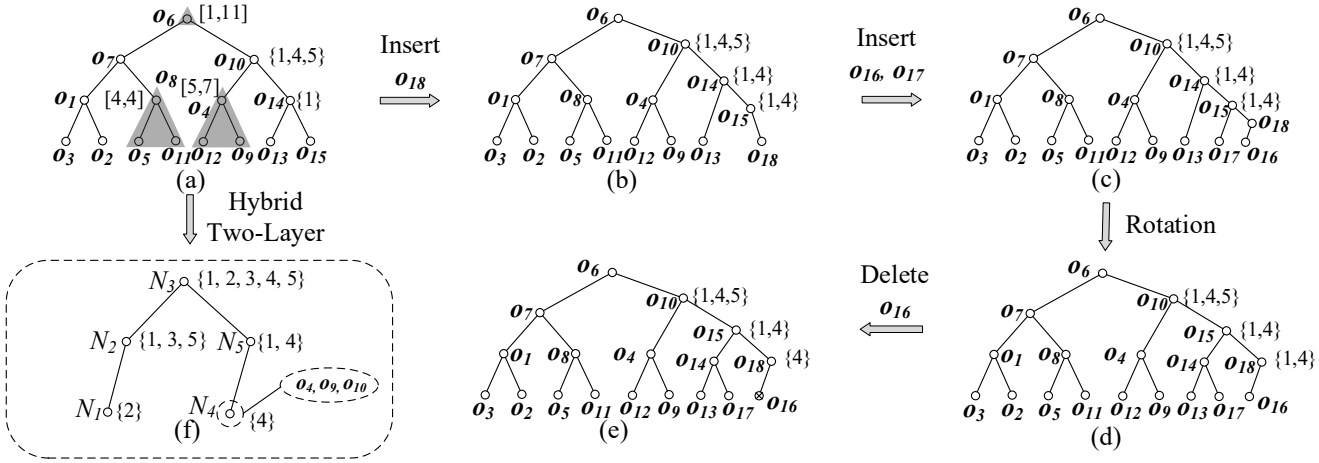


Figure 2: An example of the insertion/deletion with BST ($\alpha = 0.2$).

check is conducted on each backtracked node through the *Maintain* procedure (Line 9). If node u violates the balance condition, we re-balance node u through rotation and update the data maintained by u with the data maintained by two child nodes of u (Lines 11-14).

Theorem 3.7. *The RangePQ index $\mathcal{T}$ can handle each insertion in $O(KM + \log n)$ amortized expected time.*

Proof. We analyze the time cost of insertion step by step. First, we perform a binary search traversal on the structure $\mathcal{T}$ to find the position where object $\boldsymbol{e}$ is to be inserted and create a new node containing the inserted object e. This step takes $O(\log n)$ time. During the search process, we update the auxiliary structures maintained by the node u for each accessed node u, expected within $O(1)$ time. For each traversed node on the path, the maintained coarse cluster ID set performs the same update operation. Note that the time complexity of calculating the coarse cluster ID to which $\boldsymbol{e}$ belongs is related to the clustering algorithm; in the k-Means algorithm we use, this part takes $O(KM)$ time. After the insertion is complete, we recursively check all the nodes on the path, and if a node is found to be imbalanced, we re-balance it through rotation operations. According to Lemma 3.4, for an imbalanced node, we re-balance it with only constant rotations, and the time cost of rotation is $O(size(u))$ because we need to reconstruct the auxiliary structures maintained by the rotated node after rotation. Once a node is re-balanced, it may become imbalanced again after $O(size(u))$ insertions within its subtree. Therefore, the amortized cost for each operation within the subtree is $O(1)$. For a single insertion, it affects the size of the subtrees of at most $\log n$ nodes, and we charge an amortized cost of $O(1)$ for each node. Overall, the cost of insertion is $O(KM + \log n)$ amortized time. □

Deletion of RangePQ index. Alg. 4 shows the pseudo-code for deletion. When an object $\boldsymbol{e}$ is removed from $\mathcal{O}$, we start from the root node to find the node containing $\boldsymbol{e}$ (Lines 1-6). During the search process, it updates the coarse cluster set $u.SP$ and $u.num$ for each node u it visits. It searches downward according to the key of u until it finds the corresponding node. Then, it marks u as invalid but does not delete u (Line 7). At the same time, it records the number of invalid nodes in the tree, *inv*. This deletion does not affect the insertion process because it does not affect the actual size of the left and right subtree leaves of each node. If the existing node containing $\boldsymbol{e}$ is marked as invalid at the time of insertion $\boldsymbol{e}$, we simply mark it as valid. Clearly, *inv* can be maintained within $O(1)$ time. When $2 \cdot inv > size(root(\mathcal{T}))$, the entire tree is rebuilt (Line 8). Note that $size(u)$ includes the number of all invalid and valid nodes in the subtree of u. Also, the number of valid nodes is at least half of the total number of objects in the tree, thus it does not affect the time complexity of insertion and query algorithms.

Theorem 3.8. *The RangePQ index $\mathcal{T}$ can handle each deletion in $O(\log n)$ amortized expected time.*

Proof. For deletion, to find the node to be deleted, it takes $O(\log n)$ time. Updating the auxiliary structures maintained by the affected node requires a total expected time of $O(\log n)$. Besides, since the tree is only rebuilt with $O(n \log n)$ time after deleting $\Omega(n)$ objects, the amortized cost of each deletion is expected $O(\log n)$. □

Example 3.9. Continuing with the object set from Example 2.4 and the initial index $\mathcal{T}$ shown in Fig. 2(a), only partial SP sets of nodes are displayed due to space constraints. First, $\boldsymbol{o}_{18}$ is inserted into $\mathcal{O}$, it then searches on $\mathcal{T}$ and places it as the right child of the node containing $\boldsymbol{o}_{15}$. Then, we update the SP set of the relevant nodes based on $\boldsymbol{o}_{18}$'s coarse cluster ID. All nodes remain balanced, so the structure after insertion is shown in Fig. 2(b). Subsequent insertions of $\boldsymbol{o}_{16}$ and $\boldsymbol{o}_{17}$ proceed similarly, with the index updated in Fig. 2(c). After inserting $\boldsymbol{o}_{17}$, the node containing $\boldsymbol{o}_{14}$ becomes imbalance and it leads to a node re-balancing, with the corrected structure shown in Fig. 2(d). The removal of $\boldsymbol{o}_{16}$ involves marking its node, updating $u.num[i]$ if $\boldsymbol{o}_{16}$ belongs to cluster i, and adjusting the SP set if $u.num[i]$ drops to zero. Ancestor nodes are updated accordingly, but no balancing is required due to the nature of the deletion. The final tree structure is as shown in Fig. 2(e).

3.3 Hybrid Two-Layer Index

The above index efficiently handles range-filtered ANN searches and index updates. However, its space cost is not linear, making it impractical to store entirely in memory for large datasets. To address this, we introduce a *hybrid two-layer structure*, termed *RangePQ+*, which compresses the original tree by having each node contain multiple objects. This new structure maintains the object set $\mathcal{O}$ using a modified index that compresses the tree into two layers: the original tree structure as the first layer and a second layer where each node, representing a compressed subtree, contains multiple objects. These nodes are managed using linear space cost data structures, details of which will be discussed later. This hybrid structure maintains query efficiency and the original time complexity for insertions and deletions while reducing space costs.

RangePQ+ index structure. The proposed hybrid two-layer scheme RangePQ+ is implemented by compressing consecutive objects into a reduced number of nodes. The main idea of the hybrid two-layer compression technique is to first sort objects by attribute values, assuming that all attribute values are unique. When attributes are the same, we can further deduplicate them by key values. Then, we sequentially generate $\zeta = \Theta(n/K)$ nodes, each of which contains $\epsilon = \Theta(K)$ objects with consecutive attribute values, where ϵ is a hyper-parameter set by the user. Subsequently, for the compressed nodes, we create a tree index $\mathcal{T}_{\mathcal{H}\ell}$ to support ANNS. Similarly, for each node in $\mathcal{T}_{\mathcal{H}\ell}$, we retain two values, lp, and rp, to save the minimum and maximum attribute values in its subtree. Additionally, each node also stores the minimum and maximum attribute values of the objects held in the corresponding node, denoted as Clp and Crp. To support ANNS queries, for PQ-index information, we also associate the union of coarse cluster IDs $u.PN$ of all objects stored in each node. Each node also maintains a set SP, which preserves the union set of coarse cluster IDs of objects contained in all nodes within the subtree. Besides, we maintain a hash table $u.HT$ for each node to maintain all the objects within the node, which is indexed using the coarse cluster ID of each object as the key. By using this hash table, we can quickly extract objects from a given cluster at each node, which facilitates the query processing.

Query with RangePQ+ index. The query algorithm for the RangePQ+ index is similar to that of the original RangePQ. Overall, we perform efficient queries of the union of coarse cluster IDs within a range, using the coarse cluster IDs maintained by nodes on the tree. Note that special processing is required when a node is not completely contained within the interval, i.e., when only some of the objects in the node are contained within the interval. Alg. 5 shows the pseudo-code for performing range queries on index $\mathcal{T}_{\mathcal{H}\ell}$. We initialize three sets C, R, and NS. Then, we obtain the union of coarse cluster IDs of nodes in $\mathcal{T}_{\mathcal{H}\ell}$ that are entirely contained within the range $[l, r]$ by calling the *HybridIndexSetUnion* algorithm. The *HybridIndexSetUnion* algorithm performs a recursive process similar to the *IndexSetUnion* procedure. When the $[u.Clp, u.Crp]$ of a node u is completely contained within the range, we directly use $u.PN$ to update set C, and update set NS (Line 14). Otherwise, we repeat the process from Lines 7-14 of Algorithm 1, replacing *IndexSetUnion* with *HybridIndexSetUnion* (Line 15). Note that in $\mathcal{T}_{\mathcal{H}\ell}$, some nodes may only have part of their objects contained within the range $[l, r]$. Fortunately, these cases only occur in nodes

Algorithm 5: RangePQ+ Query

Input: RangePQ index $\mathcal{T}_{\mathcal{H}\ell}$, the query vector $\boldsymbol{q}$, the attribute query range $[\ell, r]$, the number of results k
Output: Top-k approximate nearest neighbors in the range

1 $C \leftarrow \emptyset, R \leftarrow \emptyset, NS \leftarrow \emptyset$;
2 $HybridIndexSetUnion(root(\mathcal{T}_{\mathcal{H}\ell}), \ell, r, C, NS)$;
3 $HybridEndPointUnion(root(\mathcal{T}_{\mathcal{H}\ell}), \ell, \ell, r, C, NS)$;
4 $HybridEndPointUnion(root(\mathcal{T}_{\mathcal{H}\ell}), r, \ell, r, C, NS)$;
5 $SearchByCCenters(\boldsymbol{q}, C, \ell, r, R, NS)$;
6 **return** top-k nearest objects of R;
7 **procedure** *HybridEndPointUnion(u, ep, ℓ, r, C, NS)*:
8 **if** $ep \in [u.Clp, u.Crp]$ **then**
9 Update set C by scanning the objects saved by u with the range of $[\ell, r]$, $NS \leftarrow NS \cup \{u\}$;
10 **else if** $ep < u.Crp$ **then**
11 $HybridEndPointUnion(u.left, ep, \ell, r, C)$;
12 **else** $HybridEndPointUnion(u.right, ep, \ell, r, C)$;
13 **procedure** *HybridIndexSetUnion(u, ℓ, r, C, NS)*:
14 **if** $[u.Clp, u.Crp] \in [\ell, r]$ **then** $C \leftarrow C \cup u.PN, NS \leftarrow NS \cup \{u\}$;
15 Repeat the operations of Lines 7-14 in Algorithm 1 with replacing *IndexSetUnion* with *HybridIndexSetUnion*;

located at the left and right endpoints of the range $[l, r]$. Therefore, we perform additional separate handling of the left and right endpoints of the query range by invoking *HybridEndPointUnion* (Lines 3-4). It conducts recursive searches based on the attribute values of objects maintained at each node (Lines 7-12). When a node containing the endpoint ep is found, the sets C and NS are accordingly updated (Line 9). Finally, we use *SearchByCCenters* to conduct an approximate top-k nearest neighbor query. Since index $\mathcal{T}_{\mathcal{H}\ell}$ is a hybrid two-layer structure, for the fetch object algorithm in the original *SearchByCCenters* we first use a similar tree-based recursive approach to locate. For the interior of each node u we fetch the objects by the corresponding hash table $u.HT$.

Theorem 3.10. *The RangePQ+ index $\mathcal{T}_{\mathcal{C}}$ can answer a range filtered ANN search in $O(d \cdot Z + C_Q(M + \log \zeta) + L(M + \log k + \log \zeta) + \epsilon)$ time, where C_Q is the number of coarse clusters that contains objects in $\mathcal{O}_Q$. The space cost of this index is $O(n)$.*

Proof. In the index $\mathcal{T}_{\mathcal{H}\ell}$, the total number of nodes is $\zeta = \Theta(n/\epsilon)$. Therefore, the query time for *Hybrid IndexSetUnion* is $O(C_Q \log \zeta)$. For the left and right endpoint queries, it takes $O(\log \zeta)$ time to find the corresponding node, and then spends $O(\epsilon)$ time to update the candidate coarse cluster IDs set C. The analysis of the time spent on *SearchByCCenters* is similar to that in Theorem 3.8. Overall, the total time cost is $O(d \cdot Z + C_Q(M + \log \zeta) + L(M + \log k + \log \zeta) + \epsilon)$. The space cost of RangePQ+ mainly comes from the SP set maintained at each node, which can be bounded by:

$$O(\sum_{i=0}^{\log \zeta} K \cdot \frac{\zeta}{2^i}) = O(\sum_{i=0}^{\log \zeta} \frac{n}{2^i}) = O(n). \tag{4}$$

The first equation in Eqn. 4 is derived by $\zeta = \Theta(n/K)$, and the second equation arises from the convergence of the summation of a

Algorithm 6: RangePQ+ Insertion

Input: Hybrid RangePQ index $\mathcal{T}_{\mathcal{H}}$, new object $\boldsymbol{e}$
Output: Hybrid RangePQ index $\mathcal{T}_{\mathcal{H}}$ after insertion

1 $Hybrid\text{-}Insert(root(\mathcal{T}_{\mathcal{H}}), \boldsymbol{e})$;
2 **procedure** *Hybrid-Insert(u, e)*:
3 **if** $attr(\boldsymbol{e}) < attr(u.\boldsymbol{o})$ **then** $v \leftarrow u.left$;
4 **else** $v \leftarrow u.right$;
5 **if** $attr(\boldsymbol{e}) \in [u.Clp, u.Crp]$ *or* $v = \emptyset$ **then**
6 Add object $\boldsymbol{e}$ into node u;
7 **if** *u needs to split* **then** Split u into two nodes and insert the latter node into the right subtree of u;
8 **return**;
9 **end**
10 Update auxiliary structures maintained by internal node u with the coarse cluster ID of $\boldsymbol{e}$ and $attr(\boldsymbol{e})$;
11 $Hybrid\text{-}Insert(v, \boldsymbol{e})$;
12 $Maintain(u)$;

Algorithm 7: RangePQ+ Deletion

Input: RangePQ index $\mathcal{T}_{\mathcal{H}}$, object $\boldsymbol{e}$ is to be deleted
Output: RangePQ index $\mathcal{T}_{\mathcal{H}}$ after insertion

1 $u \leftarrow root(\mathcal{T}_{\mathcal{H}})$;
2 **while** $attr(e) \notin [u.Clp, u.Crp]$ **do**
3 Update auxiliary structures maintained by internal node u with the coarse cluster ID of $\boldsymbol{e}$ and $attr(\boldsymbol{e})$;
4 **if** $attr(e) < u.Clp$ **then** $u = u.left$;
5 **else** $u = u.right$;
6 **end**
7 Delete $\boldsymbol{e}$ from node u and update *inv* according to the number of objects in node u;
8 **if** $2 \cdot inv > size(root(\mathcal{T}_{\mathcal{H}}))$ **then** Rebuild the index $\mathcal{T}_{\mathcal{H}}$;

geometric series. The space occupied by other auxiliary structures on the tree is also $O(n)$. Hence the space cost is $O(n)$. □

Example 3.11. We continue with the object set from Example 2.4. The original index structure is shown in Figure 2 (a). When using the hybrid two-layer structure to compress the index, assume ϵ is set to 3. After sorting the original set, it becomes $\{\boldsymbol{o_3}, \boldsymbol{o_1}, \boldsymbol{o_2}, \boldsymbol{o_7}, \boldsymbol{o_5}, \boldsymbol{o_8}, \boldsymbol{o_{11}}, \boldsymbol{o_6}, \boldsymbol{o_{12}}, \boldsymbol{o_4}, \boldsymbol{o_9}, \boldsymbol{o_{10}}, \boldsymbol{o_{13}}, \boldsymbol{o_{14}}, \boldsymbol{o_{15}}\}$, and we sequentially split them into five nodes $\{N_1, \ldots, N_5\}$. Then, based on the new nodes, we construct the corresponding index $\mathcal{T}_{\mathcal{H}}$ and maintain the coarse cluster set information for each node. The final structure is as shown in Figure 2 (f), with the *SP* set maintained by each node attached.

Insertion of RangePQ+. Alg. 6 presents the pseudo-code for inserting a new object $\boldsymbol{e}$ into index $\mathcal{T}_{\mathcal{H}}$. To insert $\boldsymbol{e}$, we start at the root node $root(\mathcal{T}_{\mathcal{H}})$ and find the right node u where $\boldsymbol{e}$ should be inserted, and then add it to u (Line 6). If the number of objects contained in the node exceeds $2 * \epsilon$, we split this node and evenly divide the stored objects into two nodes, u_{pre} and u_{suf}, in order. Then, let u replace with u_{pre}, and insert u_{suf} into the right subtree of u (Line 7). For the visited node u, we update auxiliary structures maintained by node u with the coarse cluster ID of $\boldsymbol{e}$ and $attr(\boldsymbol{e})$ (Line 10). Then, we continue to search for the node v that needs to be inserted recursively (Line 11). Finally, during the backtracking phase, we use *Maintain* to keep the tree balanced (Line 12). We have the following theorem for the time complexity of insertion.

Theorem 3.12. *The RangePQ+ index $\mathcal{T}_{\mathcal{H}}$ can handle each insertion in $O(KM + \log \zeta)$ amortized expected time.*

Proof. First, finding the node where the insertion is needed takes $O(\log \zeta)$ time. The time to insert $\boldsymbol{e}$ into the corresponding node is bouned in $O(\epsilon)$. When a node splits and a new node is inserted into the tree, the time consumed is $O(\epsilon \log \zeta)$. However, we note that this step only occurs after a node has more than ϵ inserted objects, so the amortized cost of this step is $O(\log \zeta)$. At the same time, we note that the effect on the tree size is 1, so the amortized cost of rotations to maintain tree balance is also $O(\log \zeta)$. Combining the time spent calculating the coarse center to which $\boldsymbol{e}$ belongs, the total time cost is $O(KM + \log \zeta)$. □

Deletion of RangePQ+ Index. Alg. 7 shows the pseudo-code for deleting an object $\boldsymbol{e}$ from index $\mathcal{T}_{\mathcal{H}}$. To delete $\boldsymbol{e}$, we start from the root node $root(\mathcal{T}_{\mathcal{H}})$ to find the node containing $\boldsymbol{e}$. We also update auxiliary structures maintained by node u with the coarse cluster ID of $\boldsymbol{e}$ and $attr(\boldsymbol{e})$ for currently accessed node u (Line 3). Then, based on the attribute range maintained by u, we determine the node for the next iteration (Lines 4-5). When we find the node containing object $\boldsymbol{e}$, we remove $\boldsymbol{e}$ from u. When the number of objects maintained in u is less than $\epsilon/2$, we increase *inv* by one (Line 7). Note that when inserting, if the number of objects maintained by a node changes from less than $\epsilon/2$ to more than $\epsilon/2$, we decrease *inv* by one. When $2 \cdot inv$ exceeds the number of nodes within the entire index, we directly reconstruct the entire index (Line 8).

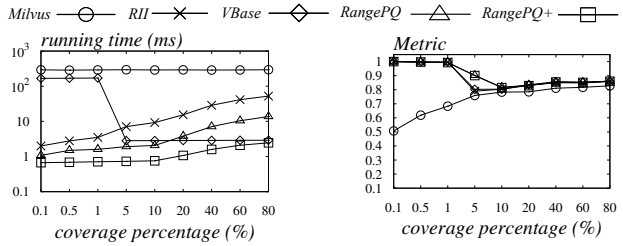
Theorem 3.13. *The RangePQ+ index $\mathcal{T}_{\mathcal{H}}$ can handle each deletion in $O(\log n)$ amortized expected time.*

Proof. For deletion, to find the node that needs to be deleted, we spend $O(\log \zeta)$ time. Updating the auxiliary structure maintained by the affected node requires a total expected time of $O(\log \zeta)$. Furthermore, since we only rebuild the tree in $O(n \log n)$ time after deleting $\Omega(n)$ data points, the cost of each deletion is $O(\log n)$ amortized expected time. □

4 RELATED WORK

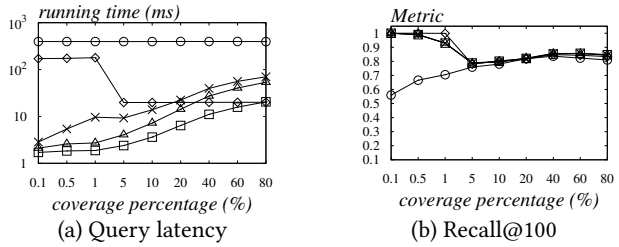
The ANN search is a classic problem in the field of databases and information retrieval, with a research history of more than 30 years [5]. Existing ANN algorithms can be divided into four types:

Locality Sensitive Hashing (LSH). LSH [22] is a method for indexing high-dimensional data. The technique involves hashing input items so that similar items map to the same buckets with high probability (they are locality-sensitive). Variants such as E2LSH[15] and FALCONN[3] are highlighted for their application in approximate retrieval of high-dimensional data, offering tunable performance with theoretical error guarantees, but require high redundancy and additional space overhead to ensure accuracy. The query (resp. space) cost is dominated by $O(n^{\rho})$ (resp. $O(n^{1+\rho})$). This class



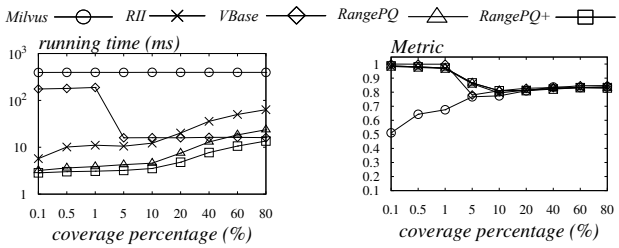
(a) Query latency (b) Recall@100

Figure 3: Query performance on SIFT dataset.



(a) Query latency (b) Recall@100

Figure 4: Query performance on GIST dataset.



(a) Query latency (b) Recall@100

Figure 5: Query performance on WIT dataset.

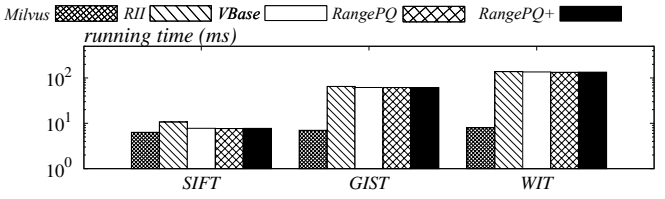


Figure 6: Insertion cost of index.

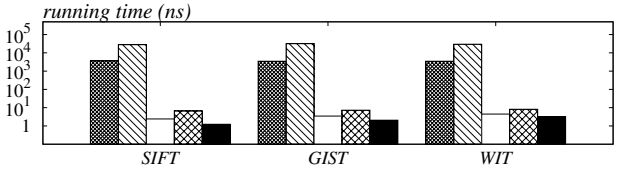


Figure 7: Deletion cost of index.

of methods provides reliable theoretical guarantees but the practical space overhead is large [38] compared with other methods. Some research works also aim to directly generate hash functions through learning methods based on data characteristics [13, 40].

Tree-based indexes. These indexes mainly consider the design of recursively splitting the entire set $\mathcal{O}$ into subsets corresponding to the subtrees of the search tree. like the k-d tree [8], PKD-tree [42], FLANN [37], RPTree [14], and ANNOY [9], are used for organizing data in a hierarchical structure, allowing for logarithmic query times in lower dimensions. These structures vary in their construction, from using principal component analysis to random projections for dividing the data space. These types of methods are mainly used to design algorithms for exactly finding nearest neighbors.

Graph-based indexes. Graph-based indexes construct a navigable graph over the dataset, where nodes represent objects, and edges connect similar objects. Techniques such as the k-nearest neighbor graph [16], monotonic search networks [17], and small world graphs [31, 32] are discussed. Given a query vector, the algorithm starts from a random initial node and traverses the graph to the node closest to the query. These indexes are celebrated for their empirical performance in high-dimensional spaces, providing efficient query execution by exploiting the graph's navigable properties. The disadvantage is that it tends to consume memory space, and the time cost for building the data structure is expensive.

Product Quantization (PQ). This method involves partitioning the high-dimensional space into multiple low-dimensional subspaces and quantizing each subspace separately. This approach is notable for reducing storage requirements and facilitating faster ANN searches [6, 23]. Therefore, PQ is a common method for handling large-scale data. Current billion-level search systems are usually based on the PQ method, especially in combination with indexes based on inverted file system. IVFADC is a notable index that combines an inverted file system with product quantization, optimizing both storage and search efficiency. Existing works have also explored the application of the PQ method in other scenarios, such as hardware acceleration [4, 11, 25].

5 EXPERIMENTS

We compare the proposed RangePQ and RangePQ+ against the state-of-the-art solutions in various aspects through experiments. We also conduct experiments to examine the impact of input parameters. All experiments are conducted on a Linux machine equipped with an Intel Xeon(R) CPU at 2.20GHz and 256GB of memory.

5.1 Experimental Settings

Datasets. We use the following three real-world datasets tested in related research [34, 51, 52]: (i) SIFT, which consists of 128-dimensional feature vectors extracted from multiple images. It provides one million base vectors, ten thousand query vectors, and one hundred thousand training vectors. We use its one million base vectors as the input data. (ii) GIST, which is composed of 960-dimensional feature vectors extracted from images. It provides one million base vectors, ten thousand query vectors, and half a million training vectors. We used its one million base vectors as the input data. (iii) WIT, which consists of 2048-dimensional embedding vectors generated using Wikipedia images through ResNet-50.

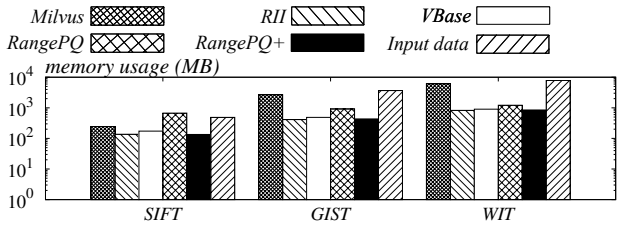
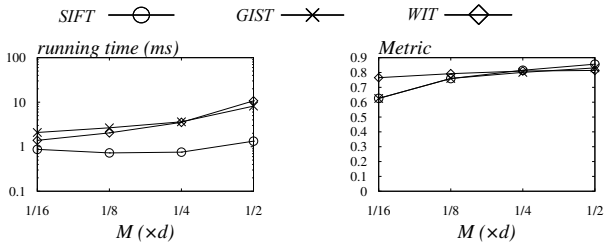


Figure 8: Memory usage of index.



(a) Query latency (b) Recall@100

Figure 9: Impact of parameter M on all datasets.

It contains more than six million vectors, and we randomly sample one million vectors as input data. Following the previous work [51, 52], for SIFT and GIST, we uniformly generate a random integer key from range $[1, 10^4]$ for each object as its attribute value. For WIT, we use the size of the image as the attribute value.

Competitors. We include the following methods in our experimental comparisons: (i) Milvus [45] is a popular vector database system, and we use `IVF_PQ` as its built-in index. (ii) RII [34] is an index that supports efficient ANN search for a subset. (iii) VBase [51] is an advanced system that efficiently supports range filtered queries. (iv) RangePQ is the index of $O(n \log n)$ space that proposed in Section 3.1. (v) RangePQ+ is the hybrid two-layer index of linear space proposed in Section 3.3. Note that we do not include SeRF as it does not support dynamic updates. Our code and more experimental details can be found in [1].

Evaluation Metrics and Parameters. Following the previous work [23], we use Recall@100 to measure the quality of range filtered ANN search results. In terms of parameters, we set the size ϵ of objects accommodated per node in RangePQ+ to 10^4. For the number L of objects computed, for the SIFT and WIT (resp. GIST) datasets, we set the base value L_{base} to 10^3 (resp. 3×10^3). For the base ratio parameter r_{base}, we set it to 10%. For the number of subspaces M of PQ-codes for all methods, we set it to $d/4$, where d is the number of dimensions of the original dataset. For all indices, we set K to $\sqrt{n}$. We conduct experiments for parameters L_{base}, M, and ϵ, which will be elaborated on later. For VBase and RII, we keep the common parameters the same and then tune their additional parameters so that queries across all tested coverage of ranges achieve the best practical query efficiency while the quality of the returned results is satisfactory, with a recall of over 0.8.

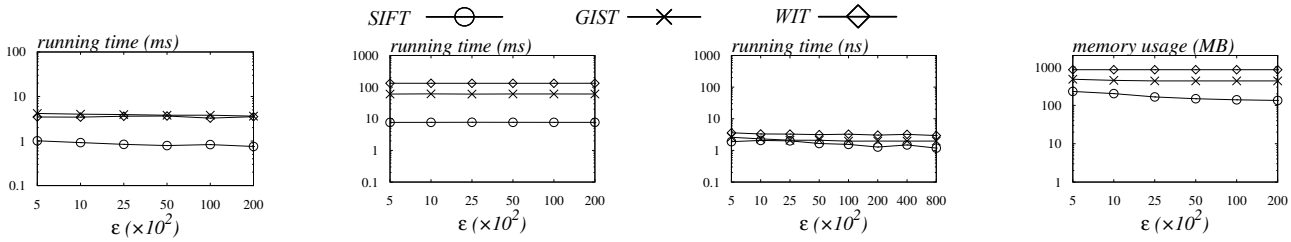
5.2 Experimental Comparisons

Exp 1: Query performance. In the first set of experiments, we test the range filtered ANN search of all indices. Figures 3-5 show the query performance of all indices across all datasets. For the SIFT and GIST datasets, we use their provided query vectors. For the WIT dataset, we randomly sample 1000 vectors not from the input data as query vectors. We test query results for different coverage of query ranges Q, which include $\{0.1\%, 0.5\%, 1\%, 5\%, 10\%, 20\%, 40\%, 60\%, 80\%\}$. The figures show the query time and the Recall@100 of the query results, with data points representing the average of all tested query results. We can observe that the proposed RangePQ+ has a clear advantage across all query ranges. Notably, our RangePQ is up to 20x faster than RII and up to two orders of magnitude faster than VBase and Milvus, while always achieving the highest recall. For VBase, it uses the built-in indexes for linear scanning when the query range coverage is less than a set threshold. Clearly, scanning the vectors incurs a high running cost. When the range coverage is high, it changes to use the ANN index and then post-prune the vectors violating the range filter. By using the ANN index, the query time is dramatically reduced, which shows the importance of ANN indexes. We also find that RangePQ+ outperforms RangePQ, due to better cache friendliness and lower height of the BST in fetching objects within RangePQ+.

Exp 2: Update efficiency. Next, we test the update performance of all indices. Figure 6 shows the time taken to insert data into all indices across all datasets. It is observed that the time taken for insertion is almost the same for all methods except Milvus. This is mainly because the major time cost of insertion is spent on finding the coarse cluster where the object belongs, which takes $O(KM)$ time. For the objects inserted, Milvus first places them into a segment. When the size of this segment reaches a certain threshold, it creates a separate index for this segment. This means that during searches, it has to traverse all objects in the unprocessed segment, leading to decreased query efficiency as shown in our Exp 1. Figure 7 shows the time taken to delete data from all indexes across all datasets. RangePQ+ has a clear advantage over other methods. RangePQ+ is faster than RangePQ because it requires updating fewer auxiliary structures and has smaller constants. RII takes more time because it needs to update an external data frame, which is used for filtering on objects.

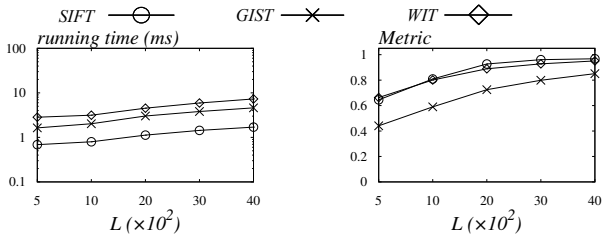
Exp 3: Indexing space cost. Figure 8 shows the memory usage of all indices across all datasets. We also report the space cost of the input data as a comparison. The space used by RangePQ+ is significantly less than that used by RangePQ. Additionally, the space usage of RangePQ+ is similar to that of RII and VBase, as they all utilize linear space. Milvus stores PQ-codes using floats, so its space consumption is slightly larger than other methods. Almost all methods use less space than the original data size, as PQ-codes effectively compress the data.

Exp 4: Impact of parameter M. In this set of experiments, we test the impact of the number M of subspaces in PQ-codes on the query performance. Based on previous work and recommended settings [23, 34], we set the number of codes Z per codebook in each subspace to 256. Figure 9 shows the query performance of RangePQ+ under different settings of M, where M is set to $\{d/16, d/8, d/4, d/2\}$, and d represents the number of original dimensions of the dataset.



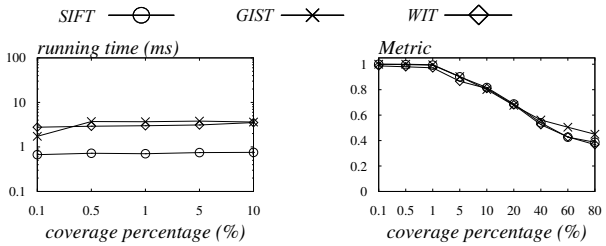
(a) Query time (b) Insertion time (c) Deletion time (d) Memory Usage

Figure 10: Impact of parameter ϵ on all datasets



(a) Query latency (b) Recall@100

Figure 11: The impact of different parameter L for queries in the fixed range coverage case on all datasets.



(a) Query latency (b) Recall@100

Figure 12: The impact of different ranges for queries in the fixed-L case on all datasets.

We find that when M is set to $d/4$, there is the best trade-off between query time and accuracy.

Exp 5: Impact of parameter ϵ. In this set of experiments, we tested the impact of different ϵ parameters (Defined in Sec. 3.3) on RangePQ+. Figure 10 shows the performance of RangePQ+ across all datasets with various settings of ϵ. We observe that as ϵ decreases, memory usage tends to increase, which is due to the first layer of the tree structure containing more nodes. Based on the experimental results, we choose an ϵ setting of 10^4 as the default parameter because it achieves a good trade-off in all respects.

Exp 6: Effectiveness of the adaptive L policy. In this set of experiments, we test the effectiveness of the adaptive L policy. Specifically, we test the impact of scanning different numbers of objects on query performance for a fixed range coverage percentage case. This result is also used to indicate how the parameters should be set. In addition to this, we also test the query performance under different ranges without the adaptive L policy. Figure 11 shows the impact of varying parameter L for RangePQ+ across all datasets. For all queries, we set the range filter to cover 10% of the elements, which also defines the setting for r_{base}. We test the query performance for all datasets with L set to $\{500, 1000, 2000, 3000, 4000\}$. We find that when L is set to 1000 (resp. 3000) for the SIFT and WIT (resp. GIST) dataset with a recall of over 0.8. Therefore, we set L_{base} to the corresponding value. Figure 12 shows the impact of varying coverage percentage for RangePQ+ across all with a fixed L policy, where L is set to 1000 (resp. 3000) for the SIFT and WIT (resp. GIST) dataset. We can find that as range coverage goes up, Recall@100 goes down very significantly. When adaptive L is used, the experimental results will have good quality query results on all ranges as shown before. This shows the effectiveness of adaptive L.

6 CONCLUSIONS

In this paper, we study the range filtered ANN search problem. We propose a lightweight index structure, RangePQ+, that efficiently supports range filtered ANN search and dynamic updates. For the query algorithm, the main idea of the proposed index is to combine PQ-index with tree structure indexing; for updates, we use the weighted balanced technique to ensure that the update cost of the index remains within $O(\log n)$. By further applying a hybrid two-layer structure, we reduce the space cost of the full-fledged solution RangePQ+ from $O(n \log n)$ to $O(n)$, while keeping the query and update efficiency. Extensive experiments demonstrate the effectiveness of the proposed method. For future work, we plan to explore other types of ANN indexes for handling range filtered ANN search in dynamic scenarios.

REFERENCES

[1] 2024. Code and technical report. https://github.com/oldjang/RangePQ.

[2] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM* 51, 1 (2008), 117–122.

[3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *NeurIPS*. 1225–1233.

[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] Sunil Arya and David M. Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *SODA*. 271–280.

[6] Artem Babenko and Victor S. Lempitsky. 2015. Tree quantization for large-scale similarity search and classification. In *CVPR*. 4240–4248.

[7] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. In *ECCV*, Vol. 11216. 209–224.

[8] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[9] Erik Bernhardsson. 2018. *Annoy: Approximate Nearest Neighbors in C++/Python.* https://pypi.org/project/annoy/ Python package version 1.13.0.

[10] Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. 2012. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval*. 17.

[11] Davis W. Blalock and John V. Guttag. 2017. Bolt: Accelerated Data Mining with Fast Vector Compression. In *SIGKDD*. 727–735.

[12] Norbert Blum and Kurt Mehlhorn. 1980. On the Average Number of Rebalancing Operations in Weight-Balanced Trees. *Theor. Comput. Sci.* 11 (1980), 303–320.

[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 Neighborhood Search. In *NeurIPS*.

[14] Sanjoy Dasgupta and Yoav Freund. 2008. Random projection trees and low dimensional manifolds. In *STOC*. 537–546.

[15] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *PoCG*. 253–262.

[16] Wei Dong, Moses Charikar, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*. 577–586.

[17] 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.

[18] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135.

[19] Martin Grohe. 2020. word2vec, node2vec, graph2vec, X2vec: Towards a Theory of Vector Embeddings of Structured Data. In *PODS*.

[20] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *SIGKDD*. 855–864.

[21] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *ICML*.

[22] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. 604–613.

[23] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128.

[24] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[25] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Trans. Big Data* 7, 3 (2021), 535–547.

[26] Donald Ervin Knuth et al. 1973. *The art of computer programming*. Vol. 3.

[27] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *ICML*. 1188–1196.

[28] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[29] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. 2017. FEXIPRO: Fast and Exact Inner Product Retrieval in Recommender Systems. In *PSIGMOD*. 835–850.

[30] Defu Lian, Haoyu Wang, Zheng Liu, Jianxun Lian, Enhong Chen, and Xing Xie. 2020. LightRec: A Memory and Search-Efficient Recommender System. In *WWW*. 695–705.

[31] 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.

[32] Yury A. Malkov and Dmitry 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.

[33] Rosalind B Marimont and Marvin B Shapiro. 1979. Nearest neighbour searches and the curse of dimensionality. *IMA Journal of Applied Mathematics* 24, 1 (1979), 59–70.

[34] Yusuke Matsui, Ryota Hinami, and Shin'ichi Satoh. 2018. Reconfigurable Inverted Index. In *ACM Multimedia Conference on Multimedia Conference, MM*. ACM, 1715–1723.

[35] Antoine Miech, Dimitri Zhukov, Jean-Baptiste Alayrac, Makarand Tapaswi, Ivan Laptev, and Josef Sivic. 2019. HowTo100M: Learning a Text-Video Embedding by Watching Hundred Million Narrated Video Clips. In *ICCV*. 2630–2640.

[36] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *NeurIPS* 26 (2013).

[37] Marius Muja and David Lowe. 2009. Flann-fast library for approximate nearest neighbors user manual. (2009).

[38] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of vector database management systems. (2023).

[39] Mattis Paulin, Matthijs Douze, Zaïd Harchaoui, Julien Mairal, Florent Perronnin, and Cordelia Schmid. 2015. Local Convolutional Features with Unsupervised Training for Image Retrieval. In *ICCV*. 91–99.

[40] Ruslan Salakhutdinov and Geoffrey E. Hinton. 2007. Learning a Nonlinear Embedding by Preserving Class Neighbourhood Structure. In *AISTATS*, Vol. 2. 412–419.

[41] Murray Shanahan. 2024. Talking about Large Language Models. *Commun. ACM* 67, 2 (2024), 68–79.

[42] Chanop Silpa-Anan and Richard I. Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *CVPR*.

[43] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. 2018. PAM: parallel augmented maps. In *PPoPP*, Andreas Krall and Thomas R. Gross (Eds.). 290–304.

[44] Yufei Tao. 2022. Algorithmic Techniques for Independent Query Sampling. In *PODS*. 129–138.

[45] 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*. 2614–2627.

[46] 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.

[47] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB*. 194–205.

[48] 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.

[49] Shitao Xiao, Zheng Liu, Weihao Han, Jianjin Zhang, Yingxia Shao, Defu Lian, Chaozhuo Li, Hao Sun, Denvy Deng, Liangjie Zhang, Qi Zhang, and Xing Xie. 2022. Progressively Optimized Bi-Granular Document Representation for Scalable Embedding Based Retrieval. In *WWW*. 286–296.

[50] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *SIGMOD*. 2241–2253.

[51] 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*. 377–395.

[52] 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.