Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search

Anonymous Author(s)

ABSTRACT

Given a set $\mathbb{O}$ 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 $\mathbb{O}$ 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 RangePQ, a novel indexing scheme that supports efficient range filtered ANN searches 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.

ACM Reference Format:
Anonymous Author(s). 2025. Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search. In *Proceedings of International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In recent years, the rise of large language models [48] and advancements in machine learning [34] have greatly increased the demand for managing high-dimensional data [52]. High-dimensional data, or vectors, are crucial today as many machine learning models compress real-world objects into feature vectors. These models transform various data types, such as images and graphs, into vectors that encapsulate essential information in high-dimensional space [25, 26, 33, 41, 42, 46]. Consequently, vector queries have become vital for applications like online search [15] and recommendation systems [35, 36, 57]. The expanding applications and data volumes have also driven advancements in vector database management systems [52, 56, 59].

A vector database is a set $\mathbb{O}$ of n objects, each represented as a vector in $\mathbb{R}^d$. A fundamental query is finding the nearest neighbor of a given query vector $\boldsymbol{q} \in \mathbb{R}^d$ [32]. Due to the curse of dimensionality [39, 55], exact nearest neighbor search in high-dimensional space is difficult, and thus the focus often shifts to approximate nearest neighbor (ANN) searches [2]. ANN algorithms are essential tools for extracting relevant information from large vector databases and are used in systems like Apache Lucene [12] and Milvus [52]. They support top-k queries, returning the k vectors closest to the query vector $\boldsymbol{q}$. Often, objects have additional attributes like popularity or price that influence search outcomes. Applying attribute-based range filters during an ANN search narrows the search to a subset of $\mathbb{O}$ meeting specific filtering criteria. For instance, Google Multisearch integrates attribute filters with top-k searches based on image vectors. In e-commerce, each item has a feature vector and an associated price. A typical query might seek the top-k items closest to a querying item vector, with prices not more than a threshold t. Such range-filtered ANN searches are common in vector databases, and many studies have addressed this topic [44, 52, 56, 59, 60].

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 criteria [59]. 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 [52]. These simple query strategies would scan many irrelevant objects, resulting in inferior query performance.

Most recently, several graph-based indices have been specifically designed to support more efficient range-filtered ANN search. One such method is SeRF [60], which utilizes the graph-based ANN search index HNSW [38]. SeRF compresses indexes built for different ranges to avoid retrieving objects that do not meet the filter criteria during the search. However, this approach faces several challenges. First, due to the compression of graphs constructed within various attribute ranges, the worst-case space overhead can reach $O(n^2 M_H)$, where M_H is the maximum out-degree of nodes in the HNSW graph. For large datasets, this space cost is clearly unacceptable. To mitigate this, SeRF employs a graph compression technique called MaxLeap, which results in significant information loss, leading to sub-optimal query performance, as observed in

SIGMOD '25, June 2025, Berlin, Germany
© 2025 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

experiments. Second, SeRF struggles with dynamic updates, such as inserting or deleting objects. Its construction process requires objects to be inserted in order of their attribute values, with newly inserted objects needing an attribute value larger than all existing objects in $\mathbb{O}$; otherwise, the index must be rebuilt. Similarly, SeRF is inefficient in handling deletions, as it lacks a mechanism to update the index after object removal. Another approach, proposed in [20], constructs separate graph indexes for multiple range combinations. During the search, it selects a pre-built index to answer the query, but the index may include objects outside the query range, requiring further filtering to ensure the results meet the range criteria. This method incurs significant space overhead due to redundancy in building multiple overlapping graph indexes. Besides, it is unclear how to efficiently update the index when the dataset changes.

Product Quantization (PQ) 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 [29]. 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@10 rate above 0.8, indicating good accuracy [9]. In many applications, this recall is already satisfactory [31]. When higher quality query results are required, reranking techniques based on the original vectors can be employed to obtain more accurate results [22]. Due to its impressive performance, PQ-indexes have been widely adopted in real-world systems like Faiss [19, 30], ScaNN [27], Milvus [52], and AnalyticDB [56]. Yet, challenges persist with PQ-based indices in range-filtered ANN search. Traditional ANN-first and range-first approaches often yield sub-optimal results for previously discussed reasons.

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. Recent methods, such as those used in graph indices [20, 60], often build dedicated ANN indices—like multiple HNSW indices—for specific attribute. This approach, however, is resource-intensive: for an attribute of interest, an ANN index of size $O(nM_H \log n)$ must be created, where M_H is a parameter of the HNSW graph. For example, we may be interested in products filtered by price range, sales, or ratings of an online shop. It is evident that these methods fail to leverage existing ANNS index within the system, resulting in additional time and space overhead.

RangePQ distinguishes itself by utilizing a common PQ index and introducing a lightweight index that encodes coarse cluster information derived from this PQ-index for a given attribute *attr*. Specifically, we construct a binary search tree (BST) $\mathcal{T}$ for the attribute *attr* to encode this information. In the BST, the key at each node corresponds to the attribute value of an object, and each node is associated with a range of attribute values. Each node records *lp* and *rp* indicating the smallest and largest attribute value in the subtree within the node, respectively. The root node *r* represents the entire attribute value range of the object set $\mathbb{O}$. This hierarchical partitioning continues recursively, mapping each node *u* to a specific range $[u.lp, u.rp]$. At each node *u* in the BST $\mathcal{T}$ associated with the range $[u.lp, u.rp]$, we maintain a set *u.SP* of coarse cluster IDs. If a coarse cluster with ID *i* contains any data point whose attribute *attr* falls within $[u.lp, u.rp]$, we include *i* in *u.SP*. As we will show, the space cost of this index design is $O(n \cdot \log K)$, where *n* is the number of nodes in the BST and *K* is the number of clusters. To reduce the space cost to $O(n)$, we introduce a hybrid two-layer structure called RangePQ+, significantly lowering the space requirements compared to existing solutions.

Using the proposed index $\mathcal{T}$, we efficiently process a range filtered ANN query by identifying the clusters containing objects within the specified range. This direct access to relevant clusters enhances query performance without requiring additional indexing structures. Utilizing these clusters and index $\mathcal{T}$, we further retrieve the top *L* relevant objects close to the query vector $\mathbf{q}$ for subsequent search, based on the distances between the cluster centers and $\mathbf{q}$. By leveraging the encoded information in the BST, our method facilitates efficient data retrieval within specified ranges. RangePQ+ offers significant advantages: *(i)* Unlike ANN-first methods, it avoids computing distances for irrelevant vectors; *(ii)* Compared to range-first methods, it does not require retrieving all objects within the range; *(iii)* In contrast to existing graph-based indices [20, 60], it employs lightweight BST for a range filtered ANN query and shares a common ANN index, thereby avoiding the excessive space costs.

Despite having a lightweight and efficient index structure, updating the index when the dataset changes remains a challenging problem. In our solution, at each node *u* of the BST $\mathcal{T}$ with range $[u.lp, u.rp]$, we maintain a set *u.SP* of coarse cluster IDs that contain data points whose attribute *attr* falls within $[u.lp, u.rp]$. Efficiently maintaining this information during updates is non-trivial. On the one hand, we need the BST to remain balanced to ensure a bounded depth for search operations, thus avoiding a degraded $O(n)$ search time. However, existing balanced BSTs such as AVL trees and Red-Black trees complicate efficient updates in our index structure. These trees perform rotations to re-balance their height, and a rotation at the root can affect the entire subtree. Consequently, the sets *u.SP* at the affected nodes may need to be updated, potentially involving up to $O(K \log n)$ coarse cluster IDs, where *K* is the number of coarse clusters. To address this issue, we propose an efficient update strategy that reduces the update cost to $O(\log n)$. This strategy allows us to maintain the balance of the BST and the correctness of the cluster ID sets *u.SP* efficiently during dataset changes. 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.
- Experiments on real high-dimensional datasets show RangePQ+ improves query times tenfold while maintaining accuracy and update efficiency comparable to existing PQ-based methods.
- We further demonstrate that compared to graph-based specialized methods for range-filtered ANN search, the proposed method achieves an excellent trade-off among index size, query performance, and update overhead.

2 BACKGROUND

2.1 Preliminaries

Let $\mathbb{O}$ 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 $\mathbb{O}$, a query vector $\boldsymbol{q}$, and a positive integer k, the Nearest Neighbor (NN) search returns a set T with k objects that has the top-k smallest distance to $\boldsymbol{q}$.

The exact search of NN is expensive since it takes too much computation [54]. 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 $\mathbb{O}$, 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 $\mathbb{O}$, it satisfies that $dis(\boldsymbol{q}, \boldsymbol{o}_i) \le c \cdot dis(\boldsymbol{q}, \boldsymbol{o}_i^*)$.

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 [29]. It is defined as the fraction of query objects that the nearest neighbor is contained in the top k results. In many applications [44, 52, 56, 59], each object in the database $\mathbb{O}$ is associated with a specific attribute *attr* of interest. For the user, the goal is to filter elements from the set $\mathbb{O}$ 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 result to query vector $\boldsymbol{q}$ on the set $\mathbb{O}_Q = \{\boldsymbol{o} | attr(\boldsymbol{o}) \in Q \land \boldsymbol{o} \in \mathbb{O}\}$.

Example 2.4. Consider an object set $\mathbb{O} = \{\boldsymbol{o}_1, \boldsymbol{o}_2, \ldots, \boldsymbol{o}_{15}\}$ 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. When the query vector $\boldsymbol{q}$ is associated with a range filter in range $[4, 5]$, the subset filtered by the range is $\mathbb{O}_Q = \{\boldsymbol{o}_5, \boldsymbol{o}_6, \boldsymbol{o}_8, \boldsymbol{o}_{11}, \boldsymbol{o}_{12}\}$. 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 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:

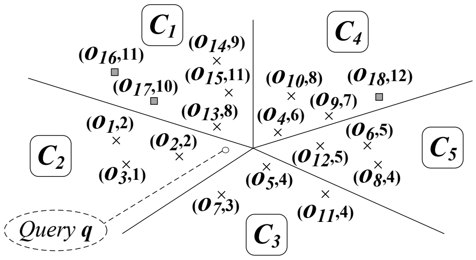


Figure 1: An example of an object set that is divided into five clusters $\{C_1, C_2, \ldots, C_5\}$. Circular point: query vector $\boldsymbol{q}$. Square points: new objects added in latter examples.

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

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

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:

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

where $\bar{x}^{(i)}$ is the ID of the closest centroid of $\boldsymbol{x}^{(i)}$ from the i-th sub-codebook. We call $\bar{x}$ 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. A value $A(m, z)$ in the distance table 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)* [29], 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 $\mathbb{O}$ in Example 2.4, we divide the object set into 5 coarse clusters. For example, 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 ANN 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 [52] 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. SeRF [60] combines with HNSW graph [38] to compress $O(n^2)$ intervals into a single ANN index. Each SeRF graph edge is annotated with four values, l, r, b, e, indicating its validity based on query filters $[x, y]$ where $x \in [l, r]$ and $y \in [b, e]$. Objects are sorted by attribute value and sequentially inserted into HNSW-based graph, creating M_H new edges per insertion. When an object's out-degree exceeds M_H, pruning reduces it to M_H and sets valid intervals for pruned edges. Queries use standard ANN search, traversing only edges meeting range filters. The method has $O(n^2 M_H)$ space overhead in the worst case. Hence, they use a method called MaxLeap for further compression, which reduces space by sacrificing query performance. 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.

SuperPostfiltering. SuperPostfiltering pre-builds graph indexes for a specific range set [20]. Given a parameter β, the set of ranges at level i is $\{[j \cdot \beta^i + 1, (j+2) \cdot \beta^i] \mid j \geq 0 \wedge (j+2)\beta^i \leq n\}$. For a query range containing m elements, it can find a range that covers it, where i satisfies $\beta^{i-1} \leq m \leq \beta^i$. After finding such a range, it queries the index built for that range and then applies filtering. SuperPostfilter demonstrates competitive query performance. However, due to the overlap between internal ranges at each level and the creation of multiple search graph indexes across different levels, it consumes a large amount of space. Besides, it does not support any form of updates. Hence, it is ineffective when the dataset is large and dynamically changing.

VBase. VBase [59] is built on the iterator model [24], allowing objects in the index to be traversed one by one using the `Next` interface. To process a range-filtered query, VBase first traverses objects in the ANN index according to the standard ANN search process. Upon visiting a new object, it checks for *relaxed monotonicity* to determine if the search is steadily deviating from the query vector. It then applies the range filter to the traversed objects; if an object meets the filter criteria, it is added to the result set. The search stops when relaxed monotonicity is met, returning top-k nearest objects. Unlike other systems [52, 56, 58] that filter after a top-k search and struggle to ensure exactly k results due to challenges in setting k', VBase avoids performance issues caused by trial-and-error. It achieves equivalent results with optimal k' and improves efficiency using attribute indexes and cost-based query plan selection.

RII. The Reconfigurable Inverted Index (RII) [40] was initially designed to manage ANN searches with dynamically created subsets $\mathbb{O}_s$ of $\mathbb{O}$ by using the set S, of object IDs of each object in $\mathbb{O}_s$, as input. Unlike traditional methods that assume a static input set $\mathbb{O}$, 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 $\mathbb{O}$ 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-$\lceil \frac{KL}{|\mathbb{O}_s|} \rceil$ 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. RII retrieves elements in S within these clusters. Once L IDs or all candidate lists are processed, it sorts the IDs by distance and returns the top-k nearest elements based on approximate distances from a precomputed table A. If fewer than θ elements are found in S, RII performs a linear scan over $\mathbb{O}_s$, calculating distances directly to ensure k results. Index reconstruction is triggered when substantial size changes occur to maintain query efficiency. For range-filtered queries, RII retrieves IDs meeting the range criteria before applying its query algorithm to generate results.

3 OUR SOLUTION

As outlined in Sec. 2.2, PQ-based indexing methods partition the dataset into K coarse clusters, each represented by a centroid, enabling efficient ANN search by limiting computations to the most relevant clusters. But when applied to range-filtered queries, existing PQ-based methods suffer from efficiency issues. The primary issues include (i) Irrelevant cluster scanning: Clusters closest to the query vector may not contain any objects satisfying the specified range, leading to redundant computations. (ii) Unnecessary accesses of objects: Existing approaches further generate unnecessary accesses when retrieving objects that satisfy the range filter in the order of the distance from the cluster centers to the query vector $\boldsymbol{q}$. Since objects in the corresponding clusters may not meet the range filter, which also results in sub-optimal performance.

To address these limitations, we propose a new structure that efficiently returns relevant coarse clusters and a specified number of relevant objects, sorted by distance from these coarse cluster centers to the query vector $\boldsymbol{q}$, and with attribute values within $[\ell, r]$. We utilize a BST to encode the range information linked to cluster IDs and propose a two-step query method. Step (i) Relevant clusters identification: this step exclusively identifies coarse clusters containing objects within the range $[\ell, r]$. Step (ii) Refined Retrieval: subsequently, we retrieve the top L relevant objects from these clusters, ordered by the distance of query vector $\boldsymbol{q}$ to the coarse centers for the following search ensuring that results meet both proximity and range criteria. Even with efficient query processing algorithms, efficiently supporting updates remains a significant challenge. As mentioned in Sec. 1, conventional update strategies can incur update overhead as high as $O(K \log n)$, where K is typically set to $\Theta(\sqrt{n})$. Therefore, we propose a weight-based amortized update algorithm characterized by its ability to average the number of updates based on the number of nodes, ensuring that the amortized cost for each update can be reduced to $O(\log n)$, independent of K, significantly lowering the cost of updates. Finally, to make the index more lightweight, we employ a hybrid two-layer structure that reduces the index size to linear while keeping high query and update efficiency. Next, we present our indexing scheme RangePQ that supports efficient range-filtered ANN searches and index updates.

3.1 Index Scheme

Next, we elaborate on how our structure addresses range filtered queries. 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 ([50, 51]). *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 $O(K)$ changes in the nodes of the rotated subtree, significantly altering cluster IDs and making updates prohibitively expensive. To tackle this issue, we will use a weight-balancing strategy to avoid frequent rebalancing. We will show that with such a weight-balancing strategy, 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. The proposed method is an independent component based on the PQ index, which means that it will not modify its internal structure and affect other functionalities of the systems.

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. This means that even if a lot of objects have the same attribute value, they will have no effect on the balance of the tree. Therefore, the proposed scheme is insensitive to the distribution of the attributes of the objects. 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.

Example 3.2. Continuing with the object from Example 2.4, the initial object set is $\mathcal{O} = \{\mathbf{o}_1, \mathbf{o}_2 \ldots, \mathbf{o}_{15}\}$. To construct a corresponding index $\mathcal{T}$, its construction process is similar to that of a standard BST. First, we sort the objects according to their attribute values, and the sorted order is $\{\mathbf{o}_3, \mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_7, \mathbf{o}_5, \mathbf{o}_8, \mathbf{o}_{11}, \mathbf{o}_6, \mathbf{o}_{12}, \mathbf{o}_4, \mathbf{o}_9, \mathbf{o}_{10}, \mathbf{o}_{14}, \mathbf{o}_{13}, \mathbf{o}_{15}\}$. Then we select the median element $\mathbf{o}_6$ as the root node and proceed recursively to create the left and right subtrees of the root node. The built structure is shown in Fig. 2 (a). When the tree structure of index $\mathcal{T}$ is created, we backtrack up from the leaf nodes, gradually updating the auxiliary information of each node by aggregating the information of the child nodes during the backtracking process. As an example, the *SP* set of the node $\mathbf{o}_{10}$ includes the cluster IDs $\{1, 4, 5\}$. This is because the *SP* set of $\mathbf{o}_4$ includes $\{4, 5\}$ (since $\mathbf{o}_4$ and $\mathbf{o}_9$ are in Cluster 4, and $\mathbf{o}_{12}$

Algorithm 1: RangePQ-Query($\mathcal{T}$, $\boldsymbol{q}$, $[\ell, r]$, k)

```
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($u$, $\ell$, $r$, $C$, $NS$):
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         $IndexSetUnion(u.right, \ell, r, NS)$;
```

Algorithm 2: SearchByCCenters($\boldsymbol{q}$, C, $[\ell, r]$, R, NS)

```
1  for each $i \in C$ do
2      Compute asymmetric distance between $\boldsymbol{q}$ and coarse center $\boldsymbol{c_i}$
         by using distance table;
3  Sort cluster IDs in $C$ in increasing order of distance to $\boldsymbol{q}$;
4  for each $i \in C$ in increasing order of distance from $\boldsymbol{c_i}$ to $\boldsymbol{q}$ do
5      $cnt_{old} \leftarrow |R|$;
6      while true do
7          $\boldsymbol{np} \leftarrow FetchNewObject(NS, i, j = |R| - cnt_{old} + 1, \ell, r)$
8          if $\boldsymbol{np} == \emptyset$ then break;
9          Append $\boldsymbol{np}$ into $R$ and compute the asymmetric distance
             between $\boldsymbol{np}$ and $\boldsymbol{q}$ by using distance table;
10         if $|R| == L$ then return search results $R$;
11 return $R$;
12 procedure FetchNewObject($NS$, $i$, $cnt$, $\ell$, $r$):
13     for node $u \in NS$ do
14         if $[u.lp, u.rp] \subseteq [\ell, r]$ then
15             if $cnt \leq u.num[i]$ then
16                 return $FindObjectFromNode(u, i, cnt)$ ;
17             $cnt \leftarrow cnt - u.num[i]$;
18         else
19             if $u.P == c$ then
20                 if $cnt == 1$ then return $u.o$;
21                 $cnt \leftarrow cnt - 1$;
22     return $\emptyset$ ;
23 procedure FindObjectFromNode($u$, $i$, $cnt$):
24     Fetch the $cnt$-th object in cluster ID $i$ from the subtree rooted
         at $u$ using $u.num[i]$, at node $u$.
```

is in Cluster 5), and the SP set of $\mathbf{o}_{14}$ includes $\{1\}$ (as $\mathbf{o}_{13}, \mathbf{o}_{14}, \mathbf{o}_{15}$ are all in Cluster 1). Therefore, by merging the cluster IDs of the children and the cluster ID of $\mathbf{o}_{10}$, its SP set is derived as $\{1, 4, 5\}$.

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

Definition 3.3 (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 3.4. *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 3.5 ([14]). *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; 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 11-14). 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 with pseudo-code 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 1-2). After calculating the distances, all the clusters are sorted in increasing order of distances (Line 3). Next, we traverse the clusters with ID in C based on distance order. For each cluster, we perform the fetching and distance computation of the relevant objects (Lines 4-10). 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 7). 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 13-21). 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 17). 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 9). When the size of set R reaches the set parameter L, we return the set R directly (Line 10). Finally, the result set R is returned. We have the following theorem for query time complexity of RangePQ.

Theorem 3.6. *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 include 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 3.7. 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 $\boldsymbol{o}_8$ 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*.

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 [14];
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($\mathcal{T}$, $\boldsymbol{e}$)

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

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 number of elements covered by the query range r_Q. We define L as $max(L_{base} \cdot \frac{r_Q}{r_{base}}, L_{base})$ where r_{base} is the base parameter for the number of elements covered. 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

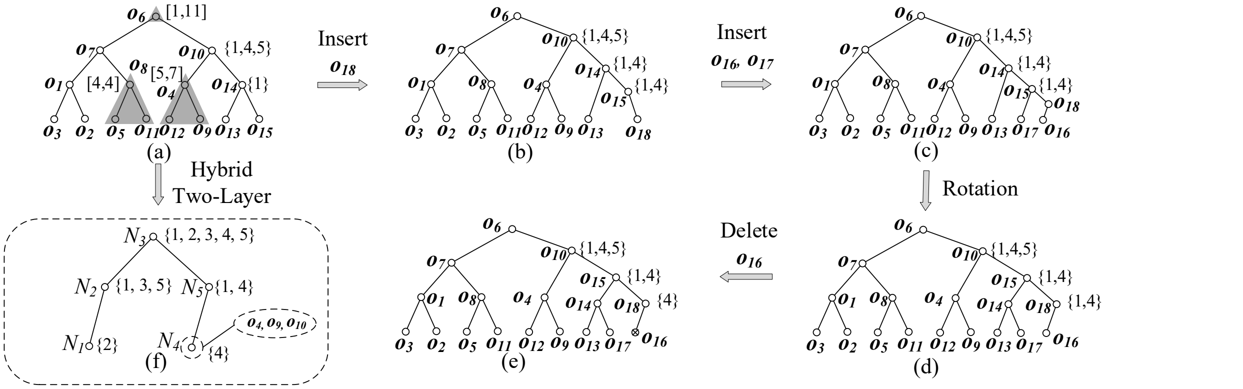


Figure 2: An example of updates with BST ($\alpha = 0.2$). In (a), the objects with attributes are $\{(\mathbf{o}_1, 2), (\mathbf{o}_2, 2), (\mathbf{o}_3, 1), (\mathbf{o}_4, 6), (\mathbf{o}_5, 4), (\mathbf{o}_6, 5), (\mathbf{o}_7, 3), (\mathbf{o}_8, 4), (\mathbf{o}_9, 7), (\mathbf{o}_{10}, 8), (\mathbf{o}_{11}, 4), (\mathbf{o}_{12}, 5), (\mathbf{o}_{13}, 8), (\mathbf{o}_{14}, 9), (\mathbf{o}_{15}, 11)\}$, where the second value in each tuple is the attribute of the corresponding object. Then the set is inserted with objects $(\mathbf{o}_{16}, 11), (\mathbf{o}_{17}, 10), (\mathbf{o}_{18}, 12)$.

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 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 3.8. *The RangePQ index $\mathcal{T}$ can handle each insertion in $O(\log n)$ 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 3.5, 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 1-5). 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 6). At the same time, the number inv of invalid nodes v in the tree is updated, and this value is set to 0 when the tree is first created. 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 and set inv to 0 (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.

The existing PQ-index can easily handle updates with the pre-built coarse clusters and sub-codebooks. When an object is inserted, it is inserted by traversing all clusters, finding the nearest one and coding it according to the existing codebooks. When deleting an object, we can use auxiliary structures (e.g., a hash table) to directly locate the cluster where the object needs to be deleted. When the number of updated objects exceeds $\Theta(n)$, it is typically necessary to rebuild the entire PQ-index to ensure query efficiency. The rebuild of the BST layer can be completed in $O(n \log n)$ time. Hence, the amortized update time can also be bounded by $O(\log n)$. Since the PQ index is maintained globally in the vector database, it needs to be updated in the system. Consequently, in our theoretical analysis, we do not charge the update cost of PQ-index in the update costs of RangePQ. But in our experiment, we report the end-to-end update time, which includes the time to update the PQ-index.

Theorem 3.9. *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 $O(\log n)$. □

Algorithm 5: RangePQ+ Query($\mathcal{T_H}$, $\boldsymbol{q}$, $[\ell, r]$, k)

```
1  $C \leftarrow \emptyset, R \leftarrow \emptyset, NS \leftarrow \emptyset$;
2  $HybridIndexSetUnion(root(\mathcal{T_H}), \ell, r, C, NS)$;
3  $HybridEndPointUnion(root(\mathcal{T_H}), \ell, \ell, r, C, NS)$;
4  $HybridEndPointUnion(root(\mathcal{T_H}), r, \ell, r, C, NS)$;
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 $HybridIndexSetUnion$;
```

Example 3.10. 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 shortly. 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. Next, for the compressed nodes, we create a tree index $\mathcal{T_H}$ to support ANN search. Similarly, for each node in $\mathcal{T_H}$, 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 ANN search 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, indexed using the coarse cluster ID of each object as the key. With this hash table, we can quickly extract objects from a given cluster at each node, facilitating query processing.

Algorithm 6: RangePQ+ Insertion($\mathcal{T_H}$, $\boldsymbol{e}$)

```
1  $Hybrid\text{-}Insert(root(\mathcal{T_H}), \boldsymbol{e})$;
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      Update auxiliary structures maintained by internal node $u$ with
         the coarse cluster ID of $\boldsymbol{e}$ and $attr(\boldsymbol{e})$;
10     $Hybrid\text{-}Insert(v, \boldsymbol{e})$;
11     $Maintain(u)$;
```

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 range, i.e., when only some of the objects in the node are contained within the range. Alg. 5 shows the pseudo-code for performing range queries on index $\mathcal{T_H}$. We initialize three sets *C*, *R*, and *NS*. Then, we obtain the union of cluster IDs of nodes in $\mathcal{T_H}$ 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 Alg. 1, replacing $IndexSetUnion$ with $HybridIndexSetUnion$ (Line 15). Note that in $\mathcal{T_H}$, some nodes may only have part of their objects contained within the range $[l, r]$. Fortunately, these cases only occur in nodes 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

Algorithm 7: RangePQ+ Deletion($\mathcal{T}_{\mathcal{H}}$, $\boldsymbol{e}$)

```
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 $e$ and $attr(e)$;
4     if $attr(e) < u.Clp$ then $u = u.left$;
5     else $u = u.right$;
6 Delete $e$ from node $u$ and update $inv$ according to the number of
    objects in node $u$;
7 if $2 \cdot inv > size(root(\mathcal{T}_{\mathcal{H}}))$ then Rebuild the index $\mathcal{T}_{\mathcal{H}}$;
```

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 $\mathcal{T}_{\mathcal{H}}$ 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 each node u, we fetch the objects by the corresponding hash table $u.HT$.

Theorem 3.11. *The RangePQ+ index $\mathcal{T}_{C}$ 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 $\mathcal{T}_{\mathcal{H}}$, 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 3.9. 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 geometric series. The space occupied by other auxiliary structures on the tree is also $O(n)$. Hence the space cost is $O(n)$. □

Example 3.12. 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 $2\cdot\epsilon$, 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 9). Then, we continue to search for the node v that needs to be inserted recursively (Line 10). Finally, during the backtracking phase, we use *Maintain* to keep the tree balanced (Line 11). We have the following theorem for the time complexity of insertion.

Theorem 3.13. *The RangePQ+ index $\mathcal{T}_{\mathcal{H}}$ can handle each insertion in $O(\log\zeta)$ 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 6). 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 7).

Theorem 3.14. *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 databases and information retrieval, with over 30 years of research [6]. Current ANN algorithms are categorized into four types:

Locality Sensitive Hashing (LSH). LSH [28] indexes high-dimensional data by hashing similar items into the same buckets with high probability. Variants like E2LSH [17] and FALCONN [3] support approximate high-dimensional retrieval with tunable performance and theoretical guarantees but incur significant redundancy and space overhead. The query and space costs of LSH are $O(n^{\rho})$ and $O(n^{1+\rho})$, respectively. While providing strong theoretical guarantees, LSH has higher practical space overhead compared to other methods [44]. Some studies focus on learning hash functions based on data characteristics [15, 47].

Tree-based indexes. These indexes mainly consider the design of recursively splitting the entire set $\mathbb{O}$ into subsets corresponding to the subtrees of the search tree. K-d tree [10], PKD-tree [49], FLANN [43], RPTree [16], and ANNOY [11], 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 [1] build a navigable graph, where nodes represent objects and edges connect similar ones. Methods like k-nearest neighbor graphs [18], monotonic search networks [21], and small world graphs [37, 38] start a query from a random node, traversing on the graph to find the node closest to the query. They excel in high-dimensional spaces in practice but often require high memory and preprocessing costs.

Vector Quantization (VQ). These methods compress vectors to enable approximate distance computations in a reduced space, thus lowering storage costs and speeding up ANN searches [8, 29]. Product Quantization (PQ) is a leading technique, frequently combined with inverted file indexes (e.g., IVFADC) for billion-scale search. PQ has also been applied in hardware acceleration [4, 13, 30].

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 with a single thread on a Linux machine with an Intel Xeon(R) CPU at 2.20GHz and 768GB of memory.

5.1 Experimental Settings

Datasets. We use the following three real-world datasets tested in related research [40, 59, 60]: (i) SIFT, which consists of 128-dimensional feature vectors extracted from multiple images. It contains up to one billion base vectors, and ten thousand query vectors. We use one million vectors from base vectors as the object set. (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 object set. (iii) WIT, which consists of 2048-dimensional embedding vectors generated using Wikipedia images through ResNet-50. It contains more than six million vectors, and we randomly sample one million vectors as the object set as the previous work [60]. Following the previous works [59, 60], 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. Each dataset contains one attribute, and we construct the index based on this attribute. To further test the query performance of the proposed method under datasets with skewed attribute values, we generated attribute values for SIFT using Zipfian distribution (shape parameter $a = 2$), denoted as SIFT-ZIP dataset.

Competitors and Evaluation Metrics. We include the following methods, all using PQ-index as the backbone, in our experimental comparisons: (i) Milvus [52] with `IVF_PQ` as its built-in index, (ii) RII [40], (iii) VBase [59], (iv) our proposed RangePQ with an $O(n \log n)$ space index as described in Section 3.1, and (v) RangePQ+, the hybrid two-layer index with linear space described in Section 3.3. Following the line of previous PQ work [7, 9, 23, 29, 31], we use *Recall@k* to measure the quality of range-filtered ANN search results. To provide a more comprehensive comparison, we also include three recent graph-based indices designed for range filtering queries: (vi) SeRF [60], (vii) SuperPostFiltering, dubbed as SuperPostFilter [20], and (viii) ACORN [45], a hybrid search method that employs a predicate-agnostic compression technique. This method can be regarded as a heuristic acceleration method based on post-filtering. Since graph-based methods maintain the original vectors, they are typically used in scenarios where higher query quality is required. These methods generally use recall to measure the quality of the results, which is defined as the proportion of true k-nearest neighbors among the k query results. When comparing with these methods, we follow this standard. It is important to note that SeRF and SuperPostFiltering do not support dynamic updates. We will demonstrate that our proposed RangePQ+ achieves comparable query performance to graph-based indices while also supporting dynamic updates and offering better scalability.

Parameters. For parameters, we set the size ϵ of objects accommodated per node in RangePQ+ to 10^4. For the number L of objects computed, for SIFT and WIT (resp. GIST) datasets, we set the base value L_{base} to 10^3 (resp. 3×10^3). For the base parameter r_{base}, we set it to 10^5. For the number M of subspaces in PQ-codes for all methods, we set it to $d/4$, where d is the number of dimensions of the input dataset. For all indices, we set K to $\sqrt{n}$. We also conduct experiments for parameters L_{base}, M, and ϵ, which will be elaborated on later. For k in range-filtered ANN query, we set it to 10. For VBase and RII, we keep the common parameters the same and 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 has a recall@10 over 0.8. For graph-based methods, all methods use parameters as specified in their respective papers to gain a recall above 0.9.

5.2 Comparison with PQ-Index Methods

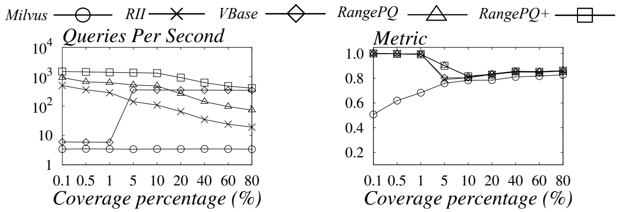
Exp 1: Query performance. In the first set of experiments, we test the range filtered ANN search of all indices. Fig.s 3-5 show the query performance of all indices across all datasets. For the SIFT and GIST datasets, we use their provided query vectors. For the WIT dataset, we randomly sample 1000 vectors 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 queries per second (QPS) and the Recall@10 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 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 QPS is dramatically increased, 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+. Fig. 6-7 show the query performance on SIFT-ZIP and GIST-ZIP, with skewed attribute values. Consistent with earlier results, RangePQ+ significantly outperforms other methods and demonstrates robustness to the distribution of attribute values.

Exp 2: Update efficiency. Next, we test the update performance of all indices. We test the average time it took to insert and delete 10^4 objects, where the objects were randomly selected from the data set. Fig. 8 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 because the experiments show end-to-end time, where the main time cost of insertion is spent on finding the coarse clusters to which the objects belong, 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. Fig. 9 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. Fig. 11 shows the impact of dataset updates on the query performance of all methods. We report QPS for all methods while achieving the same metric level of search results as in Exp. 1, across varying numbers of object updates. The results show that RangePQ+ achieves stable query performance after updates, verifying its ability to handle dynamic updates to the dataset effectively.

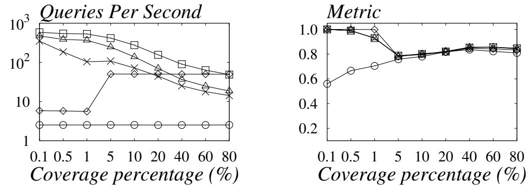
Exp 3: Index size and scalability test. Fig. 10 shows the index size of all methods across all datasets. The index size of RangePQ+ is significantly less than that used by RangePQ and comparable to that of RII and VBase, as these methods also use linear space indexes. Milvus additionally constructs other indexes for different types of hybrid query processing, resulting in slightly higher space cost compared to other methods. All methods process range filtered ANN search in small space by utilizing PQ. The index size of the common PQ index is around 10% of the space of original vectors.

Then, we assess the scalability of these methods using SIFT datasets ranging from 1 million to 100 million vectors. Fig.s 12(a) and (b) respectively show the queries per second (QPS) when achieving the same required quality of search results and the index size for different methods as the data size increases. For methods that cannot build an index within three days, we omit the corresponding results. As shown in Fig. 12(a), RangePQ+ achieves a steady advantage. As the dataset becomes larger, the advantage of RangePQ+ becomes more obvious. When the dataset size reaches 100 million, RangePQ+ is more than an order of magnitude faster than VBase and RII. This demonstrates the excellent scalability of RangePQ+. Fig. 12(b) shows the index sizes of the different methods when the size of the dataset increases. RangePQ+ has a comparable index size with other methods while saving a lot of space than RangePQ.



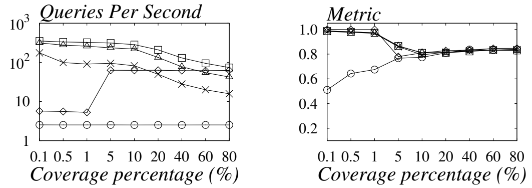
(a) Queries per second (b) Recall@10

Figure 3: Query performance on SIFT dataset.



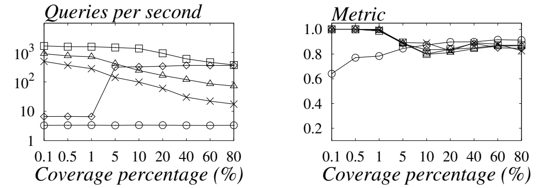
(a) Queries per second (b) Recall@10

Figure 4: Query performance on GIST dataset.



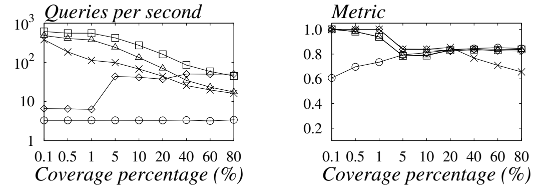
(a) Queries per second (b) Recall@10

Figure 5: Query performance on WIT dataset.



(a) Queries per second (b) Recall@10

Figure 6: Query performance on SIFT-ZIP dataset.



(a) Queries per second (b) Recall@10

Figure 7: Query performance on GIST-ZIP dataset.

5.3 Comparison with Graph-based Methods

Based on the previous experiments, we observe that our proposed method, RangePQ+, outperforms existing quantization-based approaches. To further demonstrate its effectiveness, we compare RangePQ+ with graph-based indices specifically designed for range-filtered ANN search. For a fair comparison, we integrate RangePQ+ with the state-of-the-art quantization method from [22] and employ reranking [53] and fastscan [5] techniques to enhance query performance. Fastscan technology in PQ-index, akin to prefetching in graph-based ANNS [1], is a common technique [22, 27]. The PQ-index is initially developed for scenarios with limited storage capacity. Since graph indexes typically store original vectors for

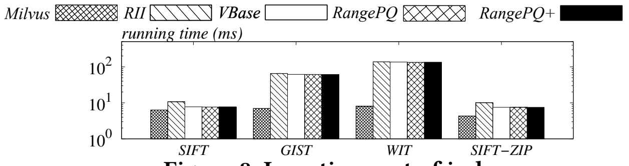


Figure 8: Insertion cost of index.

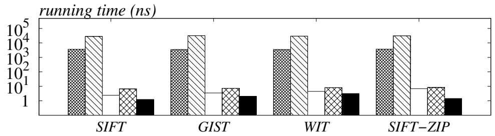


Figure 9: Deletion cost of index.

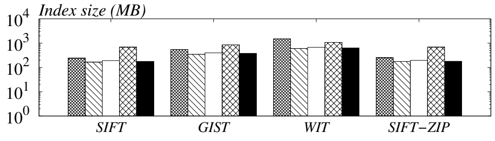
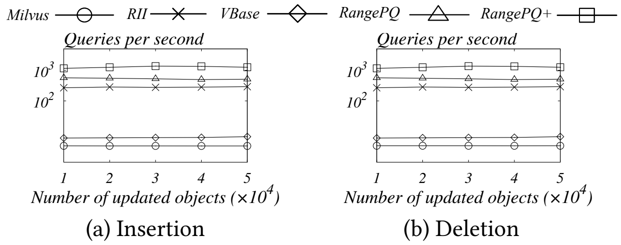


Figure 10: Index size.

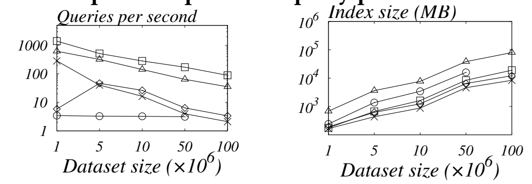
searching, they require a larger space overhead and provide a high recall of query results. In scenarios requiring high recall, PQ indexes can employ the exact distance of original vectors for re-ranking to enhance the quality of query results [22, 30].

Exp 4: Query efficiency. We evaluate the performance of our RangePQ+ against competitors on GIST and WIT datasets using range filters with varying coverage. Fig. 13 and Fig. 14 shows the QPS and recall on GIST and WIT datasets, respectively. In Fig. 13, RangePQ+ shows a stable QPS performance consistently maintaining a recall of query results above 0.9. In contrast, SeRF and ACORN fail to return search results with recall higher than 0.9 within 100ms for certain queries with less than 1% range coverage, due to the structure information loss in their pruning strategies. We observed that SuperPostFilter failed to achieve a recall above 0.9 within 100ms for some small ranges, e.g., ranges with 0.1% coverage, on the WIT dataset. This issue may stem from precision degradation caused by its attribute value storage method, which uses floats, resulting in inaccuracies for small ranges. Although SuperPostFilter exhibits certain advantages over RangePQ+, its efficiency gain came at the cost of a significantly larger index size. Specifically, on the GIST (or WIT) dataset, the index sizes for SeRF, SuperPostFilter, ACORN and RangePQ+ were 0.39GB (or 0.38GB), 10.6GB (or 15.2GB), 4.3GB (or 4.1GB) and 0.13GB (or 0.12GB), respectively. Notably, SuperPostFilter required up to 100 times more index space than RangePQ+, highlighting the space efficiency of our approach. We can find that RangePQ+ achieves stable performance, comparable to graph-based methods. It can be further observed that graph-based methods are more sensitive to dataset characteristics, as graph structures are significantly influenced by the dataset. For instance, the performance of graph-based indices differs significantly on GIST and WIT datasets. In contrast, quantization methods like RangePQ+ demonstrated greater stability across different datasets. Considering lightweight nature of RangePQ+ and its ease of updates, it provides an excellent balance between index size, query efficiency, and update capability. We also conduct additional experiments where coverage was held fixed at 0.5%, 1%, 5%, and 10% on the GIST and WIT datasets, then analyzed the trade-off between QPS and recall in figures 15-18. Nonetheless, these figures indicate that RangePQ+ provides stable performance comparable to graph-based methods, while offering a lightweight index structure and efficient dynamic updates, making it particularly well-suited to dynamic settings.



(a) Insertion (b) Deletion

Figure 11: Impact of updates to query performance on SIFT.

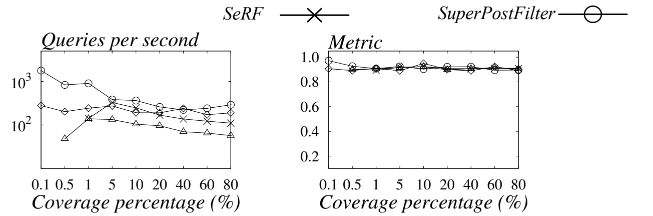


(a) Range 1% (b) Index size

Figure 12: Varying the size of the SIFT dataset.

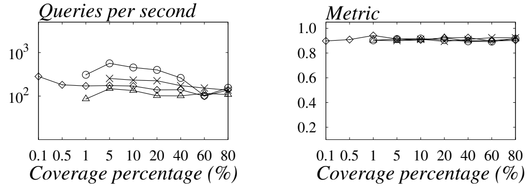
Exp 5. Impact of updates. Next, we examine the impact of updates on our RangePQ+ index compared to state-of-the-art graph-based indices. In this experiment, we report the QPS for RangePQ+ at the same required recall level of search results across varying numbers of updates. In other words, we evaluate the cost of maintaining the same recall during dynamic updates. Fig. 19(a) and Fig. 19(b) show the QPS as we gradually insert and delete 5×10^4 records, respectively. As we can observe, the QPS of RangePQ+ and ACORN are insensitive to dynamic updates since both methods can support index updates to reflect the latest datasets. Yet, the other two will have degraded QPS with the increasing number of updated objects. The reason is that these methods are static and cannot include the changes into their indexes. To reach the same level of recall, it hence needs to visit more objects and hence a degration of the QPS. This again confirms our RangePQ+ is a better choice for dynamic scenarios, especially when a light-weighted index is desired.

Exp. 6: Scalability test. Finally, we assess the scalability of the four algorithms in our experiments. We sample datasets ranging from 1 million to 100 million vectors from the SIFT dataset to examine their scalability. Similar to the previous experiments, Fig.s 20(a) and (b) show the QPS and index sizes for which the different methods satisfy the recall requirement as the data size increases, respectively. For methods that cannot build an index within three days, we omit the corresponding results. Firstly, we observe that graph-based indices are generally less scalable compared to PQ-based indices. Specifically, SeRF, Superposfiltering, and ACORN fail to complete index processing within three days for dataset sizes of 100 million, 10 million, and 50 million, respectively. Besides, as observed in Fig. 20(a), SuperPostFilter achieves high QPS but at the expense of a significantly larger index size—over 70 times larger than our RangePQ+ when the data size is 5 million, as shown in Fig. 20(b). RangePQ+ and SeRF exhibit comparable QPS, but RangePQ+ consumes four times less index space than SeRF. Besides, SeRF does not support dynamic updates. Additionally, RangePQ+ offers a significant advantage over ACORN. When the dataset size reaches



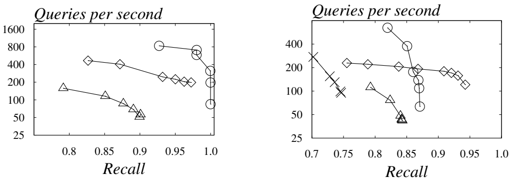
(a) QPS (b) Recall

Figure 13: Query performance on the GIST dataset.



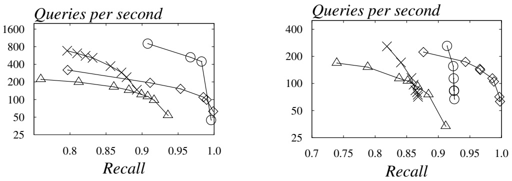
(a) QPS (b) Recall

Figure 14: Query performance on the WIT dataset.



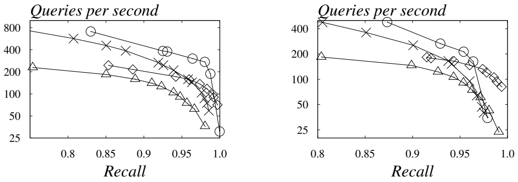
(a) GIST (b) WIT

Figure 15: QPS vs Recall on 0.5% range.



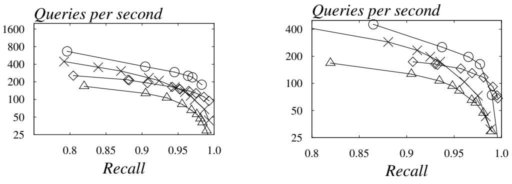
(a) GIST (b) WIT

Figure 16: QPS vs Recall on 1% range.



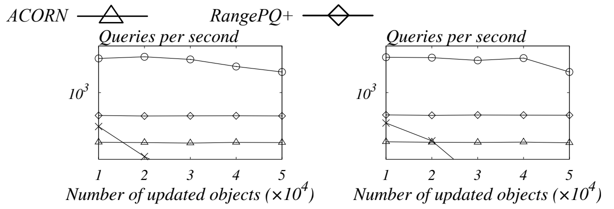
(a) GIST (b) WIT

Figure 17: QPS vs Recall on 5% range.



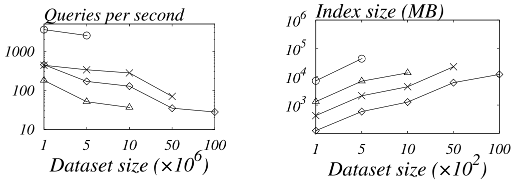
(a) GIST (b) WIT

Figure 18: QPS vs Recall on 10% range.



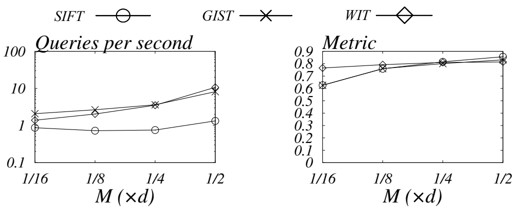
(a) Insertion (b) Deletion

Figure 19: Impact of updates to query performance on SIFT.



(a) Range 1% (b) Index size

Figure 20: Varying the size of the SIFT dataset.

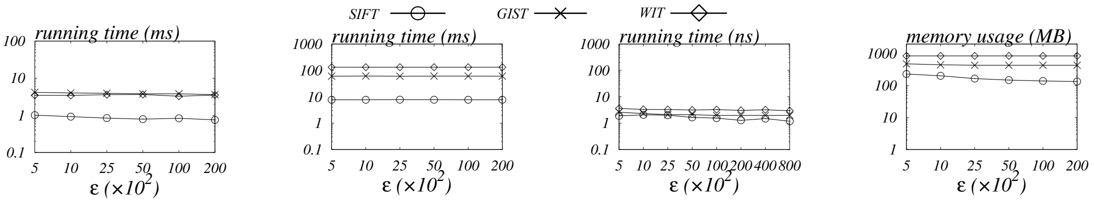


(a) Query latency (b) Recall@10

Figure 21: Impact of parameter M on all datasets.

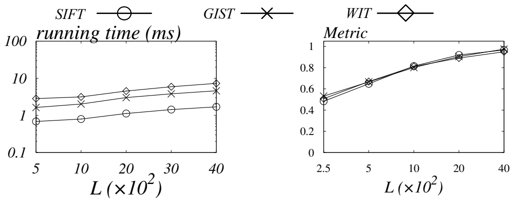
10 million, RangePQ+ has three times the QPS of ACORN and its index size is 7 times smaller than ACORN. This demonstrates that RangePQ+ offers a superior trade-off between query efficiency and index size. Moreover, RangePQ+ shows remarkable efficiency in index construction time, building the index in under one minute for all dataset sizes. In comparison, SeRF requires over 9,000 seconds to build the index when the data size is 5 million, while SuperPostFilter (resp. ACORN) takes over 90,000 (resp. 16000) seconds for the same dataset size. This substantial difference underscores the scalability and efficiency of our method. Moreover, the standard ANN indices used by RangePQ+, such as the PQ-index and the one described in [22], can be constructed in less than one day when the data size is 100 million, which is faster than the construction time required by SeRF. Since these indices are necessary to support basic ANN searches (excluding range-filtered queries), it is reasonable to assume that they are already available. Besides, RangePQ+ achieves comparable query performance to SeRF while being lightweight and supporting updates, a feature not available in SeRF and SuperPostFilter. While ACORN supports dynamic updates, its QPS is far lower than that of our RangePQ+. These make RangePQ+ a highly efficient and flexible solution for large-scale applications.

Exp 7: 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 [29, 40], we set the number of codes Z per codebook in each subspace to 256. Fig. 21 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 22: Impact of parameter ϵ on all datasets



(a) Query latency (b) Recall@10

Figure 23: Impact of parameter L on all datasets.

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

Exp 8: 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 22 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 9: Impact of parameter L. In this set of experiments, we test the impact of changing the number of accessed objects, L, on the results under a fixed query range coverage percentage. This will be used to guide how to set the base parameter L_{base} for RangePQ+. Figure 23 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 the SIFT and WIT (resp. GIST) dataset with L set to $\{250, 500, 1000, 2000, 4000\}$ (resp. $\{750, 1500, 3000, 6000, 12000\}$). We find that when L is set to 1000 (resp. 3000) for the SIFT and WIT (resp. GIST) dataset, the recall performance is satisfactory. Therefore, we set L_{base} to the corresponding value.

6 FUTURE WORK

In this section, we discuss potential future extensions, including multithreaded processing, handling limited memory environments, and supporting more complex range filters.

Concurrent multithreaded processing. Currently, our solution works for single-core environment. To extend to multithreaded environments, our proposed method could employ fine-grained locking (e.g., mutexes) at each node, allowing concurrent reads but risking contention at high concurrency. For RangePQ+, a promising direction is to adopt a *two-layer hierarchical locking* scheme, which introduces different levels of locks for upper-layer operations and lower-layer nodes to balance contention and access control. Additionally, auxiliary structures—such as the *SP* sets—could be transitioned to lock-free implementations using techniques like compare-and-swap operations. Future research may explore more optimized concurrent update strategies, such as multi-layer hierarchical locking to reduce conflicts and load balancing methods to ensure an even distribution of workloads across the structure.

Extension of external memory. For memory-limited scenarios, partitioning the dataset and index is crucial. In this setting, vectors are stored on external storage, while in-memory BSTs maintain cluster metadata to facilitate efficient lookups. The vectors are partitioned by clusters and fetched on demand, minimizing I/O overhead. Caching strategies (e.g., LRU and prefetching) can be explored to boost performance. Alternatively, disk-friendly structures (e.g., B-tree or B^+-tree) may replace BSTs, with dynamic updates handled via deferred updates or incremental rebalancing. Write-optimized structures (e.g., append-only logs) could temporarily buffer changes before merging, and adaptive caching coupled with parallel I/O offers additional opportunities for optimization.

More complicated range-filtered queries. RangePQ+ mainly focuses on continuous range filters, but we plan to expand our approach to more complex scenarios, including unions and intersections of multiple range filters as well as non-numeric attributes. For unions, we can merge sorted ranges into non-overlapping segments; for intersections, we can similarly combine the relevant portions. The resulting sets of coarse clusters can then be processed via our *SearchByCCenters* method. In the case of non-numeric attributes (e.g., strings), we can map them (e.g., lexicographically) to numeric values to convert the problem into a range-filtered ANN search. These extensions will be a key direction for future work. Additionally, exploring multi-attribute range-filtered ANN search presents an interesting direction for future work.

7 CONCLUSIONS

In this paper, we study the range filtered ANN search problem and propose a lightweight RangePQ+ index, that efficiently supports range filtered ANN search and dynamic updates. Extensive experiments show the effectiveness of our RangePQ+.

REFERENCES

[1] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore L. Willke. 2023. Similarity search in the blink of an eye with compressed indices. *Proc. VLDB Endow.* 16, 11 (2023), 3433–3446.

[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] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2016. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. In *VLDB*, Vol. 9. 12.

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

[7] Artem Babenko and Victor S. Lempitsky. 2012. The inverted multi-index. In *CVPR*. 3069–3076.

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

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

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

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

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

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

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

[15] 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*.

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

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

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

[19] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]

[20] Joshua Engels, Benjamin Landrum, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2024. Approximate Nearest Neighbor Search with Window Filters. *arXiv preprint arXiv:2402.00943* (2024).

[21] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.

[22] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 3 (2024), 167.

[23] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. 2946–2953.

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

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

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

[27] 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*.

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

[29] 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.

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

[31] Yannis Kalantidis and Yannis Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. 2329–2336.

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

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

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

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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).

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

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

[45] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proc. ACM Manag. Data* 2, 3 (2024), 120.

[46] 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.

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

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

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

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

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

[52] 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.

[53] Jingdong Wang, Ting Zhang, Nicu Sebe, Heng Tao Shen, et al. 2017. A survey on learning to hash. *TPAMI* 40, 4 (2017), 769–790.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

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

[59] 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.

[60] 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.