DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter

Anonymous Author(s)

ABSTRACT

Recent advancements in AI have enabled models to map real-world entities, such as product images, into high-dimensional vectors, making approximate nearest neighbor search (ANNS) crucial for various applications. Often, these vectors are associated with additional attributes like price, prompting the need for range-filtered ANNS where users seek similar items within specific attribute ranges. Naive solutions like pre-filtering and post-filtering are straightforward but inefficient. Specialized indexes, such as SeRF, SuperPostFiltering, and iRangeGraph, have been developed to address these queries effectively. However, these solutions do not support dynamic updates, limiting their practicality in real-world scenarios where datasets frequently change.

To address these challenges, we propose *DIGRA*, a novel dynamic graph index for range-filtered ANNS. DIGRA supports efficient dynamic updates while maintaining a balance among query efficiency, update efficiency, indexing cost, and result quality. Our approach introduces a dynamic multi-way tree structure combined with carefully integrated ANNS indices to handle range filtered ANNS efficiently. We employ a lazy weight-based update mechanism to significantly reduce update costs and adopt optimized choice of ANNS index to lower construction and update overhead. Experimental results demonstrate that DIGRA achieves superior trade-offs, making it suitable for large-scale dynamic datasets in real-world applications.

ACM Reference Format:
Anonymous Author(s). 2025. DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter. In *Proceedings of International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In today's landscape of big data and AI, effectively managing high-dimensional vectors is essential for modern applications like recommendation engines [22], image retrieval [16], and natural language processing [10, 26]. Machine learning models convert various data types into vector representations, enabling real-world applications to perform queries within vector spaces [17, 21, 27, 29]. A key task in this domain is identifying the nearest vector to a given query vector, which is critical for functionalities such as object recognition [35] and personalized content delivery [8].

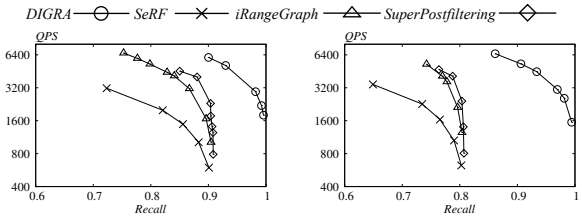
However, as vector dimensionality increases, traditional exact search methods become inefficient due to the curse of dimensionality, making approximate nearest neighbor search (ANNS) a more practical solution. ANNS techniques enable the efficient retrieval of vectors similar to a query vector, supporting large-scale vector search operations. Vector databases like Milvus [31] and AnalyticDB [33] enhance search precision by integrating ANNS with attribute-based filtering. For example, e-commerce platforms can provide product recommendations based on image similarity while also applying filters for price or user age [28, 31, 38]. To handle high-dimensional vectors and range-filtered queries effectively, modern vector databases utilize various algorithms to accelerate ANNS with range filters. Although combining ANNS with range filtering is crucial for many applications, existing methods often fall short in addressing this challenge efficiently.

Common approaches to manage such queries involve pre-filtering or post-filtering techniques. Pre-filtering filters the dataset based on attribute criteria and then conducts nearest neighbor search. Post-filtering, conversely, conducts ANNS first and then filters the results according to the attribute constraints. These methods are straightforward to implement and support dynamic updates to the dataset, making them compatible with database systems and widely adopted in various vector databases [31, 33, 38]. However, they exhibit sub-optimal performance on large datasets. Pre-filtering can be time-consuming due to the need for extensive preliminary scans, while post-filtering may miss many similar objects, leading to delays in query completion.

To gain better query efficiency for ANNS with range filters, recent research has designed specialized algorithms that construct dedicated indexes to enable efficient query processing. One such method is SeRF [39], which addresses ANNS with range filters by utilizing Hierarchical Navigable Small World (HNSW) graphs. HNSW is a popular index structure for ANNS that organizes data points into a hierarchical, multi-layered graph, allowing efficient navigation through high-dimensional spaces by connecting vertices (vectors) via edges based on proximity. This structure facilitates fast approximate searches by traversing from higher levels of the hierarchy down to the most relevant vertices. In SeRF, the approach involves compressing $O(n^2)$ HNSW graphs—each corresponding to different range filters—into a single, unified graph. This method theoretically incurs a space overhead of up to $O(n^2M)$, where M is a parameter in HNSW that controls the maximum number of connections per vertex, significantly impacting memory usage. To mitigate this substantial space requirement, SeRF employs a compression technique called MaxLeap, which reduces the number of edges by allowing longer-range connections. However, this compression leads to the loss of crucial graph structure information, resulting

SIGMOD '25, June 2025, Berlin, Germany

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX



(a) 10% insertion (b) 20% insertion

Figure 1: Impact of insertion on SIFT.

in suboptimal query performance. Additionally, because the construction of SeRF must follow a specific attribute order, inserting or deleting objects affects many subsequent HNSW constructions, rendering it incapable of supporting dynamic updates.

SuperPostFiltering [12] addresses the efficiency issue by creating graph indices for various predefined range combinations. During the query phase, it selects the appropriate pre-constructed index to answer the query. While this method can improve query efficiency, it generates a large number of redundant graph indices, leading to significant memory overhead. Moreover, when objects are inserted or deleted from the dataset, the set of maintained ranges changes substantially, making it impractical to support dynamic updates. A more recent method, iRangeGraph [36], utilizes a segment tree to manage indices, which reduces overlaps between graph indices and achieves both stable query performance and reduced space overhead. However, segment trees are inherently static structures that cannot efficiently handle updates. As noted in [36], iRangeGraph does not support dynamic insertion or deletion of objects, limiting its applicability in environments where datasets change over time.

In real-world scenarios, vector databases are dynamic, with new data frequently added and old data removed. For instance, as illustrated in Fig. 1(a) (resp. Fig. 1(b)), starting with 90% (resp. 80%) vectors from a base dataset of 1 million SIFT vectors, the addition of the other 10% (resp. 20%) vectors significantly degrades the search efficiency of static indexes like SeRF, irangeGraph, and SuperPostFiltering when handling range-filtered ANNS. Specifically, to achieve a high recall rate of 90%, the Queries Per Second (QPS) performance deteriorates markedly. This degradation becomes more pronounced with the insertion of 20% new data. Results on other datasets are similar, as seen in Sec. 5. In fact, when a substantial portion of new elements is added, relying on the static index built on the original dataset leads to significant drops in recall. While one might consider periodically rebuilding the index to handle updates, this approach is impractical due to the high computational costs and resource consumption involved in rebuilding large-scale indices. Moreover, determining the optimal frequency for rebuilding is challenging: Rebuilding too often incurs excessive overhead, while doing it infrequently leads to outdated indices and degraded query performance.

To address this limitation, we propose **DIGRA**[1], an effective index that achieves a superior trade-off among query efficiency, update efficiency, indexing cost, and query result quality (in terms of recall). As we can see from Figure 1, our DIGRA, which supports dynamic index updates, consistently maintains high query efficiency and high recall, even as new data is added, underscoring the importance of supporting index updates.

[1]Dynamic Graph Index for Range-filtered ANN

Achieving a balance among all four aspects—query efficiency, update efficiency, indexing cost, and result quality—is challenging because previous methods struggle with updates due to their static nature or costly rebuilding processes when data changes. Specifically, existing solutions face difficulties because they: *(i)* encode the graph index and range information using static structures that are not amenable to efficient updates; *(ii)* build multi-level graph index structures that necessitate complete or partial rebuilding when the dataset changes, incurring significant update costs. To overcome the challenges, DIGRA includes the following innovative designs:

Dynamic multi-way tree structure. We employ a dynamic multi-way tree T to organize the index based on the attribute ϕ of range filtering. Each node in T uses ϕ as the key and stores mapped vector IDs as values. By utilizing split and merge operations similar to those in B-trees, we can adjust the tree structure efficiently during insertions and deletions. This dynamic approach allows the index to accommodate data updates seamlessly.

Efficient range query handling. For any arbitrary range $[\ell, r]$, we can efficiently identify at most two nodes in T in $O(\log n)$ time such that all vectors with attribute values within this range are stored in the subtrees of these two nodes. This property enables us to retrieve the vectors matching the range filter without scanning irrelevant parts of the tree, thereby enhancing query efficiency.

Integration of ANNS indices at tree nodes. To accelerate approximate nearest neighbor search (ANNS) within the filtered ranges, we build an ANNS index G_u (e.g., a Navigable Small World (NSW) or a Hierarchical Navigable Small World (HNSW) graph index) at each internal node u of T. This index is constructed over the vectors stored in the subtree rooted at u. During a range-filtered ANNS query, we first locate the relevant nodes u and v, perform ANNS searches using G_u and G_v, and then merge the results to obtain the top-k answers. This hierarchical indexing leverages the tree structure to improve search performance.

Lazy weight-based update mechanism. Maintaining the ANNS indices G_u at each node u in the multi-way tree T during dynamic updates presents a significant challenge, especially when split and merge operations modify the number of elements in affected subtrees. For instance, splitting a node can impact numerous descendants, necessitating the costly rebuilding of their ANNS indices. To address this, we introduce a *lazy weight-based update mechanism* that reduces the update cost to an amortized $O(c_{upd} \cdot \log n)$ per update, where c_{upd} represents the time complexity of inserting an object into an ANNS index. Additionally, we employ a background rebuilding strategy to eliminate amortization effects, ensuring consistent update performance without degrading query efficiency. This approach effectively balances the need for dynamic updates with the preservation of high query performance and recall quality.

Optimized choice of ANNS index. While it might initially seem advantageous to use HNSW graphs at each node of our multi-way tree—given HNSW's typically higher Queries Per Second (QPS) compared to other graph-index methods that support update like NSW while achieving similar recall rates [32], we recognize that HNSW's primary strength lies in its inherent hierarchical structure. Since our multi-way tree already provides a hierarchical organization, incorporating HNSW would result in redundant layering of hierarchies. Therefore, we opt for NSW graphs as the ANNS indices

at each node for several reasons. First, NSW graphs have significantly lower construction and update costs, being up to an order of magnitude less computationally intensive to build and maintain compared to HNSW graphs [32], thereby reducing overall indexing and update overhead. Second, the flat structure of NSW graphs aligns seamlessly with the hierarchical organization of the multi-way tree, facilitating efficient searches within subtrees without introducing unnecessary complexity.

In summary, we make the following contributions.

- By combining the multi-way tree and NSW structures, DIGRA achieves efficient range-filtered searches with high recall, without relying on redundant graph indices.
- With the structural design of DIGRA, we introduce an efficient lazy weight-based update mechanism, ensuring real-time index maintenance without the need for complete index rebuilding.
- We provide a rigorous theoretical analysis showing that the update overhead of DIGRA is only $O(\log n)$ times larger than that of inserting a point in a vanilla NSW, which translates to at least an $O(n/\log n)$ reduction in update costs compared to traditional approaches.
- Extensive experiments on real-world high-dimensional vector datasets demonstrate that DIGRA maintains excellent query performance, while reducing update overhead by five orders of magnitude compared to methods that require index rebuilding.

2 PRELIMINARIES

2.1 Problem Definition

Let O be a set of n vectors in $\mathbb{R}^d$. The distance between any two vectors $\mathbf{p}, \mathbf{q} \in \mathbb{R}^d$ is denoted by $\delta(\mathbf{p}, \mathbf{q})$, where δ could represent a metric such as the Euclidean distance. In high-dimensional spaces, many applications can be formulated as nearest neighbor search (NNS) problems, defined as follows:

Definition 2.1 (Nearest Neighbors Search (NNS)). Given an object set O consisting of n vectors, a query vector $\mathbf{q} \in \mathbb{R}^d$, and a positive integer $k \leq n$, the NNS returns a set $R^* = \{\mathbf{o}_1^*, \mathbf{o}_2^*, \ldots, \mathbf{o}_k^*\}$ of k vectors from O with the top-k smallest distances to $\mathbf{q}$.

Exact NNS in high-dimensional spaces is computationally expensive due to the curse of dimensionality. As the number of vectors and their dimensions required for NNS have increased significantly, exact NNS becomes impractical in many cases. Hence, approximate nearest neighbor search (ANNS) is widely used, which sacrifices less query quality for higher query efficiency. In the literature, *recall* is commonly used to measure the quality of ANNS. If R^* is the exact NNS result and R is the ANNS result, the recall is defined as $\text{Recall}(R) = \frac{|R \cap R^*|}{|R^*|} = \frac{|R \cap R^*|}{k}$. The goal of ANNS query processing is to maximize query throughput while keeping a high recall.

In many applications, each object in the set O is associated with a specific attribute (e.g., date, price), and users may want to perform NNS on objects whose attributes fall within a given range. For example, if vectors represent image embeddings and the attribute is the publication date, users might seek images similar to a given image that were published within a specific date range. This problem is known as NNS with a range filter, formally defined as follows:

Definition 2.2 (Nearest Neighbor Search with Range Filter). Given an object set O where each object $\mathbf{o} \in O$ has an attribute $\phi(\mathbf{o})$, a query vector $\mathbf{q} \in \mathbb{R}^d$, a positive integer k, and a range $Q = [\ell, r]$, the NNS with range filter returns the nearest neighbors to $\mathbf{q}$ within the set $O_Q = \{\mathbf{o} \in O \mid \mathbf{o} \in O, \phi(\mathbf{o}) \in Q\}$, i.e., the objects satisfying the range filter on attribute ϕ.

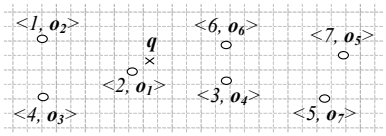


Figure 2: Example of an object set

For NNS with range filter queries, we also focus on the approximate version to gain high query efficiency while achieving high recall.

Example 2.3. As shown in Figure 2, consider a set of objects $O = \{\mathbf{o}_1, \mathbf{o}_2, \ldots, \mathbf{o}_7\}$ where each object is associated with an attribute value on ϕ. For example, $\mathbf{o}_2$ has an attribute value of 1 on ϕ. Given a query vector $\mathbf{q}$, represented by a cross, performing a NNS on O returns $\mathbf{o}_1$, as it is the closest to $\mathbf{q}$. When we perform a NNS with a range filter on attribute ϕ, where the query range is $[3, 6]$ and k is set to 2, the subset after applying the range filter is $O_Q = \{\mathbf{o}_3, \mathbf{o}_4, \mathbf{o}_6, \mathbf{o}_7\}$. Then, $\mathbf{o}_4$ and $\mathbf{o}_6$ are the query answers that meet the filter condition.

2.2 Graph-based ANNS and HNSW

Given a set O of n vectors in d-dimensional space $\mathbb{R}^d$, graph-based methods construct a directed proximity graph G, where each vertex represents a vector in O, and directed edges connect certain pairs of vertices. Let vertex u and vertex v corresponds to two vectors $\mathbf{o}_u$ and $\mathbf{o}_v$, respectively, with the distance between them, $\delta(u, v)$, typically defined using a distance metric in $\mathbb{R}^d$. The (out-)neighbors of a vertex u, denoted $N_G(u)$, are chosen based on proximity to u following a specific selection rule. Most graph-based methods utilize an identical greedy search algorithm, as shown in Alg. 1. For simplicity, we use $\delta(u, \mathbf{q})$ to denote the distance between $\mathbf{o}_u$ and the query vector $\mathbf{q}$. The ANNS begins at an initial entry vertex $entry$, returning ef vectors closest to $\mathbf{q}$ among the visited vertices. To find the final set of k nearest neighbors, where $k \leq ef$, the algorithm selects the k closest vectors from the returned set. For improved performance, it is recommended to set $ef \geq k$.

HNSW (Hierarchical Navigable Small World) is a popular graph-based method for ANNS. Unlike traditional methods that use a single proximity graph, HNSW leverages a multi-layer structure inspired by skip lists. Each layer contains a proximity graph, referred to as a Navigable Small World (NSW). If HNSW has L layers, the proximity graph at layer i is denoted as $H[i]$. The number of vertices decreases exponentially from the base layer $H[0]$ to the top layer $H[L]$, similar to the hierarchical structure of skip lists.

ANNS on HNSW. In HNSW, the search begins at the top layer and descends through the layers to reach layer 0, where the main search is performed. Starting from layer l, it finds the vertex closest to query $\mathbf{q}$ at each layer, using it as the entry point for the next layer, until reaching layer 0. Once at layer 0, a greedy search, $\text{GreedySearch}(H[0], ep_0, \mathbf{q}, ef)$, is applied to find the closest vertices to $\mathbf{q}$. Finally, it returns the k nearest objects from the search.

HNSW Construction. The HNSW graph is built by inserting vertices incrementally into an initially empty graph. Let M be the maximum out-degree parameter. When inserting a vertex u, the

Algorithm 1: GreedySearch

Input: Graph G, entry point $entry$, query vector $\mathbf{q}$, ef
Output: Query result
Mark $entry$ as visited and set $C \leftarrow \{entry\}$, $W \leftarrow \{entry\}$;
while $|C| > 0$ **do**
  $u \leftarrow$ extract nearest element from C to $\mathbf{q}$;
  $f \leftarrow$ get furthest element from W to $\mathbf{q}$;
  if $\delta(u, \mathbf{q}) > \delta(f, \mathbf{q})$ **then** break;
  for $v \in N_G(u)$ **do**
    if *v is not visited* **then**
      Mark v as visited;
      if $\delta(v, \mathbf{q}) < \delta(f, \mathbf{q})$ *or* $|W| < ef_s$ **then**
        $C \leftarrow C \cup \{v\}$, $W \leftarrow W \cup \{v\}$;
      if $|W| > ef_s$ **then**
        Remove f from W;
return W;

Algorithm 2: GraphInsertion

Input: Graph G, node u, entry node $entry$, Parameters M, ef_{con}
$C \leftarrow GreedySearch(G, u, entry, ef)$;
$N_G(u) \leftarrow prune(C, u, M)$;
for $v \in N_G(u)$ **do**
  $N_G(v) \leftarrow N_G(v) \cup u$;
  if $|N_G(v)| > M$ **then**
    $N_G(v) \leftarrow prune(N_G(v), v, M)$;
procedure *prune(C,u,M)*:
  Sort elements in C by the distance to u;
  $R \leftarrow \emptyset$;
  for $v \in C$ **do**
    if *not exists* $w \in R$ *s.t.* $\delta(v, w) < \delta(u, v)$ **then**
      $R \leftarrow R \cup \{w\}$;
      if $|R| = M$ **then**
        break;
  return R;

algorithm first determines the highest layer L_u in which u will appear, where L_u is determined by a truncated geometric distribution. After determining the layers, u is inserted into the proximity graphs from layer L_u down to layer 0. The maximum out-degree is $2M$ in layer 0 and M in all other layers. The pseudo-code about vertex insertion into a proximity graph is described in Alg. 2. First, the ef_c nearest vertices to u are selected as edge candidates and pruned to M edges, forming $N(u)$ (Lines 1-2). For each $v \in N(u)$, u is added to $N(v)$, and pruning is performed if $|N(v)| > M$ (Lines 3-6). We say an edge (u, v) is dominated if there exists an edge (u, w) in the graph s.t. $\delta(u, w) < \delta(u, v)$ and $\delta(v, w) < \delta(u, v)$. During pruning, candidates are sorted by distance to u (Line 8), and edges that are not dominated are iteratively added to set R (Lines 10-14).

2.3 Existing Solutions

Prefiltering. Prefiltering is a straightforward method. It pre-sorts all elements based on attribute ϕ. Given a query vector $\mathbf{q}$ and a range filter $[l, r]$, it first does a binary search on the sorted attribute list to find the start and end positions of elements satisfying the range filter, and then scans this subset, computes the distance between $\mathbf{q}$ and each element, and returns the closest k element.

Postfiltering. Postfiltering method builds an ANNS index on all data. For a query vector $\mathbf{q}$ and a range filter $[a, b]$ on attribute ϕ, it first retrieves the $K = k$ nearest neighbors using the ANNS index. The range filter is then applied to these elements to check if at least k of them satisfy the filter. If fewer than k elements meet the criteria, K is multiplied by a constant factor $m > 1$, and the ANNS query is repeated. This process continues until k elements satisfying the filter are found. The final result is the k closest elements to $\mathbf{q}$ that meet the range filter, based on the most recent ANNS query.

SeRF. An idea to perform range-filtered ANNS is to build a dedicated ANNS index for each possible query range on attribute ϕ. SeRF [39] is based on this concept, utilizing HNSW as the index for each range on attribute ϕ. To reduce redundancy, SeRF compresses the index by minimizing duplicate edge storage. However, this method leads to the creation of $O(n^2)$ indices, which becomes impractical for large datasets. SeRF addresses this issue by recognizing that certain edges are shared across indices for continuous ranges. When an edge appears in the indices for intervals $[x, y]$, for all $x \in [l, r]$ and $y \in [b, e]$, SeRF avoids storing the edge multiple times. Instead, it stores the edge once, associating it with the four values l, r, b, e. During a query with a range filter $[L, R]$, SeRF performs ANNS on the HNSW index comprising vertices that satisfy the range filter and edges where $L \in [l, r]$ and $R \in [b, e]$. If the attribute values are independent of the object set, the expected space overhead of SeRF is $O(Mn \log n)$. However, in the worst case, space consumption can grow to $O(nM^2)$, and construction time may reach $O(M^2n^2)$. To mitigate this, SeRF employs a compression technique that omits the storage of certain edges. While this reduces space usage, excessive pruning can degrade the quality of query results. Besides, SeRF does not support arbitrary data insertion or deletion, making it unsuitable for large datasets that require dynamic updates.

SuperPostfiltering. SuperPostfiltering [12] avoids building indices for all $O(n^2)$ possible query ranges for attribute ϕ. It constructs graph indices for ranges on attribute ϕ within a predefined range set R. This set consists of $\log_\beta n$ levels, where β is a constant. At level i, the ranges are defined as $R[i] = \{[j \cdot \beta^i + 1, (j+2) \cdot \beta^i] \mid j \geq 0 \wedge (j+2) \cdot \beta^i \leq n\}$. For a query range containing m elements, a corresponding range in $R[i]$ can be found such that $\beta^{i-1} \leq m \leq \beta^i$, ensuring the number of elements in this range does not exceed $2\beta m$. Once the appropriate range is identified, Postfiltering is applied on the index for that range. While offering good query performance, significant overlap between ranges at each level leads to high space consumption. Despite the theoretical space complexity of $O(nM \log n)$, the actual space usage can be several times larger than the original dataset. Furthermore, SuperPostfiltering has the same issue as SeRF that does not support updates.

iRangeGraph. The iRangeGraph index [14] is built on a static segment tree over attribute ϕ, where each node corresponds to an HNSW index covering the elements within the range of that node. For query, iRangeGraph does not perform a greedy search on a preconstructed graph at a specific node in the segment tree. Instead, it constructs the search graph during the query process. For an ANNS query with a range filter $[l, r]$, when the greedy search accesses an element u, the algorithm dynamically searches for M out-neighbors v that satisfy $l \leq \phi(v) \leq r$. This is done by traversing the segment tree from the root to the leaf node associated with u,

resulting in $O(\log n)$ nodes being examined. These neighbors are then added to the search queue. iRangeGraph achieves a good trade-off between efficiency and accuracy. However, since the segment tree is a static structure, it cannot efficiently handle updates to dynamic data. Updating iRangeGraph is time-consuming because it requires modifying the multi-level structure at each node, and thus it is difficult to extend iRangeGraph to support updates effectively.

3 OUR SOLUTION

Next, we elaborate on our proposed DIGRA index. In Section 3.1, we introduce the dynamic multi-way tree structure, T, which organizes data points in order based on their attribute values ϕ. At each node u in T, we maintain a Navigable Small World (NSW) graph index, constructed for all vectors falling within the sub-tree rooted at node u. This dynamic structure enables efficient handling of data insertions and deletions while keeping the index updated seamlessly. Next, we further explain how to do query processing in Section 3.2. Here, we demonstrate how to use the concept of *α-approximate range coverage* to efficiently identify at most two nodes in T that can include all vectors satisfying the range filter. This reduces the need to scan irrelevant portions of the tree, thereby improving query performance. In Section 3.3, we explain the index construction process using the query processing algorithm. In Section 3.4, we introduce our *lazy weight-based update mechanism*, designed to maintain high update efficiency in the NSW graph indices during dynamic changes. We show that the update time complexity can be bounded by $O(c_{upd} \cdot \log n)$, where c_{upd} is the cost of inserting an object into the NSW graph. This balances the need for efficient updates while preserving query performance and index quality.

3.1 Index Structure

The proposed DIGRA index uses a multi-way tree structure T to manage the entire set. To distinguish, we use "node" in tree structures and "vertex" in graph structures. The tree structure T is similar to a B-tree, where each node can store up to B children, and it separates its children by storing several key values. For each object $\mathbf{o}$, we use attribute value $\phi(\mathbf{o})$ as the key. For a node u in T, we use $u.child[i]$ to represent its i-th child, and $u.key[i]$ to represent its i-th stored key value. Thus, all attribute values of objects in the subtree of $u.child[i]$ are guaranteed to be within the range $[u.key[i-1], u.key[i])$. For node u, we maintain an NSW-style search graph, represented as $u.G$. It is constructed for set O_u, the object set containing all the objects stored in the subtree rooted at u. To speed up the search, we additionally maintain an entry node for node u, represented as $u.en$, which is a leaf node in the subtree of u to help obtain the search entry in the current graph $u.G$. Each node in T further stores a value $layer$ to indicate its level, with leaf nodes having a layer value of 0. For a non-leaf node u, its $layer$ value is the layer value of its child plus one, i.e., $u.layer = u.child[0].layer + 1$. Similar to B-trees, DIGRA guarantees that all child nodes of any given node are assigned identical layer values. An important property of the multi-way tree structure in DIGRA is that we keep the tree T weight-balanced, whose definition is as follows.

Definition 3.1. (Weight Balance) Let T be a multi-way tree with a branching factor B. The tree T is said to be *weight balanced* if: For every non-root node u in T at layer i, the size of u, denoted by size(u), satisfies $\frac{B^i}{4} \leq \text{size}(u) \leq B^i$, where size$(u)$ is defined as the number of leaf nodes in the subtree rooted at u.

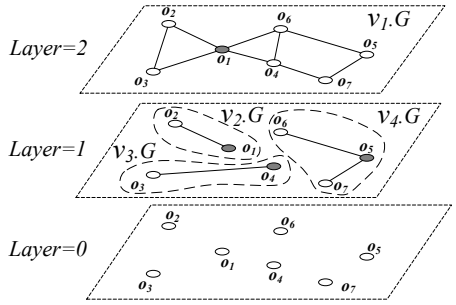


Figure 3: Example of DIGRA index

The weight of a node u is defined as its $size(u)$. We will utilize this weight balance condition to develop a lazy, weight-based update strategy that ensures update costs remain bounded. For now, we assume that the index structure is weight-balanced in our discussion of Sections 3.1-3.2.

Lemma 3.2. *The height of T is $O(\log n)$.*

It is straightforward to verify using the weight balance condition and thus omitted.

As to be detailed in Section 3.2, for any arbitrary range $Q = [\ell, r]$ on the attribute ϕ, there exist at most two nodes u and v in the tree T such that the union of their associated object sets, $O_u \cup O_v$, includes all objects O_Q that satisfy the range Q. Formally, this relationship is expressed as: $O_Q \subseteq O_u \cup O_v$. It is important to note that this approach may result in the inclusion of objects outside the specified range Q. However, we will demonstrate that the number of such irrelevant objects is bounded by a constant factor, ensuring that the majority of the objects within $O_u \cup O_v$ are relevant. Subsequently, we utilize the NSW indices maintained at nodes $u.G$ and $v.G$ to incrementally retrieve the ANNS results, thereby preserving the efficiency of the ANNS and avoid retrieving many irrelevant objects whose attribute value on ϕ is out of the range Q.

One of our key observations is that our multi-way tree inherently includes a hierarchical structure. Consequently, we maintain an NSW index at each node without introducing an additional hierarchical layer. During the search algorithm, we enable skip-based searching, allowing an ANNS search within a node to leverage the graph indices of its subtree nodes to locate the appropriate entry points. From this perspective, the proposed DIGRA index resembles a specialized HNSW structure. Specifically, starting from the root node, it corresponds to a comprehensive NSW graph akin to the bottom layer in HNSW. As we traverse down the tree layers, this graph becomes partitioned according to the objects stored in the child nodes. Specifically, let H_u to be the multi-layer graph structure with $u.G$ as the bottom layer. By visiting all the graph of the nodes on the path from the leaf node where $u.en$ is located to u, we implicitly construct H_u. This design offers several advantages. First, each layer retains only a single NSW graph, ensuring efficient updates during our lazy weight-based updates. Second, during search execution, the ability to utilize child nodes to identify entry vertices helps maintain query efficiency comparable to that of HNSW.

Example 3.3. Following the example set O in Example 2.3, we construct a DIGRA index for this dataset with a branching factor $B = 3$. The resulting balanced three-layer tree T, maintained by the DIGRA index, is illustrated in Figure 4(a). In this figure, the numbers within the nodes represent object IDs. The objects in the leaf nodes are ordered by their attribute values as follows: $\{\mathbf{o}_2, \mathbf{o}_1, \mathbf{o}_4, \mathbf{o}_3, \mathbf{o}_7, \mathbf{o}_6, \mathbf{o}_5\}$. The NSW indices maintained at each internal node, specifically v_1, v_2, v_3, v_4, of tree T are depicted in Figure 3. For graph indices corresponding to layers other than 0, the entry point of each respective graph index is highlighted in gray. For instance, the graph $v_4.G$ comprises the nodes $\mathbf{o}_5, \mathbf{o}_6, \mathbf{o}_7$, with $\mathbf{o}_5$ designated as the entry point of $v_4.G$.

3.2 Query Processing

Given the multi-way tree T maintained by our DIGRA index, a key property is the ability to identify a set of nodes in T that disjointly and exactly cover a query range Q, formally defined as follows.

Definition 3.4 (Disjoint Exact Coverage). Given an arbitrary range $Q = [\ell, r]$, a set of nodes $C = \{u_1, u_2, \ldots, u_x\}$ in tree T is said to *disjointly cover* Q exactly if the following conditions are satisfied:

$$\bigcup_{u \in C} O_u = O_Q \quad \text{and} \quad O_{u_i} \cap O_{u_j} = \emptyset \quad \forall\, i \neq j,$$

where O_Q is the set of objects whose attribute ϕ falls into Q and O_u is the set of objects falling into the subtree rooted at u.

Lemma 3.5. *DIGRA can identify a set C of $O(B \log n)$ nodes in T that disjointly covers an arbitrary query range $Q = [\ell, r]$ on the attribute ϕ in $O(B \log n)$ time.*

Proof. Let $C = \{u | O_u \subseteq O_Q \wedge O_{p(u)} \nsubseteq O_Q\}$, where $p(u)$ represents the parent node of u. We first show that C is a disjoint exact coverage. For each element $o \in O_Q$, consider the leaf $leaf_o$ where o is located. Since the $O_{leaf_o} \subseteq O_Q$ but $O_{root} \nsubseteq O_Q$, and the number of objects in the subtrees of nodes along the path from $root$ to the $leaf_o$ are continuously decreasing, there must be exact one node u along the path satisfying $u \in C$ and $o \in O_u$. We have that $\bigcup_{u \in C} O_u \supseteq O_Q$. And it is not difficult to see that nodes in C is disjoint, because the object set of subtree of each node will only intersect with its ancestors but a node and its ancestor will not both be included in C. Also we can directly derive $\bigcup_{u \in C} O_u \subseteq O_Q$ from the definition of C. Therefore, we have that C is a disjoint exact coverage.

Next, we are going to prove the lemma by showing that the number of nodes at the same level in C does not exceed $2B$. We will prove this by contradiction. Let the range that each node u represents to be $[\ell_u, r_u]$. Suppose that there are $2B + 1$ nodes in C in layer i, denoted as u_1, ..., u_{2B+1} s.t. $\ell_{u_1} \leq r_{u_1} \leq \ell_{u_2} \leq r_{u_2} \leq \ldots \leq \ell_{u_{2B+1}} \leq r_{u_{2B+1}}$. Consider $p(u_{B+1})$ satisfying $O_{p(u_{B+1})} \nsubseteq O_Q$ in layer $i + 1$. Because there are at most B children of $p(u_{B+1})$ in layer i, u_1 and u_{2B+1} cannot be children of $p(u_{B+1})$. Since $O_{u_1} \cap O_{p(u_{B+1})} = \emptyset$ and $O_{u_{2B+1}} \cap O_{p(u_{B+1})} = \emptyset$, we have $\ell \leq \ell_{u_1} < \ell_{p(u_{B+1})} \leq r_{p(u_{B+1})} < r_{u_{2B+1}} \leq r$. That means $O_{p(u_{B+1})} \subseteq O_Q$, which is a contradiction. Therefore, we have shown that the number of nodes in C does not exceed $2B$ at each level, which implies that the number of nodes in C is $O(B \log n)$. □

Algorithm 3: DIGRA-Query

Input: Query vector $\mathbf{q}$, range $[\ell, r]$, query parameters k, ef_s
Output: Query result

$hNode \leftarrow findHighNode(root, \ell, r)$;
if $hNode.layer == 0$ **then return** $\{hNode.en\}$; ;
$lId \leftarrow findPostion(hNode, \ell)$;
$rId \leftarrow findPosition(hNode, r)$;
$entryList \leftarrow emptylist$;
if $rId > lId + 1$ **then**
  $entry \leftarrow findEntry(H_{hNode}, \mathbf{q})$;
  $entryList.push(entry)$;
  $G \leftarrow hNode.G \cup (\bigcup_{ch \in hNode.child} ch.G)$;
end
else
  $lNode \leftarrow findNode(highNode.child[lId], \ell)$;
  $rNode \leftarrow findNode(highNode.child[rId], r)$;
  $lEntry \leftarrow findEntry(H_{lNode}, \mathbf{q})$;
  $rEntry \leftarrow findEntry(H_{rNode}, \mathbf{q})$;
  $entryList.push(lEntry, rEntry)$;
  $lG \leftarrow lNode.G \cup (\bigcup_{ch \in lNode.child} ch.G)$;
  $rG \leftarrow rNode.G \cup (\bigcup_{ch \in rNode.child} ch.G)$;
  $G \leftarrow lG \cup rG \cup hNode.G$;
end
$W \leftarrow RangeGreedySearch(G, entryList, \mathbf{q}, ef_s, \ell, r)$;
return k objects closest to $\mathbf{q}$ in W;

A natural approach would be to perform an ANNS on the NSW graph $u.G$ for each node $u \in C$, then merge the results and return the top-k answers. However, this strategy results in unnecessarily high query costs since we need to take $O(B \log n)$ such ANN searches. To alleviate this issue, we find that it is feasible to include some objects with attribute value on ϕ out of the range $[\ell, r]$ and then prune them, which can save costs. The key is to bound such irrelevant objects. Thus, we propose α-approximate range coverage.

Definition 3.6. (α-Approximate Range Coverage) Given a range $R = [\ell, r]$ and a dataset O, where O_R represents the objects in O falling within the range R, for a range $R' = [\ell', r']$ satisfying $R \subseteq R'$, if there exists a constant α such that $\frac{|O_{R'}|}{|O_R|} \leq \alpha$, we refer to the range query R' on dataset O as an α-approximate coverage for R.

Theorem 3.7. *Given an arbitrary query range $Q = [\ell, r]$, we can find at most two disjoint nodes u and v in T with $O(\log n)$ time such that their associated range $Q' = [\ell', r']$ is a $4B$-approximate range coverage of Q, i.e., $|O_u \cup O_v|/|O_Q| \leq 4B$.*

Proof. Let $C = \{u | O_u \nsubseteq O_Q \wedge \exists c \in u.child\ s.t. O_c \subseteq O_Q\}$. We first show that C is a $4B$-approximate range coverage. For each element $o \in O_Q$, consider the leaf $leaf_o$ where o is located. Since the $O_{leaf_o} \subseteq O_Q$ but $O_{root} \nsubseteq O_Q$, and the number of objects in the subtrees of nodes along the path from $root$ to the $leaf_o$ are continuously decreasing, there must be exact one node u along the path satisfying $u \in C$ and $o \in O_u$. Hence $\bigcup_{u \in C} O_u \supseteq O_Q$ and nodes in C is disjoint. Assume that ch_u is the child of u satisfying $O_{ch_u} \subseteq O_Q$ for $u \in C$. Note that $S = \{ch_u | u \in C\}$ is disjoint since C is disjoint, and obviously $\bigcup_{ch \in S} O_{ch} \subseteq O_Q$. According to Definition 3.1, we have $|O_u| \leq 4B \cdot |O_{ch_u}|$. Hence we have

$\sum_{u \in C} size(u) \leq 4B \cdot |O_u| = 4B \cdot \sum_{u \in C} |O_{ch_u}| \leq 4B \cdot |\bigcup_{ch \in S} O_{ch}| \leq 4B \cdot |O_Q|$.

Next, we are going to prove that $|C| \leq 2$ by contradiction. Let the range that each node u represents is $[\ell_u, r_u]$. Suppose there are 3 nodes in C in layer i, denoted as u_1,u_2,u_3 s.t. $\ell_{u_1} \leq r_{u_1} \leq \ell_{u_2} \leq r_{u_2} \leq \ell_{u_3} \leq r_{u_3}$ Consider u_2 satisfying $O_{u_2} \nsubseteq O_Q$. Since $O_{u_1} \cap O_Q \neq \emptyset$ and $O_{u_3} \cap O_Q \neq \emptyset$, there must be $\ell \leq r_{u_1} \leq \ell_{u_2} \leq r_{u_2} \leq \ell_{u_3} \leq r$. That means $O_{u_2} \subseteq O_Q$, which is a contradiction. Therefore, we can use at most two nodes in C as a $4B$-approximate range coverage. □

Theorem 3.7 enables us to focus on at most two nodes, u and v, and their associated NSW indices, $u.G$ and $v.G$, to efficiently process the approximate nearest neighbor (ANN) search. Our results demonstrate that this strategy yields approximately a 4x speed-up in query processing compared to directly solving the problem using disjoint exact coverage. We will show the detailed experimental results in Sec. 5. Building on this idea, we now present the detailed query algorithm.

Query algorithm. Alg. 3 is the pseudo-code for answering a query. First, we find the lowest layer node $hNode$ that covers the range $[\ell, r]$ in the tree in $O(\log n)$ time (Line 1). If $hNode.layer = 0$, it implies that the query range contains only one leaf node, and we directly return the node's entry (Line 2). Next, we check if $hNode$ can serve as an approximate range cover by identifying which of its children contain ℓ and r (Lines 3-4). If range $[\ell, r]$ covers at least one child ch of $hNode$, we perform ANNS on $hNode$ (Lines 6-9). Let $size_{[\ell,r]}(O)$ be the number of objects in dataset O with attribute in range $[\ell, r]$. As the tree is weight-balanced, $size(ch) \leq size_{[\ell,r]}(O)$ and $size(hNode) \leq 4B \times size(ch)$. Thus, ANNS on $hNode$ provides a $4B$-approximate range coverage for $[\ell, r]$.

If $hNode$ cannot cover $[\ell, r]$, we use two nodes for the search (Lines 11-20). Specifically, we locate two nodes each covered by $[\ell, r]$ across multiple children (Lines 12-13). Starting from the lId/rId child of $hNode$, we recursively visit the right/left-most children until they are fully covered by $[\ell, r]$. This takes $O(\log n)$ time, bounded by the tree height. Assuming $[\ell, r]$ covers one child each of $lNode$ and $rNode$, we have $size(lNode) + size(rNode) \leq 4B \times (size(lc) + size(rc)) \leq 4B \times size_{[\ell,r]}(O)$. Therefore, we achieve a $4B$-approximate coverage using $lNode$ and $rNode$. We optimize NSW search by leveraging a hierarchical structure. First, we use the hierarchy to find entries closer to $\mathbf{q}$, accelerating search convergence (Lines 7, 12-13). Second, we expand edge coverage by incorporating edges from the same vertices at other graph layers (Lines 9, 17-19). Since there can be two search entries, and not all vertices in the graph satisfy the range filter, we use the *RangeGreedySearch* algorithm, similar to Alg. 1, but with two modifications: (i) initialization uses multiple entries, and (ii) a vertex v is added to the set W only if $\ell \leq \phi(v) \leq r$.

3.3 Index Construction

The DIGRA index is built with a bottom-up approach using a dynamic multi-way tree structure. Data objects are first placed into leaf nodes, and an NSW graph is constructed for each leaf. These leaf nodes are then grouped into the higher-layer nodes, with each new node maintaining an NSW graph that covers the vectors within its subtree. This process continues upwards until the entire tree is formed. The NSW graph index of each internal node is built based on the object set from its child nodes, where the graph index is used to achieve efficient search. Next, we introduce the details of index construction. Alg. 4 shows the pseudo-code for building the DIGRA index. We first create leaf nodes for all objects and push them into Q_0 (Lines 2-5). From bottom to top, we iteratively extract $branch$ nodes from the queue of the lower layer as children of newly created node, construct a graph on the new node, and push it into the queue of the new layer (Lines 6-16). The $branch$ is usually set to be $\lceil B/2 \rceil$ here. This process repeats until only one node remains in the new layer. We return this node as the root (Line 17).

In Alg. 5, we show the process of building the graph for node u. First, we set entry and layer for u (Lines 1-2). Since we can obtain a multi-layer structure by visiting any path from a leaf node to u, we randomly select a leaf node from the subtree of u as its entry en. Then, we need to establish the NSW for the new node by establishing edges for each $\mathbf{o}$ in the subtree of u (Lines 3-16). Recalling the process of constructing NSW, it first finds the closest ef_{con} vertices in the graph for each vertex. If all ef_{con} vertices

Algorithm 4: DIGRA-Build

Input: The sorted objects O, construction parameter M, ef_c
Output: Root node of index DIGRA

```
1  Initialize queues Q, h ← 0;
2  for o in O do
3      nd ← createNode(), nd ← 0, nd.en ← o;
4      Q_0.push(nd)
5  end
6  while |Q_h| > 1: do
7      while Q_h is not empty do
8          if |Q_h| ≤ B then branch ← |Q_h| ;
9          else branch ← ⌈B/2⌉ ;
10         u ← createNode();
11         u.child ← branch nodes dequeued from Q_h;
12         Refresh(u, M, ef_c);
13         Q_{h+1}.push(u);
14     end
15     h ← h + 1;
16 end
17 return Q_h.front();
```

Algorithm 5: Refresh

Input: node u,construction parameter M, ef_{con}

```
1  u.en ← selectEn(u);
2  u.layer ← u.child[0].layer + 1;
3  for o ∈ O_u do
4      R ← ∅;
5      for v ∈ u.child do
6          if u ∈ O_v then R ← R ∪ N_{v.G}(o);
7          else
8              entry ← findEntry(H_v, o);
9              C ← GreedySearch(s.G, entry, o, ef_con);
10             R ← R ∪ prune(C, o, M);
11         end
12     end
13     N_{u.Graph}(o) ← prune(R, o, M);
14 end
```

are selected as neighbors of this vertex, the graph density will be too high. Therefore, through *prune* in Alg. 2, only M vertices are chosen as neighbors. The intuition behind pruning is that if two vertices are close to each other, there is likely to be an edge between them, so we only keep one of them to be a neighbor to sparsify the graph and ensure search performance. However, since the NSW is constructed incrementally starting from an empty graph, vertices inserted earlier will have difficulty finding vertices inserted later as neighbors. To overcome this, for each $\mathbf{o}$, we first independently find ef_{con} neighbors and select M edges through pruning in the graph of each child of u except the child that $\mathbf{o}$ belongs to (Lines 5-12). Then we further sparsify the found $M \cdot |u.child|$ edges to M edges through pruning (Line 13). This allows each vertex to obtain information from the entire set O_u, ensuring search performance. Note that we used the same method as in Sec. 3.2 to find entry, thereby improving the speed (Line 8).

Lemma 3.8 shows the time complexity of the $Refresh$ operation:

Lemma 3.8. *For node u in the DIGRA index, the time cost of performing the refresh operation is $O(size(u) \cdot c_{upd})$.*

Proof. We analyze the time complexity of inserting each object into $u.G$ during $Refresh$. First, each object in O_u needs to find its neighbors in $O(B)$ graphs, which costs $O(c_{upd})$. Next we construct its edge in $u.G$, which costs $O(c_{upd})$. The total cost of each object in $Refresh$ is $O(c_{upd}) + O(c_{upd}) = O(c_{upd})$. Hence time complexity of $Refresh$ is $O(size(u) \cdot c_{upd})$. □

In the following theorem, we obtain the time complexity of building DIGRA index and the space occupied by DIGRA.

Theorem 3.9. *Algorithm 4 constructs a DIGRA index on O with $O(Mn \log n)$ space complexity in $O(n \log n \cdot c_{upd})$ time.*

Proof. We analyze the time and space complexity separately for the graph index and the tree index. Similar to a B-tree, the number of nodes on the tree is $O(n)$, which means the space occupied by the tree structure is $O(n)$. As for the graph, each object belongs to only one graph in each layer, and each object has M outgoing edges in one graph. The graph index's space complexity is $O(Mn \log n)$. Therefore, the total space usage of the index is $O(n)+O(Mn \log n) = O(Mn \log n)$. The time complexity of constructing the tree structure, excluding establishing the graph index (i.e., excluding the $Refresh$ operation), is $O(n)$. Next, we analyze the time cost of all refresh operations. Since each object is refreshed in exactly one node in each layer, we have $\sum_{u.layer=i} size(u) = n$ for each layer i. So the refresh cost of each layer is $O(n \cdot c_{upd})$. The time complexity of constructing all graphs is $O(n \cdot c_{upd} \cdot \log n)$. Hence the total complexity is $O(n) + O(n \log n \cdot c_{upd}) = O(n \log n \cdot c_{upd})$. □

3.4 Index Update

The key to updating the DIGRA index is to ensure that while maintaining the tree's balance condition, the ANN index maintained by each node remains up-to-date and consistent with the updated set in the corresponding subtree. Our update algorithms ensure the query performance of DIGRA by maintaining both the tree's balance and the consistency of the ANN index. It includes two steps: *(i)* Start from the root and search downwards until the node that needs to be updated is found. During this process, we need to update the ANN index maintained by each node. *(ii)* Backtrack on the tree after the update, and for each node violating the balance condition, we rebalance it via a split or merge operation. When the tree is balanced, the cost of the first step does not exceed the time required to update on $O(\log n)$ vanilla NSW graphs. Therefore, ensuring efficient updates lies in bounding the time overhead of the second step. When a node u of size $size(u)$ undergoes a split or merge operation, the time consumed is $O(size(u) \cdot c_{upd})$, which is the time cost of the refresh operation. Here, we use the proposed lazy update mechanism, to allow each initially balanced node u to withstand $\Omega(size(u))$ update operations before triggering a rebalancing operation, thereby reducing the total update overhead to $O(c_{upd} \log n)$. Next, we show the details of the update algorithms.

Algorithm 6: DIGRA-Insert

```
   Input: object o, construction parameter M, ef_con
1  if root = NULL then
2  |  root ← createNode(), root.layer ← 0, root.en ← o
3  end
4  else
5  |  insertNode(root, o);
6  |  if |root.child| > B then
7  |  |  nRoot ← createNode();
8  |  |  nRoot.child[0] ← root, root ← nRoot;
9  |  |  splitNode(root, 0),
10 |  end
11 end
12 procedure insertNode(node, o, M, ef_con):
13 |  id ← findPosition(node, o);
14 |  if node.layer = 1 then
15 |  |  newNode ← createNode();
16 |  |  newNode.en ← o, entry ← o;
17 |  |  insertKeyAndChild(node, id, o, newNode);
18 |  end
19 |  entry ← vertex closest to o in N_{node.child[id].G}(o);
20 |  GraphInsertion(node.G, o, entry, M, ef_con);
21 |  if |node.child[id].child| > B then
   |  |  splitNode(node, id, M, ef_con) ;
22 procedure splitNode(node, id. M, ef_con):
23 |  n1 ← node.child[id], n2 ← createNode();
24 |  n1, n2 ← splitEvenly(n1);
25 |  insertChild(node, id, n2);
26 |  Refresh(n1, M, ef_con), Refresh(n2, M, ef_con);
```

Insertion of DIGRA index. Alg. 6 outlines the pseudo-code of insertion. If the current tree is empty, we create a new node and make it the root node (Lines 1-3). Otherwise, starting from the root, we insert a new object $\mathbf{o}$ recursively (Line 5). We first find the position in tree to insert the new node, create a leaf node and insert this leaf node into the corresponding position (Lines 14-18). Then we insert $\mathbf{o}$ into the graph of each node on the backtracking path (Lines 19-20). Next, if any child of the node has greater than B children after the insertion, we split the child (Line 21). The splitting process is similar to B-tree splitting. When we split the node $n1$, we divide the key and children of $n1$ into two parts, keeping one part for itself and moving the other part becoming the new node $n2$, and insert $n2$ after $n1$ (Lines 23-25). As the subtree of $n1$ and $n2$ changed,

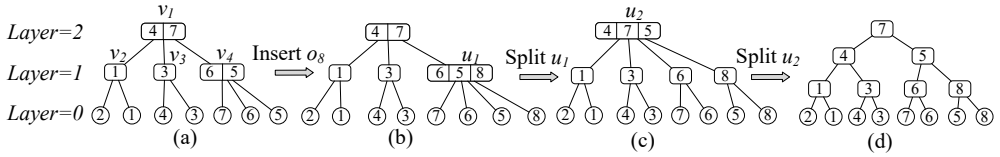


Figure 4: An example of the insertion with DIGRA ($B = 3$).

we rebuild $n1.G$ and $n2.G$ by $Refresh$ (Line 26). When the insertion is finished, if the number of children of the root node exceeds B, we will split the root node and make it a child of the new root node (Lines 6-10). To analyze the time complexity of insertion for DIGRA, we first analyze the time complexity of the $Split$ operation:

Lemma 3.10. *The time of a split operation to u is $O(size(u) \cdot c_{upd})$.*

Proof. First, updating children and keys requires $O(1)$ operations. As n_1 and n_2 are children of node u, their combined size is $O(\text{size}(u))$. By Lemma 3.8, two refresh operations cost $O(\text{size}(u) \cdot c_{\text{upd}})$, resulting in an overall time cost of $O(\text{size}(u) \cdot c_{\text{upd}})$. □

Assuming that after the last split on node u, the next split occurs after $\Omega(\text{size}(u))$ insertions in u's subtree by using the lazy update strategy elaborated later, we have the following theorem:

Theorem 3.11. *The DIGRA index can handle each insertion operation in $O(c_{upd} \log n)$ amortized time.*

Proof. We analyze the time cost of the insertion operation step by step. First, we find the location to insert the object $\mathbf{o}$ on DIGRA, create a leaf node, and insert it, which costs $O(\log n)$. Then we insert $\mathbf{o}$ into the graph of each visited node u by calling $GraphInsertion$. The time complexity of this step is $O(c_{upd} \log n)$. Next, we need to check all nodes on the recursive path. If we find that the number of children of a node u exceeds B, we perform a split operation. The time cost of split is $O(size(u) \cdot c_{upd})$. But a split only happens after $\Omega(size(u))$ insertions in the subtree rooted at u. Therefore, the amortized cost for each insertion in the subtree is $O(c_{upd})$. For a single insertion, it affects up to $O(\log n)$ nodes, and we charge an amortized cost of $O(c_{upd})$ for each node. Therefore, the amortized cost of the split caused by inserting an object is $O(c_{upd} \log n)$. Overall, the cost of insertion is amortized time of $O(c_{upd} \log n)$. □

Example 3.12. With the object set from Example 2.3, we demonstrated how the structure of the tree changes after insertion. We have constructed the tree structure of DIGRA shown in Fig. 4 (a) with $B = 3$ using $\mathbf{o}_1$ to $\mathbf{o}_7$, and now we want to insert $\mathbf{o}_8$. We locate the corresponding position for $\mathbf{o}_8$ insertion and create a leaf node. In the structure shown in Fig. 4 (b), we find that after inserting $\mathbf{o}_8$, the number of children of u_1 exceeds 3. We split u_1 into two nodes, with one node containing key $\mathbf{o}_6$ and children $\mathbf{o}_7$, $\mathbf{o}_6$, and the other node containing key $\mathbf{o}_8$ and child $\mathbf{o}_5$, $\mathbf{o}_8$. We then add them as new children of their parent node u_w, with $\mathbf{o}_5$ serving as the key splitting the two new nodes as shown in Fig. 4 (c). u_2 also exceeds 3 children and is the root. We create a new root, with the nodes generated from splitting u_2 becoming the children of the new root, and the key splitting these two nodes, $\mathbf{o}_7$, becoming the key in the new root. Then we obtain the final tree shown in Fig. 4 (d).

Deletion of DIGRA index. Alg. 7 outlines the pseudo-code for dealing with deletions. We first mark the key for deletion (Line 1). Note that when conducting $RangeGreedySearch$ in query processing, we will put vertices marked for deletion into the search queue C, but not into the result queue W. Then we proceed with the recursive deletion process starting from the root node (Line 2). We find the corresponding leaf node and delete it (Lines 5-8). After deleting the leaf node, we need to check if the number of children of $child[id]$ of the backtracked node is less than $\lceil B/2 \rceil$. If so, we look for a sibling node whose number of children is greater than $\lceil B/2 \rceil$. If we find such a node, we transfer a child from the sibling node and refresh the graphs of the two nodes (Lines 10-18). If we cannot find such a sibling node, we merge it with one of its siblings on the left or right (Lines 20-21). When merging $n1$ and $n2$, we append the children and keys of $n2$ to $n1$ (Lines 25-26). Then we remove $n2$, and reconstruct the graph of $n1$ (Line 28). Finally, if the root node has only one child left after deletion, we will remove the root node and set its child as the new root node (Line 3). Similar to insertion, we present the time complexity of $Merge$ and $Deletion$ below.

Lemma 3.13. *The time cost of merge operation is $O(size(u) \cdot c_{upd})$.*

Proof. We can achieve the changes in children and keys through $O(1)$ operations and bound the $Refresh$ operation by $O(size(u) \cdot c_{upd})$ since $O(size(n1)) = O(size(u))$. Therefore the overall time complexity is $O(size(u) \cdot c_{upd})$. □

Assuming that after the last merge on node u, the next merge on u occurs after $\Omega(\text{size}(u))$ deletions in its subtree by using the lazy update strategy explained later, we have the following theorem:

Theorem 3.14. *The deletion of DIGRA index can handle each deletion operation in $O(c_{upd} \log n)$ amortized time.*

Proof. First, we find the position of the object to be deleted on DIGRA and remove the corresponding leaf node. The time complexity of this part is $O(\log n)$. Next, we need to check all nodes on the recursive path. If we perform child transfer and refresh operations. The transfer operation can be completed in $O(1)$ time, and the cost of the refresh operation can be limited to $O(size(u) \cdot c_{upd})$. If we perform a merge operation, it also costs $O(size(u) \cdot c_{upd})$. These cases only occur $\Omega(size(u))$ deletions in the subtree rooted at u. Therefore, the amortized cost for each operation in the subtree is $O(c_{upd})$. For a single deletion, it affects up to $O(\log n)$ nodes, and we charge an amortized cost of $O(c_{upd} \log n)$ for each node. Overall, the cost of deletion is $O(c_{upd} \log n)$ amortized time. □

Example 3.15. Consider the object set from Example 3.12. We have constructed the tree structure of DIGRA shown in Fig. 5 (a) with $B = 3$ using $\mathbf{o}_1$ to $\mathbf{o}_8$, and now we want to delete $\mathbf{o}_1$. We locate the corresponding position for $\mathbf{o}_1$ and delete the leaf node. After deletion, the number of children of v_1 is less than 2. We merge v_2 and v_3 with $\mathbf{o}_4$ serving as the key splitting them. In the structure shown in Fig. 5 (b), we find that v_1 has only one child, so we merge

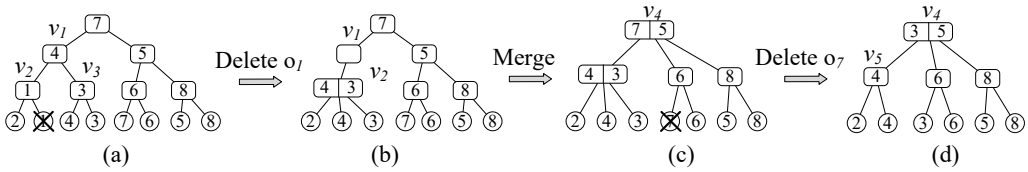


Figure 5: An example of the deletion with DIGRA ($B = 3$).

Algorithm 7: DIGRA-Delete

Input: Delete key of element key, construction parameter M, ef_{con}

```
1  markAsDeleted(key);
2  eraseNode(root, key);
3  if |root.key| = 0 then root ← root.child[0]; ;
4  procedure eraseNode(node, key):
5      id ← findPosition(node, key);
6      if node.layer = 1 then removeKeyAndChild(node, id) ;
7      else
8          eraseNode(node.child[id], key);
9          if |node.child[id].child| < ⌈B/2⌉ then
10             if id > 0 and |node.child[id − 1].child| > ⌈B/2⌉
                then
11                 redistributeKeys(node, id);
12                 Refresh(node.child[id − 1], M, ef_con);
13                 Refresh(node.child[id], M, ef_con);
14             end
15             else if id < |node.child| and
                |node.child[id + 1].child| > ⌈B/2⌉ then
16                 redistributeKeys(node, id);
17                 Refresh(node.child[id], M, ef_con);
18                 Refresh(node.child[id + 1], M, ef_con);
19             end
20             else if id > 0 then mergeNodes(node, id − 1); ;
21             else mergeNodes(node, id, M, ef_con) ;
22         end
23     end
24 procedure mergeNodes(node, id, M, ef_con):
25     n1 ← node.child[id], n2 ← node.child[id + 1];
26     n1 ← append(n1, n2)
27     removeChild(node, n2),
28     Refresh(n1, M, ef_con);
```

v_2 and its sibling with $\mathbf{o}_4$ serving as the key splitting them. Then we delete $\mathbf{o}_7$ from the index shown in Fig. 5 (c). As v_5 has 3 children, we transfer a child from v_5 to obtain the index shown in Fig. 5 (d).

Lazy update. The B-tree only limits the number of branches and is not a very effective balancing strategy in extreme cases. Consider a B-tree with $B = 3$. Suppose a node u has two children v_1 and v_2, where all nodes in the subtree of child v_1 have 2 branches, and all nodes in the subtree of child v_2 have 3 branches. Thus we no longer have $size(u) = O(size(v_1))$, since $size(v_1) = 2^i$ and $size(u) = 3^i + 2^i$. This makes it difficult to obtain an α-approximate range coverage. To improve this, we trigger a split or merge when the balance condition introduced in Definition. 3.1 is not met. We replace the condition at line 21 in Alg. 6 with the check if the size of the node is greater than B^i, and replace the condition at lines 9,10,15 in Alg. 7 with the check if the size to the node is less or greater than $B^i/4$, where i is the layer of the node. To balance the tree, we do not need to perform split and merge on all of the nodes every update. According to the previous work [5], if we use such a balancing strategy, we have the following theorem:

Theorem 3.16. *It takes $\Omega(size(u))$ insertions or deletions to make the balanced node u after a merge or split operation unbalanced again.*

Recap that Theorems 3.11 and 3.14 both require this lazy update strategy to achieve the desired time complexity. The above theorem affirms that such a lazy update strategy is doable, showing the final time complexity of insertion and deletion to be both $O(c_{upd} \log n)$.

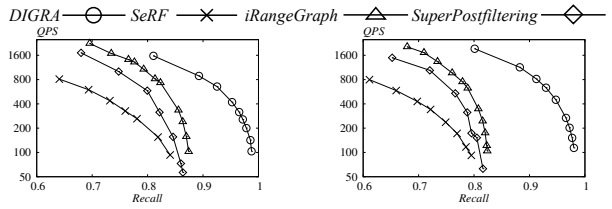
4 RELATED WORK

Approximate Nearest Neighbor Search. Approximate nearest neighbor search (ANNS) methods are generally categorized into three types: LSH-based [2, 3, 9, 18, 34], PQ-based [4, 6, 7, 14, 19, 20], and graph-based [11, 13, 23, 24, 30]. Locality-Sensitive Hashing (LSH) [18] hashes similar items into the same bucket with high probability. While LSH and its variants offer adjustable performance and theoretical error guarantees, they require more space to maintain accuracy compared to other methods. Product Quantization (PQ) [19] partitions high-dimensional space into multiple low-dimensional subspaces and quantizes each separately. PQ and its extensions [4, 6, 7, 14, 20] trade some accuracy for reduced space usage, making them suitable for billion-scale datasets. Graph-based methods [11, 13, 23, 24, 30] build a proximity graph where nodes represent vectors and edges connect similar vectors, using greedy search for ANNS. Notable graph-based approaches like HNSW [24], NSG [13], and DiskANN [30] achieve a strong balance between efficiency and accuracy and are widely adopted in industry.

ANNS with attribute filtering. Several algorithms and systems have been developed for attribute-filtered ANN queries [12, 15, 25, 31, 37–39]. Some systems, like Milvus [31], support general attribute filters and predicates, employing different strategies for various ANN searches. RII [25], based on PQ, first filters vectors by predicates before applying a query strategy. However, previous works [12, 37, 39] show that general methods are sub-optimal for specific attribute queries, where specialized designs often perform better. Beyond range-filtering [12, 37, 39], Filtered-DiskANN [15] target queries with attributes such as date, price range, or language.

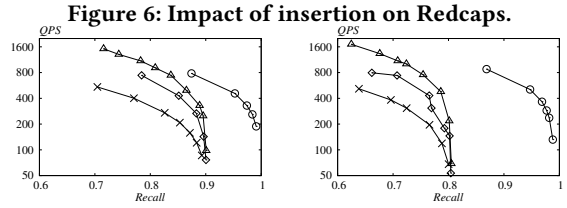
5 EXPERIMENTS

We compare the proposed DIGRA against the state-of-the-art solutions in various aspects through experiments. All experiments are conducted on a Linux machine equipped with an Intel Xeon(R) CPU at 2.30GHz and 768GB of memory using a single thread.



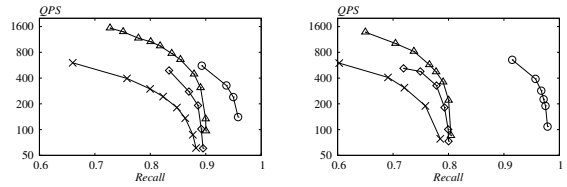
(a) 10% insertion (b) 20% insertion

Figure 6: Impact of insertion on Redcaps.



(a) 10% insertion (b) 20% insertion

Figure 7: Impact of insertion on GIST.



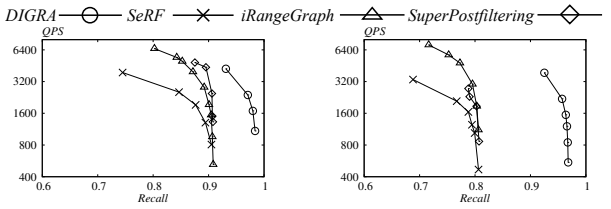
(a) 10% insertion (b) 20% insertion

Figure 8: Impact of insertion on WIT.

5.1 Experimental Settings

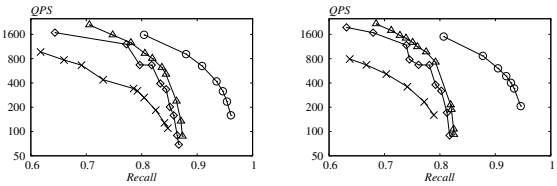
Datasets. We use the following four real-world datasets and their query settings tested in related research for range-filtered ANNS [12, 24, 39]: (i) SIFT, which consists of 128-dimensional feature vectors extracted from multiple images. It provides one million base vectors and ten thousand query vectors. (iv) Redcaps, which consists of 512-dimensional embedding vectors generated by CLIP on the RedCaps images. It contains more than 11.2 million vectors. (iii) GIST, which consists of 960-dimensional feature vectors extracted from images. It provides one million base vectors and one thousand query vectors. (iv) WIT, which consists of 2048-dimensional ResNet-50 embedding vectors of an image from Wikipedia. It contains more than six million vectors. For SIFT and GIST, we use base vectors as input, where each vector is assigned a random integer from the range $[1, 10^4]$ as an attribute, and select one thousand out of the provided vectors to be our query vectors. For Redcaps, we randomly extract one million vectors with the timestamp of the image as the attribute value and create a set of one thousand query vectors by asking ChatGPT-4 to come up with queries for an image search system and embedding using CLIP. For WIT, we randomly extract one million vectors with the size of the image as the attribute value and randomly extract one thousand vectors as query vectors.

Competitors. We include the following methods in our experimental comparisons: (i) SeRF [39], which is a static range-index method that builds a compressed index for all possible query ranges based on HNSW. As reported in the paper, we evaluate SeRF with MaxLeap optimization. (ii) SuperPostfiltering [12], which is a static method based on Vamana [30]. (iii) iRangeGraph [37], which is a static method of constructing graphs during the search process based on HNSW. (iv) Prefiltering, which first uses binary search to find all objects that satisfy the range constraint, then performs a linear scan on these objects to find the nearest neighbor. Our code and more experimental details can be found in [1].



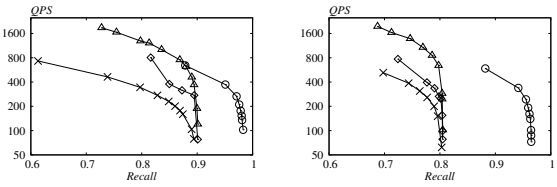
(a) 10% deletion (b) 20% deletion

Figure 9: Impact of deletion on SIFT.



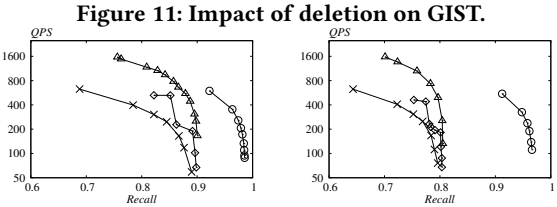
(a) 10% deletion (b) 20% deletion

Figure 10: Impact of deletion on Redcaps.



(a) 10% deletion (b) 20% deletion

Figure 11: Impact of deletion on GIST.



(a) 10% deletion (b) 20% deletion

Figure 12: Impact of deletion on WIT.

Evaluation Metrics and Parameters. Following existing benchmarks [36, 39], we use *QPS* (Queries Per Second) to measure efficiency and use *Recall* to show the accuracy. Suppose that R^* is the exact k nearest neighbors and R is the result of ANNS, the $Recall(R)$ is define as $Recall(R) = \frac{|R \cap R^*|}{|R^*|} = \frac{|R \cap R^*|}{k}$. We set $k = 10$ for all experiments. In terms of construction parameters, for our method, we set $B = 3$ for the multi-way tree structure. The NSW graph construction process includes two parameters M and ef_{con}. The number M is set to 16 for SIFT, 32 for GIST, Redcaps and WIT. The base ef_{con} is set to 400 for all datasets. We also conduct experiments to examine the impact of these parameters. For SeRF and iRangeGraph, the parameter M of their HNSW graph are set to the same as DIGRA. Other parameters of these two methods are their default settings. As SuperPostfiltering is based on Vamana, we use its recommended parameters, i.e., $\beta = 2$, $ef = 500$, $m = 64$ for all datasets. Apart from Prefiltering, we adjust parameters during the search to control the QPS-Recall trade-off during the query phase. For SuperPostfiltering, the parameters are k and $final\_multiply$, and for other methods, the parameter is ef in greedy search. The

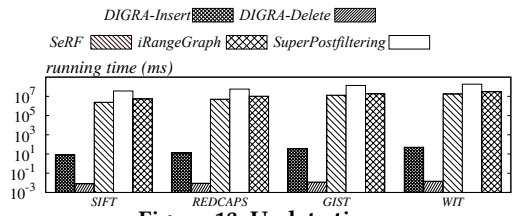


Figure 13: Update time.

default coverage of query ranges is set to 10%, i.e., 10% of the objects falling into the range.

5.2 Experimental Comparisons

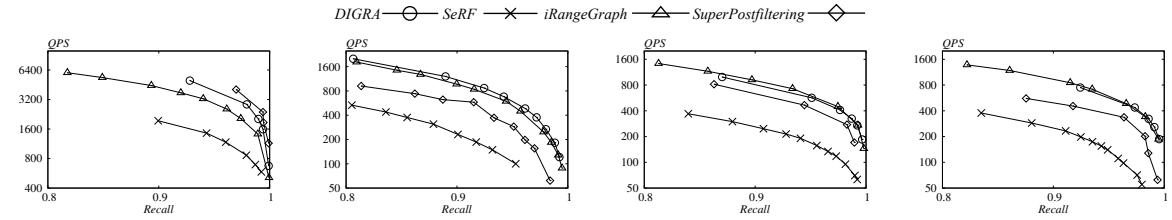
Exp 1: Impact of update. Figures 6-8 illustrate the query performance of various methods across three datasets: RedCaps, GIST, and WIT after data insertion (SIFT results are presented in Figure 1 of Section 1). In this set of experiments, we start with 80% or 90% of vectors from these datasets, and then add the remaining 10% or 20% vectors to the dataset. Figures 9-12 illustrate the query performance of various methods after data deletion. In this set of experiments, we start with all of vectors from these datasets, and then remove the 10% or 20% vectors from the dataset. For SeRF, iRangeGraph, and SuperPostFiltering, as they are static indexes, we use the index built before the update to process the range-filtered ANNS queries. We can observe that the query performance of SeRF, SuperPostFiltering, and iRangeGraph degrades significantly, as these methods are designed for static data. We can also find that as the amount of data inserted/deleted increases, the query performance of indexes designed for static data further degrades, with recall unable to exceed 0.85. In contrast, our DIGRA maintains stable query performance even with dynamic data changes. As seen from the curve, our DIGRA is insensitive to data changes, showing its strong capability in handling ANNS with range filters for dynamic vector data.

Exp 2: Update efficiency. To evaluate update efficiency, we start with the full dataset, remove 10% of the vectors to test deletion efficiency, and then reinsert the 10% to measure insertion efficiency. We report the average time for each insertion and deletion operation. For static methods such as SeRF, SuperPostFiltering, and iRangeGraph, we report their index rebuild time for comparison. The update times for all methods across different datasets are shown in Figure 13. The proposed method, DIGRA, demonstrates exceptional update performance, with insertion and deletion operations being over five and eight orders of magnitude faster, respectively, than static indexes that require rebuilding. Additionally, DIGRA-Delete outperforms DIGRA-Insert. This is because, with randomly generated updates, most deletions only require marking objects for deletion in the ANN index of affected nodes without reconstructing the tree nodes or their ANN indices. The millisecond-level update capability of DIGRA highlights its superior performance in handling ANNS with range filters in dynamic vector data scenarios.

Exp 3: Query performance on static data. In this experiment, we demonstrate that even without changes to the dataset, our proposed method, DIGRA, remains highly efficient for range-filtered ANNS while maintaining the same level of recall. Figure 14 presents the query performance of all methods on static datasets with a range coverage of 10%, indicating that 10% of the objects fulfill the range condition Q on attribute ϕ. Prefiltering is excluded due to its QPS being less than 50. Our index and query strategies, designed for dynamic updates, continue to deliver excellent performance on static datasets compared to other methods. Analyzing the results, SeRF shows suboptimal query performance across all datasets, primarily due to its heavy compression. While SuperPostfiltering slightly outperforms DIGRA on the SIFT dataset, it lags behind both DIGRA and iRangeGraph on other datasets. Notably, our method outperforms iRangeGraph on low-dimensional datasets like SIFT and Redcaps and performs comparably on high-dimensional datasets like GIST and WIT. This can be attributed to iRangeGraph's dynamic edge selection strategy, which incurs significant computational costs in low-dimensional spaces. In contrast, in high-dimensional spaces, the cost of distance computations becomes the dominant factor, affecting all methods similarly. In other words, our DIGRA supports efficient updates and also does so without any compromise to query efficiency—even outperforming other methods in various scenarios. This demonstrates that DIGRA effectively balances the need for dynamic updates with high query performance, making it highly suitable for both dynamic and static datasets.

Exp 4: Indexing cost. Figure 15 displays the memory usage of all methods across various datasets. Note that Prefiltering does not need an additional index, meaning its memory usage is equivalent to the input data size. SeRF incurs only a slightly higher memory cost compared to Prefiltering, significantly lower than the theoretical maximum space cost of $O(n^2M)$. This efficiency is achieved through an aggressive compression strategy, which, however, compromises performance and query accuracy. Previous experimental results indicate that both Prefiltering and SeRF do not deliver satisfactory query performance to achieve the same level of recall. On the other hand, SuperPostFiltering incurs excessive memory usage to achieve comparable performance as our DIGRA and iRangeGraph, requiring up to five times the size of the input data. In contrast, our proposed method, DIGRA, maintains a memory footprint comparable to iRangeGraph while delivering outstanding query performance. This balance ensures that DIGRA achieves high efficiency without incurring unreasonable memory costs, making it a superior choice for range-filtered ANNS tasks. For indexing time, Figure 13 has already shown the indexing times for the other methods. For DIGRA, the indexing time is comparable to that of iRangeGraph and is omitted here for brevity.

Exp 5: Impact of query range. In this set of experiment, we examine the impact of the range coverage to the query performance. Figures 16-17 show the query performance of different methods across all datasets when varying the range filter coverage ratio. We adjust the range filter across {1%, 5%, 10%, 20%, 40%} and display the corresponding QPS and recall. We can observe that DIGRA performs well, maintaining recall above 0.9 while ensuring high QPS. Because SeRF uses MaxLeap to compress ANN index, which loses a significant amount of information, its query performance is inferior to other methods, especially when the coverage ratio of range filter is less than 10%. On the SIFT dataset, SuperPostFiltering achieves the best QPS, but it consumes a large amount of space to store pre-built ANN indices as shown in Exp. 4. DIGRA also outperforms iRangeGraph in terms of recall, consistent with our observations in Exp. 3. Similar experimental results can be observed on the Redcaps datasets. On the GIST dataset, DIGRA performs



(a) SIFT (b) Redcaps (c) GIST (d) WIT

Figure 14: Query performance.

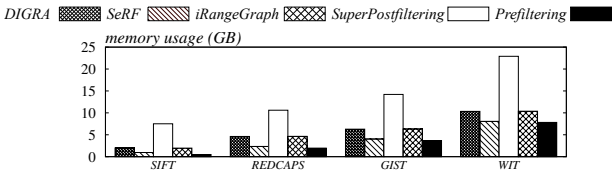
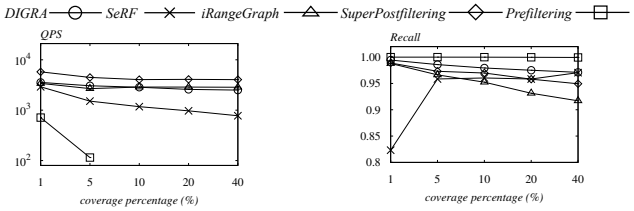
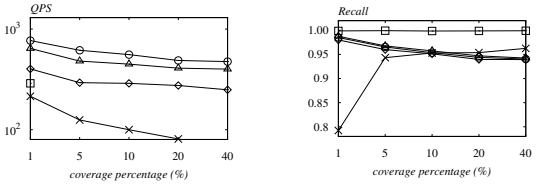


Figure 15: Memory usage.



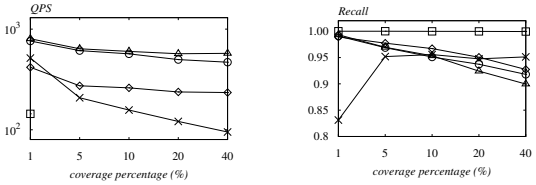
(a) Queries per second (b) Recall

Figure 16: Impact of query range on SIFT.



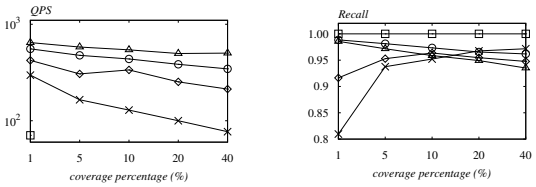
(a) Queries per second (b) Recall

Figure 17: Impact of query range on Redcaps.



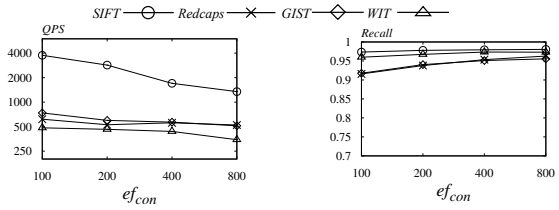
(a) Queries per second (b) Recall

Figure 18: Impact of query range on GIST.



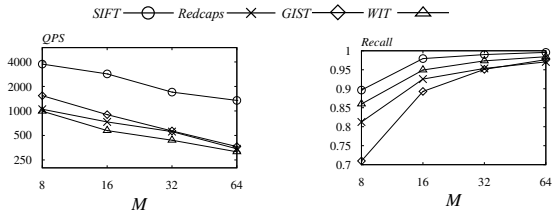
(a) Queries per second (b) Recall

Figure 19: Impact of query range on WIT.



(a) Queries per second (b) Recall

Figure 20: Impact of ef_{con} to query performance.



(a) Queries per second (b) Recall

Figure 21: Impact of M to query performance.

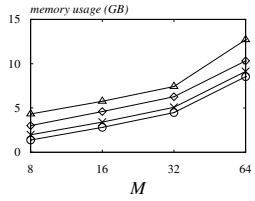
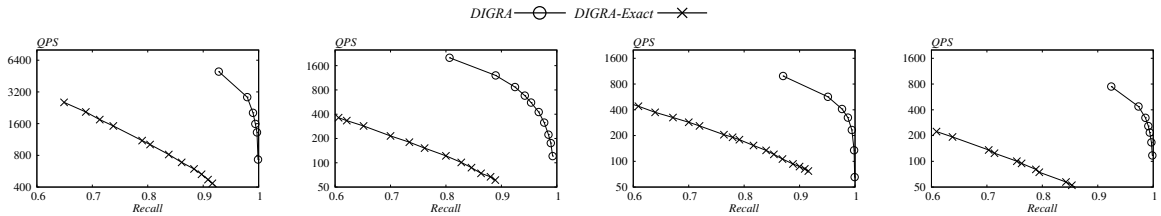


Figure 22: Impact of M to memory usage.

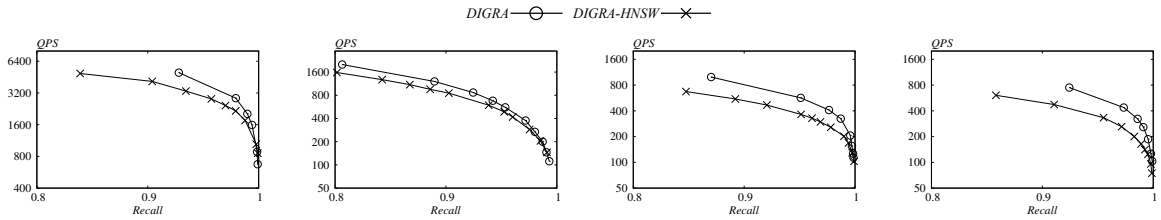
similarly to iRangeGraph and significantly outperforms SuperPostFiltering in terms of QPS with similar recall. On the WIT dataset, DIGRA achieves higher recall than iRange and SuperPostFiltering, while having lower QPS than iRangeGraph. Overall, DIGRA delivers stable and superior query performance across various query ranges while at the same time supporting updates.

Exp. 6: Impact of parameter ef_{con}. In this set of experiment, we examine the impact of parameter ef_{con}. Fig. 20 shows how varying ef_{con} affects performance. Overall, increasing ef_{con} slightly improves recall but marginally reduces QPS. This occurs because a larger ef_{con} causes the out-edges of each vertex to connect to more distant vertices, thereby enhancing search accuracy. However, it also expands the search radius, leading to a minor decrease in QPS. Based on the experimental results, we set $ef_{con} = 400$ as the default parameter, as it achieves a good balance between recall and QPS.



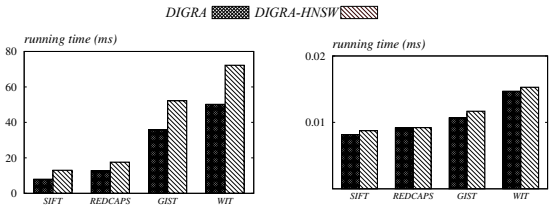
(a) SIFT (b) Redcaps (c) GIST (d) WIT

Figure 23: Exact coverage v.s. approximate coverage.



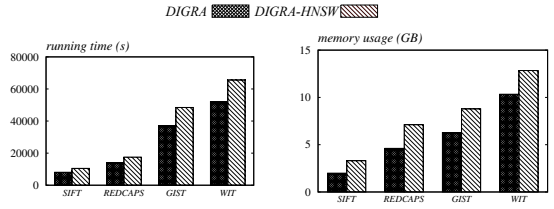
(a) SIFT (b) Redcaps (c) GIST (d) WIT

Figure 24: DIGRA-HNSW v.s. DIGRA on query performance.



(a) Insertion time (b) Deletion time

Figure 25: DIGRA-HNSW v.s. DIGRA on update time .



(a) Indexing time (b) Memory usage

Figure 26: DIGRA-HNSW v.s. DIGRA on preprocessing cost.

Exp. 7: Impact of parameter *M*. Fig. 21 - 22 shows the impact of parameter M. Increasing M leads to higher space consumption but strengthens the graph's connectivity, resulting in the exploration of more nodes during greedy search. This improvement in connectivity enhances recall but adversely affects QPS. Based on the experimental results, we set $M = 16$ for the SIFT dataset and $M = 32$ for other datasets as the default parameters to achieve a balance between query performance and space usage.

Exp. 8: Exact coverage v.s. approximate coverage. In this experiment, we compare the query performance of the search strategy based on the approximate coverage composed of 2 nodes with the exact coverage composed of $O(B \log n)$ nodes mentioned in Sec. 3.2. Fig. 23 shows the comparison in query performance between the two strategies. Note that the recall of strategy based on exact coverage can be higher than shown in the figure, but this would further reduce the QPS, which we do not display. The strategy based on approximate coverage adopted by DIGRA outperforms the strategy based on exact coverage. Although there may be some redundant vertices in the approximate coverage search, searching on at most two graphs allows for stronger connections between vertices, which means the search converges faster. In contrast, the exact coverage search involves $O(B \log n)$ nodes, and the graphs on these $O(B \log n)$ nodes are not interconnected. It is time-consuming to find query vector $\mathbf{q}$'s neighborhood on all $O(B \log n)$ graphs to make the search converge. This experiment demonstrates the effectiveness of our search strategy.

Exp. 9: DIGRA v.s. DIGRA-HNSW. In this set of experiments, we compare our full-fledged DIGRA against a modified version of DIGRA that maintains a complete HNSW index at each node, dubbed as DIGRA-HNSW. Recall that in DIGRA, each node holds only a single-layer NSW, forming an HNSW-like hierarchical index with a tree structure. Results are shown in Fig. 24 - 26. Unlike DIGRA-HNSW, which performs greedy search on two unrelated graphs, DIGRA utilizes the tree structure to link the graphs via cross-layer edges, enabling faster search convergence and superior query performance. For deletion, DIGRA and DIGRA-HNSW perform similarly as deletions are primarily marked. However, DIGRA surpasses DIGRA-HNSW in insertion efficiency, construction time, and memory usage, underscoring the advantages of our optimized choice of ANNS index at each node.

6 CONCLUSIONS

In this paper, we introduce DIGRA, an effective index that achieves a superb balance among query efficiency, update efficiency, indexing cost, and query result quality (measured by recall). Compared to existing methods, DIGRA supports dynamic updates without compromising query efficiency or increasing indexing space, making it as competitive as all current static methods. Additionally, it can process insertions in milliseconds and deletions in microseconds.

REFERENCES

[1] 2024. Code and technical report. https://anonymous.4open.science/r/VDB-DIGRA.

[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] Lars Arge and Jeffrey Scott Vitter. 2003. Optimal External Memory Interval Management. *SIAM J. Comput.* 32, 6 (2003), 1488–1508.

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

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

[8] 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*.

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

[10] Hiroyuki Deguchi, Taro Watanabe, Yusuke Matsui, Masao Utiyama, Hideki Tanaka, and Eiichiro Sumita. 2023. Subset Retrieval Nearest Neighbor Machine Translation. In *ACL*. 174–189.

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

[12] 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).

[13] 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.

[14] 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.

[15] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *WWW*. 3406–3416.

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

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

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

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

[26] Yuxian Meng, Xiaoya Li, Xiayu Zheng, Fei Wu, Xiaofei Sun, Tianwei Zhang, and Jiwei Li. 2022. Fast Nearest Neighbor Machine Translation. In *ACL*. Association for Computational Linguistics, 555–565.

[27] 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).

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

[29] 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.

[30] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 13748–13758.

[31] 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.

[32] 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.

[33] 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.

[34] Jiuqi Wei, Botao Peng, Xiaodong Lee, and Themis Palpanas. 2024. DET-LSH: A Locality-Sensitive Hashing Scheme with Dynamic Encoding Tree for Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 17, 9 (2024), 2241–2254.

[35] Paul Wohlhart and Vincent Lepetit. 2015. Learning descriptors for object recognition and 3D pose estimation. In *CVPR*. 3109–3118.

[36] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. arXiv:2409.02571

[37] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. *CoRR* abs/2409.02571 (2024).

[38] 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.

[39] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 1 (2024), 26 pages.