GTI: Graph-based Tree Index with Logarithm Updates for Nearest Neighbor Search in High-Dimensional Spaces

Ruiyao Ma
Zhejiang University
ryma@zju.edu.cn

Yifan Zhu
Zhejiang University
xtf_z@zju.edu.cn

Baihua Zheng
Singapore Management University
bhzheng@smu.edu.sg

Lu Chen
Zhejiang University
luchen@zju.edu.cn

Congcong Ge
Zhejiang University
gcc@zju.edu.cn

Yunjun Gao
Zhejiang University
gaoyj@zju.edu.cn

ABSTRACT

Nearest neighbor search (NNS) is fundamental for high-dimensional space retrieval and impacts various fields, such as pattern recognition, information retrieval, recommendation systems, and vector database management. Among existing NNS methods, graph-based methods often excel in query accuracy and efficiency. However, these methods face significant challenges, including high construction costs and difficulties with dynamic data updates. Recent efforts have focused on combining graph methods with hashing, quantization, and tree-based approaches to address these issues, but problems with large index sizes and update performance remain unresolved. In response, this paper proposes GTI, a novel, lightweight, and dynamic graph-based tree index for high-dimensional NNS. GTI constructs a tree index built across the entire dataset and employs a lightweight graph index at the level 1 of the tree to significantly reduce graph construction costs. It also features effective data insertion and deletion algorithms that enable logarithmic real-time updates. Additionally, we have developed an effective NNS algorithm for GTI, which not only achieves approximate search performance on par with SOTA graph-based methods but also supports exact NNS. Extensive experiments on six real-world datasets demonstrate that GTI achieves an approximately 10× improvement in update efficiency compared to SOTA tree-based methods, while achieving search effectiveness comparable to SOTA approximate NNS methods. These results underscore the potential of GTI for effective application in dynamic and evolving scenarios.

PVLDB Reference Format:
Ruiyao Ma, Yifan Zhu, Baihua Zheng, Lu Chen, Congcong Ge, and Yunjun Gao. GTI: Graph-based Tree Index with Logarithm Updates for Nearest Neighbor Search in High-Dimensional Spaces. PVLDB, 18(4): 986 - 999, 2024.
doi:10.14778/3717755.3717760

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ZJU-DAILY/GTI-Graph-based-Tree-Index.

Proceedings of the VLDB Endowment, Vol. 18, No. 4 ISSN 2150-8097.
doi:10.14778/3717755.3717760

1 INTRODUCTION

Nearest neighbor search (NNS) is a crucial technique for identifying the closest data points to a query within a dataset, and it plays a fundamental role in high-dimensional space retrieval. This technique is widely applied across various domains, including pattern recognition [21, 39], information retrieval [25, 72], recommendation systems [47, 56], retriever-augmented generation [63], and vector database management [53, 70]. For instance, top-k queries, which leverage ranking function scores to retrieve the k most important query results, covering various query models, data access methods, and ranking functions [34], are similar to NNS based on object-query similarity. Thus, variants of NNS are applicable to answer top-k queries. Building on index-free approaches [17, 18] and using R-tree [32] for dataset preprocessing, recent top-k studies [48, 49] effectively combine skyline queries with top-k queries, serving a role similar to NNS. However, such tree-based methods tailored for top-k queries face limitations due to the curse of dimensionality [9, 64], resulting in reduced search performance in high-dimensional spaces. Over the past few decades, numerous methods have been proposed to enhance NNS efficiency, including tree-based methods, hash-based methods, quantization-based methods, and graph-based methods. Among these, graph-based methods stand out for representing neighbor relationships effectively and delivering highly accurate query results through greedy algorithms. They achieve an optimal balance between query efficiency and accuracy [59], and have become a preferred and popular choice in many practical applications [52].

Despite their advantages, graph-based methods face significant challenges. Index construction can be time-consuming, often taking hours or even days for billion-scale datasets, even with parallelization [6, 71]. While parallel processing accelerates construction, it also significantly increases memory usage, which limits scalability and complicates large dataset management. Additionally, graph-based methods struggle with dynamic data updates, making them less suitable for scenarios with frequent data changes. For example, major platforms like Microsoft [46] and Facebook [38] integrate graph-based methods for their search engines but face issues with real-time updates. When new data, such as recent developments in ChatGPT, become available, these methods may not update promptly, leading to degraded query performance. To address this, incremental construction approaches have been proposed [44, 45], which support data insertion but often fail with data

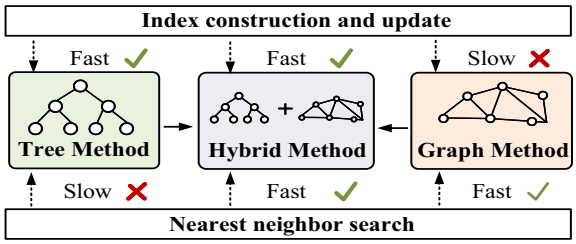


Figure 1: Motivating Example

deletions, as seen with social media content removal. Periodic reconstruction methods offer a solution for handling both insertions and deletions [55], but they incur high reconstruction costs.

Recent studies have explored hybrid approaches (e.g., integrating graph-based methods with hashing, quantization or trees) to alleviate the construction and update bottlenecks of graph-based methods. For instance, ELPIS [6] partitions the dataset using a tree index and builds subgraphs within leaf nodes, which reduces construction costs by avoiding a complete graph over the entire dataset. However, ELPIS still faces challenges with frequent data updates. Similarly, Aguerrebere et al. [1] employ quantized compressed vectors to construct and search graphs, aiming to reduce memory usage and speed up the construction and search processes, yet the graph index still struggles with updates. LSH-APG [71] leverages a lightweight LSH framework to improve search speeds and pruning conditions, thereby reducing construction and search costs while supporting data insertions and deletions. Nevertheless, its lazy deletion strategy necessitates periodic index adjustments, and the coexistence of hash and graph indexes can lead to a large index size, potentially affecting query performance.

Hybrid methods offer a promising solution by combining different approaches to overcome the limitations of existing methods. As illustrated in Fig. 1, tree-based methods are efficient in construction and well-suited to dynamic scenarios but often suffer from poor search performance in high dimensional spaces. In contrast, graph-based methods excel in fast NNS but are hampered by high construction and update costs. To address these challenges, we propose a hybrid tree-graph index that leverages the strengths of both methods: supporting dynamic updates, and maintaining efficient query performance, especially in high-dimensional spaces. However, achieving this requires overcoming three key challenges.

Challenge 1: Optimizing graph construction costs. Graphs are known for their excellent query performance but come with expensive construction costs. Although parallelization can accelerate construction, it requires substantial memory, limiting scalability and complicating the management of large datasets. Recent approaches [6, 23] attempt to mitigate these costs by hierarchically partitioning the dataset individually with the tree index to accelerate graph construction. However, they still struggle with the significant memory demands of maintaining complex neighbor relationships between objects. To address this, we propose a multi-granularity partitioning approach. By hierarchically partitioning the data using the tree index and selectively building graphs at one internal level of the tree, we can effectively reduce overhead and more efficiently manage neighbor relationships.

Challenge 2: Achieving real-time graph updates. Graph-based methods face difficulties in efficiently updating data due to their reliance on complex neighbor structures when performing searches. Frequent changes, such as product listings on e-commerce platforms like Alibaba's Taobao and content updates on social media platforms like Twitter and Instagram, create dynamic environments where existing tree-graph methods [6, 16, 23, 35, 50] struggle. To overcome this, we design efficient data insertion and deletion algorithms that enable logarithmic real-time updates. Our approach separates updates into two components: tree updates for all objects and graph updates for the selected internal nodes. This design ensures that overall update complexity remains $O(\log n)$ on average, leveraging the logarithmic time complexity of tree updates and the relatively small number of graph nodes.

Challenge 3: Maintaining efficient search performance. The hybrid tree-graph method requires a top-down search through the tree during the search phase, which can limit search performance. Furthermore, the over-reliance on tree partition information may limit the global navigation structure of the graph, resulting in declining query accuracy. Additionally, existing hybrid tree-graph methods tend to support only approximate queries, making them unsuitable for applications requiring exact searches, such as obstacles locating by robotics and autonomous vehicles for navigation and path planning, and disease diagnosis based on medical imaging. To address this, our approach diverges from traditional methods by conducting searches directly on the graph, bypassing the performance limitations of tree traversal and maintaining efficient graph query performance. By leveraging both global neighbor relationships in the graph and local neighbor relationships in the tree's leaf nodes, we ensure comprehensive dataset reachability and high query accuracy. Furthermore, our hybrid structure supports both efficient approximate and exact queries, capitalizing on the advantages of both the tree and graph components.

In summary, our key contributions are:

- *Lightweight and dynamic graph-based tree index.* We introduce GTI, a novel dynamic Graph-based Tree Index for NNS in high-dimensional spaces. GTI features a tree index built spanning the entire dataset and a lightweight graph index constructed from nodes at level 1 of the tree. This design significantly reduces both index construction time and space costs.
- *Efficient data update algorithms.* We propose effective update strategies for dynamic scenarios, encompassing algorithms for both data insertion and deletion that support logarithmic real-time updates on the proposed GTI.
- *Efficient and general search algorithms.* Building on the reduced construction costs and robust update capabilities, we develop an efficient approximate NNS algorithm for GTI, achieving performance comparable to SOTA graph-based methods. Meanwhile, GTI supports exact NNS, encompassing all queries traditionally supported by tree indexes.
- *Extensive experiments.* We conduct comprehensive experimental evaluations on six real-world datasets, demonstrating GTI's efficiency and effectiveness. GTI achieves an approximately 10× improvement in update efficiency compared to SOTA tree-based methods, offers significantly lower construction cost compared to graph-based methods, and maintains search effectiveness on par with SOTA

approximate NNS methods. These results underscore GTI's potential for widespread application in dynamic scenarios.

The rest of this paper is organized as follows. We review previous work in Section 2 and present the problem statement in Section 3. Subsequently, we introduce our newly proposed lightweight and dynamic graph-based tree index, GTI, in Section 4 and detail the nearest neighbor search process in Section 5. Finally, we report comprehensive experimental studies in Section 6 and conclude the paper with future directions in Section 7.

2 RELATED WORK

In this section, we review the existing work on nearest neighbor search in high-dimensional spaces.

2.1 Basic solution for nearest neighbor search

Numerous methods have been proposed to efficiently answer nearest neighbor search in high-dimensional spaces. They can be classified into four main categories: hash-based methods, quantization-based methods, tree-based methods, and graph-based methods.

Hash-based methods. Hash-based methods [30, 31, 43, 65, 69] map high-dimensional data objects into several low-dimensional hash buckets and answer queries by locating the hash buckets of real answers. Although hash-based methods provide theoretical guarantees on precision, they face several challenges, including low-quality results and inadequate support for range queries and other complex search functionalities.

Quantization-based methods. Quantization-based methods [29, 36, 54, 60, 64] partition data objects based on quantified feature values and search for candidate points with the same quantization value as the final results. However, quantization errors can make it difficult to achieve high query accuracy, especially in high-dimensional spaces.

Tree-based methods. Tree-based methods [4, 7, 11–13, 19, 24, 32, 40, 41, 51, 57, 61, 67, 68, 74] divide the data space into subspaces using pivot mapping or hyperplane partitioning and construct tree indexes to manage these subspaces. By reducing the search space to points within overlapping subspaces, tree-based methods can support both approximate and exact searches with relatively low indexing overhead and can adapt flexibly to dynamic scenarios.

Graph-based methods. Graph-based methods leverage the nearest neighbor relationships among objects to construct proximity graphs, facilitating approximate nearest neighbor search. These methods generally provide superior query performance in terms of both accuracy and efficiency compared to other methods [5]. Existing graph-based methods often rely on base graphs such as Delaunay Graph, Relative Neighborhood Graph, K-Nearest Neighbor Graph and Minimum Spanning Tree [59]. Despite their advantages, graph-based methods face high construction cost and difficulties with updates [71]. To mitigate construction costs, approaches such as approximate proximity graph [22, 27, 28] and parallel construction strategies [27, 28, 45] have been proposed, though they still result in large index sizes. On the other hand, to handle updates, periodic reconstruction methods [55] have been developed, but they are hindered by high reconstruction costs.

2.2 Hybrid solutions

Given the superior query performance of graph-based methods, there has been exploration of graph-based hybrid solutions that aim to enhance overall performance by integrating graphs with trees [16, 26, 35, 50], hash tables [37], and quantized objects [42]. While such hybrid methods improve the graph search performance, they still face significant challenges, including high graph construction costs and difficulties in handling dynamic data operations.

Recent research has shifted focus towards reducing construction costs and supporting dynamic real-world applications. Prominent examples include ELPIS [6] and LANNS [23] that effectively reduce graph construction costs through hierarchical partitioning information from trees, LVQ [1] that provides a generalized quantization framework to improve data compression and accelerate graph navigation, LSH-APG [71] that employs a lightweight LSH framework to provide high-quality search seeds and pruning conditions for graph, and MNG [55] that employs inner product technique to accelerate graph navigation and introduces a lazy deletion strategy with periodic rebuild operation to facilitate graph updates.

However, the graph size of the aforementioned methods remains considerable, posing challenges for graph storage in main memory and for handling dynamic updates. The recent study SPFresh [66] introduces a lightweight incremental rebalancing protocol designed specifically for updating posting lists stored on disk. While it adopts a rebuild strategy that works effectively with small graphs based on posting centroids, this method is inherently tailored for disk-based data management systems. Consequently, SPFresh is not generalizable for main memory data management and fails to support scenarios requiring real-time updates. Motivated by these challenges and the promising potential of graph-tree hybrid methods [62], we propose a novel dynamic graph-based tree index, GTI. GTI addresses construction bottlenecks by building a lightweight graph on top of a tree index and supports efficient real-time updates through graph-based single-point retrieval and logarithmic tree update operations. Consequently, GTI achieves more efficient index updates and comparable construction efficiency to tree-based methods, while maintaining the high search performance of approximate graph-based methods and supporting exact similarity searches.

Table 1: Symbols and description

Notation	Description
q, o, O	A query, an object and an object set
n	The number of objects in object set O, i.e., $\|O\|$
$d(\cdot,\cdot)$	A distance function
N, e, v	A tree node, a tree entry and a graph vertex
N_c	The node capacity of the tree
m	The graph sparsity

3 PROBLEM FORMULATION

We begin by formally defining the nearest neighbor search and range queries. Table 1 summarizes the frequently used notations.

Nearest neighbor search is a fundamental problem in database management and artificial intelligence, and is widely applied to high-dimensional spaces. The k-nearest neighbor search (k-NNS) problem is defined as follows:

Definition 3.1. **(k Nearest Neighbor Search.)** Given a set of objects O in a high-dimensional space, a query object q, a distance function $d(\cdot,\cdot)$, and an integer k, a k-nearest neighbor search (k-NNS) finds a set S of $\min(k, |O|)$ objects in O that are closest to q. Formally, $k\text{-NNS}(q,k) = \{S|\ S \subseteq O \wedge |S| = \min(k,|O|) \wedge \nexists s \in S, o \in O - S, d(q,s) > d(q,o)\}$.

Exact k-NNS in large high-dimensional datasets can be computationally intensive. A more efficient and potentially more practical

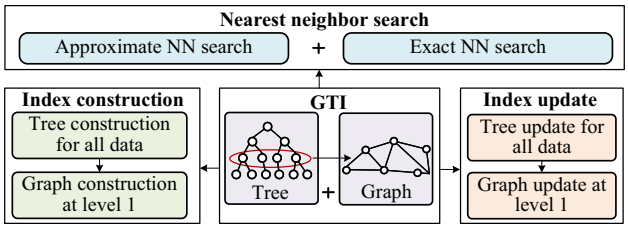


Figure 2: The overall framework of GTI

alternative is the approximate nearest neighbor search (ANNS), defined as follows:

Definition 3.2. **(ϵ-Approximate k Nearest Neighbor Search.)** Given a set of objects O in a high-dimensional space, a query object q, a distance function $d(\cdot,\cdot)$, an integer k, and a small constant $\epsilon > 0$, an approximate k nearest neighbor search (AkNNS) finds a set S of $\min(k, |O|)$ objects in O such that $\text{A}k\text{NNS}(q,k) = \{S|\ S \subseteq O \wedge |S| = \min(k,|O|) \wedge \nexists s \in S, o \in O - S, d(q,s) > (1+\epsilon)d(q,o)\}$.

To simplify modeling and evaluation, the metric *recall* can be used to approximate the exact value of ϵ, which can be defined as follows [30]:

$$recall(\text{A}k\text{NNS}(q,k)) = \frac{|k\text{-NNS}(q,k) \cap \text{A}k\text{NNS}(q,k)|}{|k\text{-NNS}(q,k)|}, \quad (1)$$

In addition to nearest neighbor search, range queries are also commonly used in database management, which are defined as:

Definition 3.3. **(Range Query.)** Given a set of objects O in a high-dimensional space, a query object q, a distance function $d(\cdot,\cdot)$, and a search radius r, a range query (RQ) finds all objects in O that are within a distance r from the query q. Formally, $RQ(q,r) = \{o|\ o \in O \wedge d(q,o) \leq r\}$.

In this paper, we focus on how GTI efficiently answers general nearest neighbor search in dynamic scenarios, supporting both approximate and exact queries. Additionally, due to the nature of its hybrid tree-graph structure, our proposed method GTI naturally supports exact range queries.

4 INDEX

In this section, we present an overview of our graph-based tree index GTI, examine the construction and update strategies to support dynamic scenarios effectively, and provide a theoretical analysis of space consumption and time complexity.

4.1 Overview

Graph-based methods excel at representing neighbor relationships and achieve efficient approximate NNS, while tree-based methods offer low construction and update costs. Inspired by this, we propose the Graph-based Tree Index (GTI), which combines the querying advantages of graphs with the dynamic benefits of tree indexes. GTI supports lightweight index construction and dynamic real-time updates, while enabling efficient and comprehensive nearest neighbor searches. As illustrated in Fig. 2, GTI integrates tree index and graph index built on one internal level of the tree. We first utilize a tree structure to hierarchically partition the dataset and then selectively construct a graph at the tree's one internal level, thus reducing overall construction costs. Meanwhile, GTI supports flexible updates through efficient tree and lightweight graph updates, achieving logarithmic complexity on average. Finally, GTI accommodates both approximate and exact nearest neighbor searches efficiently, meeting diverse query requirements.

4.2 Index Structure

The proposed hybrid index GTI consists of two key components: (i) **a tree index** that efficiently partitions datasets hierarchically, maintains local neighbor information within nodes, and reduces construction and update overhead; and (ii) **a graph index**, lightly constructed at one internal level of the tree, which establishes global adjacency relationships and enables fast navigation to locate query regions. For a clearer understanding, Fig. 3 provides a comprehensive illustration of the GTI structure.

The Tree Index. The tree index is built upon the complete dataset, with nodes (entries) maintaining local neighborhood relationships. To efficiently construct the tree index and support data updates, we use a structure similar to the M-tree [19], which exhibits strong performance among dynamic indices according to a recent survey [15]. Specifically, the tree organizes data hierarchically using ball partitioning, where each node represents a hypersphere defined by a center and radius. Leaf nodes correspond to the smallest hyperspheres that contain neighboring objects, while non-leaf nodes represent larger hyperspheres encompassing groups of nearby hyperspheres. The tree is incrementally constructed from the bottom up, creating nested hyperspheres that progressively partition the dataset into clusters.

Each node in GTI comprises multiple entries, with each entry holding an object and the distance pd from the object to the center of its parent entry. Furthermore, a non-leaf entry includes a pointer to its subtree node, along with the center point and the radius of the hypersphere represented by the subtree node. As illustrated in Fig. 3, node N_2 contains three non-leaf entries e_3, e_4, and e_5. The last entry, e_5, points to its subtree node N_6, which represents a hypersphere (depicted as a circle in a two-dimensional plane) with center o_3 and radius r_5, and stores the distance $d(o_3, o_7)$ to the center of its parent entry e_1. The number of entries in a node is constrained by the node capacity N_c. As shown in Fig. 3, given a node capacity of 3, there are up to 3 entries within each node. Additionally, leaf and internal nodes may have different node capacities in practice to accommodate the graph (see Sec. 6.3 for details).

The Graph Index. The graph index is constructed from nodes located at the level immediately above the leaf entries of the tree (e.g., Level 1 in Fig. 3 where leaf entries are at Level 0). This design maintains global neighborhood relationships to facilitate rapid query navigation. Leveraging the hierarchical structure of SOTA graph-based method HNSW [45], the GTI graph enables efficient and high-quality searches by starting from the top layer of the graph and progressively delves deeper. Specifically, the top-layer graph contains the fewest vertices and the longest edges. As we move to the lower layers, the edges become shorter and the number of vertices increases. Vertices in the upper layer are always included in the lower layers, culminating in the bottom layer, which includes all vertices. The graph is built incrementally. For each new vertex, a random integer l is chosen, defining the maximum number of layers to which the vertex belongs, e.g., $l = 2$ implies that a vertex only appears in two layers, layer 0 and layer 1. A proximity graph is then built incrementally for all vertices at each layer. By using an exponentially decaying probability for l, the expected number of layers in the graph scales logarithmically. This random strategy to determine the maximum layer height for vertices has been demonstrated to be effective [45].

To enhance search quality and mitigate the potential issue of local optima due to incomplete vertices on the graph, each vertex at the bottom layer of the graph stores a pointer to the corresponding leaf node in the tree. This pointer directs to the child node of the corresponding tree entry. When accessing the bottom layer of the graph during the search, we simultaneously search both the vertices on the graph and the associated leaf nodes in the tree, leveraging both global neighbor relationships on the graph and the local neighbor relationships within the leaf nodes in the tree. As illustrated in the *Tree* and *Graph* parts of Fig. 3, vertex o_3 at the bottom layer of the graph corresponds to entry e_5 in level 1 of the tree and has a pointer to the child node of e_5, i.e., the leaf node N_6. When a search accesses vertex o_3 on the bottom layer of the graph, it also accesses objects within N_6 through the pointer. Due to space constraints and to enhance the clarity of our drawing, we only include the pointers for vertices o_3 and o_6 and omit pointers for the other vertices in the bottom layer of the graph shown in Fig. 3.

Additionally, we maintain an in-degree radius table that stores the maximum distance between each vertex and its reverse neighbors (in-degree edges) across all layers. This facilitates locating reverse neighbors of each vertex to support deletion operations on the graph (see Section 4.4). As shown in the *Graph* part of Fig. 3, the reverse neighbors of o_2 at different layers of the graph are $\{o_1, o_7, o_9\}$ respectively, with the maximum distance being $d(o_2, o_9)$.

4.3 Index Construction

Graph indexes accelerate approximate nearest neighbor searches but are expensive to construct and require substantial memory to store neighborhood relationships. Optimization techniques, such as refining graph structure [22, 27, 28], employing parallelization [27, 28, 45], and leveraging tree partitioning information [6, 23] can mitigate computational overhead. However, challenges remain due to high space costs, as the graph must index the entire dataset.

In contrast to existing methods that index every data object directly, our approach first utilizes a tree index to hierarchically partition the dataset and then constructs a graph index from the nodes located at level 1 of the tree. This approach reduces both the time and space costs. During graph construction, we maintain an in-degree radius table to facilitate efficient index maintenance. Additionally, to enhance index search quality, we incorporate pointers from graph vertices to the tree's leaf nodes.

Algorithm 1 outlines the construction process of our index. The input to the algorithm is an object set O, the node capacity N_c, and the graph sparsity m which controls the maximum number of the neighbors of each vertex in the graph. The output is the GTI index I. Firstly, the GTI index is initialized by encompassing both a tree index and a graph index (line 1). Subsequently, it builds the tree index to hierarchically partition the entire dataset following the insertion operation of the M-tree [19] (lines 2–4), which is one of the SOTA tree-based methods according to the survey [15].

Next, the graph index is constructed on the entries located at level 1 immediately above the leaf entry level of the tree (lines 5–10). Initially, all entries from level 1 of the tree are retrieved (line 5), and an empty in-degree radius table of the graph is initialized (line 6). Subsequently, the *center* of each entry is incrementally inserted into the graph (lines 7–9). Similar to the SOTA graph-based method HNSW [45], vertices are inserted into the graph at various

Algorithm 1: Index Construction

Input: an object set O, the node capacity N_c, and the graph sparsity m
Output: the GTI index I

1: $I_T \leftarrow \emptyset, I_G \leftarrow \emptyset$
2: **foreach** $o_i \in O$ **do**
3: create leaf entry e, $e.center \leftarrow o_i$, $e.pd \leftarrow \infty$
4: InsertTree(I_T, e, N_c) // tree construction
5: $E \leftarrow$ entries in $Level$ 1 of I_T
6: $IR \leftarrow \emptyset$ // in-degree radius table
7: **foreach** $e_i \in E$ **do**
8: $v \leftarrow e_i.center$, $leaf \leftarrow e_i.child$
9: InsertGraph(I_G, v, IR, m) // graph construction
10: add pointer from v to $leaf$
11: **return** $I_T \cup I_G$

layers, and a top-down graph search is utilized to find neighbors for each vertex within their respective layers. The maximal number of neighbors of a vertex in the upper layers of the graph is m, and that in the bottom layer is $2m$ [45]. Meanwhile, we also synchronize the recording of the maximum in-degree edge distance (radius) of each vertex and store it in the in-degree radius table (line 9), to support subsequent efficient index updates. Subsequently, in the bottom layer of the graph, a pointer is added to each vertex, directing it to the corresponding leaf node of its associated entry in the tree (line 10). Finally, the tree index and the graph index are returned together, forming the finalized GTI index (line 11).

4.4 Index Updating

In real-life scenarios, data insertion and deletion are very frequent. Existing graph-based methods often face challenges with these dynamic updates due to their reliance on maintaining global navigational relationships. For instance, the graph-based method NSG integrated into Alibaba's Taobao e-commerce platform struggles with dynamic updates [28]. While some graph methods [44, 45] handle data insertion through incremental construction, they typically face difficulties with data deletion, such as in scenarios involving product discontinuation. Although periodic reconstruction strategies with lazy updates [55] have been proposed to address both data insertion and deletion, the high cost of reconstruction remains a significant limitation.

Our GTI index combines the strengths of tree and graph structures, enabling flexible data insertion and deletion while preserving efficient graph query performance post-update. The update process of GTI index involes two components: updating the tree index and updating the graph index. The tree index update is highly efficient, requiring adjustments only to the affected local nodes. For the graph index update, we design an efficient data update algorithm and utilize the exact search capabilities of the tree structure to assist the update on the graph. Since the graph index is built on the entries on the tree's level one rather than the entire dataset, graph update operations are relatively lightweight. Consequently, the GTI index update operates within logarithmic complexity relative to the tree update (as detailed in Section 4.5). We detail the object insertion and deletion algorithms for the GTI index below.

Object Insertion. The insertion of an object into the GTI index involves both tree and graph insertions. When a new object comes,

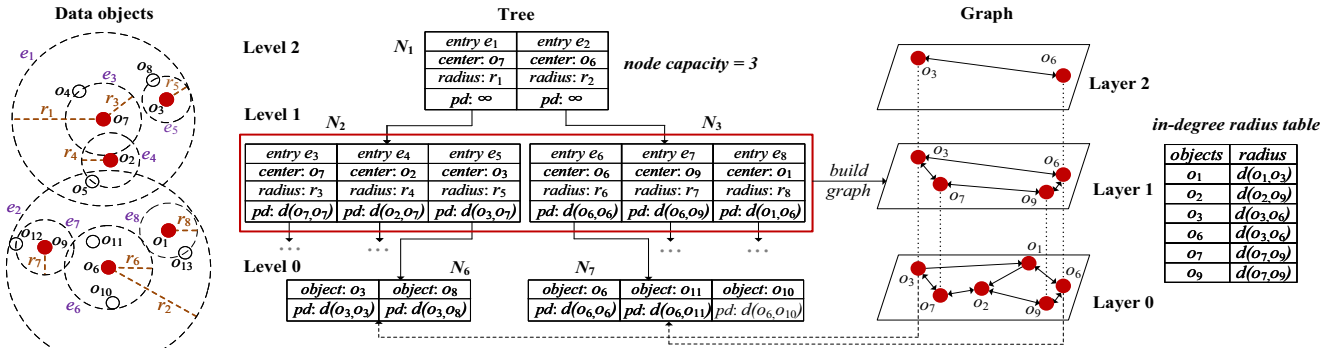


Figure 3: The structure of our proposed graph-based tree index, GTI

Algorithm 2: Object Insertion

Input: a new object o to be inserted, and a GTI index I
Output: the updated GTI index I

1: $I_T, N_c \leftarrow$ tree index information from I
2: create leaf entry e, $e.center \leftarrow o$, $e.pd \leftarrow \infty$
3: InsertTree(I_T, e, N_c) // tree insertion
4: **if** *insertion of* e *leads to leaf node split* **then**
5: $\quad I_G, IR, m \leftarrow$ graph index information from I
6: $\quad v \leftarrow center$ of the new entry in I_T's *Level* 1
7: $\quad$ InsertGraph(I_G, v, IR, m) // graph insertion
8: $\quad$ add pointer from bottom-layer v to new entry's child leaf node
9: **return** $I_T \cup I_G$

Algorithm 3: Object Deletion

Input: an object o, a GTI index I, and a search parameter ef
Output: the updated GTI index I

1: $I_T \leftarrow$ tree index information from I
2: $I_G, IR, m \leftarrow$ graph index information from I
3: $o' \leftarrow$ **A**k**NNSearch**(o, I, ef, 1)
4: **if** $o' \neq o$ **then** $o' \leftarrow$ RangeSearchT(o, 0, I_T)
5: **if** $o' \neq o$ **then return** I // o does not exist in I
6: DeleteTree(o', I_T) // tree deletion
7: **if** *deletion of* o' *leads to leaf node underflow* **then**
8: $\quad v \leftarrow center$ of the deleted entry in I_T's *Level* 1
9: $\quad$ delete pointer to v's leaf node from bottom-layer of the graph
10: $\quad C \leftarrow$ RangeSearchT(v, $IR[v]$, I_T)
11: $\quad N \leftarrow$ reserve neighbors of v in C
12: $\quad$ DeleteGraph(I_G, N, v, IR, m) // graph deletion
13: $\quad$ **foreach** $r \in$ *objects to be reinserted* **do**
14: $\quad\quad I_G \leftarrow$ InsertGraph(I_G, r, IR, m) // graph reinsertion
15: **return** $I_T \cup I_G$

it is first inserted into the tree index. If this insertion triggers a node split in the tree, i.e., a new entry is added into level 1 of the tree, the graph must also be updated.

Algorithm 2 outlines the process for updating the GTI index with a new object o. The algorithm takes as input the new object o and the current GTI index I, and outputs the updated GTI index I. Firstly, we insert the new object into the tree index in GTI (lines 1–3). Next, we determine if the graph index needs updating. If the insertion of o causes a leaf node split in the tree, leading to a new entry in level 1, GTI synchronizes the graph insertion (lines 4-8). It first initializes the graph index information, including the graph I_G, the in-degree radius table and the graph sparsity (line 5). Then, the algorithm inserts the new entry's *center* as a vertex to the graph, similar to HNSW (lines 6–7). It also adds a pointer from the new vertex at the graph's bottom layer to the new entry's child node in the tree (line 8). Note that we follow the confirmed *M_LB_DIST* [19] strategy for node splitting, ensuring that each node split results in only one new entry in the upper layer. If no leaf node split occurs, level 1 of the tree remains unchanged, and no graph update is required. Finally, the GTI index update completes (line 9).

Object Deletion. The deletion of objects in the GTI index involves two main processes: tree deletion and graph deletion. First, we locate and efficiently remove the target object from the tree index. If this deletion causes a node underflow in the tree, such as the removal of an entry from level 1 of the tree, a graph update becomes necessary. When a vertex in the graph needs to be deleted, we remove it along with its corresponding in-degree and out-degree edges across all layers. Deleting out-degree edges is straightforward, as they are directly stored in the graph's adjacency list. However, deleting in-degree edges is more complex due to the potentially large and variable number of in-degree edges. Directly storing all in-degree edges could lead to significant space inefficiencies. Consequently, finding all in-degree edges, or reverse neighbors, involves traversing all vertices, which is time-consuming. To address this challenge, we develop an efficient graph deletion algorithm. This algorithm leverages the maximum in-degree radius of vertices stored in the in-degree radius table, combined with range queries on the tree, to efficiently locate and delete all reverse neighbors, ensuring accurate edge removal.

Algorithm 3 outlines the object deletion process in the GTI index. It takes as input an object o, a GTI index I, and a search parameter ef, and outputs the updated GTI index I. The process begins by retrieving information about both the tree and graph indexes (lines 1–2). To locate the object to be deleted, it first conducts an approximate 1-NN search using the GTI index (line 3). If this search fails to locate the object, this algorithm proceeds with an exact range query [19] on the tree index, using o as the center and 0 as the radius to find the object (line 4). Should the exact range query also fail, it indicates that the object to be deleted does not exist, and the deletion operation terminates (line 5). Our algorithm prioritizes the AkNN search to locate the target object due to its superior performance compared to the exact range search.

If the object is successfully located, the deletion process continues (lines 6-15). The object is initially removed from the tree according to the deletion rules of the M-tree [19] (line 6). Subsequently, we check whether a graph update is necessary. If the

Algorithm 4: AkNN Search

Input: a query object q, a GTI index I, a search parameter ef, and an integer k
Output: approximate k nearest neighbors of q

1: $I_T, I_G \leftarrow$ tree and graph indices from I
2: $C \leftarrow \emptyset$ // set for the current NNs
3: $p \leftarrow$ entry point of I_G // a random vertex in the top-layer graph
4: $L \leftarrow$ number of layers in the graph
5: **foreach** $l \leftarrow L-1$ *to* 1 **do**
6: $\quad C \leftarrow$ SearchUpperLayer(q, p, 1, l, I_G)
7: $\quad p \leftarrow$ nearest object from C to q
8: $C \leftarrow$ **AefNNSBottomLayer**(q, p, ef, I)
9: **return** k nearest neighbors in C

deletion causes a leaf node to become empty, leading to the removal of both the leaf node and its parent entry in level 1, the graph must be updated to reflect these changes (lines 7-14). In this context, the vertex v corresponding to the *center* of the deleted entry is identified in the graph (line 8). The pointer to the corresponding leaf node is then removed from the graph bottom layer (line 9). To address the remaining connections, an exact range query is performed on the tree with a radius equivalent to v's in-degree radius (line 10). The results from this range query include all reverse neighbors of v across different layers in the graph (line 11). For example, if object o_2 in Fig. 3 is to be deleted, the corresponding range query radius from the in-degree radius table is $d(o_2, o_9)$, and the identified reverse neighbors are $\{o_1, o_7, o_9\}$. Subsequently, we delete the corresponding vertex, along with all its associated edges from the graph (line 12). It is important to note that such deletions may reduce the number of neighbors for some vertices, potentially causing them to violate the graph construction rules. Meanwhile, visiting the neighbors of these vertices does not guarantee to find their true nearest neighbors. To address this, the algorithm reinserts these vertices into the graph to re-establish their correct set of neighbors (lines 13-14). Finally, the algorithm returns the updated GTI index to complete the object deletion process (line 15).

4.5 Index Complexity Analyses

Space Consumption of Index. Our index consists of two components, i.e., the tree index and the graph index. Let n denote the cardinality of the object set (i.e., $|O|$) and N_c denote the node capacity. The tree index is built on the entire dataset and is balanced. Its space cost depends only on the number of tree nodes, which is $O(n)$. For graph index, it is a hierarchical graph built based on nodes located at level 1 of the tree. Its space cost depends on the number of vertices and the edges. The number of vertices is $O(\frac{n}{N_c})$, while the number of edges is related to the number of layers, which is $O(\log \frac{n}{N_c})$. Considering that the number of neighbors of each vertex in each layer of the graph is a constant related to m, the number of edges can be estimated as $O(\log \frac{n}{N_c} \cdot m)$. Thus, the space cost of the graph index is $O(\frac{n \cdot m}{N_c} \cdot \log \frac{n}{N_c})$. Consequently, the overall space complexity of our index is $O(n + \frac{n \cdot m}{N_c} \cdot \log n)$.

Time Complexity of Index Construction. Index construction consists of two parts: tree construction and graph construction. For the tree index, the overall complexity for constructing the tree index is $O(n \log n)$. For the graph index built on the nodes located at level 1 of the tree, a top-down search is conducted to find each vertex's neighbors at each layer of the graph. The time complexity of this search is equivalent to the complexity of searching in HNSW, which is $O(\log \frac{n}{N_c})$ [45, 55]. HNSW [45] demonstrates logarithmic search complexity in low-dimensional spaces, while MNG [55] unifies the search complexity of HNSW to $O(n'^{\frac{2}{D}} \log n')$, where D is the dimensionality, and n' is the number of graph vertices. When the dimensionality is high, as in this paper ($D = 128$ and $n' = 1,000,000$), $n'^{\frac{2}{D}} = 1.2 \approx 1$, so the time complexity can be simplified to $O(\log n')$ for both low-dimensional and high-dimensional spaces. Therefore, the overall complexity for graph construction is $O(\frac{n}{N_c} \log n)$. Consequently, the overall time complexity of index construction for GTI is $O(n \log n + \frac{n}{N_c} \log n)$.

Time Complexity of Index Updating. Index updating involves two main operations: object insertion and deletion. For object insertion, we first add the object to the tree. If this insertion causes a tree node split in the tree - an event that is relatively infrequent compared to the overall update - we then proceed to update the graph. Due to the logarithmic nature of both tree and graph structures, the time complexity for insertion is $O(\log n)$. For object deletion, we first locate the object to be deleted through a range search of radius 0, which has a complexity of $O(\log n)$ according to previous studies [14]. After locating the object, we remove it from the tree, which also has a time complexity of $O(\log n)$. If the deletion causes a node underflow in the tree - another relatively rare event - we then perform a deletion operation on the graph. This includes locating reverse neighbors using a tree range query, deleting nodes and edges across all layers, and re-inserting a few objects into the graph. The time complexities for these operations are $O(\log n)$, $O(\log \frac{n}{N_c})$, and $O(\log \frac{n}{N_c})$, respectively. Therefore, the time complexity of object deletion is $O(\log n)$. In conclusion, the overall time complexity for index updating remains logarithmic.

5 NEAREST NEIGHBOR SEARCH

In this section, we propose efficient algorithms for nearest neighbor search in high-dimensional spaces using GTI, including approximate nearest neighbor search and exact nearest neighbor search.

5.1 Approximate Nearest Neighbor Search

In this section, we detail the approach to achieving efficient approximate query performance within the tree-graph structure of GTI. Given a query q, the approximate nearest neighbor search finds the approximate nearest neighbors of q in the dataset. Existing hybrid tree-graph methods [6, 26, 35] often rely on a top-down tree search to accelerate the graph search by leveraging the tree's partition information. However, their search performance is constrained by the need for hierarchical tree traversal and an over-reliance on tree partitioning, which can misguide the search and lead to local optima. Consequently, these methods fail to fully exploit the global navigation advantages offered by the graph structure.

To address the aforementioned issues, our approach deviates from the traditional approaches of first searching the tree and then the graph. Instead, we initiate the search directly within the graph. This strategy avoids the performance limitations imposed by hierarchical traversal and misleading tree partitioning, leading to more efficient graph-based query performance. During the graph search, we simultaneously leverage both global neighborhood relationships within the graph and local neighborhood relationships of tree leaf nodes. This dual approach ensures comprehensive coverage of the

Algorithm 5: AefNNS at Bottom Layer

Input: a query object q, an entry point p, a search parameter ef, and a GTI index I
Output: approximate ef nearest neighbors of q

1: $I_T, I_G \leftarrow$ tree and graph indices from I
2: $V \leftarrow \{p\}$ // the set of visited vertices
3: $Q \leftarrow \{\langle p, d(q,p)\rangle\}$ // the priority queue to store candidate vertices
4: $A \leftarrow \{p\}$ // the global nearest neighbors of q
5: $dis_{max} \leftarrow \infty$ // the local maximum distance to q among objects in A retrieved from the graph
6: **while** $Q \neq \emptyset$ **do**
7: $\quad \langle c, dis\rangle \leftarrow Q.pop()$
8: $\quad$ **if** $dis > dis_{max}$ **then break**
9: $\quad$ **foreach** $c' \notin V \wedge c' \in$ *neighbors of* c *in bottom layer of* I_G **do**
10: $\quad\quad V \leftarrow V \cup \{c'\}$
11: $\quad\quad$ **if** $d(c', q) < dis_{max}$ **then**
12: $\quad\quad\quad$ update Q, A, dis_{max} with $\langle c', d(c', q)\rangle$
13: $\quad\quad\quad$ **foreach** $s \in$ *leaf node in* I_T *pointed to by* c' **do**
14: $\quad\quad\quad\quad$ update A with $\langle s, d(s, q)\rangle$ if possible
15: **return** ef nearest neighbors in A

dataset and preserves the integrity of neighborhood relationships, ultimately improving the quality of query results.

Algorithm 4 outlines the procedure for performing an approximate k nearest neighbor search using our index. The algorithm takes as input a query object q, a GTI index I, a search parameter ef, and an integer k, and outputs approximate k nearest neighbors of q. The process begins by retrieving the tree and graph information from the GTI (line 1). It then initializes a set C to hold the candidate nearest neighbors for graph expansion (line 2), selects a random vertex p from the top layer of the graph as the entry point (line 3), and records the total number of layers L in the graph (line 4).

Next, the algorithm traverses the graph from the top layer down to layer 1 (lines 5–7). It employs a hierarchical search method of HNSW [45], similar to the process of finding neighbors for each point during the graph's incremental construction. Specifically, starting at the entry point p in the top layer, the algorithm performs a greedy search [28, 45] to expand the neighbors of the current vertex until the approximate 1-NN neighbor for the query is found (line 6). It uses this 1-NN as the entry point for the next lower layer (line 7), iterating through each layer until reaching layer 1. Once q's 1-NN in layer 1 is identified, the algorithm proceeds to search for q's approximate ef-NNs within the bottom layer of the graph (line 8). This process includes both the neighbors directly accessible in the bottom layer of the graph and the leaf nodes in the tree structure pointed to by the graph vertices at the bottom layer (i.e., layer 0). Finally, the k nearest neighbors from the candidate set are returned as the final results for the query (line 9).

Algorithm 5 outlines the details of searching the bottom layer of the graph to find the approximate ef-NNs for a given query object. The algorithm takes as input a query object q, an entry point p, a search parameter ef, and a GTI index I, and outputs q's approximate ef nearest neighbors. The process begins by retrieving the tree and graph indexes from the GTI (line 1). The algorithm then initializes several structures: a set V to track visited vertices, a priority Q to store candidate vertices sorted by their distances to q, a set A to store the current nearest neighbors of q, and a distance dis_{max} to keep track of the local maximum distance to q among objects

Algorithm 6: kNN Search

Input: a query object q, a GTI index I, a search parameter ef, and an integer k
Output: exact k nearest neighbors of q

1: $C \leftarrow$ **AkNNSearch**(q, I, ef, k)
2: $I_T \leftarrow$ tree index from I
3: $R \leftarrow k$SearchT(C, I_T, k)
4: **return** k nearest neighbors in R

in A retrieved from the graph (lines 2-5). The search proceeds by exploring layer 0 of the graph and the leaf nodes pointed to by the visited vertices until Q becomes empty or other termination criteria are met (lines 6-14). Specifically, the algorithm retrieves the vertex c from Q that is closest to q and compares its distance to q with dis_{max}. If c's distance exceeds dis_{max}, the algorithm terminates the search early, as the remaining vertices in Q are unlikely to contribute significantly to the query results (lines 7-8). Otherwise, the algorithm continues to examine each un-visited neighbor c' of c in the bottom layer of the graph.

For each candidate vertex c', if its distance to q is within dis_{max}, the algorithm enqueues c' to Q, adds c' to A and updates dis_{max} if necessary. Specifically, dis_{max} is updated if including c' in A changes q's distance to its ef-th nearest neighbor among graph objects in A (lines 9-12). Note that dis_{max} captures the local maximum distance to q from existing result objects in A that are retrieved from the graph, excluding those from the tree. The algorithm then examines each object s in the leaf nodes pointed to by c'. If s is closer to q than q's furthest object in A, the algorithm updates the result set A with s (lines 13-14). For instance, accessing object o_6 in the bottom layer triggers access to additional objects $\{o_{10}, o_{11}\}$ within leaf node N_7 pointed to by o_6, as illustrated in Fig. 3. Notably, the access of objects in leaf nodes does not influence the search condition within the graph. Instead, it updates A if any objects in the leaf nodes are closer to q than the current objects in A, without altering dis_{max}. Finally, the algorithm returns the ef nearest neighbors from A (line 15).

5.2 Exact Nearest Neighbor Search

Benefiting from the tree-graph structure of our GTI index, we support the exact nearest neighbor search efficiently. Traditional exact k-NNS methods on tree indexes treat k-NNS as a specialized range query [19]. The radius is initially set to infinite. As the search progresses, the radius is updated to the distance from q to its current k-th nearest neighbor and is gradually reduced until the true k-NNs are found. However, an excessively large radius early in the search can lead to inefficient tree pruning, causing unnecessary calculations. To address this inefficiency, we propose initializing the search radius with a high-quality approximate k-th nearest neighbor distance, derived from the approximate k-NN search algorithm described in Section 5.1. Such a high-quality initial radius helps improve pruning efficiency and accelerate the search process.

Algorithm 6 outlines the exact k nearest neighbor search process using our index. The algorithm takes as input a query object q, a GTI index I, a search parameter ef, and an integer k. It outputs the exact k nearest neighbors of q. First, we execute an approximate k-NNS to find the approximate k-NNs of the query (line 1). Next, the algorithm performs the exact k-NNS on the tree index using the high-quality initial radius instead of an infinite radius traditionally

used in tree-based searches [19] (lines 2–3). Finally, the algorithm returns the results of the exact search (line 4).

Remark. Our GTI method integrates the SOTA M-tree [15], which inherently supports exact range queries, such as finding all objects within a given search radius from a query. However, the M-tree index only supports exact range queries using metrics that satisfy the triangle inequality. For scenarios where the triangle inequality is not satisfied, the M-tree index employed in GTI can be substituted with a Cover tree index [8, 33].

5.3 Search Complexity Analyses

The approximate nearest neighbor search in GTI consists of two components: searching for neighbors within the hierarchical graph and examining corresponding leaf nodes in the tree. The time complexity for searching the hierarchical graph, constructed from the entries at level 1 of the tree, equals the number of layers multiplied by the average number of neighbors in graph. This can be expressed as the search path length times the cost of a single search. Considering that the number of neighbors in graph is a constant related to m, and the number of layers in the graph is $O(\log \frac{n}{N_c})$ [45, 55], the complexity becomes $O(m \log n)$.

Each candidate vertex in the bottom layer of the graph has N_c more neighbors connected to the associated leaf entry in the tree index. As a result, the number of neighbors of each vertex in the bottom layer of graph is $m' = 2m + N_c$. Note that the size of candidate vertexes is related to the search parameter ef, the complexity of searching their corresponding leaf nodes is $O(N_c \cdot ef)$. Consequently, the overall time complexity of the approximate search on GTI is $O(m \log n + N_c \cdot ef)$.

6 EXPERIMENTS

In this section, we conduct empirical experiments to evaluate the performance of our proposed graph-based tree index GTI against its competitors, considering the construction and update costs, the search performance, and the scalability.

6.1 Experimental Settings

Datasets. We employ six widely used real-life datasets for evaluating NNS methods [6, 28, 45, 55, 58, 59, 71, 73] in our study. These datasets are generated from different data types and vary in cardinality and distributions: (i) *Deep* [71], image vectors extracted from the last layers of a convolutional neural network; (ii) *Msong* [3], a collection of one million western popular music pieces; (iii) *Gist* [2], one million images with 960 dimensions; (iv) *Color* [10], image features extracted from Flickr; (v) *Turing* [58], Bing textual queries encoded by Turing AGI v5; and (vi) *Bigann* [2], SIFT vectors representing image feature descriptions. Table 2 summarizes all the datasets used, where *Dim.* denotes the dataset dimension. The query workloads in this paper are derived from *Deep*, *Msong*, *Gist*, *Turing* and *Bigann*, or sampled from *Color*.

Baselines. To evaluate our proposed graph-based tree index GTI, we compare it with multiple baseline methods: (i) two well-established and efficient tree-based methods for exact NNS and range queries [15], including the dynamic index M-tree [19] and the static index MVPT [11, 12]; (ii) two SOTA graph-based methods for ANNS [59], including HNSW [45] and NSG [28], where the advanced MNG [55] update strategy is applied to both methods; and (iii) two recent hybrid methods for ANNS, including the dynamic hash-graph combination method LSH-APG [71], and the static tree-graph combination method ELPIS [6]. Note that we do not select MNG as a separate baseline, because it requires fine-tuning for different datasets and becomes NSG when adopting the edge selection strategy with $\tau = 0$. In addition, LVQ [1] was excluded from the experiments, as it is a quantization-based framework that can be integrated with graph-based and hybrid methods, including our GTI. While combining LVQ with these methods might offer efficiency and storage benefits, LVQ has several limitations: (i) LVQ could potentially increase the update overhead of other methods, as its updates scale linearly with dataset size [1], making it incompatible with GTI, which supports logarithmic updates; (ii) LVQ accelerates calculations using AVX512, but this instruction set is not widely supported, including on the processor used in our experiments and the latest Intel(R) Xeon(R) E Processor series [20]; and (iii) LVQ's compression can reduce accuracy in graph construction and search, lowering query recall for approximation methods. Therefore, we do not apply LVQ to baselines in our experiments.

Table 2: Dataset statistics

Dataset	Cardinality	Dim.	Size (GB)	Type
Deep	1, 000, 000	256	0.96	Image
Msong	992, 272	420	1.56	Audio
Gist	1, 000, 000	960	3.58	Image
Color	5, 000, 000	282	5.27	Image
Turing	100, 000, 000	100	37.63	Text
Bigann	100, 000, 000	128	48.06	Image

Table 3: Evaluation parameters

Parameter	Value
Integer k	1, 5, **10**, 20, 50
Search radius r (×0.01%)	1, 2, 4, **8**, 16
Leaf node capacity N_c	**2**, 4, 6, 8, 10
Graph sparsity m	4, 8, **16**, 32, 64
Proportion of dataset (%)	**1**, 5, 10, **20**, 40, 50, 60, 80, **100**

Configuration. We implement GTI and the baseline methods in C++. To ensure fairness in comparison, we clear the caches between query workloads and disable certain optimizations employed by some baselines. These optimizations, designed to improve distance computations or hardware configurations, can be applied to both GTI and all the baselines. They include normalized distance for NSG, AVX-512 support for LSH-APG, and parallel computing optimization for ELPIS. All experiments are conducted on a Linux server with a 2.20GHZ Intel(R) Xeon(R) CPU and 256GB of memory.

Parameters and Performance Metrics. In this study, we evaluate the performance of our proposed index GTI and its competitors by examining the impact of several key parameters. Specifically, we vary the integer k for k-nearest neighbor search, the search radius r for range queries that is adjusted based on the number of qualified objects required, the leaf node capacity N_c that controls the number of objects in each leaf node, the graph sparsity m that controls the number of neighbors in the graph, and the proportion of the datasets used in the experiments. Table 3 provides an overview of these key parameters and their corresponding values, with default values highlighted in bold. To ensure a fair evaluation, we use 1 million objects as the default size for datasets larger than 1 million. Consequently, the default proportions are set as follows: *Color*, *Turing* and *Bigann* at 20%, 1% and 1%, respectively, while *Deep*, *Msong*

Table 4: Index construction cost of different methods

Method		*Deep*		*Msong*		*Gist*		*Color*		*Turing*		*Bigann*	
		Time (s)	*PM* (MB)	Time (s)	*PM* (MB)	Time (s)	*PM* (MB)	Time (s)	*PM* (MB)	Time (s)	*PM* (MB)	Time (s)	*PM* (MB)
Tree-based Methods	M-tree	10.32	29.64	15.79	29.94	24.33	30.02	11.35	29.13	8.07	30.87	7.07	29.24
	MVPT	67.08	3.82	102.97	3.79	201.58	3.82	33.16	3.82	26.55	3.82	17.49	3.82
Graph-based Methods	HNSW	285.62	375.12	285.95	376.29	1221.12	382.69	240.54	378.34	286.35	377.80	142.08	377.81
	NSG	453.86	1697.78	537.31	1607.13	1469.09	1611.39	408.99	1654.68	1087.48	1763.69	297.27	1657.62
Hybrid Methods	LSH-APG	201.85	434.56	195.04	430.36	298.15	408.65	231.32	434.56	170.82	441.41	129.76	439.96
	ELPIS	108.92	385.36	188.52	386.53	355.52	382.69	122.54	388.58	122.86	385.44	62.96	385.46
	GTI	71.25	162.67	106.35	162.77	218.94	170.56	70.68	169.91	57.38	166.87	35.76	158.14

and *Gist* are set at 100%. Baseline methods are tested with their default parameter settings. We assess the efficiency and effectiveness of GTI and its competitors by measuring several metrics, including index construction cost, update time, search time, and search recall (defined in Equation (1)). All graph-oriented components of graph-based and hybrid methods are built in parallel, and the update methods are also processed in parallel. Each search measurement is based on the average performance across 100 random queries.

6.2 Construction and Update Performance

Construction Performance. We first compare the construction costs of GTI with those of competitors across all datasets. The results are presented in Table 4, where *PM* denotes the peak memory usage during index construction (excluding the size of the dataset). Overall, tree-based methods (M-tree, MVPT) exhibit significantly lower construction costs compared to graph-based methods (HNSW and NSG) and hybrid methods (LSH-APG, ELPIS and GTI). This is because their efficient hierarchical partitioning approach is less complex than the computation and storage of neighborhood relationships required by graph-based and hybrid methods. As a result, it is more cost-effective in terms of construction while necessitating more complex search traversals. In addition, hybrid methods leverage auxiliary indexes such as hash or tree structures to accelerate the construction of internal graph indexes, resulting in a shorter construction time than graph-based methods.

Among graph-based and hybrid methods, GTI incurs the lowest construction overhead. Specifically, GTI has the shortest construction time, comparable to MVPT, approximately 4× faster than graph-based methods, and about 2× faster than hybrid methods on average. Notably, NSG takes the longest construction time due to the need to build two graphs: first, the *k*NNG, and then the subsequent NSG built using the *k*NNG.

In terms of peak memory usage during index construction, our proposed method GTI uses the least memory among graph-based and hybrid methods, with a footprint at least 2× smaller than that of graph-based and hybrid methods. This efficiency stems from GTI's approach of first using a tree for fast hierarchical partitioning and then constructing a graph index only for nodes located at level 1 of the tree, rather than on the entire dataset. This strategy effectively reduces both the time and space costs of index construction. Although ELPIS writes to disk during index construction, it requires loading the entire index and graph into memory during search operations, resulting in a higher peak memory usage compared to GTI. Additionally, MVPT maintains a consistent index size due to its balanced tree structure based on pivot points, which is unaffected by data type variations.

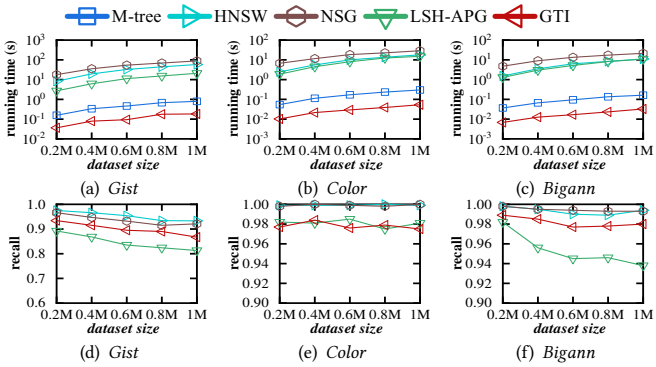


Figure 4: Update performance vs. *dataset size* (the recall of exact methods are omitted as the values are always 1)

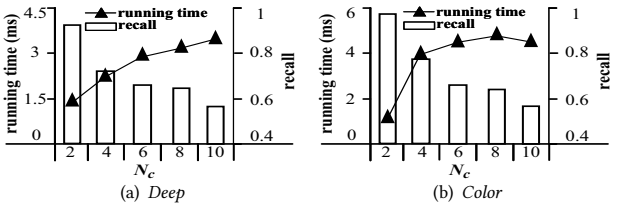
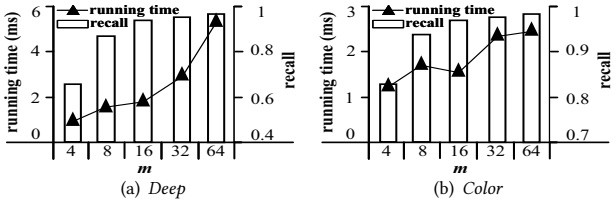


Figure 5: Effect of the leaf node capacity N_c on NNS

Update Performance. Next, we investigate the impact of update operations on various methods while varying the dataset size. Specifically, we perform update operations including inserting and removing objects, which correspond to 0.1% of the cardinality of the test datasets. Meanwhile, we intersperse query operations within the updates. This approach simulates real-world dynamic scenarios, such as frequent changes in product listings on e-commerce platforms like Alibaba's Taobao, where updates and user queries may occur concurrently.

For M-tree, we perform the exact *k*NN searches, while for other methods, we conduct approximate *k*-NNS. Although GTI supports various query types, we focus solely on approximate *k*-NNS in this evaluation. We compare average running time and search recall against baselines that support updates, where we adopt MNG's periodic graph reconstruction update strategy for HNSW, NSG, and LSH-APG. Notably, LSH-APG is an approximate method and only supports object deletion based on IDs rather than the objects themselves. Since real-world scenarios often require object-based deletions, we adapt MNG's update strategy to support object-based deletions in LSH-APG.



(a) *Deep* (b) *Color*

Figure 6: Effect of the graph sparsity m on NNS

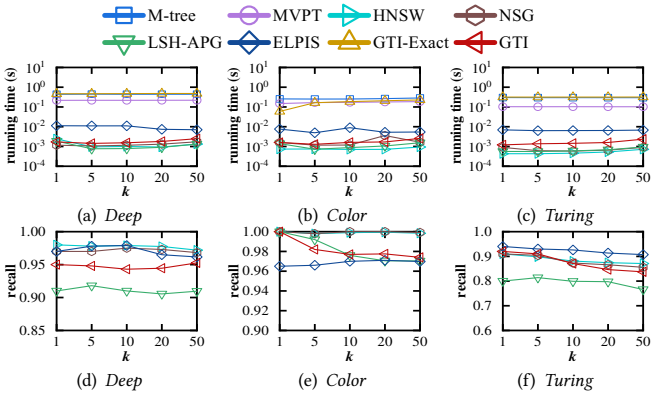
The results, shown in Fig. 4, highlight the comparative performance. Thanks to its tree-graph hybrid structure and efficient update strategy, GTI significantly outperforms other methods in update performance. GTI is approximately 10× faster than the next best tree-based method and two orders of magnitude faster than other graph-based and hybrid methods. Additionally, GTI's update time scales similarly with cardinality as M-tree, i.e. logarithmic complexity, which supports our theoretical analysis. However, due to slower query times of the M-tree, particularly for locating deletion points, its overall update operation time is longer than our proposed method. In terms of search recall post-updates, GTI maintains performance comparable to other SOTA approximate methods, with the recall of over 0.97 on *Color* and *Bigann*. This demonstrates the effectiveness and robustness of our proposed update strategy.

6.3 Search Performance

We proceed to investigate the search performances of GTI and its competitors by varying four key parameters: node capacity N_c, graph sparsity m, the number k for k-NNS, and the search radius r for exact range queries. Additionally, we analyze the trade-off between recall and search time through recall-time curves.

Impact of N_c and m. Fig. 5 illustrates the approximate k-NNS performance of GTI as leaf node capacity N_c varies. When N_c increases, search time tends to rise while recall decreases. This observation aligns with our complexity analysis in Section 5.3. An increase in N_c results in a greater number of objects within each tree leaf node, leading to an increase in the total number of neighbors that need to be visited during search. Although the graph size and height (i.e., the search path length $O(\log \frac{n}{N_c})$) decreases, this effect is minor due to the dataset size $n \gg N_c$. Consequently, the search time increases with larger N_c. Additionally, a larger N_c reduces the size of the graph and impairs global navigation performance and recall, even though the number of objects in the leaf nodes increases. Therefore, a smaller N_c generally results in better performance.

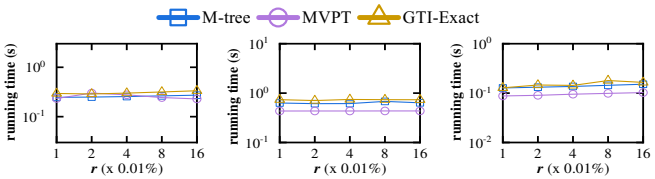
Fig. 6 illustrates the approximate k-NNS performance as graph sparsity m varies. The results reveal that as m increases, both the search time and recall rise, highlighting a trade-off between search efficiency and accuracy. This observation is in line with our complexity analysis presented in Section 5.3. Increased m means more neighbors ($O(2m + N_c)$) must be accessed during each search step, leading to longer search time and higher recall. Despite the increasing complexity, GTI maintains a high search recall (above 0.9) even with smaller m, due to effective neighbor relationships within tree nodes. For our experiments, we use $N_c = 2$ and $m = 16$ as their default values, which strike a good balance between search time and recall. For the capacity of internal nodes in the tree, we use commonly accepted values like 64 as the default setting.



(a) *Deep* (b) *Color* (c) *Turing*

(d) *Deep* (e) *Color* (f) *Turing*

Figure 7: Impact of integer k on NNS (the recall of exact methods are omitted as the values are always 1)

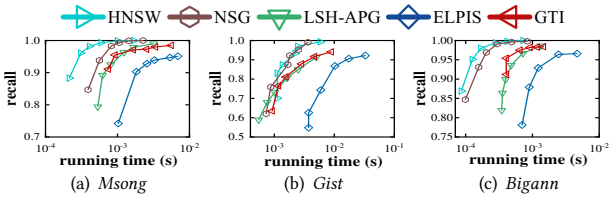


(a) *Msong* (b) *Gist* (c) *Bigann*

Figure 8: Impact of search radius r on exact range queries

Effects of r and k. Figs. 7 and 8 illustrate the performance of kNNS and range queries for different algorithms, while varying the integer k and the search radius r. Note, our GTI supports both approximate k-NNS (shown as GTI) and exact k-NNS and range query (shown as GTI-Exact). As shown in Fig. 7, approximate k-NNS methods (HNSW, NSG, LSH-APG, ELPIS, and GTI) are about two orders of magnitude faster than exact k-NNS methods (M-tree, MVPT, GTI-Exact). This speed advantage comes at the expense of some query accuracy, demonstrating that approximate methods significantly enhance efficiency by sacrificing accuracy. Meanwhile, this also indicates that the time cost of AkNNS remains negligible compared to exact search methods, while still maintaining high recall. Therefore, using AkNNS first to locate the object to be deleted during the deletion process is more efficient in most cases. Even in extreme cases where most target objects do not exist, the time cost incurred by the AkNNS remains negligible.

Besides, regardless of whether the k-NNS is exact or approximate, our method performs comparably to leading SOTA methods in terms of both time and accuracy, reflecting the efficacy of our search strategy. Notably, on datasets where approximate methods struggle to maintain high recall, such as *Turing*, our GTI also supports exact queries to meet high-precision requirements when necessary. Additionally, ELPIS exhibits relatively slower performance among the approximate methods due to its need to sequentially search subgraphs across multiple leaf nodes. GTI also supports exact range queries due to the benefits of its internal tree index. As shown in Fig. 8, GTI's performance in exact range queries is comparable with other efficient range query methods. Overall, GTI provides efficient graph-like approximate query performance, while also supporting exact queries. This dual capability makes GTI more versatile and efficient compared to other methods, enabling it to better accommodate diverse query requirements.



(a) *Msong* (b) *Gist* (c) *Bigann*

Figure 9: The recall vs. time for approximate k-NNS

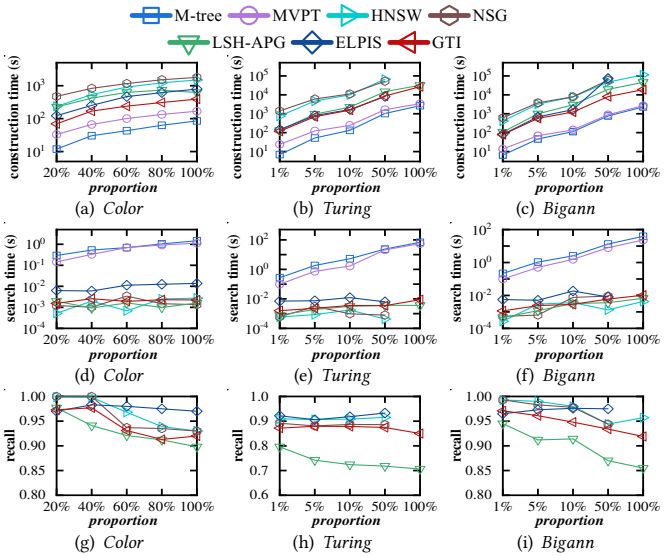
The recall vs. time. Increasing the number of objects examined in an ANNS method can enhance accuracy, but this improvement comes at the cost of reduced query efficiency. An algorithm is deemed to have superior query performance if it achieves a given target recall level within a shorter search time. Fig. 9 presents the approximate k-NNS recall-time of GTI compared to other approximate methods. We observe the following trends: (i) As search time increases, all algorithms show improved recall, adhering to the fundamental principle of ANNS methods where increased search time yields higher accuracy. Furthermore, the search time required to achieve a given recall grows almost exponentially as recall improves. (ii) While GTI does not outperform all approximate methods in terms of recall due to its reduced proximity graph size - resulting in significantly lower construction and update costs - its performance is still comparable. GTI maintains competitive graph-like approximate k-NNS performance, differing from the best methods by only a few milliseconds at similar recall levels, and achieving high query accuracy. Furthermore, GTI uniquely supports both exact k-NNS and range queries, a capability not provided by any other approximate methods.

6.4 Scalability Analysis

Finally, we present the construction and k-NNS performance with varying dataset proportions, as shown in Fig. 10. Note GTI performs only approximate k-NNS. For construction time, we focus on cases where the construction time is 36 hours or less; algorithms exceeding this limit are considered unable to build the index within a reasonable timeframe, such as ELPIS on *Bigann* and *Turing*, and HNSW on *Turing*.

We observe that the construction time increases with dataset cardinality. Tree-based methods (M-tree and MVPT) generally have shorter construction times compared to graph-based (HNSW, NSG) and hybrid (LSH-APG, ELPIS, and GTI) methods. The latter methods require establishing complex neighbor relationships to accelerate the search process significantly. Among graph-based and hybrid methods, GTI demonstrates notably better construction time performance, showcasing its efficiency. For instance, on the largest *Bigann* dataset with 100 million objects, GTI's construction time is only 5 hours, whereas other graph-based and hybrid methods take half a day to nearly two days.

Additionally, increasing dataset size poses memory issues on NSG on *Bigann* and *Turing*, while ELPIS and HNSW cannot complete index construction on *Bigann* and *Turing* within the limited timeframe due to prolonged disk writes from oversized leaf nodes and the complexity of multi-layer neighbor relations, respectively. In terms of search performance, we find that search time generally increases with dataset size, while the recall of graph-based and hybrid methods decreases. Exact tree-based methods, in contrast,



(a) *Color* (b) *Turing* (c) *Bigann*
(d) *Color* (e) *Turing* (f) *Bigann*
(g) *Color* (h) *Turing* (i) *Bigann*

Figure 10: Index and k-NNS performance vs. proportion of dataset (the recall of exact methods are omitted as the values are always 1)

are 2-4 orders of magnitude slower than approximate graph-based and hybrid methods, which trade off construction performance for improved search efficiency. Among the graph-based and hybrid methods, GTI shows comparable search time and recall. This indicates that GTI scales effectively with increasing data sizes.

7 CONCLUSIONS

In this paper, we introduce GTI, a new, agile graph-based tree index designed for NNS in high-dimensional spaces. GTI comprises a tree index spanning the entire dataset and a lightweight graph index constructed at the upper tree level. This approach significantly reduces the time and space costs associated with index construction. We also present efficient algorithms for data insertion and deletion, enabling real-time updates with logarithmic complexity on GTI. Furthermore, we propose approximate nearest neighbor search algorithms for GTI, achieving efficient graph-like query performance compared to SOTA methods. Additionally, GTI supports both exact NNS and range search functionalities. Extensive experiments demonstrate that GTI achieves efficient index construction with minimal time and space costs, and surpasses state-of-the-art methods in update efficiency. Extensive experiments on six real-world datasets demonstrate the efficient search performance and strong scalability of the proposed method. These results highlight the superior efficiency of GTI, making it a promising solution for various real-life applications. Moving forward, we plan to explore NNS with the learned index or consider the GPU architecture to further enhance efficiency.

ACKNOWLEDGMENTS

This work was supported in part by the NSFC under Grants No. (62025206, U23A20296, and 62102351), Zhejiang Province's "Lingyan" R&D Project under Grant No. (2024C01259), and the Key Lab of Big Data Intelligent Computing of Zhejiang Province. Congcong Ge is the corresponding author of the work.

REFERENCES

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

[2] Laurent Amsaleg and Hervé Jegou. 2010. Datasets for approximate nearest neighbor search. *Retrieved from http://corpus-texmex.irisa.fr/* (2010).

[3] Anon. 2011. Million Song Dataset Benchmarks. *Retrieved from http://www.ifs.tuwien.ac.at/mir/msd/* (2011).

[4] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. 2018. HD-Index: Pushing the Scalability-Accuracy Boundary for Approximate kNN Search in High-Dimensional Spaces. *Proc. VLDB Endow.* 11, 8 (2018), 906–919.

[5] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.* 87 (2020), 101374.

[6] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. Elpis: Graph-Based Similarity Search for Scalable Data Science. *Proc. VLDB Endow.* 16, 6 (2023), 1548–1559.

[7] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*. 322–331.

[8] Alina Beygelzimer, Sham M. Kakade, and John Langford. 2006. Cover trees for nearest neighbor. In *ICML*, Vol. 148. 97–104.

[9] Christian Böhm. 2000. A cost model for query processing in high dimensional data spaces. *ACM Trans. Database Syst.* 25, 2 (2000), 129–178.

[10] Paolo Bolettieri, Andrea Esuli, Fabrizio Falchi, Claudio Lucchese, Raffaele Perego, and Fausto Rabitti. 2009. Enabling content-based image retrieval in very large digital libraries. In *Proceedings of the SecondWorkshop on Very Large Digital Libraries*.

[11] Tolga Bozkaya and Z. Meral Özsoyoglu. 1997. Distance-Based Indexing for High-Dimensional Metric Spaces. In *SIGMOD*. 357–368.

[12] Tolga Bozkaya and Z. Meral Özsoyoglu. 1999. Indexing Large Metric Spaces for Similarity Search Queries. *ACM Trans. Database Syst.* 24, 3 (1999), 361–404.

[13] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn J. Keogh. 2014. Beyond one billion time series: indexing and mining very large time series collections with i SAX2+. *Knowl. Inf. Syst.* 39, 1 (2014), 123–151.

[14] Edgar Chávez, Gonzalo Navarro, Ricardo A. Baeza-Yates, and José L. Marroquín. 2001. Searching in metric spaces. *ACM Comput. Surv.* 33, 3 (2001), 273–321.

[15] Lu Chen, Yunjun Gao, Xuan Song, Zheng Li, Yifan Zhu, Xiaoye Miao, and Christian S. Jensen. 2023. Indexing Metric Spaces for Exact Similarity Search. *ACM Comput. Surv.* 55, 6 (2023), 128:1–128:39.

[16] 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 *NIPS*. 5199–5212.

[17] Paolo Ciaccia and Davide Martinenghi. 2017. Reconciling Skyline and Ranking Queries. *Proc. VLDB Endow.* 10, 11 (2017), 1454–1465.

[18] Paolo Ciaccia and Davide Martinenghi. 2020. Flexible Skylines: Dominance for Arbitrary Sets of Monotone Functions. *ACM Trans. Database Syst.* 45, 4 (2020), 18:1–18:45.

[19] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB*. 426–435.

[20] Intel Corporation. 2023. Intel(R) Xeon(R) E Processors. *Retrieved from https://www.intel.com/content/www/us/en/products/details/processors/xeon/e.html* (2023).

[21] Thomas M. Cover and Peter E. Hart. 1967. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theory* 13, 1 (1967), 21–27.

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

[23] Ishita Doshi, Dhritiman Das, Ashish Bhutani, Rajeev Kumar, Rushi Bhatt, and Niranjan Balasubramanian. 2021. LANNS: A Web-Scale Approximate Nearest Neighbor Lookup System. *Proc. VLDB Endow.* 15, 4 (2021), 850–858.

[24] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules Against Data Series Similarity Search. *Proc. VLDB Endow.* 15, 10 (2022), 2005–2018.

[25] Myron Flickner, Harpreet S. Sawhney, Jonathan Ashley, Qian Huang, Byron Dom, Monika Gorkani, Jim Hafner, Denis Lee, Dragutin Petkovic, David Steele, and Peter Yanker. 1995. Query by Image and Video Content: The QBIC System. *Computer* 28, 9 (1995), 23–32.

[26] Cong Fu and Deng Cai. 2016. EFANNA : An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph. *CoRR* abs/1609.07228 (2016).

[27] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.* 44, 8 (2022), 4139–4150.

[28] 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.

[29] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 4 (2014), 744–755.

[30] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*. 518–529.

[31] Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. 2020. iDEC: Indexable Distance Estimating Codes for Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 13, 9 (2020), 1483–1497.

[32] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.

[33] Michael E. Houle and Michael Nett. 2013. Rank Cover Trees for Nearest Neighbor Search. In *SISAP (Lecture Notes in Computer Science)*, Vol. 8199. 16–29.

[34] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.* 40, 4 (2008), 11:1–11:58.

[35] Masajiro Iwasaki. 2015. Neighborhood Graph and Tree for Indexing Highdimensional Data. *Yahoo Japan Corporation. Retrieved August* 22 (2015), 2020.

[36] 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.

[37] Zhongming Jin, Debing Zhang, Yao Hu, Shiding Lin, Deng Cai, and Xiaofei He. 2014. Fast and Accurate Hashing Via Iterative Nearest Neighbors Expansion. *IEEE Trans. Cybern.* 44, 11 (2014), 2167–2177.

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

[39] Atsutake Kosuge and Takashi Oshima. 2019. An Object-Pose Estimation Acceleration Technique for Picking Robot Applications by Using Graph-Reusing k-NN Search. In *First International Conference on Graph Computing*. 68–74.

[40] Michele Linardi and Themis Palpanas. 2018. Scalable, Variable-Length Similarity Search in Data Series: The ULISSE Approach. *Proc. VLDB Endow.* 11, 13 (2018), 2236–2248.

[41] Michele Linardi and Themis Palpanas. 2020. Scalable data series subsequence matching with ULISSE. *VLDB J.* 29, 6 (2020), 1449–1474.

[42] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2021. HVS: Hierarchical Graph Structure Based on Voronoi Diagrams for Solving Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 15, 2 (2021), 246–258.

[43] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. 2020. VHP: Approximate Nearest Neighbor Search via Virtual Hypersphere Partitioning. *Proc. VLDB Endow.* 13, 9 (2020), 1443–1455.

[44] 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.

[45] 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.

[46] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *SIGPLAN*. 270–285.

[47] Yitong Meng, Xinyan Dai, Xiao Yan, James Cheng, Weiwen Liu, Jun Guo, Benben Liao, and Guangyong Chen. 2020. PMD: An Optimal Transportation-Based User Distance for Recommender Systems. In *ECIR (Lecture Notes in Computer Science)*, Vol. 12036. 272–280.

[48] Kyriakos Mouratidis, Keming Li, and Bo Tang. 2021. Marrying Top-k with Skyline Queries: Relaxing the Preference Input while Producing Output of Controllable Size. In *SIGMOD*. 1317–1330.

[49] Kyriakos Mouratidis and Bo Tang. 2018. Exact Processing of Uncertain Top-k Queries in Multi-criteria Settings. *Proc. VLDB Endow.* 11, 8 (2018), 866–879.

[50] Javier Alvaro Vargas Muñoz, Marcos André Gonçalves, Zanoni Dias, and Ricardo da Silva Torres. 2019. Hierarchical Clustering-Based Graphs for Large Scale Approximate Nearest Neighbor Search. *Pattern Recognit.* 96 (2019), 106970.

[51] Beng Chin Ooi. 1987. Spatial kd-Tree: A Data Structure for Geographic Database. In *Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung, Darmstadt, 1.-3. April 1987, Proceedings (Informatik-Fachberichte)*, Vol. 136. 247–258.

[52] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *The VLDB Journal* 33 (2024), 1591–1615.

[53] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *SIGMOD*. 597–604.

[54] John Paparrizos, Ikraduya Edian, Chunwei Liu, Aaron J. Elmore, and Michael J. Franklin. 2022. Fast Adaptive Similarity Search through Variance-Aware Quantization. In *ICDE*. 2969–2983.

[55] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proc. ACM Manag. Data* 1, 1 (2023), 54:1–54:27.

[56] Badrul Munir Sarwar, George Karypis, Joseph A. Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In *WWW*. 285–295.

[57] Patrick Schäfer and Mikael Högqvist. 2012. SFA: a symbolic fourier approximation and index for similarity search in high dimensional datasets. In *EDBT*. 516–527.

[58] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. 2022. Results of the NeurIPS'21 challenge on billion-scale approximate nearest neighbor search. In *NIPS*. 177–189.

[59] 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.

[60] Runhui Wang and Dong Deng. 2020. DeltaPQ: Lossless Product Quantization Code Compression for High Dimensional Similarity Search. *Proc. VLDB Endow.* 13, 13 (2020), 3603–3616.

[61] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A Data-adaptive and Dynamic Segmentation Index for Whole Matching on Time Series. *Proc. VLDB Endow.* 6, 10 (2013), 793–804.

[62] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Graph- and Tree-based Indexes for High-dimensional Vector Similarity Search: Analyses, Comparisons, and Future Directions. *IEEE Data Eng. Bull.* 46, 3 (2023), 3–21.

[63] Zijie J Wang and Duen Horng Chau. 2024. MeMemo: On-device Retrieval Augmentation for Private and Personalized Text Generation. In *SIGIR*. 2765–2770.

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

[65] Yair Weiss, Antonio Torralba, and Robert Fergus. 2008. Spectral Hashing. In *NIPS*. 1753–1760.

[66] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the Symposium on Operating Systems Principles*. 545–561.

[67] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2020. Massively Distributed Time Series Indexing and Querying. *IEEE Trans. Knowl. Data Eng.* 32, 1 (2020), 108–120.

[68] Yuxiang Zeng, Yongxin Tong, and Lei Chen. 2023. LiteHST: A Tree Embedding based Method for Similarity Search. *Proc. ACM Manag. Data* 1, 1 (2023), 35:1–35:26.

[69] Huayi Zhang, Lei Cao, Yizhou Yan, Samuel Madden, and Elke A. Rundensteiner. 2020. Continuously Adaptive Similarity Search. In *SIGMOD*. 2601–2616.

[70] Yunan Zhang, Shige Liu, and Jianguo Wang. 2024. Are There Fundamental Limitations in Supporting Vector Data Management in Relational Databases? A Case Study of PostgreSQL. In *ICDE*. 3640–3653.

[71] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards Efficient Index Construction and Approximate Nearest Neighbor Search in High-Dimensional Spaces. *Proc. VLDB Endow.* 16, 8 (2023), 1979–1991.

[72] Chun Jiang Zhu, Tan Zhu, Haining Li, Jinbo Bi, and Minghu Song. 2019. Accelerating Large-Scale Molecular Similarity Search through Exploiting High Performance Computing. In *BIBM*. 330–333.

[73] Yifan Zhu, Ruiyao Ma, Baihua Zheng, Xiangyu Ke, Lu Chen, and Yunjun Gao. 2024. GTS: GPU-based Tree Index for Fast Similarity Search. *Proc. ACM Manag. Data* 2, 3 (2024), 142.

[74] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2016. ADS: the adaptive data series index. *VLDB J.* 25, 6 (2016), 843–866.