6-2024

GTS: GPU-based Tree Index for Fast Similarity Search

Yifan ZHU

Ruiyao MA

Baihua ZHENG
Singapore Management University, bhzheng@smu.edu.sg

Xiangyu KE

Lu CHEN

See next page for additional authors

Published in Proceedings of the ACM on Management of Data: SIGMOD 2024. DOI: 10.1145/3654945

GTS: GPU-based Tree Index for Fast Similarity Search

YIFAN ZHU, Zhejiang University, China
RUIYAO MA, Zhejiang University, China
BAIHUA ZHENG, Singapore Management University, Singapore
XIANGYU KE, Zhejiang University, China
LU CHEN, Zhejiang University, China
YUNJUN GAO, Zhejiang University, China

Similarity search, the task of identifying objects most similar to a given query object under a specific metric, has gathered significant attention due to its practical applications. However, the absence of coordinate information to accelerate similarity search and the high computational cost of measuring object similarity hinder the efficiency of existing CPU-based methods. Additionally, these methods struggle to meet the demand for high throughput data management. To address these challenges, we propose GTS, a GPU-based tree index designed for the parallel processing of similarity search in general metric spaces, where only the distance metric for measuring object similarity is known. The GTS index utilizes a pivot-based tree structure to efficiently prune objects and employs list tables to facilitate GPU computing. To efficiently manage concurrent similarity queries with limited GPU memory, we have developed a two-stage search method that combines batch processing and sequential strategies to optimize memory usage. The paper also introduces an effective update strategy for the proposed GPU-based index, encompassing streaming data updates and batch data updates. Additionally, we present a cost model to evaluate search performance. Extensive experiments on five real-life datasets demonstrate that GTS achieves efficiency gains of up to two orders of magnitude over existing CPU baselines and up to 20× efficiency improvements compared to state-of-the-art GPU-based methods.

CCS Concepts: • **Information systems → Main memory engines**.

Additional Key Words and Phrases: Metric Space, Concurrent Similarity Search, GPU-based Index

ACM Reference Format:
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* xx, xx, Article xx (August 2024), 27 pages. https://doi.org/XXXXXXX.

1 INTRODUCTION

Similarity search constitutes a fundamental challenge in the realms of information retrieval and data mining, involving the efficient identification of objects in a dataset that are most similar to a given query object [15]. This process carries profound significance across diverse domains, including multi-media retrieval, decision making, and visualization recommendation [12, 37, 42, 55]. Operating within metric spaces, similarity search quantifies object similarity via distance metrics, enabling efficient high-dimensional data management across a wide spectrum of data types [17].

Authors' addresses: Yifan Zhu, Zhejiang University, China, xtf_z@zju.edu.cn; Ruiyao Ma, Zhejiang University, China, ryma@zju.edu.cn; Baihua Zheng, Singapore Management University, Singapore, bhzheng@smu.edu.sg; Xiangyu Ke, Zhejiang University, China, xiangyu.ke@zju.edu.cn; Lu Chen, Zhejiang University, China, luchen@zju.edu.cn; Yunjun Gao, Zhejiang University, China, gaoyj@zju.edu.cn.

ACM 2836-6573/2024/8-ARTxx
https://doi.org/XXXXXXX.XXXXXXX

The recent research [29] highlights the pressing need for a general-purpose database capable of handling diverse cancer omics data, spanning molecular omics, molecular interaction, imaging, textual data, etc. While existing solutions like R-Trees [26] excel in handling spatial data, and graph-based methods [54] are tailored for high-dimensional vector data, the metric space offers a flexible framework for managing *dynamic* data of *various types* with *distinct measures*, such as word cosine distance for semantic measurement and edit distance for DNA strings [53]. Similarity search in metric spaces is designed with this versatility in mind, providing a general-purpose solution. In addition, metric indexes serve as pivotal solutions for enhancing the scalability of vector databases and offering cost-effective updates for social media and online commercial databases, addressing key issues identified in recent advancements [51, 54], such as the inevitably heavy construction cost and update complexities for graph-based methods.

Despite its importance, similarity search faces several challenges in achieving efficiency. The absence of coordinate information in metric spaces restricts the utilization of geometric structure knowledge of data points [43]. Additionally, the computation of distances between objects, like edit distance [59], incurs significant computational overhead. Existing CPU-based methods, such as MVP-tree [9, 10], Spacing-filling curve and Pivot-based B^+-tree [16], and EGNAT [44, 48], often fall short in meeting the demands of efficient similarity search. Furthermore, recent technological advancements have heightened the demand for managing and querying large-scale dynamic data, as seen in the exponential growth of DNA data generation facilitated by sequencing technologies [25]. These evolving requirements render CPU-based methods insufficient to meet the timely query needs of similarity search.

Recent advancements in similarity search have turned towards GPU-accelerated methods [61]. This shift is primarily attributed to the potential parallelism of independent and simple calculations, which has shown promising outcomes [6, 38]. These methods can be categorized into *distance table-based* and *tree index-based* methods. Distance table-based methods [6, 20, 23, 30, 34, 39] employ table structures to facilitate concurrent calculations across all objects, followed by subsequent filtering of unnecessary objects. While GPUs exhibit remarkable concurrent performance enhancements in similarity search compared to CPU-based methods, this approach necessitates a significant number of *unnecessary distance computations* between *all* objects and the query.

On the other hand, tree index-based methods [35, 36, 38] construct tree-structured indexes to enhance pruning efficiency and utilize GPU cores for concurrent searches *on various trees*. However, GPUs face certain limitations in terms of individual computational capacity and core diversity compared to their CPU counterparts [50]. As a result, current GPU-based methods struggle to achieve *concurrent traversal of each tree index*, thus failing to fully exploit the computation potential of GPUs. Consequently, their efficiency improvements over CPU-based methods remain limited.

In this context, some researchers have explored GPU-based approximate similarity search [58, 60, 61]. However, they fail to support exact query-answering and lack versatility when it comes to handling a wide range of distance metrics. Our goal is to develop an effective GPU-based index *optimized to harness the full computational potential of GPUs while avoiding unnecessary distance computations*. We aim to achieve efficient batch similarity search *in general metric spaces*, where the only available information is the distance metric function. Nevertheless, to achieve this goal, we must confront and overcome three key challenges.

Challenge I: Performing data pruning in parallel on a tree index poses significant difficulties in GPU-based parallelization. CPU-based approaches efficiently implement tree-based indexes through by-level pruning methods to enhance query efficiency by reducing unnecessary distance computations. However, parallelizing this process on GPUs is challenging due to inherent architectural limitations, as elaborated below: **(1)** *Top-Down Traversal*: Pruning in the tree structure

necessitates a top-down traversal from the root to the leaves, where calculations in succeeding levels depend on those performed at the current level. This interdependency poses a significant hurdle to effective parallelization across levels. **(2)** *Non-Contiguous Memory Allocation*: In tree structures, nodes at the same level are often stored in non-contiguous memory locations. Consequently, during GPU parallel computation, a linear traversal is necessary to allocate different nodes for concurrent processing. Unfortunately, this allocation process cannot be parallelized efficiently. **(3)** *Limited GPU Core Capabilities*: GPU cores are typically optimized for parallelizing simple and similar tasks. Conversely, the tree structure may consist of multiple data objects that demand distinct operations. This disparity makes it challenging to efficiently parallelize such diverse tasks on GPUs [35].

To address these challenges, this paper proposes a novel strategy aimed at *reorganizing tree structures within tables* to optimize GPU resources during the pruning process. Initially, a tree-based index structure is built using iteratively chosen pivots, and data partitioning methods are employed to balance the tree and control its height, thereby reducing wastage of GPU resources resulting from the top-down search approach. Subsequently, table-based strategies are utilized to store the indexed tree, allowing consecutive storage of tree nodes at the same level. Such organization enables the parallel allocation of simple GPU tasks for non-continuous nodes at the same level, effectively optimizing GPU computing resources and avoiding GPU thread waiting issues caused by serial task allocation.

Additionally, we have designed a *cost model* to enhance the parallel computation efficiency of nodes at different levels. The performance of similarity search is closely tied to both object pruning capability and parallel computing efficiency, directly impacted by *the node capacity*. A higher node capacity implies a lower tree height, limiting pruning capability but enabling more GPU cores for parallel computation of child nodes. Thus, by utilizing the proposed cost model to optimize node capacity, we achieve a well-balanced trade-off between pruning capability and parallel computing efficiency, significantly improving search performance. This improvement is also validated by experimental results.

Challenge II: Enhancing memory management for batch similarity queries. Batch similarity queries often involve independent query objects, demanding a considerable amount of memory to store intermediate results. However, surpassing the GPU's memory limit can trigger the memory deadlock issue, which obstructs the release of occupied memory for further computation. To address this, we propose a *two-stage hierarchical query strategy*. In the first stage, batches of concurrent queries are processed in upper-level tree nodes within the index. When the storage capacity of intermediate results approaches a predefined limit, we introduce a node-priority concurrent query strategy, granting access to lower-level tree nodes. This strategy involves grouping the intermediate results of each query and subsequently processing each group sequentially to obtain real results, with the queries in each group computed in parallel. By adopting this approach, we effectively circumvent memory deadlocks and enable high-concurrency similarity queries within our proposed search method.

Challenge III: Supporting efficient data updates. Existing CPU-based approaches rely on pre-computed distances between data objects and pivot or cluster centers for data pruning [17]. However, dynamic updates to the dataset can impact the index structure, potentially reducing query performance. For instance, a large number of object insertions may alter the overall object distribution, rendering the existing index unsuitable for capturing the characteristics of the updated dataset. Moreover, GPU cores face constraints in supporting dynamic data space allocation [56], posing challenges in allocating space for newly created tree nodes and modifying existing tree structures in GPU-based tree indexes. To address these challenges, we take into account the unique characteristics of GPUs, which offer powerful computing capabilities but lack dynamic

space allocation abilities. Our proposed solution leverages a *compact, high-speed cache table* to implement peak-valley update operations. For low-frequency object updates, we directly update the data in this cache table, avoiding the need to modify the storage structure of the GPU-based tree index. When the cache table exceeds a given size constraint, we perform *batch updates* to transfer the data into the tree index. In scenarios involving extensive batch data updates, we adopt an *index reconstruction strategy*, capitalizing on the GPU's superior parallel computing performance, which offers a time complexity of $O(\log^3 n)$. As a result, for substantial data updates, we opt to rebuild the tree index entirely. This approach not only enables efficient batch data updates but also guarantees that data updates do not compromise the query efficiency.

In summary, our key contributions are as follows:

- *GPU-based indexing.* We introduce GTS, a GPU-based Tree Index for Similarity Search, which efficiently handles metric range queries and metric k-nearest neighbor queries.
- *Tree-structured index with table-based storage.* We devise a novel pivot-based tree index with a table-based index structure, enabling parallelized calculations of non-continuous tree nodes at the same level for the first time.
- *Concurrent similarity search.* We develop concurrent search methods for our proposed GPU-based-tree index to prevent memory deadlock. Additionally, we introduce a cost model that balances concurrency and pruning capability.
- *Dynamic updates.* We propose effective update strategies tailored for dynamic scenarios, including both streaming data updates and batch data updates.
- *Extensive experiments.* We conduct comprehensive experimental evaluations on five real datasets, demonstrating the remarkable efficiency gains achieved by GTS. It outperforms existing CPU baselines by up to two orders of magnitude and even surpasses GPU-based general methods by up to 20× in terms of efficiency.

The rest of this paper is organized as follows. We review the previous works in Section 2, and present the problem statement in Section 3. The GPU-based tree index is introduced in Section 4, and the similarity search process with the cost model is detailed in Section 5. Comprehensive experimental studies and our findings are reported in Section 6. Finally, Section 7 concludes the paper and offers directions for future research.

2 RELATED WORK

CPU-based Similarity Search. CPU-based similarity search in metric spaces has received significant attention. Existing approaches can be categorized into *table-based* methods, *tree-based* methods, and *hybrid* methods that combine the previous two groups of approaches. Table-based methods pre-compute the *full* set of element-wise distances between pivots (reference points) and objects, manage them in a large table for quick access, and utilize triangle inequality to narrow down search spaces. Prominent examples include List of Clusters [13, 14], LAESA [45], and EPT [49]. On the other hand, tree-based methods, including BST [32], M-tree [19], MVP-tree [9, 10], spacing-filling curve, and pivot-based B$^+$-tree [16], employ hierarchical tree structures to manage objects in a top-down manner, which rapidly identifies candidate items that meet the search criteria, thereby reducing the search space and improving query efficiency. Hybrid methods aim to leverage the advantages of both table-based and tree-based approaches. For instance, CPT [46] links the distance table with M-tree leaves, while GNAT [11] and EGNAT [44, 48] store the distance table of the minimum bounding box in tree nodes. These methods combine pre-computed distance information with hierarchical data structures, resulting in high single-point query performance, making them suitable for a wider range of similarity search tasks.

Despite the progress made in CPU-based similarity search methods, they face limitations [50]. A primary challenge is the *limited parallelism* offered by CPUs, which are predominantly designed for sequential processing. Furthermore, the *computational intensity* of search operations can strain CPU performance, making real-time query processing for large-scale datasets a challenging task. To address the increasing demand for higher search performance, recent studies have turned to GPU-accelerated approaches. Capitalizing on the inherent advantages of GPUs, such as massive parallelism and superior computational power, GPU-based methods have the potential to overcome the shortcomings of CPU-based techniques. They unlock more efficient concurrent similarity search and significantly improve query throughput.

GPU-accelerated Table-based Methods. By utilizing table structures, we can efficiently perform similarity calculations between database objects and the query concurrently, enabling the parallel filtering of objects that meet the query conditions. Notable examples like Pivot-based Heaps [6], ES4D [34], and COSS [20] employ pivots, lookup arrays, and bin arrays to efficiently compute distances and enhance search termination or support general similarity search. Additionally, G-Grid [39] leverages GPU-searchable grid tables for exact k-NN queries in road networks, while GPU-based Batch search [30] optimizes brute-force, approximate, and compressed-domain search to facilitate billion-scale similarity search. The recent Dr. Top-k algorithm [23] introduces delegate-centric concepts and system optimizations to achieve fast multi-GPU top-k computation. However, existing table-based methods still suffer from the need to calculate similarity distances for *all* database objects and the query, resulting in numerous unnecessary computations that hinder search efficiency.

GPU-accelerated Tree-based Methods. To address the issue of unnecessary distance computations, researchers have proposed GPU-based tree indexes. For instance, G-tree [35] combines the efficiency of R-tree in low-dimensional space with a best-first strategy for concurrent tree-based similarity search on GPUs. G-PICS [38] aims at parallelized processing by constructing various spatial trees with different partitioning methods. LBPG-tree [36] optimizes the usage of multiple GPUs by compacting and sorting candidate tree nodes and optimizing search schedules for concurrent queries. Aiming at optimizing computing resource utilization, some methods [41, 52, 57] leverage a level-based concurrent construction approach to build the index. While these approaches represent significant advancements, challenges persist in concurrent similarity search. Existing GPU-based tree indexes often employ similarity search by (1) applying fixed-size thread blocks and (2) sequentially processing each tree node. This approach may lead to certain blocks having an excessive number of answers to verify, causing size overflow and potential memory deadlock issues. Conversely, other blocks may remain idle due to the absence of leaf nodes to explore, leading to a significant workload imbalance.

In contrast to common CPU-based practices that link tree elements with pre-computed distance table cells, our novel GPU-based tree index, GTS, is efficiently *maintained within a table*. This design allows us to concurrently compute non-continuous tree nodes at the same level via sort and coding strategies, and ensures uniform and straightforward computing tasks across all CPU cores, achieving higher efficiency. Additionally, we introduce two-stage search methods that enhance *memory utilization* and a cost model for evaluating *node capacity*, resulting in substantial improvements in concurrency and throughput. Furthermore, we have developed effective streaming data updates and batch update strategies to efficiently manage dynamic scenarios. These collective improvements position GTS as a powerful and efficient solution for concurrent similarity search tasks on GPU-based architectures.

Table 1. Symbols and description

Notation	Description
q, o, p	A query, an object, and a pivot in a metric space
O	An object set in a metric space
n	The number of objects in set O
$d(\cdot,\cdot)$	A distance metric
N	A tree node
N_list, T_list	The node list and the table list
$MRQ(\cdot,\cdot)$	A metric range query
$MkNNQ(\cdot,\cdot)$	A metric k nearest neighbour query

3 PROBLEM FORMULATION

We formally introduce the metric space and the similarity search below. For clarity, Table 1 summarizes frequently used notations.

A metric space is represented as a tuple (M, d), where M denotes the domain of objects, and d serves as the distance metric used to quantify the similarity between any pair of objects (o_1, o_2) within this space. The distance metric d must adhere to the fundamental principles, including *symmetry*, *non-negative*, *identity*, and *triangle inequality*, i.e., $d(o_1, o_2) \leq d(o_1, o_3) + d(o_3, o_2)$. Distance metrics, such as Manhattan distance, Euclidean distance, and word edit distance [59], satisfy the aforementioned conditions and are commonly employed in metric spaces. An essential advantage of the metric space is its ability to accommodate various data models since it does not impose any requirements on the data type [17].

Building upon these principles, we can formally define two types of metric similarity searches:

Definition 3.1. **(Metric Range Query.)** Given an object set O, a query object q, and a search radius r in a metric space, a metric range query (MRQ) finds all objects in O that are within a distance r from q. Formally, $MRQ(q, r) = \{o |\ o \in O \wedge d(q, o) \leq r\}$.

Definition 3.2. **(Metric k Nearest Neighbor Query.)** Given an object set O, a query object q, and an integer k in a metric space, a metric k nearest neighbor query (MkNNQ) finds a set S of k objects in O that are most similar to q. Formally, $MkNNQ(q, k) = \{S |\ S \subseteq O \wedge |S| = k \wedge \forall s \in S, o \in O - S, d(q, s) \leq d(q, o)\}$.

Consider the metric space for a string dataset shown in Fig. 1, with object set $O = \{o_1, o_2, ..., o_9\}$ and word edit distance $d(\cdot,\cdot)$ measuring similarity, a metric range query $MRQ(o_1, 2)$ aims to find objects that can be transformed to o_1 within 2 edit operations (insertion, deletion, or replacement). Consequently, $MRQ(o_1, 2)$ returns the set $\{o_1, o_2, o_3\}$. A metric 3 ($k = 3$) nearest neighbor query $MkNNQ(o_1, 3)$ finds 3 objects from O that can be modified with the least operations to become most similar to the query object o_1. Thus, $MkNNQ(o_1, 3) = \{o_1, o_2, o_3\}$. Note that if the distance from the query to its k-th nearest neighbor is known in advance, an MkNNQ can be efficiently answered by performing an MRQ. For instance, when finding $MkNNQ(o_1, 3)$, we can obtain the answer by performing $MRQ(o_1, 2)$, assuming that the search distance $r = 2$ from o_1 to its 3^{rd} nearest neighbor has been provided in advance.

Indexing a metric space that lacks coordinate information can be challenging, as traditional mathematical tools used in vector spaces cannot be directly applied. To tackle this, a common approach is to select a number of reference points, known as *pivots*, to map the data objects into a low-dimensional space [43, 62]. Each chosen pivot serves as an individual dimension, representing the distances from the pivot to all the data objects. This transformation allows the metric space to be represented in a structured form, similar to a vector space, enabling further analysis.

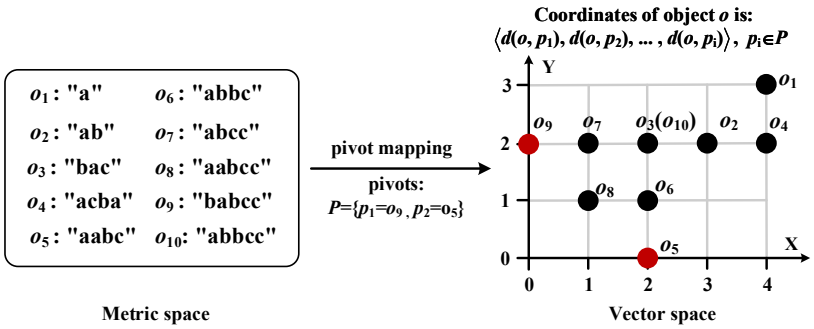


Fig. 1. Metric space and pivot mapping

When employing a pivot set, such as $P = \{p_1 = o_9, p_2 = o_5\}$, to map a metric space into a vector space, each object o is represented as a coordinate $\langle d(o, p_1), d(o, p_2)\rangle$, as depicted in Fig. 1. Thus, the distance between the objects o_1 and o_2 can be approximated by $|d(o_1 - p_i) - d(o_2 - p_i)|$, using the pivot p_i in the coordinate (dimension) D_i of the mapped vector space, which is smaller than the real distance $d(o_1, o_2)$ according to the triangle inequality. Therefore, when answering a similarity search, an efficient solution is to first search the mapped vector space with indexes, and then verify each found object to return the real answers. For instance, when answering $MRQ(o_1, 1)$, it is more efficient to search the vector space along each coordinate to find objects that are within distance 1 to o_1 along any dimension to find the candidate objects $\{o_1, o_2, o_4\}$ for $MRQ(o_1, 1)$ in the metric space. By verifying each candidate, $\{o_1, o_2\}$ are returned as the real answers. This pivot-based indexing technique effectively reduces the search space and speeds up the search process, making it a valuable solution for efficiently handling metric spaces without explicit coordinate information.

4 INDEX

In this section, we present an overview of our GPU-based tree index designed for similarity search, along with the structure of our proposed index. Subsequently, we delve into the construction and update strategies employed to support dynamic scenarios effectively. To solidify the understanding of our approach, we also provide a theoretical analysis that covers space consumption and time complexity of the proposed method.

4.1 Overview

Unlike CPUs that possess a few powerful cores optimized for handling complex tasks efficiently, GPUs offer abundant computing resources, making them ideal for *parallel* processing of numerous *simple computational tasks* independently. Therefore, linear continuous structures such as tables are well-suited for the parallel nature of GPUs, as each GPU core can process individual values in the table simultaneously. On the other hand, tree indexes, with hierarchical structures for pruning, can effectively narrow down the search space and accelerate the search process.

Motivated by these factors, we design a GPU-based tree index called GTS, which fully *harnesses the pruning properties of tree structures* and the *parallelism capabilities of GPUs* to achieve efficient similarity search in metric spaces. The framework of GTS is depicted in Fig. 2. Firstly, we construct a GPU-based tree index in a hierarchical manner, where upper-level nodes split the objects into lower-level nodes using pivot mapping and object partitioning. To accommodate dynamic scenarios, we design stream data updates and batch updates, catering to different update requirements. Finally, we utilize the GPU-based tree index to concurrently process multiple queries in a batch,

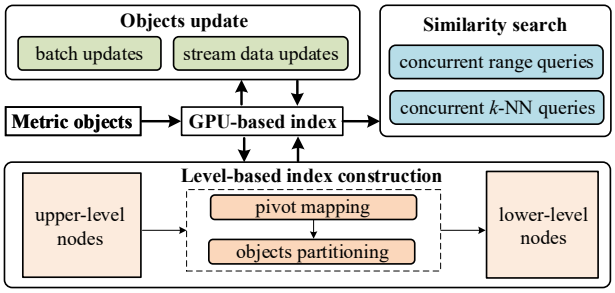


Fig. 2. The overall framework of our work GTS

significantly accelerating general similarity search in metric spaces, including range queries and k-nearest neighbor (k-NN) queries in batches. These techniques effectively exploit the pruning characteristics of the tree index and the parallelism of GPUs to enhance the overall search performance.

4.2 Index Structure

Our proposed GPU-based tree index GTS consists of two key components: **the tree index** and **the table list**, as illustrated in Fig. 3. The balanced tree index efficiently partitions the data objects after pivot mapping, facilitating hierarchical data management and achieving excellent pruning efficiency. To ensure efficient parallel computations on GPUs, we employ a node list structure to manage the tree nodes and store the distances from the pivots to the objects within each node using the table list, which is stored sequentially and contiguously. For a clearer understanding, Fig. 3 provides an illustrative example of our indexing structure applied to the metric space featured in Fig. 1. In the following, we will provide a comprehensive explanation of the detailed structure.

The Tree Index. Similar to the MVPT [9, 10], which is renowned as the most efficient CPU-based in-memory metric index [17], the tree nodes of our proposed GTS also maintain crucial information such as the selected pivot, the minimum distance from the enclosed objects to the pivot (denoted as min_dis), and the size of the data objects under their management. To ensure efficient utilization of GPU resources during similarity calculations, we store all the tree nodes in a node table list. Specifically, these nodes are linearly linked, and their ID numbering follows the principles of a full multi-way tree. By designating the ID of the root node as 1, the ID of the j-th child node w.r.t. node N_i is determined as follows [38]:

$$ID = (i-1) * N_c + j + 1, \tag{1}$$

where N_c denotes the node capacity, representing the maximum number of child nodes each node has. For example, in Fig. 3, each node N_i ($1 \leq i < 7$) is linked to the next node N_{i+1}, and the j-th child node of N_i is $N_{(i-1)*N_c+j+1}$, e.g., the second child node of N_3 is N_7. This design maximizes the GPU resources by enabling the parallelized computation for non-continuous tree nodes at the same level, and will be detailed in Section 4.3. Additionally, to maximize space utilization, we precompute the theoretical maximum height $max\_h = \lceil log_{N_c}(|O|+1) \rceil - 1$ of the index, where $|O|$ is the total number of the objects. We then set the height bound h to be $(max\_h - 1)$, making some nodes at the last level overfull and not eligible for splitting. For instance, the size (= 3) of node N_7 exceeds the node capacity (= 2).

Furthermore, to store the object partitioning information, which includes the objects and their distances to the pivots in each tree node, we maintain the start position of each node in the table list (denoted as pos). Let's consider node N_3 in Fig. 3 as an example, where the pivot is o_9, the start

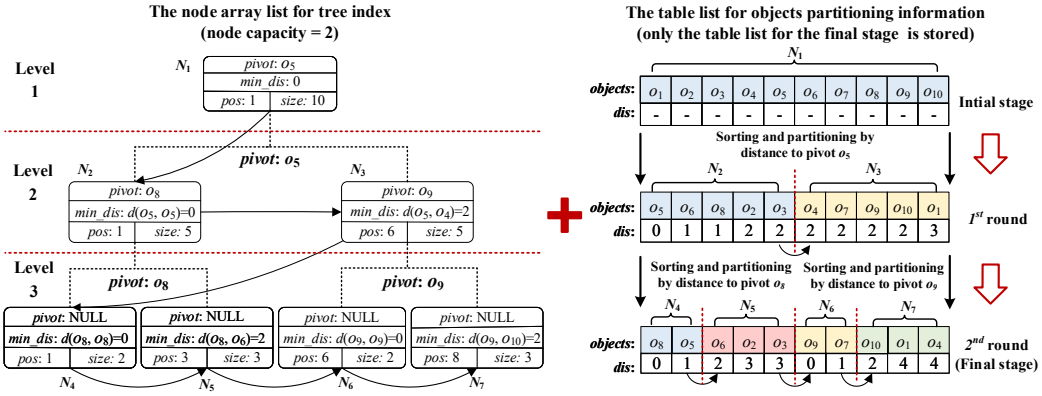


Fig. 3. The structure of our proposed GPU-based tree index, GTS

position of node N_3 in the table list is 6, and its *size* is 5. Thus, according to the second level of the table list, we can deduce that N_3 maintains the objects o_4, o_7, o_9, o_{10}, and o_1. It is important to note that these objects are mapped by the pivot o_5 in the parent node of N_3 and the minimum distance min_dis from the pivot o_5 to all data objects in node N_3 is 2.

The Table List. We use a table list to maintain the object partitioning information for each level of the tree. Considering the example shown in Fig. 3, at the first level, node N_1 contains all ten objects since no data objects are partitioned yet. Next, using pivot o_5 to map objects in N_1, N_1 is split into N_2 and N_3. N_2 maintains the first five objects with the smallest distances to the pivot, while N_3 maintains the rest. Therefore, $\{o_5, o_6, o_8, o_2, o_3\}$ are partitioned into N_2 in ascending order based on their distances to the pivot o_5, while $\{o_4, o_7, o_9, o_{10}, o_1\}$ are allocated to N_3 in the same manner. Subsequently, the objects in N_2 and N_3 are partitioned again using pivots o_8 and o_9 respectively, resulting in the creation of nodes N_4 to N_7 in the third level. The table list records the partitioning information for all the tree nodes. Notably, since lower-level tree nodes are partitioned from the upper-level nodes, we can obtain the objects' partitioning information of upper-level nodes by combining the table list information of the lower-level objects. For example, the objects partitioning information of node N_3 can be derived by combining the list information of its child nodes N_6 and N_7. Therefore, to optimize memory consumption, we store the table list solely for the last level (i.e., leaf level) of the tree.

4.3 Index Construction

The index construction process in GPU-based solutions involves data transfer from main memory to GPU and the data management via the index maintained within the GPU memory. Existing GPU-based tree construction strategies often assign each GPU core to construct individual tree nodes [33, 47]. However, this approach encounters challenges when dealing with tree nodes that manage a large number of objects. Since each GPU core can only handle relatively simple computational tasks, splitting such tree nodes becomes problematic. For example, attempting to split the root node into second-layer tree nodes would require storing all the objects in a single GPU core, which is practically infeasible due to memory limitations. To overcome this limitation, some methods [28, 38] build multiple trees, with each tree managing only a small number of objects. While this approach increases parallelism during index construction, it introduces significant space consumption when handling high query throughput. Querying multiple trees for each query object consumes considerable space, thereby restricting the number of queries that can be answered in parallel and causing workload imbalances across the trees. To address these issues, we propose

Algorithm 1: Index Construction

Input: an object set O, a distance metric d, and the node capacity N_c that controls the tree height
Output: the node list N_list and the table list T_list

1: $h \leftarrow \lceil log_{N_c}(|O|+1) \rceil - 1$
2: $N\_list[1] \leftarrow$ the initialized root node with $size = |O|, pos = 0$
3: $layer \leftarrow 1, pos \leftarrow 1, N_s \leftarrow 1$ // initialize the current layer $layer$, the starting position pos in the node list, and the number N_s of nodes in the current layer
4: **foreach** $o_i \in O$ ***in parallel*** **do**
5: $\quad T\_list[i].object \leftarrow o_i$ // initialize the table list
6: **while** $layer \leq h$ **do**
7: $\quad N\_list, T\_list \leftarrow$**Mapping**$(d, pos, N_s, N\_list, T\_list)$
8: $\quad N\_list, T\_list \leftarrow$**Partitioning**$(N_c, pos, N_s, N\_list, T\_list)$
9: $\quad pos \leftarrow (pos-1) * N_c + 2$
10: $\quad N_s \leftarrow N_c * N_s, layer \leftarrow layer + 1$
11: **return** N_list, T_list

a top-down construction algorithm based on distance encoding to achieve high-level parallelism. Instead of assigning each GPU core to individual nodes [33, 47], we construct nodes at the same level simultaneously using all the GPU cores to facilitate parallelism. This is because the distance computation of each node is independent and all nodes at each level are continuously stored. Specifically, we partition the objects following a top-down manner. At each level i, we simultaneously map the objects with pivots to all the tree nodes at the current level i. Next, we employ a *global sort algorithm with distance encoding* to concurrently partition the objects at the level $(i+1)$. Since all objects are sorted and partitioned together using the existing table stored in global memory, we can fully utilize the GPU cores without being limited by space constraints. Finally, we construct a single tree that maintains all the objects, breaking the construction bottleneck encountered by multi-tree methods [28, 38]. In the following, we detail the construction of GTS.

Index Construction. Algorithm 1 outlines the construction process of our index. Firstly, it calculates the tree height (denoted as h) based on the dataset size $|O|$ and the node capacity N_c (line 1). Next, the algorithm initializes the root node, the current layer number, the starting position of the root node in the node list, the number of nodes in the current layer, and the table list (lines 2–5). Subsequently, a while loop is executed to construct the index until the current level exceeds the height boundary (lines 6–10). In each iteration, the node list N_list of the index and the table list T_list are updated through pivot mapping (line 7) and object partitioning (line 8) operations. Finally, the algorithm completes the construction of GTS and returns N_list and T_list (line 11).

Pivot Mapping. We present the GPU-based pivot mapping process in Algorithm 2. In this algorithm, we employ the FFT [27] as the pivot selection algorithm, which selects the object farthest from existing pivots as the new pivot and can be easily parallelized by concurrently computing the distances, thereby improving efficiency. In terms of the initial pivot, several existing pivot selection algorithms, such as FFT, BPS [45], and HF [31], opt for a random object as the first pivot, as they have validated that the initial pivot has minimal impact on search performance. Moreover, the recent study [62] concludes that no algorithm exists to select the optimal pivots on any dataset. This makes it impossible to choose an optimal initial pivot with a generalized strategy. Therefore, we also choose the first pivot randomly. The pivot mapping algorithm begins by locating the nodes in the current layer from the node list N_list in parallel (lines 1–2). Next, it retrieves the objects

Algorithm 2: Mapping

Input: a distance metric d, the starting position pos, the node size N_s, the node list N_list, and the table list T_list
Output: the updated N_list and T_list

1: **foreach** $i \in [0, N_s)$ ***in parallel across blocks*** **do**
2: $N \leftarrow N\_list[pos + i], T\_tmp \leftarrow \{\varnothing\}$
3: **foreach** $j \in [0, N.size)$ ***in parallel within block*** **do**
4: $T\_tmp[j].object \leftarrow T\_list[j + N.pos].object$
5: $N.pivot \leftarrow$ the object chosen by FFT [27] from T_tmp
6: **foreach** $j \in [0, N.size)$ ***in parallel within block*** **do**
7: $T\_list[j + N.pos].dis \leftarrow d(T\_tmp[j].object, N.pivot)$
8: **return** N_list, T_list

maintained by the current node N from the table list T_list in parallel (lines 3–4). Then, the pivot is chosen from objects maintained by the temp table T_tmp using the FFT algorithm, and the distances between the pivot and the objects maintained by N are computed simultaneously (lines 5–7). Additionally, to further improve efficiency, the pivots of each node are stored in the shared memory, which is much faster than GPU global memory [38]. Finally, the algorithm returns the updated node list N_list and table list T_list (line 8).

Objects Partitioning. After completing the pivot mapping process, we obtain the distances between data objects and the pivots in each node at the current level. Subsequently, we partition the objects to the newly constructed tree nodes based on these distances. Managing the objects in each node effectively requires sorting them based on their distances to the pivot. However, performing individual sorts using each GPU core for each node incurs significant time overhead. To address this challenge, we propose a global object partition strategy to efficiently leverage all GPU computing resources. Firstly, we encode the individual distances from each object to the corresponding pivot in parallel. Subsequently, we employ a global concurrent sorting algorithm to arrange all the encoded distances in ascending order. This arrangement groups the objects that should be partitioned to the same newly constructed tree node together. Finally, we decode the distances of each object and construct the tree nodes in the next layer in parallel. Notably, the objects in each tree node are uniformly divided based on the quantity and are then correctly indexed in the child nodes. Although identical objects with duplicate keys can be partitioned into different nodes, causing the tree nodes to overlap, this strategy ensures a balanced tree structure with efficient and accurate similarity search.

Algorithm 3 presents the detailed steps to partition the objects. Firstly, the algorithm normalizes the individual distances derived through pivot mapping in the current layer to the range [0,1) in parallel (lines 1–2). Next, it encodes the distances of each object, considering the information of the node to which the object belongs (lines 3–6). Specifically, given an object o that belongs to the node N_i, the distance dis from o to the pivot $N_i.pivot$, and the upper bound max of dis, the encoded distance dis' of o is determined as $dis' = \frac{dis}{max+1} + i$. By doing so, the integer part of dis' records the node belonging information of o, while the decimal part of dis' records the distance from o to the corresponding pivot. After applying the sorting algorithm offered by GPU on the encoded distances (line 7), the algorithm partitions the objects in each node concurrently (lines 8–19). This enables simultaneous computation of non-continuous tree nodes at the same level. For each node N in the current layer, the algorithm first decodes the distances of each object belonging to N in parallel (lines 10–11). Then, after initializing the newly constructed child nodes N' of N in the next

Algorithm 3: Partitioning

Input: the node capacity N_c, the starting position pos, the node size N_s, the node list N_list, and the table list T_list
Output: the updated N_list and T_list

1: get the maximum distance max in the T_list **in parallel**
2: normalize each distance dis in the T_list to $\frac{dis}{max+1}$
3: **foreach** $i \in [0, N_s)$ ***in parallel across blocks*** **do** // encoding
4: $N \leftarrow N\_list[pos + i]$
5: **foreach** $j \in [0, N.size)$ ***in parallel within block*** **do**
6: $T\_list[j + N.pos].dis \leftarrow i + T\_list[j + N.pos].dis$
7: sort all the distances in T_list using GPU
8: **foreach** $i \in [0, N_s)$ ***in parallel across blocks*** **do**
9: $N \leftarrow N\_list[pos + i]$
10: **foreach** $j \in [0, N.size)$ ***in parallel within block*** **do** // decoding
11: $T\_list[j + N.pos].dis \leftarrow (T\_list[j + N.pos].dis - i) * (max + 1)$
12: $avg\_size \leftarrow \lfloor \frac{N.size}{N_c} \rfloor$
13: **foreach** $j \in [0, N_c)$ ***in parallel within block*** **do**
14: $N' \leftarrow N\_list[(N.pos - 1) * N_c + j + 2]$
15: $N'.pos \leftarrow N.pos + j * N_c$
16: **if** $j < N_c - 1$ **then** $N'.size \leftarrow avg\_size$
17: **else** $N'.size \leftarrow N.size - avg\_size * (N_c - 1)$
18: $N'.min\_dis \leftarrow T\_list[N'.pos].dis$
19: **return** N_list, T_list

layer, the algorithm distributes the objects evenly into each new node N' (lines 12-18). Finally, the object partitioning completes and the algorithm returns the updated N_list and T_list (line 19).

4.4 Index Updating

To cater to dynamic scenarios, we have devised efficient data update strategies for our GTS index. Data updates can encompass insertions, deletions, and modifications, where each modification can be regarded as a deletion followed by an insertion. To handle different application scenarios, data updates can take two forms: 1) Real-time or incremental updates are suitable for continuously arriving data streams, such as real-time monitoring and real-time data analysis; and 2) Batch updates are designed to handle large volumes of data updates, typically encountered in scenarios involving database batch management. To effectively accommodate these distinct update scenarios, we introduce two separate index update strategies: *Stream Data Updates* and *Batch Updates*.

Stream Data Updates. When data arrives in a sequential streaming fashion, traditional data updates for tree index can lead to *structure changes*, potentially affecting search efficiency and incurring a logarithmic time cost to update corresponding tree nodes. Nonetheless, incrementally updating indexes can be time-consuming on GPU, whose architecture fails to support efficient individual operations [50]. Motivated by the LSM-Tree update strategy that reduces write amplification by buffering writes in memory and subsequently merges them using sequential operations [40], we propose *a lazy strategy with a cache list* to handle streaming data updates. This approach aims to harness the superior processing power of GPUs. Specifically, when new objects arrive, instead of directly updating the index, we store the newly inserted data in a cache list. For data deletions,

we first locate their positions based on the ID. If the data is present in the cache list, we remove it directly. Otherwise, i.e., it is stored in the index, we mark the corresponding position in the T_list for later deletion. When the size of the cache list exceeds a given limit, we efficiently reconstruct the entire index using the objects in both the index and the cache list, and then clear the cache. As depicted earlier, our construction strategy fully leverages the processing power of the GPU, enabling the index GTS to be efficiently reconstructed. For instance, the index for 10 million objects can be rebuilt within just 2 seconds, minimizing disruption to the user. Moreover, as the entire index is reconstructed, update operations on GTS have no adverse effects on search performance.

When answering similarity queries, we perform separate searches on both the index and the cache list, merging the results to retrieve final answers. Given that the cache list is much smaller than the dataset, we employ a brute-force table-based method for querying, utilizing GPU parallelism to compute the distance between each object in the cache list and the query and then validate all the results. Furthermore, our search strategy ensures accurate query outcomes by accounting for the presence of existing objects and newly inserted objects in the tree index and the cache list, while disregarding removed objects that are no longer accessible.

Batch Updates. Our index is stored in contiguous structures, which consist of a list-based structure and a distance table. However, updating operations on these structures can be costly, especially for list modifications. For instance, inserting elements in the middle of the list necessitates moving half of the elements, which is time-consuming. To address this issue, when significant bulk updates are encountered, we opt for direct index reconstruction. Leveraging our efficient parallel index construction method, we find that performing a direct batch reconstruction of the modified dataset is both feasible and effective, as demonstrated in Section 6.2.

4.5 Complexity Analyses

Space Consumption of Index. Our index consists of two components, i.e., the tree index N_list and the table list T_list. Let n denote the cardinality of the object set (i.e., $|O|$) and N_c denote the node capacity. For the tree index, it is balanced and essentially consists of tree nodes stored consecutively in N_list, so its space cost depends only on the number of tree nodes, which is $O(\frac{(N_c)^{h+1}-1}{N_c-1})$, i.e., $O(n)$, where h is the height of the tree and equals $\lceil log_{N_c}(|O|+1)\rceil - 1$. Conversely, T_list records the partitioned data objects and their respective distances to the pivots. As the partitioning is applied to the entire dataset in a disjoint manner, the space cost of T_list directly correlates to the dataset size, yielding $O(n)$. Consequently, the overall space complexity of our index is $O(n)$.

Time Complexity of Index Construction. Index construction at each layer primarily involves two key steps: pivot mapping and object partitioning. In the pivot mapping step at each level, we compute the distance between each data object and its corresponding pivot, incurring the time complexity of $O(\frac{n}{C})$, where C represents the GPU concurrent computing power. For objects partitioning, the dataset is firstly sorted in ascending order based on the distances to the pivots, incurring the cost of $O(\lceil \frac{n}{C} \rceil \log^2 n)$ [30]. Next, the data partitioning with the complexity of $O(\frac{n}{C})$ is performed. In summary, the total overhead, which involves the index construction steps for $\log n$ layers, has the time complexity of $O(\lceil \frac{n}{C} \rceil \log^3 n)$. Assuming the GPU computing power is on the same order of magnitude as the size of the dataset, the time complexity simplifies to $O(\log^3 n)$. For instance, recent GPU can be equipped with more than 16,000 cores and support $9{\times}10^{13}$ floating-point operations per second [4].

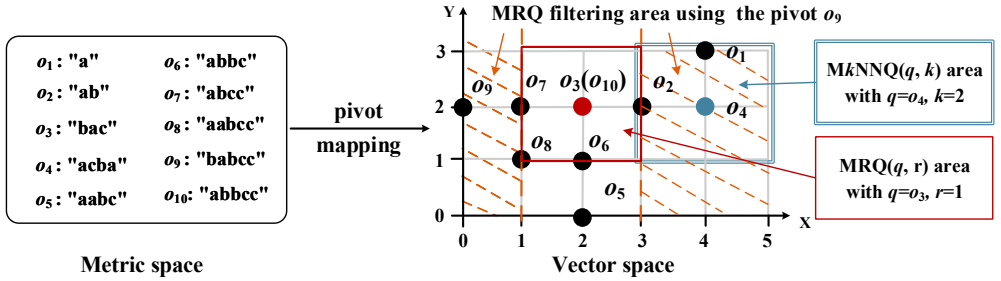


Fig. 4. Illustration of the filtering process

Time Complexity of Index Updating. Index updating involves two main scenarios: batch updates and stream data updates. For batch updates, we perform a complete reconstruction of the index, so the time complexity is consistent with that of index construction. For stream data updates, we consider data insertion, data deletion, and index reconstruction when the cache list reaches its capacity. For data insertion, we directly store the new data in the cache list, resulting in a time complexity of $O(1)$. For data deletion, we locate the data based on its ID and then either mark it for deletion in the original index structure's *T_list* or immediately remove it from the cache list, resulting in a time complexity of $O(1)$. When the cache list is full, we execute index reconstruction, with time complexity consistent with that of index construction.

5 SIMILARITY SEARCH

In this section, we propose efficient GPU-based algorithms for similarity search in metric spaces using GTS, including metric range query and metric k nearest neighbor query.

5.1 Metric Range Query in Batch

Given a metric object set O, a range query with radius r aims to find objects within O whose distances to the query object q are bounded by r. However, computing distances between the query object and unnecessary objects for answering range queries can be time-consuming. To accelerate metric range queries, we utilize the triangle inequality, following existing works [17], to filter and validate objects without unnecessary distance computations.

Lemma 5.1. *Given a pivot p, a query object q, and a search radius r in a metric space, an object o can be pruned if $|d(o,p) - d(q,p)| > r$.*

The proof of Lemma 5.1 directly derives from the triangle inequality and is omitted here. To illustrate this, let's consider the example shown in Fig. 4 for the metric space depicted in Fig. 1. In this example, the query is o_3, and the search radius is 1. Applying Lemma 5.1 with pivot o_9, we can filter all the objects o with $|d(o,o_9) - 2| > 1$ along X dimension, i.e., objects o_1, o_4, and o_9 that fall within the areas shaded by dashed lines in Fig. 4 can be pruned.

The query process in GPU-based solutions starts with loading queries from CPU to GPU, processing the queries in GPU, and finally returning the results to CPU. Similar to existing CPU-based metric indexes, GTS adopts a top-down approach to answer similarity queries using triangle inequality-based pruning techniques. In the case of range queries, GTS begins from the root node and systematically examines the tree nodes layer by layer, ensuring precise answers through the correctness of triangle inequality-based pruning, pruning only unqualified objects. For instance, given a query object q and a tree node N, let *min_dis* be the minimum distance from the pivot p of N to the objects maintained by N. If $d(q,p) + r < min\_dis$, N can be safely pruned by Lemma 5.1. Otherwise, GTS examines the subtrees of N in the next layer. However, a potential issue of exceeding the GPU's memory limit due to the space occupied by intermediate results from various

Algorithm 4: Metric Range Query

Input: a set of range queries Q, the GTS index
Output: the answers to the metric range queries

initialize Q_Res to $\{\varnothing\}$
foreach $q_i \in Q$ ***in parallel*** **do**
  $Q\_Res[i] \leftarrow (N\_list[1], q_i, r_i)$
$N\_set \leftarrow \{N\_list[1]\}$, $layer \leftarrow 1$
$N_c \leftarrow$ the node capacity of GTS
$h \leftarrow$ the tree height of GTS
Range_Q($Q, Q\_Res, N\_set, layer, N_c, h$)
Function `Range_Q`($Q, Q\_Res, N\_set, layer, N_c, h$)
$size\_GPU \leftarrow$ the available memory size of GPU
$size\_limit \leftarrow \frac{size\_GPU}{(h-layer+1)*N_c}$
$Q\_group \leftarrow Q$
if *the size of* Q_Res *>*$size\_limit$ **then**
  divide the Q_group according to Q_Res in parallel
foreach $Q' \in Q\_group$ **do**
  initialize Q'_Res to $\{\varnothing\}$
  foreach $E_i = \{N, q, r\} \in Q\_Res, j \in [0, N_c)$ ***in parallel*** **do**
    $N' \leftarrow N\_list[N.pos + j]$
    if N' *cannot be pruned for MRQ(q, r) by Lemma 5.1* **then**
      $Q'\_Res[i * N_c + j] \leftarrow \{N\_list[N.pos + j], q, r\}$
  $N'\_set \leftarrow$ the nodes in Q'_Res
  if $layer = h$ **then** output the answers to the queries of Q' by verifying the objects in the nodes of N_set'
  else Range_Q($Q', Q'\_Res, N'\_set, layer + 1, N_c, h$)

queries can lead to a memory deadlock, where memory cannot be cleared to make room for further computations. To overcome this challenge, we develop a hierarchical two-stage query strategy that effectively avoids memory deadlocks. Specifically, we perform an iterative search of the index in a top-down manner to answer multiple queries in batches, utilizing a table Q_Res to store intermediate results. Each element $E = \{N, q, r\} \in Q\_Res$ denotes that the node N needs to be searched for the range query MRQ(q, r). To maximize the utilization of computing resources, we manage the intermediate results of the query process by setting an intermediate result size limit $size\_limit$ at layer i to $\frac{size\_GPU}{h*N_c}$, where $size\_GPU$ is the GPU's available memory size, h is the tree height, and N_c is the node capacity. Hence, the size of the intermediate answer at level $i+1$ ($i+1 < h$) is bounded to $size\_limit * N_c = \frac{size\_GPU}{h}$. When the space consumption of Q_Res exceeds $size\_limit$, the queries are divided into various groups, and each group is processed linearly, ensuring the space consumption of the search process during the intermediate layers remains below the GPU memory limit. Note that the nodes in the bottom level h can directly answer each query without incurring any extra cost. Therefore, our proposed strategy effectively manages the memory size during the search of each level, preventing memory deadlocks and optimizing the utilization of computing resources.

Algorithm 4 outlines the process of the two-stage metric range query. This algorithm takes as input a set of range queries Q and the GTS index, and outputs the answer to each range query.

Firstly, we initialize all the elements in the intermediate result table Q_Res, and set N_set to the root node and the current search layer $layer$ to 1. Additionally, we obtain the node capacity N_c and the tree height h from the GTS index (lines 1–6). Next, the algorithm invokes $Range\_Q$ function to answer the range queries (line 8). It initially computes the size limit based on GPU's available memory size $size\_GPU$ and partitions the queries if needed (lines 9-10). Next, for each query group Q', the function first initializes a new intermediate result table Q'_Res for the next layer (line 15). It then concurrently finds the node N that cannot be pruned by the range query $MRQ(q, r)$ via Lemma 5.1, and updates the corresponding intermediate result Q'_Res (lines 16–20). Thereafter, the function collects all the nodes N'_set in Q'_Res (line 22). Finally, if the current layer is at the bottom level, the function uses the nodes in N'_set to answer each range query (line 22). Otherwise, it proceeds to search the next layer to answer the range queries (line 23).

5.2 Metric *k*NN Query in Batch

A metric k nearest neighbor query (MkNNQ) aims to find k objects in O that are most similar to q. Consider a metric k nearest neighbor query instance depicted in Fig. 4. The query object q is o_3, and k is 2. The answer to the query MkNNQ($q = o_3, k = 1$) is $\{o_3, o_{10}\}$. To accelerate the search process, we can apply the following lemma.

Lemma 5.2. *Given a pivot p, a query object q, and an integer k in a metric space, assuming that in the current search stage, the distance between q and its current k-th nearest neighbor is $d(q, k_{cur})$, then an object o can be pruned if $|d(o, p) - d(q, p)| \geq d(q, k_{cur})$.*

The proof of Lemma 5.2 is straightforward. If $|d(o, p) - d(q, p)| \geq d(q, k_{cur})$, it indicates that there exist at least k objects whose distances to q are not greater than o's distance. Therefore, o can be safely pruned. Let's apply Lemma 5.2 to the example query MkNNQ(o_4, 2) shown in Fig. 4. During the search, if we have already visited objects o_5 and o_6 and obtained the distance boundary $d(q, k_{cur}) = 2$, then object o_7 can be safely pruned using the pivot o_9 as $|d(o_7, o_9) - d(o_4, o_9)| = 3 > 2$.

Two typical solutions exist for kNN queries, including the best-first strategy and the range query-based strategy [15, 17]. Existing GPU-based indexes mainly apply the best-first strategy, which cannot be parallelized due to its iterative process relying on visited nodes to prune unvisited nodes, thereby suffering from unbalanced workload and memory issues [23]. To address this, we propose an alternative approach that capitalizes on GPUs' capacity for simultaneous computations by concurrently traversing tree nodes in a top-down and by-level manner. It progressively uses the distance boundary, initially set to infinity, and selects the current best k objects to derive the narrowed distance boundary, subsequently pruning lower-level tree nodes using Lemma 5.2.

Algorithm 5 presents the process of the metric kNN query in batch. Similar to the metric range query algorithm, the metric kNN query algorithm initializes the intermediate result table Q_Res, the set of nodes N_set in the current search layer $layer$, the node capacity N_c, and the tree height h (line 1). It then invokes the Knn_Q function to answer the queries (line 2). The Knn_Q function first computes the query groups Q_group in the same manner as Algorithm 4 (line 4). Next, for each group of queries Q', the function initializes the elements in the intermediate result table Q'_res (line 6). After computing the distance from each child node N' of N to the query object q, where $\{N, q\} \in Q\_Res$ (lines 7–11), the function sorts those distances via parallelized encoding and sorting strategies as in Algorithm 3 (line 12). This ensures that the distances from the objects of different nodes to the same query objects are located in contiguous segments in the temp table T_tmp. Thereafter, the function locates the k-th maximum distance $dis\_k[j]$ of each query MkNNQ(q, k) via T_tmp (line 13) and filters the unnecessary nodes by Lemma 5.2 (lines 14–17). Finally, the function decides whether to answer each MkNNQ query based on the current layer (line 19) or search the next layer (line 20).

Algorithm 5: Metric kNN Query

Input: a set of MkNN queries Q, the GTS index
Output: the answers to the metric kNN queries

initialize the Q_Res, N_set, N_c, $layer$, and h in the same step of Metric Range Query algorithm
Knn_Q($Q, Q\_Res, N\_set, layer, N_c, h$)
Function Knn_Q($Q, Q\_Res, N\_set, layer, N_c, h$)
compute the Q_group according to Q as Algo. 4 lines 9–13
foreach $Q' \in Q\_group$ **do**
    initialize Q'_Res to $\{\emptyset\}$
    foreach $E_i = \{N, q, k, d\} \in Q\_Res, j \in [0, N_c)$ ***in parallel*** **do**
        $N' \leftarrow N\_list[N.pos + j]$
        $dis \leftarrow$ the distance between $N'.pivot$ and q
        $Q'\_Res[i * N_c + j] \leftarrow \{N', q, k, dis)\}$
    $T\_tmp \leftarrow$ the sorted distances in Q'_Res via parallelized encoding and sorting strategies as Algorithm 3
    locate the current k-th maximum distance $dis\_k[i]$ of each query MkNNQ(q_i, k_i) via T_tmp
    foreach $E_i = \{N, q, k, d\} \in Q'\_Res$ ***in parallel*** **do**
        $dis \leftarrow$ the distance stored in $Q'\_Res[i]$
        if *N can be pruned by Lemma 5.2 using dis_k* **then**
            $Q'\_Res[i] \leftarrow \emptyset$
    $N'\_set \leftarrow$ the nodes in Q'_Res
    if $layer = h$ **then** output the answers to the queries in Q' by verifying the objects in the nodes of N'_set
    else Knn_Q($Q', Q'\_Res, N'\_set, layer + 1, N_c, h$)

Remark. Notably, the proposed method GTS holds the potential to handle multi-column scenarios. For instance, within the established PM-Tree framework [22], we can create a GTS index for each column and address multi-column queries by progressively combining the results of each queried attribute using Fagin's algorithm [21] and pigeon-hole principle [63].

5.3 Cost Model

In the following, we present a cost model for the similarity search, including MRQ and MkNNQ. As the queries are answered in parallel, following the GPU's parallelized schedule, our focus here is to estimate the cost for each single query.

We begin by analyzing the time cost of the metric range query. The proposed search process for the metric range query follows a top-down approach, traversing $\log_{N_c} n$ layers in the GTS index. Thus, the time complexity of the range query is $O(\sum_{i=1}^{\log_{N_c} n} \lceil \frac{S_i}{C} \rceil \cdot \log^2 S_i)$, where N_c represents the node capacity, S_i denotes the intermediate result size at layer i, and C indicates the GPU concurrent computing power. Now, let's evaluate the size of S_i. We start by assessing the probability that an object cannot be pruned by the index. Objects are partitioned into different subsets by pivot mapping at each level, and the distances between an object o and the pivots can be treated as random variables $X_1, X_2, \cdots$. According to Lemma 5.1, an object o in node N with pivot p cannot be pruned if $|d(o,p) - d(q,p)| \leq r$. Notably, an object cannot be excluded only if it cannot be pruned by pivots at every level. Thus, the probability that an object cannot be pruned at level i can be computed as:

$$P_r(o \text{ is not pruned}) = P_r(|X - Y| \leq r)^i. \tag{2}$$

Table 2. Dataset statistics

Dataset	Cardinality	Dimensionality	Distance Metric
Words	611,756	1~34	*Edit distance*
T-Loc	10,000,000	2	L_2-norm
Vector	200,000	300	*Word cosine distance*
DNA	1,000,000	108	*Edit distance*
Color	5,000,000	282	L_1-norm

Table 3. Evaluation parameters

Parameter	Value
Search radius r (x0.01%)	1, 2, 4, **8**, 16, 32
Integer k	1, 2, 4, **8**, 16, 32
Node capacity N_c	10, **20**, 40, 80, 160, 320
Number of queries in a batch	16, 32, 64, **128**, 256, 512
distinct data proportion (%)	20, 40, 60, 80, **100**
Cardinality (%)	**20**, 40, 60, 80, **100**

Here, Y denotes the distance from the query q to the pivot p, which follows the same distribution of X since q can also be regarded as a random object. Since X and Y are two independent, identically distributed random variables with variance σ^2, the mean and variance of $(X - Y)$ are 0 and $2\sigma^2$, respectively. Applying Chebyschev's inequality[15], we have

$$P_r(|X - Y| \leq r)^i \geq (1 - \frac{2\sigma^2}{r^2})^i. \tag{3}$$

Based on the above analysis, we can estimate the lower bound time complexity of MRQ as $O(\sum_{i=1}^{\log_{N_c} n} i^2 \lceil \frac{N_c^i (1-\frac{2\sigma^2}{r^2})^i}{C} \rceil \log^2 N_c)$, where n is the size of objects and N_c denotes the node capacity. Recall that MkNNQ can be answered by MRQ, and its search process also follows the top-down search manner of MRQ. Therefore, the above time complexity of MRQ also applies to MkNNQ.

Discussions. The time complexity analysis above highlights the strong correlation between search efficiency and two key factors: the node capacity N_c and the dataset characteristics. A larger N_c can lead to a reduction in the tree height, resulting in lower overhead to search different layers. However, it also allows fewer pivots to be used for pruning, limiting the pruning capability, as described in Equation (3). Let's consider the three situations as below:

(1) $n \ll C$: In this scenario, the GPU computing power exceeds the dataset size, and the overhead becomes $O(\sum_{i=1}^{\log_{N_c} n} i^2 \log^2 N_c)$, i.e., $O((\log_{N_c} n)^3 \cdot \log^2 N_c) = O(\frac{\log^3 n}{\log N_c})$. To optimize the efficiency, a large N_c is preferred, as it reduces the tree height, hence lowering the overhead to search different layers.

(2) $n \gg C$: In situations with large data sizes or high query throughput, the GPU cannot handle all the computing tasks simultaneously. The time complexity is simplified to $O(n(1 - (\frac{2\sigma^2}{r^2}))^{\log_{N_c} n} \log^2 n)$. Here, a smaller N_c results in higher efficiency.

(3) $O(n) = O(C)$: The search cost becomes complex and cannot be analyzed directly. However, considering that a few pivots can significantly reduce unnecessary distance computations [18, 43], it is suggested to use a relatively small number of pivots (hence, lower tree height) to strike a balance between the GPU parallelism and objects pruning capability, achieving high search efficiency.

6 EXPERIMENTS

In this section, we conduct empirical experiments to evaluate the performance of our proposed GPU-based tree index GTS, including the construction and update costs, the similarity search performance, and the scalability.

Table 4. Index construction cost of different methods

Method	*Words*		*T-Loc*		*Vector*		*DNA*		*Color*	
	Time (s)	Storage (MB)	Time (s)	Storage (MB)	Time (s)	Storage (MB)	Time (s)	Storage (MB)	Time (s)	Storage (MB)
BST	12.96	2.34	5.76	38.15	1.24	0.77	635.29	3.82	4.33	3.82
EGNAT	89.63	430	/	/	141.87	474.99	3447.89	637	676.01	2137.72
MVPT	0.41	2.52	10.17	38.099	12.17	0.76	33.02	3.85	35.81	3.81
GPU-Tree	3.94	6.2	87.44	105.06	1.21	2.09	28.7	10.14	6.34	10.5
LBPG-Tree	/	/	0.073	40.14	/	/	/	/	0.32	146.37
GANNS	/	/	/	/	2.35	48.82	/	/	10.28	244.14
GTS	0.11	2.63	1.1	41.1	0.2	1.05	2.47	4.11	0.54	4.1

Table 5. Update time of GTS under different cache table size

Dataset	0.01KB	0.1KB	1KB	5KB	10KB
Words [2]	0.01729s	0.01403s	**0.01368**s	0.01389s	0.01401s
T-Loc [24]	0.06865s	0.02144s	0.01667s	0.01561s	**0.01559**s
Vector [7]	0.02773s	0.01948s	0.01862s	**0.01852**s	0.01876s
DNA [3]	0.46009s	0.35260s	0.33381s	**0.31736**s	0.31738s
Color [8]	0.06106s	0.03200s	0.02992s	**0.02251**s	0.02254s

6.1 Experimental Settings

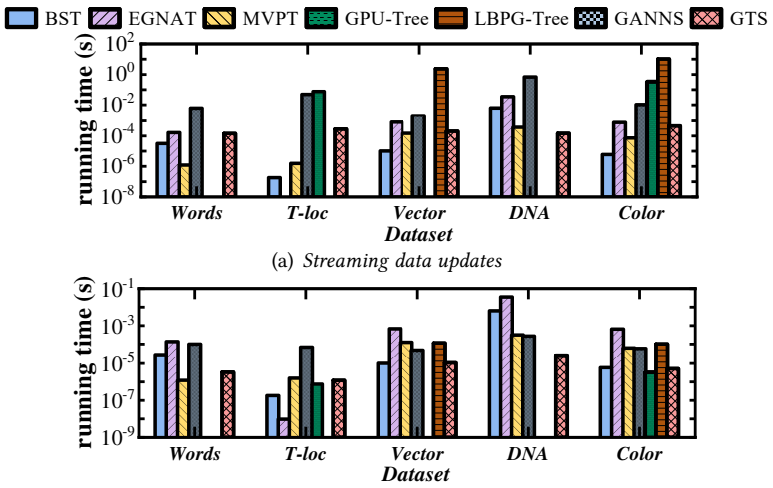
Datasets. We use five real-life datasets in our study: **(i)** *Words* [2] that contains proper nouns, acronyms, and compound words sourced from the Moby project, using edit distance to measure the similarity between words; **(ii)** *T-Loc* [24] that contains geographical locations of $10M$ Twitter-users, where the L_2-norm distance is employed as the distance metric; **(iii)** *Vector* [7] that includes 200K word embeddings of dimension 300 trained on the Spanish Billion Words Corpus, where the word cosine distance [1] is the distance metric; **(iv)** *DNA* [3] that consists of 1 million DNA data, where the edit distance is employed as the distance metric; and **(v)** *Color* [8] that contains image features extracted from *Frickr*, where the L_1-norm distance is employed to measure the similarity between features. Table 2 summarizes all the applied datasets.

Baselines. To evaluate our proposed index GTS, we first conducted a comparison against: **(i)** three most efficient CPU-based main-memory approaches for similarity search [17], including BST [32], EGNAT [44, 48], and MVPT [9, 10]; **(ii)** two GPU-based baselines for general metric spaces, including the GPU-Table that computes the distances between query and all the objects to answer MRQ and leverage Dr. Top-k algorithm [23] to answer MkNNQ, and the GPU-Tree that implements the SOTA GPU-based tree index G-PICS [38] strategy for general similarity search on single GPU by constructing multiple MVP-Trees; and **(iii)** two SOTA GPU-based special-purpose solutions, including LBPG-Tree [36] that constructs R-Trees on GPU for similarity search and GPU-based graph method GANNS [58] for vector kNN search.

Remark. Due to LBPG-Tree supports similarity search only on vector data with L_p-norm distance on *T-Loc* and *Color*, and GANNS is designed for vector similarity search on *T-Loc*, *Vector*, and *Color*, this paper reports only the available results of these two special-purpose baselines.

Configuration. All experiments were conducted on an Ubuntu server equipped with an Intel Core i9-10900X CPU, 128G of host memory, and an Nvidia Geforce RTX 2080 Ti GPU with 11G of device memory. The source code, data, and/or other artifacts have been made available [5].

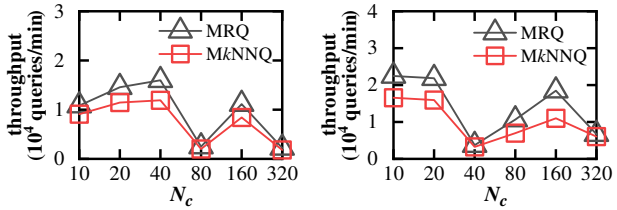
Parameters and Performance Metrics. In this study, we evaluate the performance of our proposed method GTS and its competitors, by exploring the impact of several key parameters. Specifically, we vary the search radius r for MRQ, the k for MkNNQ, the node capacity that controls



(a) *Streaming data updates*

(b) *Batch updates*

Fig. 5. Update cost



(a) *Words* (b) *Color*

Fig. 6. Effect of the node capacity N_c

the number of child nodes each tree node has, the number of queries in a batch, distinct data proportion, and the cardinality (i.e., the percentage relative to the entire dataset). Table 3 details the settings of key parameters, with default values shown in bold. In terms of cardinality, the default size of *Color* dataset is set to 20% to ensure baseline methods are executable within the limited GPU memory, while the default value is 100% for other datasets. To validate the efficiency and effectiveness of GTS, we measure several metrics, including index construction cost, update time, and throughput. Each measurement is based on the average performance across 100 randomly generated queries.

6.2 Construction and Update Performance

Firstly, we conduct a comparative analysis of the construction cost of GTS against state-of-the-art competitors. Notably, GPU-Table directly constructs one-time distance tables for online queries, eliminating index construction cost. However, EGNAT and GANNS encounter memory issues during the index construction for the *T-Loc* dataset, leading to unreported results in this regard. The results, presented in Table 4, clearly highlight the significantly low construction costs of GTS. Notably, GTS completes index construction in less than 3 seconds for each dataset, achieving 1.5~10× lower construction time cost compared to other general methods across all datasets. This advantage is attributed to the efficient utilization of continuous memory allocation for the pivot-based tree index, maximizing the usage of computing resources. Additionally, in datasets *Color* and *Vector* that exhibit similar distance computational complexity, the space overhead of GTS index on

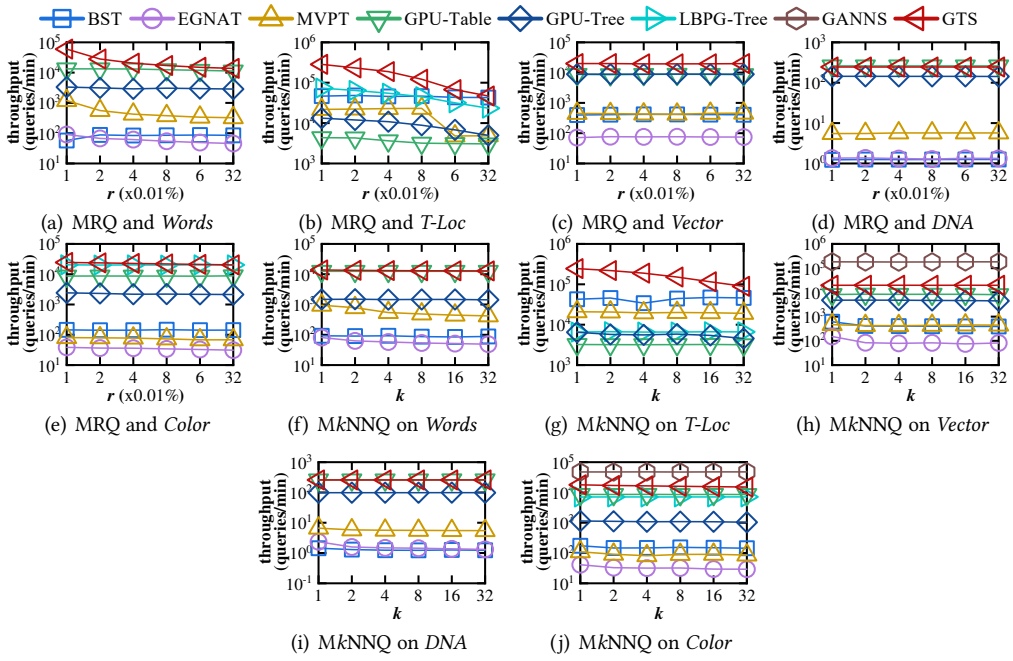
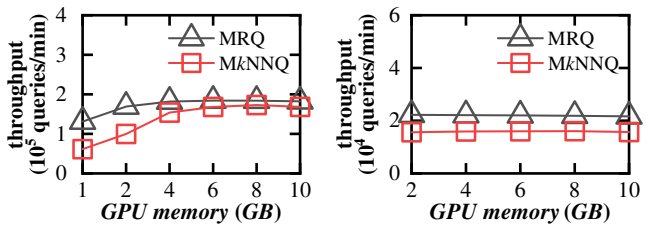


Fig. 7. Effect of r and k

Color is almost 4× that of *Vector*, due to the fivefold difference in data volume. However, the time cost of GTS index on *Color* is only 2.7× that of *Vector*. This indicates that the parallel computing capability of GPUs on *Vector* exceeds the dataset size, resulting in time complexity overhead lower than $O(n)$. This aligns with the analysis of the proposed construction cost model. Furthermore, compared with special-purpose solutions, we observe that: **(i)** LBPG-Tree exhibits a lower construction cost, while GTS supports more datasets with various distance metrics; and **(ii)** compared to GANNS, GTS exceeds in constructing a 40 × smaller index within more than 10× faster speed, and supports larger datasets, such as *T-Loc*.

Next, we investigate the impact of the cache table size on update operations. To evaluate the effect of update operations, we perform 5000 updates. Each update operation involves randomly removing an object from GTS, subsequently reinserting it, and performing a random similarity range query. Notably, the index is efficiently rebuilt once the number of objects stored in the cache table exceeds its size limit. The results in Table 5 demonstrate that GTS supports efficient streaming object updates. Notably, the update efficiency of GTS exhibits an interesting trend, initially decreasing and then increasing concerning the cache table size, which can be attributed to the trade-off between update efficiency and search efficiency. Specifically, a larger cache table size implies less available space for concurrent similarity search, leading to a higher search cost as fewer queries can be searched simultaneously. To strike a balance between update efficiency and search efficiency, we recommend setting the cache table size to be around 5KB. In real applications, the cache size could be flexibly adjusted based on our proposed index updating cost model to meet users' requirements.

We also compare the time cost of streaming data updates and batch data updates. For the former, we simulate the process of randomly removing an object and then reinserting it. For the latter, we remove 10% of the objects from the dataset randomly and then insert them back. The results, presented in Fig. 5, reveal that CPU-based methods exhibit higher efficiency in handling



(a) *T-Loc* (b) *Color*

Fig. 8. Effect of the GPU memory

streaming data updates, while GPU-based methods demonstrate higher efficiency in managing batch updates for large and complex data. Notably, GTS showcases remarkable performance for streaming data updates compared to other GPU-based methods, including LBPG-Tree and GANNS. These alternatives necessitate a complete rebuild from scratch for any data updates. Additionally, GPU-Tree, leveraging single GPU cores for complex tree structure updating, faces an efficiency bottleneck. Moreover, GTS emerges as the optimal solution for managing dynamic data updates on *DNA*. These findings imply that GTS is particularly well-suited for handling diverse cancer omics data with a large volume.

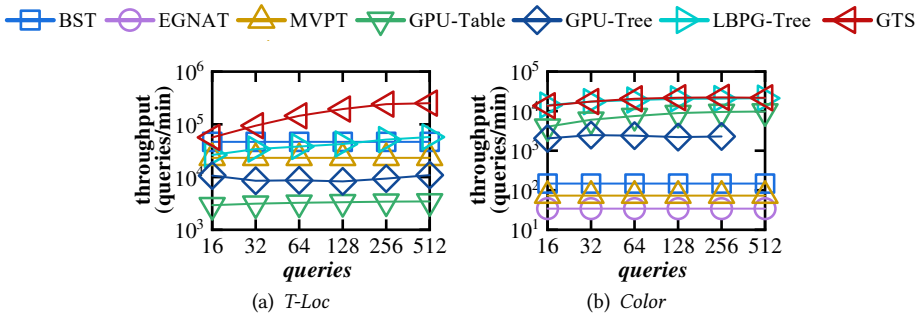
6.3 Similarity Search Performance

We proceed to explore the similarity search performances of GTS and its competitors by varying three parameters: the tuning parameter node capacity, the search radius r for MRQ, and the desired number k for MkNNQ.

Effect of Node Capacity N_c. Fig. 6 shows the GTS's performance of MRQ and MkNNQ under various node capacities N_c. Generally, the query efficiency exhibits alternatively increasing and decreasing patterns. A small N_c requires visiting more tree layers, reducing the level of computing concurrency. On the other hand, a small N_c also utilizes more pivots for object pruning, thereby avoiding a higher volume of unnecessary distance computations. Consequently, search performance depends on the trade-off between parallelism and pruning efficiency, while our evaluation validates that a small N_c can strike a balance and showcase high search performance, which is consistent with our analysis based on the proposed cost model presented in Section 5.3. Therefore, we set the node capacity to 20 for higher search efficiency.

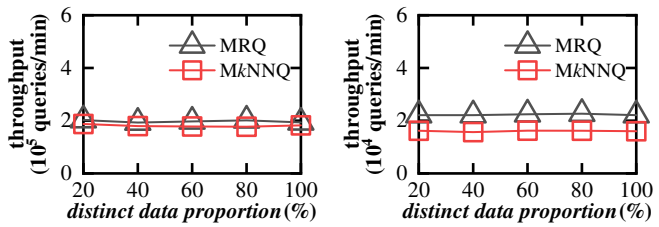
Effects of r and k. In Fig. 7, we present the performance of MRQ and MkNNQ for different algorithms on five real datasets, varying the search radius r and the integer k. Our proposed GPU-based tree index GTS demonstrates superior efficiency, outperforming all baseline methods for general metric space across all datasets. This demonstrates the effectiveness and efficiency of our concurrent similarity search algorithms. Notably, GTS achieves up to two orders of magnitude higher throughput performance compared to CPU-based methods and supports up to 20x larger throughput than GPU-based general methods. This significant speedup can be attributed to two main factors. First, the proposed tree-based index stored in a continuous table reduces unnecessary distance computations while harnessing the superior computing power of the GPU, resulting in a faster query process. Second, GTS can simultaneously answer multiple queries using a two-stage concurrent similarity search, effectively managing the memory cost for concurrent queries to optimize GPU computing resources. As a result, GTS dramatically improves the similarity search performance.

Compared with special-purpose solutions, we find that: **(i)** GTS achieves higher efficiency on all the datasets than LBPG-Tree, due to the better usage of GPU computing power; and **(ii)** though



(a) *T-Loc* (b) *Color*

Fig. 9. MRQ performance vs. the number of queries



(a) *T-Loc* (b) *Color*

Fig. 10. Effect of identical objects

GANNS outperforms GTS in MkNNQ efficiency measured in the order of milliseconds, GTS supports a broader range of datasets and offers efficient MRQ capabilities. This suggests that GTS has wider applicability and can be utilized in various scenarios.

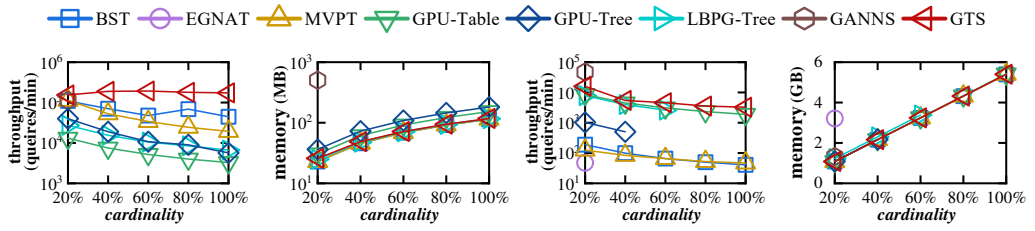
6.4 Scalability Analysis

Finally, we access the scalability of GTS by varying the GPU memory, the number of queries in a batch, the percentage of distinct objects, and dataset cardinality.

Effect of GPU memory. In Fig. 8, we present the results for MkNNQ and MRQ when varying the GPU memory size. Due to the dataset size of *Color* is 1.1G, the results of *Color* using 1G GPU memory is not reported. As observed, the throughput generally increases when more GPU memory becomes available. Notably, the throughput remains unchanged with the increasing GPU memory on *Color*. The reason behind this is that GPU computing resources are fully utilized, while only a small part of memory is sufficient for all the computing tasks, making the impact of GPU memory limited under such conditions.

Effect of the Concurrency. To demonstrate the effectiveness of our proposed similarity batch search strategy, we compare the MkNNQ performance of GTS with its competitors while varying the number of queries in a batch in Fig. 9. Firstly, we observe that GPU-Tree faces the memory deadlock problem on *Color* for 512 queries in a batch, justifying our analysis in Section 1. Besides, with the increasing number of queries, the throughput of all GPU-based methods rises as more similarity measurements can be computed in parallel. In contrast, CPU-based methods remain unaffected by the number of queries. Remarkably, GTS consistently outperforms other methods, demonstrating its superior ability to support concurrent similarity search.

Effect of identical objects. To investigate the influence of identical objects on the efficiency of similarity search, we conduct experiments with varying proportions of distinct data on the *T-Loc* and *Color* datasets. The findings presented in Fig. 10 illustrate that the search performance of



(a) *T-Loc* (b) *T-Loc* (c) *Color* (d) *Color*

Fig. 11. M*k*NNQ performance consumption vs. *Cardinality*

the proposed GTS remains unaffected by the presence of identical objects, thereby validating the effectiveness of our approach.

Effect of Dataset Cardinality. We vary the cardinality of all datasets from 20% to 100% and present the M*k*NNQ results in Fig. 11. As observed, the throughput decreases with the dataset size. Notably, the increasing dataset size poses a memory issue for EGNAt on *T-Loc* and *Color*, due to the large storage size for storing pre-computed distances. Meanwhile, some GPU-based methods, including GPU-Tree and GANNS, also face memory issue on *Color*. This is attributed to their utilization of GPU-cores for various queries, demanding a significant amount of memory to store intermediate results. Meanwhile, due to LBPG-Tree facing the dimension curse, it also runs out of memory on *Color* with a cardinality of 80%. Conversely, GTS scales well with the increasing dataset size. This is because of our proposal to dynamically group the intermediate results of each query according to the available memory. We then process each group sequentially to obtain real results, with the queries in each group computed in parallel, circumventing memory deadlocks. Thus, GTS scales effectively with increasing data sizes.

Remark. Throughout the entire experiment, GTS consistently outperforms general methods, and stands out as the optimal solution for *Words*, *T-loc*, and *DNA*. Based on these results, we can conclude that our GPU-based tree index GTS holds the promise for efficiently managing dynamic data of various types with flexible metrics. It proves to be a pivotal solution for enhancing the scalability of vector databases and offering cost-effective updates.

7 CONCLUSIONS

In this paper, we propose GTS, a highly efficient GPU-based tree index for similarity search. GTS incorporates a pivot-based tree index for object management, stored in a continuous node list, along with a cache table to support streaming data updates and batch updates for dynamic scenarios. In addition, we introduce efficient two-stage search methods that support concurrent similarity search with high query throughput. To strike a balance between the pruning capability and the parallel computing efficiency, we develop a cost-based optimization model. Extensive experiments demonstrate that our GTS outperforms state-of-the-art CPU-based and GPU-based methods, offering more efficient concurrent similarity search and scaling well with data size. These results highlight the superior efficiency and scalability of GTS, making it a promising solution for various real-life applications. Moving forward, we plan to explore approximate similarity search with learned index based on 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, Yongjiang Talent Introduction Programme (2022A-237-G), and the Key Lab of Intelligent Computing Based Big Data of Zhejiang Province. Yunjun Gao is the corresponding author of the work.

REFERENCES

[1] 2013. *Word to vectors.* https://code.google.com/archive/p/word2vec
[2] 2023. *Moby project.* https://mobyproject.org
[3] 2023. *National Library of Medicine.* http://www.ncbi.nlm.nih.gov/genome
[4] 2023. *Nvidia GPU introduction.* https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090
[5] 2023. *The source code of GTS.* https://github.com/ZJU-DAILY/GTS/
[6] Ricardo J. Barrientos, Javier A. Riquelme, Ruber Hernández-García, Cristóbal A. Navarro, and Wladimir E. Soto-Silva. 2022. Fast kNN query processing over a multi-node GPU environment. *J. Supercomput.* 78, 2 (2022), 3045–3071.
[7] Aritz Bilbao-Jayo and Aitor Almeida. 2018. Automatic political discourse analysis with multi-scale convolutional neural networks and contextual data. *International Journal of Distributed Sensor Networks* 14, 11 (2018), 1550147718811827.
[8] 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.*
[9] Tolga Bozkaya and Z. Meral Özsoyoglu. 1997. Distance-Based Indexing for High-Dimensional Metric Spaces. In *SIGMOD.* 357–368.
[10] Tolga Bozkaya and Z. Meral Özsoyoglu. 1999. Indexing Large Metric Spaces for Similarity Search Queries. *ACM Trans. Database Syst.* 24, 3 (1999), 361–404.
[11] Sergey Brin. 1995. Near Neighbor Search in Large Metric Spaces. In *VLDB*, Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio (Eds.). 574–584.
[12] Carrie J. Cai, Emily Reif, Narayan Hegde, Jason D. Hipp, Been Kim, Daniel Smilkov, Martin Wattenberg, Fernanda B. Viégas, Gregory S. Corrado, Martin C. Stumpe, and Michael Terry. 2019. Human-Centered Tools for Coping with Imperfect Algorithms During Medical Decision-Making. In *CHI.* 4.
[13] Edgar Chávez and Gonzalo Navarro. 2000. An Effective Clustering Algorithm to Index High Dimensional Metric Spaces. In *Seventh International Symposium on String Processing and Information Retrieval.* 75–86.
[14] Edgar Chávez and Gonzalo Navarro. 2005. A compact space decomposition for effective metric indexing. *Pattern Recognit. Lett.* 26, 9 (2005), 1363–1376.
[15] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Maproquín. 2001. Proximity searching in metric spaces. *Comput. Surveys* 33, 3 (2001), 273–321.
[16] Lu Chen, Yunjun Gao, Xinhan Li, Christian S. Jensen, and Gang Chen. 2017. Efficient Metric Indexing for Similarity Search and Similarity Joins. *IEEE Trans. Knowl. Data Eng.* 29, 3 (2017), 556–571.
[17] Lu Chen, Yunjun Gao, Xuan Song, Zheng Li, Yifan Zhu, Xiaoye Miao, and Christian S. Jensen. 2023. Indexing Metric Spaces for Exact Similarity Search. *Comput. Surveys* 55, 6 (2023), 128:1–128:39.
[18] Lu Chen, Yunjun Gao, Baihua Zheng, Christian S. Jensen, Hanyu Yang, and Keyu Yang. 2017. Pivot-based Metric Indexing. *Proc. VLDB Endow.* 10, 10 (2017), 1058–1069.
[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] Brian Donnelly and Michael Gowanlock. 2020. A coordinate-oblivious index for high-dimensional distance similarity searches on the GPU. In *ICS.* 8:1–8:12.
[21] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.* 66, 4 (2003), 614–656.
[22] Maximilian Franzke, Tobias Emrich, Andreas Züfle, and Matthias Renz. 2016. Indexing multi-metric data. In *ICDE.* 1122–1133.
[23] Anil Gaihre, Da Zheng, Scott Weitze, Lingda Li, Shuaiwen Leon Song, Caiwen Ding, Xiaoye S. Li, and Hang Liu. 2021. Dr. Top-k: delegate-centric Top-k on GPUs. In *SC.* 39.
[24] Bishwamittra Ghosh, Mohammed Eunus Ali, Farhana Murtaza Choudhury, Sajid Hasan Apon, Timos Sellis, and Jianxin Li. 2018. The Flexible Socio Spatial Group Queries. *Proc. VLDB Endow.* 12, 2 (2018), 99–111.
[25] Gaurav Gupta, Minghao Yan, Benjamin Coleman, Bryce Kille, Ryan A. Leo Elworth, Tharun Medini, Todd J. Treangen, and Anshumali Shrivastava. 2021. Fast Processing and Querying of 170TB of Genomics Data via a Repeated And Merged BloOm Filter (RAMBO). In *SIGMOD.* 2226–2234.

[26] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, Beatrice Yormark (Ed.). ACM Press, 47–57.

[27] Dorit S. Hochbaum and David B. Shmoys. 1985. A Best Possible Heuristic for the k-Center Problem. *Math. Oper. Res.* 10, 2 (1985), 180–184.

[28] Linjia Hu, Saeid Nooshabadi, and Majid Ahmadi. 2016. Parallel randomized KD-tree forest on GPU cluster for image descriptor matching. In *ISCAS*. 582–585.

[29] Peng Jiang, Sanju Sinha, Kenneth Aldape, Sridhar Hannenhalli, Cenk Sahinalp, and Eytan Ruppin. 2022. Big data in basic and translational cancer research. *Nature Reviews Cancer* 22, 11 (2022), 625–639.

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

[31] Caetano Traina Jr., Roberto F. Santos Filho, Agma J. M. Traina, Marcos R. Vieira, and Christos Faloutsos. 2007. The Omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient. *VLDB J.* 16, 4 (2007), 483–505.

[32] Iraj Kalantari and Gerard McDonald. 1983. A Data Structure and an Algorithm for the Nearest Point Problem. *IEEE Trans. Software Eng.* 9, 5 (1983), 631–634.

[33] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. 2013. Parallel multi-dimensional range query processing with R-trees on GPU. *J. Parallel Distributed Comput.* 73, 8 (2013), 1195–1207.

[34] Juhwan Kim, Jongseon Seo, Jonghyeok Park, Sang-Won Lee, Hongchan Roh, and Hyungmin Cho. 2022. ES4D: Accelerating Exact Similarity Search for High-Dimensional Vectors via Vector Slicing and In-SSD Computation. In *ICCD*. 298–306.

[35] Mincheol Kim, Ling Liu, and Wonik Choi. 2018. A GPU-Aware Parallel Index for Processing High-Dimensional Big Data. *IEEE Trans. Computers* 67, 10 (2018), 1388–1402.

[36] Mincheol Kim, Ling Liu, and Wonik Choi. 2022. Multi-GPU Efficient Indexing For Maximizing Parallelism of High Dimensional Range Query Services. *IEEE Trans. Serv. Comput.* 15, 5 (2022), 2910–2924.

[37] Doris Jung Lin Lee, Dixin Tang, Kunal Agarwal, Thyne Boonmark, Caitlyn Chen, Jake Kang, Ujjaini Mukhopadhyay, Jerry Song, Micah Yong, Marti A. Hearst, and Aditya G. Parameswaran. 2021. Lux: Always-on Visualization Recommendations for Exploratory Dataframe Workflows. *Proc. VLDB Endow.* 15, 3 (2021), 727–738.

[38] Zhila Nouri Lewis and Yi-Cheng Tu. 2022. G-PICS: A Framework for GPU-Based Spatial Indexing and Query Processing. *IEEE Trans. Knowl. Data Eng.* 34, 3 (2022), 1243–1257.

[39] Chuanwen Li, Yu Gu, Jianzhong Qi, Jiayuan He, Qingxu Deng, and Ge Yu. 2018. A GPU Accelerated Update Efficient Index for kNN Queries in Road Networks. In *ICDE*. 881–892.

[40] Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: a survey. *VLDB J.* 29, 1 (2020), 393–418.

[41] Lijuan Luo, Martin D. F. Wong, and Lance Leong. 2012. Parallel implementation of R-trees on the GPU. In *ASP-DAC*. 353–358.

[42] Yuyu Luo, Yihui Zhou, Nan Tang, Guoliang Li, Chengliang Chai, and Leixian Shen. 2023. Learned Data-aware Image Representations of Line Charts for Similarity Search. *Proc. ACM Manag. Data* 1, 1 (2023), 88:1–88:29.

[43] Rui Mao, Willard L. Miranker, and Daniel P. Miranker. 2012. Pivot selection: Dimension reduction for distance-based indexing. *J. Discrete Algorithms* 13 (2012), 32–46.

[44] Mauricio Marín, Roberto Uribe, and Ricardo J. Barrientos. 2007. Searching and Updating Metric Space Databases Using the Parallel EGNAT. In *ICCS*. 229–236.

[45] Luisa Micó, José Oncina, and Enrique Vidal. 1994. A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. *Pattern Recognit. Lett.* 15, 1 (1994), 9–17.

[46] Juraj Mosko, Jakub Lokoc, and Tomás Skopal. 2011. Clustered pivot tables for I/O-optimized similarity search. In *SISAP*. 17–24.

[47] Naohito Nakasato. 2012. Implementation of a parallel tree method on a GPU. *J. Comput. Sci.* 3, 3 (2012), 132–141.

[48] Gonzalo Navarro and Roberto Uribe Paredes. 2011. Fully dynamic metric access methods based on hyperplane partitioning. *Inf. Syst.* 36, 4 (2011), 734–747.

[49] Guillermo Ruiz, Francisco Santoyo, Edgar Chávez, Karina Figueroa, and Eric Sadit Tellez. 2013. Extreme Pivots for Faster Metric Indexes. In *SISAP*. 115–126.

[50] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A Hybrid B+-tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *SIGMOD*. 1523–1538.

[51] Larissa Capobianco Shimomura, Rafael Seidi Oyamada, Marcos R. Vieira, and Daniel S. Kaster. 2021. A survey on graph-based methods for similarity searches in metric spaces. *Inf. Syst.* 95 (2021), 101507.

[52] Dejun Teng, Akshay Nehe, Prajeeth Emanuel, Furqan Baig, Jun Kong, and Fusheng Wang. 2021. GPU-based Real-time Contact Tracing at Scale. In *SIGSPATIAL*. 1–10.

[53] Kyle J Tomek, Kevin Volkel, Elaine W Indermaur, James M Tuck, and Albert J Keung. 2021. Promiscuous molecules for smarter file operations in DNA-based data storage. *Nature communications* 12, 1 (2021), 3518.
[54] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 14, 11 (2021), 1964–1978.
[55] Yue Wang, Zhe Wang, Ziyuan Zhao, Zijian Li, Xun Jian, Hao Xin, Lei Chen, Jianchun Song, Zhenhong Chen, and Meng Zhao. 2022. Effective Similarity Search on Heterogeneous Networks: A Meta-Path Free Approach. *IEEE Trans. Knowl. Data Eng.* 34, 7 (2022), 3225–3240.
[56] Martin Winter, Mathias Parger, Daniel Mlakar, and Markus Steinberger. 2021. Are dynamic memory managers on GPUs slow?: a survey and benchmarks. In *PPoPP*. 219–233.
[57] Simin You, Jianting Zhang, and Le Gruenwald. 2013. Parallel spatial query processing on GPUs using R-trees. In *SIGSPATIAL*. 23–31.
[58] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2022. GPU-accelerated Proximity Graph Approximate Nearest Neighbor Search and Construction. In *ICDE*. 552–564.
[59] 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.
[60] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate Nearest Neighbor Search on GPU. In *ICDE*. 1033–1044.
[61] Jingbo Zhou, Qi Guo, H. V. Jagadish, Lubos Krcál, Siyuan Liu, Wenhao Luan, Anthony K. H. Tung, Yueji Yang, and Yuxin Zheng. 2018. A Generic Inverted Index Framework for Similarity Search on the GPU. In *ICDE*. 893–904.
[62] Yifan Zhu, Lu Chen, Yunjun Gao, and Christian S. Jensen. 2022. Pivot Selection Algorithms in Metric Spaces: A Survey and Experimental Study. *VLDB J.* 31, 1 (2022), 23–47.
[63] Yifan Zhu, Lu Chen, Yunjun Gao, Baihua Zheng, and Pengfei Wang. 2022. DESIRE: An Efficient Dynamic Cluster-based Forest Indexing for Similarity Search in Multi-Metric Spaces. *PVLDB* 15, 10 (2022), 2121–2133.

Received October 2023; revised January 2024; accepted February 2024