LM-Tree: A Hybrid Learned Index for Similarity Search in Metric Spaces

YAQI WANG, Northeastern University, China
BIN WANG*, Northeastern University, China
RUI ZHU*, Shenyang Aerospace University, China
WENLI SUN, Northeastern University, China
XIAOCHUN YANG, Northeastern University, China

Similarity search in metric spaces is a fundamental problem in data management with many applications. While numerous indices have been proposed to support similarity search, most existing approaches rely on pivot-based strategies that suffer from critical limitations. Traditional single-pivot methods offer limited pruning power, while multi-pivot techniques often become inefficient as data evolves, since pivot updates incur substantial computational overhead. In this paper, we propose LM-Tree (short for Learned M-Tree), a hybrid learned index that combines pivots and learning models with M-Tree to address the above problems. LM-Tree's key innovation lies in its self-adaptive node architecture, where each node dynamically selects an appropriate number of pivots and incorporates lightweight learning models to enhance pruning efficiency. This design enables each node to maintain a relatively large number of child nodes (or objects for leaf nodes) while consistently delivering strong pruning performance, thereby enabling LM-Tree to use a small number of nodes to index objects in metric spaces. Furthermore, we develop an efficient maintenance algorithm that handles dynamic updates, including pivot adjustments, model reforms, node splits, and merges with low overhead. Extensive experiments on both real and synthetic datasets demonstrate that LM-Tree significantly outperforms state-of-the-art methods.

CCS Concepts: • **Information systems → Data management systems**; **Data structures**; **Data access methods**; **Multidimensional range search**;

Additional Key Words and Phrases: Metric Spaces; Learned Index; Similarity Search

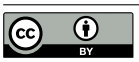
ACM Reference Format:
Yaqi Wang, Bin Wang, Rui Zhu, Wenli Sun, and Xiaochun Yang. 2026. LM-Tree: A Hybrid Learned Index for Similarity Search in Metric Spaces. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 51 (February 2026), 25 pages. https://doi.org/https://doi.org/10.1145/3786665

1 Introduction

Similarity search in metric spaces is a fundamental problem in data management, which is widely used in image retrieval, recommendation systems, bioinformatics, and more. In content-based image retrieval systems, similarity search enables efficient matching of query images against

*Corresponding author

Authors' Contact Information: Yaqi Wang, wangyq@stumail.neu.edu.cn, Northeastern University, Shenyang, Liaoning, China; Bin Wang, Northeastern University, Shenyang, Liaoning, China, binwang@mail.neu.edu.cn; Rui Zhu, Shenyang Aerospace University, Shenyang, Liaoning, China, zhurui@sau.edu.cn; Wenli Sun, Northeastern University, Shenyang, Liaoning, China, redamancyguy@163.com; Xiaochun Yang, Northeastern University, Shenyang, Liaoning, China, yangxc@mail.neu.edu.cn.



ACM 2836-6573/2026/2-ART51
https://doi.org/https://doi.org/10.1145/3786665

database entries by comparing their similarity. In bioinformatics, it could identify homologous gene sequences, facilitating critical tasks like disease diagnosis and genomic analysis.

Similarity search in metric spaces mainly includes metric range query and k nearest neighbor query [1–4]. Here, a metric space is modeled as a pair (O, d), where O represents a set of objects, and d is a distance function quantifying object similarity. In particular, the distance function d satisfies the following four properties: (i) symmetry; (ii) non-negativity; (iii) identity; and (iv) triangle inequality. Given an object set $O \subseteq \mathbb{U}$, a query object q, and a distance threshold r_q, a range query returns objects in O that are within the distance threshold r_q of q. Similarly, a k nearest neighbor query retrieves the k closest objects to q within the dataset O.

Due to the importance of similarity search in metric spaces, many efforts have been proposed. Their goal is to develop efficient indices that could provide powerful pruning ability to support similarity search [2–6]. Existing approaches can be classified into three categories based on their pivot utilization strategies: (1) *single-pivot-based*, (2) *multi-pivot-based*, and (3) *hybrid* metric indices [5–9]. Single-pivot-based indices [5, 7–9] typically employ hierarchical tree structures (e.g., M-Tree and its variants) to organize objects. These methods rely on a single pivot per node (often the node center) for partitioning and pruning. However, this design suffers from two key limitations: First, the pruning effectiveness of a single pivot is inherently constrained, frequently necessitating exhaustive scans of all child nodes or objects. Second, the extensive node count and deep tree hierarchy introduce significant computational overhead, as these methods necessitate traversing and accessing a large number of nodes within the index across multiple levels.

Multi-pivot-based methods [3, 4, 10, 11] employ multiple boundary objects as pivots within each node or partition, and organize objects based on their distance relationships to pivots. In this way, each object can be associated with multiple pivots, which significantly enhances pruning effectiveness during similarity search. However, their performance is highly sensitive to initial pivots, yet they lack mechanisms for dynamic pivot refinement as the data distribution evolves. Consequently, their pruning power degrades over time as originally selected pivots become less effective. Second, the need to store distances from all objects to multiple pivots introduces substantial storage overhead. Hybrid indices [6, 12] attempt to combine the benefits of both approaches by augmenting tree-based structures with global pivots. However, similar to multi-pivot-based methods, global pivots may become ineffective as data updates, and such indices still fail to overcome core limitations of either method. Above all, these methods lack the ability to provide strong pruning power with indices, or cannot maintain the effectiveness of pivots as data updates, and fail to efficiently support similarity search in metric spaces. An efficient index that could use as few as pivots for effective pruning, while supporting pivot set self-refinement as the data distribution evolves, is highly desired.

The description of our solution is based on the following observations. Given a node e within an M-Tree, if we could select a few proper pivots from it, these pivots could provide more powerful pruning ability than a single pivot for the node e. Additionally, if distances from objects (or child nodes) within e to a pivot follow an approximately uniform distribution, they can be efficiently modeled with a linear model, which further helps us to enhance pruning efficiency similar to learned indices [13, 14]. Lastly, if we can identify a few pivots that could utilize linear models to manage objects (or child nodes) within e and its sibling nodes, we could merge e and its sibling into fewer nodes based on these selected pivots. In this way, we could obtain an efficient index that has a smaller number of nodes, but could provide strong pruning power.

Based on these observations, we propose a novel learned index named LM-Tree (short for Learned M-Tree), built upon the M-Tree structure (details in Section 3.1). Specifically, each node contains one or a few pivots. In both inner and leaf nodes, each pivot manages a subset of child nodes or objects, respectively, with their distances from the pivot following an approximately uniform distribution.

To accelerate the search, each pivot is equipped with a lightweight linear model that approximates the distances to its managed child nodes or objects. Additionally, LM-Tree dynamically adjusts the number of child nodes (or objects in leaf nodes). This flexibility enables nodes to maintain a relatively large number of child nodes in inner nodes (or objects in the case of leaf nodes), reducing the total number of nodes that the LM-Tree should maintain. The above designs ensure that the LM-Tree achieves a small number of nodes while maintaining high pruning effectiveness.

However, we have to overcome the following three challenges. Firstly, data distributions are often complex and diverse, and it is difficult to form a proper LM-Tree for indexing objects in metric spaces by selecting as few as possible proper pivots. Secondly, it is difficult to develop efficient algorithms that utilize pivots and linear models within nodes to support similarity search. Thirdly, in dynamic scenarios, the initially chosen pivots and linear models may become ineffective over time. It is difficult to efficiently update them.

We first propose novel algorithms to support LM-Tree construction. It starts by forming a traditional M-Tree based on objects, where we use it as the initial LM-Tree. Then, we propose a two-step algorithm to form LM-Tree. In step one, we propose a segmentation-based algorithm for locally selecting proper pivots (also forming linear models) within each node of the index (details in Section 4.1.1). One of its natural properties is that it has the ability to adaptively distinguish objects that are suitable for selecting as pivots, as well as objects that should be maintained by these pivots.

In step two, we propose a node merge-based refinement algorithm to further reduce the node (also pivot and linear model) scale (details in Section 4.1.2). It evaluates whether selected pivots (also their corresponding linear models) in each node can also effectively manage elements within their nearby nodes. If so, we merge these nodes together, but retain fewer pivots within these nodes, so as to reduce the index node count. The above algorithm finally helps us achieve the goal of forming an efficient index that has a small number of nodes, but can provide strong pruning power.

We then propose search algorithms that could fully utilize pivots and learned linear models to efficiently process similarity search (details in Section 3.2). Taking the metric range query in metric spaces as an example, we first introduce a pivot-based pruning method, which can eliminate partially irrelevant elements by analyzing distance relationships between query points, pivots, and node centers (also radii). For the remaining elements, we further propose a linear model-based algorithm to reduce the search scope.

As pivots are selected from the nodes of the LM-Tree, it provides us with the ability to self-adaptively update pivots based on the distribution of nodes as data updates. We propose a deferred-based incremental maintenance algorithm. It can efficiently handle newly inserted (or deleted) objects, as well as supporting pivot (or linear model) updating, node splitting and merging, significantly reducing the overhead of pivot maintenance.

In summary, the main contributions of this paper are as follows.

- We develop a hybrid learned index LM-Tree built upon the M-Tree structure, which combines pivots and learning models to support efficient similarity search and updates. (Section 3)
- We propose an adaptive pivot selection method based on ShrinkingCone and node merging techniques. Each node dynamically selects an appropriate number of pivots and incorporates lightweight learning models to enhance pruning efficiency. (Section 4.1)
- We design efficient algorithms for incremental maintenance, including pivot adjustments, model reforms, node splits, and merges with low overhead. (Section 4.2)
- We conduct extensive experimental evaluation on three real datasets and one synthetic dataset. The experimental results demonstrate that LM-Tree significantly outperforms state-of-the-art methods. (Section 5).

2 Preliminaries

In Section 2.1, we first discuss related works on indices in metric spaces, as well as classic learned indices. In Section 2.2, we briefly discuss the traditional M-Tree. Table 1 defines the list of frequently used notations in this paper.

2.1 Related Work

In this section, we first review classic metric-space indices for exact similarity search, and then examine learned indices.

2.1.1 Metric Indices. Existing metric indices can be classified into three categories: single-pivot-based, multi-pivot-based, and hybrid indices. Single-pivot-based indices typically adopt node centers as pivots, leverage distance relationships among node centers, radii, and query points to prune irrelevant regions during query processing. Ciaccia et al. [5] propose the first balanced M-Tree that can support dynamic insertion and deletion. As a representative single-pivot-based index, M-Tree organizes objects in a hierarchical manner, where each child node (or data object in leaf nodes) is associated with a single pivot, i.e., the center of its parent node. Following M-Tree, several of its variants [2, 7, 8, 15–19] have been presented, which aim to enhance pruning power. Slim-Tree [2, 15] uses a minimum spanning tree [20, 21] to maximize distances between nodes, ensuring that the distance among objects with the same node is as small as possible. VP-Tree [9] hierarchically partitions objects within each node based on a vantage point (pivot).

Multi-pivot-based methods select a set of pivots from boundary objects, use distances from objects to these pivots to organize objects, so as to improve pruning effectiveness. MVP-Tree [3] extends the classic VP-Tree into an m-ary tree, which uses multiple vantage points per node to divide the dataset, enabling the algorithm to apply multiple triangle inequalities to rule out non-relevant regions more aggressively. SPB-tree [11] adopts a multi-pivot strategy by selecting a set of global pivots and pre-computing object-to-pivot distances. These distances are then mapped to integers using a space-filling curve. Accordingly, these mapped values are organized by a B^+-tree-based structure to support similarity search.

Hybrid indices [6, 12, 22, 23] combine advantages of single-pivot-based and multi-pivot-based methods, achieving better pruning performance. The index PM-Tree [6] builds upon the M-Tree by additionally selecting some global pivots to reduce the search space during queries. Similarly, DF-Tree [12] enhances the M-Tree by integrating global pivot filtering. GNAT [24] and its dynamic variant EGNAT [25] employ multiple pivots per node to partition the dataset using generalized hyperplane partitioning.

2.1.2 Learned Indices. Extensive research [13, 14, 26–30] has demonstrated that learned indices outperform traditional ones in query speed and storage efficiency for one-dimensional data, as well as spatial data, but few studies have explored learned indices for metric spaces. LIMS [4] is a learned metric index. It first reduces distances from objects to a set of pivots into a one-dimensional value, and then uses a one-dimensional learned index to manage these values, thereby accelerating query processing. However, these one-dimensional values cannot effectively reflect distance relationships among objects in metric spaces, leading that the index cannot provide powerful pruning ability to support similarity searches.

ZM [31] uses the Z-order space filling curve [32] to establish an ordering relationship among all points, then invokes a one-dimensional learned index RMI to support range queries. RSMI [33] introduces a recursive partitioning strategy that divides the original space, and groups objects based on model predictions. LISA [34] designs a grid-based learned index to support data updates. Flood [35] uses the query workload to learn the optimal combination of index dimensions and

Table 1. Frequent Notations

Notation	Description
$O, \|O\|$	An object set and its cardinality in a metric space
$d(\cdot,\cdot)$	A distance metric
$e(c, r)$	An index node with center c and radius r
M	The capacity of node
$\|P\|, p$	The number of pivots and a pivot
ϵ	The predefined error threshold
$MkNNQ(k)$	A k nearest neighbor query
$MRQ(q, r_q)$	A range query with query point q and radius r_q
err	The predict error of learning model

partition numbers. Qd-Tree [36] can be regarded as a workload-aware learned variant inspired by Kd-Tree [37]. [38–40] use reinforcement learning to build traditional spatial indices.

Discussion. In summary, the pruning power of single-pivot-based methods is limited by relying on a single pivot per node (often the node center) for partitioning and pruning. Multi-pivot-based methods use multiple fixed pivots for enhancing the pruning ability of indices, but are inefficient with data updates. Hybrid-based indices still fail to overcome core limitations of either method. Learned indices for metric spaces cannot always reflect distance relationships among objects in metric spaces. An efficient index that could use as few as pivots for effective pruning, while supporting pivot set self-refinement as the dataset evolves, is highly desired.

2.2 M-Tree

The M-Tree structure organizes objects into potentially overlapping metric regions, forming a hierarchical, balanced tree. Each node e in the tree is represented as a tuple $\boldsymbol{e(c, r, DL)}$, where c denotes the center of the region, r specifies the radius, and DL stores additional information depending on the node type.

For a leaf node, DL maintains information about the objects within the node e: $DL = \{(o_u, d(o_u, e.c)) \mid u = 1, 2, \dots\}$, where $o_u \in e$ represents the u-th object within e, the distance from o_u to the center $e.c$ is $d(o_u, e.c)$. This allows the algorithm to know each object's position relative to the center of the region without recomputing distances at query time.

For an inner node, DL maintains information about its child nodes: $DL = \{(child_u.c, child_u.r, d(child_u.c, e.c) - child_u.r) \mid u = 1, 2, \dots\}$, with $child_u.c$ and $child_u.r$ being the child's center and radius, $d(child_u.c, e.c) - child_u.r$ being the minimum distance from the child region to the parent center. This information is crucial for efficiently pruning irrelevant subtrees during similarity search.

When supporting similarity search based on M-Tree, the algorithm leverages precomputed distances (stored within node entries) and the triangular inequality principle to dynamically prune non-relevant search paths during query processing. For a range query $MRQ(q, r_q)$ with q and r_q being the query point and search radius, a subtree rooted at the pivot p can be safely pruned if the inequality $|d(q, p) - d(o, p)| > r_q$ holds for all objects o in its coverage region. Note, maintaining precomputed distances between pivot and each of its corresponding objects o, i.e., $d(o, p)$, is important, which enables us to prune irrelevant objects based on distance comparisons between query centers, objects, and pivots, rather than expensive runtime distance calculations. Fig. 1(b) shows the M-Tree structure using a two-dimensional Euclidean space as an example. It can be seen that the M-Tree forms a balanced hierarchical structure, with the actual objects stored in the leaf nodes. The DL corresponding to each node is shown in the table of Fig. 1(b).

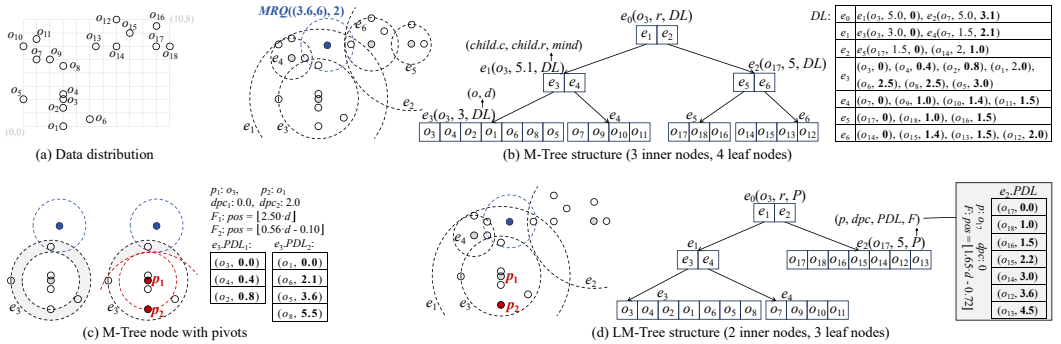


Fig. 1. The LM-Tree Index Structure.

3 LM-Tree

In this section, we propose a novel index named LM-Tree to organize objects in metric spaces. First, we introduce the LM-Tree structure. Next, we use one type of typical query named *metric range query* (*MRQ* for short), as an example to show how we apply LM-Tree to support similarity search in metric spaces.

3.1 The LM-Tree Structure

In this section, we first describe three main inspirations from M-Tree. Based on these, we further introduce the LM-Tree structure.

3.1.1 Inspirations from M-Tree. Fig. 1(a) shows the set of 18 objects in a 2-dimensional space. We form an M-Tree based on these objects, which has 3 inner nodes and 4 leaf nodes, as shown in Fig. 1(b). When we submit a range query $MRQ(q, r_q)$ with the query center q being $(3.6, 6)$ and the radius r_q being 2 on this M-Tree, 6 nodes and 10 objects are accessed, and 21 distance comparisons are performed. Among them, 7 objects are out of the search scope. A natural question is: whether an index exists with a smaller scale and powerful pruning ability for reducing the number of distance comparisons, as well as the number of objects/nodes we should access? The answer is yes, via properly modifying the M-Tree structure.

Firstly, we could use linear models to accelerate searches when comparing distance values. For any node e within an M-Tree, if we could form a proper linear model to maintain the distance relationship between the node center and child nodes of e (or objects maintained by e if e is a leaf node), multiple distance value scanning could be avoided, following the logic of searching on 1-dimensional learned indices [13, 14, 26].

Consider the example in Fig. 1(b). The center of the node e_3 is o_3, we have $d(o_3, o_3) = 0$, $d(o_4, o_3) = 0.4$, $d(o_2, o_3) = 0.8$, $d(o_1, o_3) = 2.0$, $d(o_6, o_3) = 2.5$, $d(o_8, o_3) = 2.5$, and $d(o_5, o_3) = 3.0$. Accordingly, we can use a linear model to fit distance values between the node center o_3 and other objects within e_3, i.e., $F : pos = \lfloor 1.84 \cdot d(o_i, o_3) + 0.14 \rfloor$. When the range query $MRQ((3.6, 6), 2)$ accesses e_3, M-Tree requires 7 distance comparisons (0, 0.4, 0.8, 2.0, 2.5, 2.5, and 3.0); In contrast, as will be detailed in Section 3.2, the use of F requires only 4 distance comparisons (2.0, 2.5, 2.5, and 3.0).

Secondly, although linear models can accelerate search by reducing the number of distance comparisons, they cannot reduce the number of nodes/objects that need to be accessed. Furthermore, we cannot always find a proper linear model to fit distance values across various data distributions. We address these problems by carefully selecting extra pivots for each node e, i.e., each pivot p manages a subset of child nodes within e, if e is an inner node, or a subset of objects within e if e is

a leaf node. We then individually form a linear model to maintain the distance relationship between each pivot p and its corresponding child nodes (or objects if e is a leaf node). Via introducing extra pivots, we can obtain more powerful pruning ability, as well as high quality linear models.

Back to the example in Fig. 1(c). We select objects o_3 and o_1 as pivots of node e_3, denoted as p_1 and p_2. The pivot p_1 is responsible for objects o_3, o_4, and o_2, whose distances to p_1 are 0, 0.4, and 0.8, respectively. These distances can be well approximated by a linear model: $pos = \lfloor 2.50 \cdot d \rfloor$. Similarly, the pivot p_2 is responsible for objects o_1, o_6, o_5, and o_8, with corresponding distances of 0, 2.1, 3.6, and 4.5. They can be fitted by the linear model: $pos = \lfloor 0.65 \cdot d - 0.15 \rfloor$. As will be discussed later, when performing the range query $MRQ((3.6, 6), 2)$ on e_3, the M-Tree optimized with only linear models accesses 4 objects o_1, o_6, o_8, and o_5, and performs 4 distance comparisons (values 2.0, 2.5, 2.5, and 3.0). Via introducing extra pivots and linear models, we only access 2 objects o_1, o_8, while perform 3 distance comparisons (values 0, 2.0, 5.5).

Thirdly, we could further use pivots and linear models to reduce the number of nodes within the index. As shown in Fig. 1(d), considering the node e_2, distances between o_{17} and $\{o_{17}, o_{18}, o_{16}, o_{15}, o_{14}, o_{12}, o_{13}\}$ are also approximately uniformly distributed. Therefore, we do not need to maintain $\{e_5, e_6\}$, i.e., two child nodes of e_2. Alternatively, we could use o_{17} as a pivot, use the linear model $pos = \lfloor 1.65 \cdot d - 0.72 \rfloor$ to maintain the distance relationship between o_{17} and these 7 objects. Fig. 1(d) shows the modified index, which only maintains 2 inner nodes and 3 leaf nodes.

3.1.2 LM-Tree structure. In this section, we propose a novel index named LM-Tree. It is a hybrid learned index structure combining pivots and learning models with M-Tree. Each node e in a LM-Tree is expressed by the tuple (c, r, P). Here, c and r have the same meaning as M-Tree (see Section 2.2 for details). P is a group of tuples defined as $P = \{p_v, dpc_v, PDL_v, F_v | v = 1, 2, \ldots |P|\}$, p_v denotes the v-th selected pivot in e, and $|P|$ denotes the number of pivots in e.

Given the v-th pivot p_v within e, if e is a leaf node, dpc_v denotes the distance from p_v to the node center $e.c$. PDL_v maintains a subset of objects that correspond to p_v, i.e., $PDL_v = \{o_u, d(o_u, p_v) | u = 1, 2, \ldots\}$. Here, o_u is the u-th object within PDL_v, $d(o_u, p_v)$ refers to the distance from o_u to p_v. We ensure that: (i) dpc_v is no more than the distance from any object in PDL_v to the node center $e.c$; and (ii) dpc_{v+1} is larger than the distance from any object in PDL_v to $e.c$, i.e., $\forall o_u \in PDL_v, dpc_v \leq d(o_u, e.c) < dpc_{v+1}$.

If e is an inner node, dpc_v denotes the minimum distance from the node center $e.c$ to the child node of e that contains p_v. PDL_v maintains a subset of child nodes that are corresponding to p_v, i.e., $PDL_v = \{child_u, d(child_u.c, p_v) - child_u.r | u = 1, 2, \ldots\}$. Here, $child_u$ is the u-th child node within PDL_v, and $d(child_u.c, p_v) - child_u.r$ refers to the minimum distance from $child_u$ to p_v. Similar to leaf nodes, we should ensure that dpc_v is no more than the minimum distance from any child node in PDL_v to the node center $e.c$, i.e., $\forall child_u \in PDL_v, dpc_v \leq d(child_u.c, e.c) - child_u.r$.

Regardless of inner nodes or leaf nodes, F_v is a linear model that maps each element in PDL_v to its corresponding storage position pos based on its distance to the pivot p_v, i.e., $F_v : pos = \lfloor a \cdot d + b \rfloor, d \in PDL_v$. Parameters a and b represent the slope and intercept of the linear model F_v, which are calculated in the same way as described in [14]. Due to space limitations, details are omitted.

Consider the example in Fig. 1(c). Given the leaf node e_3, objects o_3 and o_1 are selected as two pivots, denoted as p_1 and p_2, respectively. For pivot p_1, its distance to the node center is $dpc_1 = d(p_1, e_3.c) = 0$, and is associated with $PDL_1 = \{(o_3, d(o_3, p_1) = 0), (o_4, d(o_4, p_1) = 0.4), (o_2, d(o_2, p_1) = 0.8)\}$, which is fitted by the linear model $F_1 : pos = \lfloor 2.50 \cdot d \rfloor$. Similarly, pivot p_2 corresponds to $dpc_2 = d(p_2, e_3.c) = 2.0$ and $PDL_2 = \{(o_1, d(o_1, p_2) = 0), (o_6, d(o_6, p_2) = 2.1), (o_5, d(o_5, p_2) = 3.6), (o_8, d(o_8, p_2) = 4.5)\}$, with the fitted model $F_2 : pos = \lfloor 0.65 \cdot d - 0.15 \rfloor$.

3.2 Search on LM-Tree

Similarity search in metric spaces mainly includes metric range queries and k nearest neighbor queries [1–4]. The k nearest neighbor query can be converted into multiple range queries by expanding/shrinking the query radius r_q. Therefore, we focus on range queries to demonstrate how LM-Tree handles similarity search.

As shown in Algorithm 1, the algorithm accesses LM-Tree in a depth-first manner, beginning with the root node e (line 1). The search process employs *inner node access* logic to evaluate each child node e_i against the query region through distance relationship comparison: nodes completely contained within the search scope contribute all their objects to result set R, while disjoint nodes are pruned immediately. For partially overlapping nodes, the algorithm recursively applies *inner node access* (lines 2-13) if e_i remains an inner node, or switches to *leaf node access* (lines 14-21) when reaching a leaf node. This process continues until no nodes remain for access, at which point the algorithm terminates. In the following, we will explain *leaf node access* and *inner node access*.

– *Leaf node access*: We can fully leverage both the pivots and linear models maintained by each leaf node e to accelerate leaf node access. During pivot-based pruning, we access each pivot $p_v \in e.P$, if $dpc_{v+1} < d(q, e.c) - r_q$ or $dpc_v > d(q, e.c) + r_q$, all objects maintained in PDL_v can be safely pruned as stated in Theorem 1, where we do not need to further access objects within PDL_v.

Theorem 1. *Given a pivot p_v in a leaf node e, and a range query $MRQ(q, r_q)$, if either $dpc_{v+1} < d(q, e.c) - r_q$ or $dpc_v > d(q, e.c) + r_q$ holds, p_v and the objects within its corresponding PDL_v are out of the query range.*

Proof. As demonstrated in Section 3.1.2, $\forall o_u \in PDL_v, dpc_v \le d(o_u, e.c) < dpc_{v+1}$ is guaranteed. If $dpc_{v+1} < d(q, e.c) - r_q$ or $dpc_v > d(q, e.c) + r_q$, it follows that $\forall o_u \in PDL_v, d(o_u, e.c) < d(q, e.c) - r_q$ or $\forall o_u \in PDL_v, d(o_u, e.c) > d(q, e.c) + r_q$, which implies that all objects in PDL_v are out of the query range. □

If objects within PDL_v cannot be pruned via p_v, we further use the linear model F_v for pruning. As stated in PROPERTY 1, an object o_u can be pruned safely if it satisfies $d(o_u, p_v) < d(q, p_v) - r_q$ or $d(o_u, p_v) > d(q, p_v) + r_q$. Therefore, we use F_v to locate the position pos_l and pos_r of $d(q, p_v) - r_q$ and $d(q, p_v) + r_q$ in PDL_v, then objects outside the range $[pos_l, pos_r]$ can be safely pruned (See PROPERTY 1 below). For each remaining objects o_u within the range $[pos_l, pos_r]$, we compute the distance between q and o_u. If $d(o_u, q) \le r$, it is regarded as a query result object. Otherwise, it is pruned.

Property 1. *Given a pivot p_v, a query object q with a search radius r_q, let $SR(q) = [d(q, p_v) - r_q, d(q, p_v) + r_q]$ be a mapped search region. If the distance value $d(o, p_v)$ is located outside $SR(q)$, the object o can be pruned safely.*

Consider a range query $MRQ(q(3.6, 6), r_q = 2)$ (blue circle) in Fig. 1(c). When a leaf node e_3 is reached, we first compute the distance $d(q, e_3.c) = 2.04$, which less than $e_3.c + r_q = 5$, indicating that e_3 intersects the query and is should be accessed. Then, we compute the minimum and maximum distances from q to $e_3.c$, which are $d(q, e_3.c) - r_q = 4.04$ and $d(q, e_3.c) + r_q = 6.04$. As $d(q, e_3.c) + r_q = 6.04 > r_q$, we should further access e_3 via considering its pivots. The distances from pivots p_1 and p_2 to the center $e_3.c$ are $dpc_1 = 0$ and $dpc_2 = 2.0$, respectively. According to Theorem 1, as $dpc_{1+1} = 2.0 < 4.04$, objects in PDL_1 can be pruned safely. As a contrast, as $dpc_2 = 2.0 < 6.04$, objects in PDL_2 cannot be pruned. We should calculate the distance range between the query and p_2 (i.e., $d(q, p_2) - r_q = 5.02$ and $d(q, p_2) + r_q = 9.02$), further use F_2 to predict the storage positions pos_l and pos_r of 5.02 and 9.02 in PDL_2, both of which are 3. According to PROPERTY 1, objects outside the range $PDL_2[pos_l, pos_r]$ can be pruned safely. Thus, only the object in $PDL_2[3]$ (i.e., o_8) needs to be verified, others could be pruned.

Algorithm 1: Range Query Algorithm

Input: LM-Tree, a query object q, and query radius r_q
Output: Query result set R

```
1  e ← root node of LM-Tree; R ← ∅;
2  if e is an inner node then
3      for each {p_v, dpc_v, PDL_v, F_v} ∈ e.P do
4          if dpc_v ≥ d(q, e.c) + r_q then // Theorem 2
5              continue;
6          else // PROPERTY 2
7              for each child_u ∈ PDL_v[0, pos] do
8                  if child_u is contained in search scope then
9                      ∀o_j ∈ child_u, R ← R ∪ o_j;
10                 else if child_u is outside the search scope then
11                     continue;
12                 else
13                     e ← child_u and back to line 2;
14 else
15     for each {p_v, dpc_v, PDL_v, F_v} ∈ e.P do
16         if dpc_v > d(q, e.c) + r_q or dpc_{v+1} < d(q, e.c) − r_q then
17             continue; // Theorem 1
18         else // PROPERTY 1
19             for each o_u ∈ PDL_v[pos_l, pos_r] do
20                 if d(q, o_u) ≤ r_q then
21                     R ← R ∪ o_u;
22 return R;
```

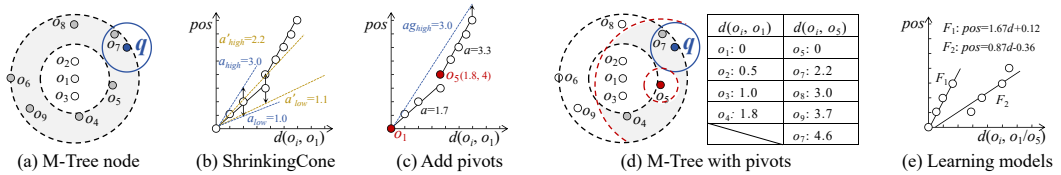
– *Inner node access:* The logic is similar to leaf node access. Three main differences are as follows. The pivot-based pruning is based on Theorem 2, while the linear model-based pruning is based on PROPERTY 2 (See below). Besides, if the child node $child_u$ of an inner node e cannot be safely pruned, we further access $child_u$.

THEOREM 2. *Given a pivot p_v in an inner node e, and a range query $MRQ(q, r_q)$, if $dpc_v > d(q, e.c) + r_q$, p_v and all child nodes within its corresponding PDL_v are out of the query range.*

PROOF. As stated in Section 3.1.2, $\forall child_u \in PDL_v, dpc_v < d(child_u.c, e.c) - child_u.r$ is ensured. Thus, if $dpc_v > d(q, e.c) + r_q$, $\forall child_u \in PDL_v$, we have, $d(child_u.c, e.c) - child_u.r > d(q, e.c) + r_q$. It implies all child nodes in PDL_v are outside the query range. □

PROPERTY 2. *Given a pivot p_v in an inner node e, and a range query $MRQ(q, r_q)$, if $d(e.child_u.c, p_v) - e.child_u.r > d(q, p_v) + r_q$, $e.child_u$ can be safely pruned.*

Discussion. LM-Tree is efficient and flexible. For efficiency, LM-Tree utilizes pivots to reduce the number of nodes and objects that need to be scanned; uses linear models to improve query efficiency by quickly locating nodes or objects need to be scanned. Additionally, as we could use fewer nodes with larger size to index objects in metric spaces (the third inspiration in Section 3.1.1), the searching algorithm only need to access a small number of nodes. For flexibility, we could adaptively select



(a) M-Tree node (b) ShrinkingCone (c) Add pivots (d) M-Tree with pivots (e) Learning models

Fig. 2. Pivot Selection and Learning Model Optimization.

pivots based on distance relationships among child nodes of inner nodes (also objects within leaf nodes). In particular, if P contains only a single pivot, it must be the node center, indicating that distances from child nodes or objects within e to its center are approximately uniformly distributed.

4 The LM-Tree Construction and Maintenance

Recalling Section 3, the key behind LM-Tree is to adaptively select pivots to enhance pruning power, form linear models to speed up query processing, and use pivots and linear models to reduce node amount. In the following, we first explain the LM-Tree construction. Next, we describe how to maintain LM-Tree as data updates.

4.1 The LM-Tree Construction Algorithms

Let $O \subseteq \mathbb{U}$ be the object set. The LM-Tree construction starts from building a traditional index M-Tree T based on objects within O. We use it as the initial LM-Tree. Then we propose a two-step method to form LM-Tree. In the first step, we propose a segment-based local construction algorithm that selects proper pivots and forms linear models for each node within T (detailed in Section 4.1.1). In the second step, we introduce a node merge-based global construction algorithm that refines the pivot selection and reduces the number of nodes, pivots, as well as linear models LM-Tree should maintain (detailed in Section 4.1.2).

4.1.1 The Segment-based Local Construction Algorithm. Our solution is based on the running example in Fig. 2(a)-(c). We fit distances from objects within the node e to its center using piecewise linear functions; objects o_1, o_2, o_3, and o_4 form a segment with a slope of 1.7, objects o_5 - o_9 form another segment with a slope of 3.3. We find that the node center in an M-Tree is effective for pruning objects located in segments with low slopes, but performs poorly in segments with high slopes as these objects have similar distances to the node center. Therefore, for objects located at high-slope segments, it is preferable to select appropriate pivots based on the local distribution of these objects, reorganize them based on their distances to the corresponding pivots, thereby improving pruning ability. In contrast, for other objects located in low-slope segments, since the node center already distinguishes objects effectively, they share a common pivot, i.e., the node center.

Above all, the process of selecting pivots and using learning models for optimization consists of the following two steps. For ease of presentation, we use leaf nodes as examples to explain the pivot selection method, but our proposed technique could also support pivot selection and learning model construction when handling inner nodes via minor modifications.

Step-1: Necessity Evaluation of Adding Pivots. Given a leaf node e within the initial LM-Tree T, we first compute distances between the node center $e.c$ and objects within e. Then, we sort these objects according to their distances to $e.c$.

Next, our goal is to effectively partition the objects within the node based on the sorted distance values, such that objects within each partition exhibit consistent patterns in their distances to the center; for example, objects with similar distances are grouped into the same partition. In

particular, we adopt the *ShrinkingCone* segmentation algorithm discussed in FITing-Tree [28] to form one (or a group of) segment(s) based on the sorted distances. Each segment corresponds to a disjoint subset of objects within e. It is represented by a linear function f with slope a and a predefined prediction error $\epsilon \geq 0$. These segments enable a distance-aware partitioning strategy, ensuring that the difference between adjacent distance values within each segment remains within a bounded range. As shown in Theorem 3, let d_i and d_{i+1} denote the distances from o_i and o_{i+1} to $e.c$, respectively. If they are within the same segment, $d_{i+1} - d_i$ and $d_i - d_{i-1}$ both fall within the range $[\max\{0, (1-2\epsilon)/a\}, (1+2\epsilon)/a]$.

Theorem 3. *Given any two adjacent objects o_i and o_{i+1} within the same segment, where d_i and d_{i+1} represent their respective distances to the node center $e.c$, $d_{i+1} - d_i$ remains within the range* $[\max\{0, (1-2\epsilon)/a\}, (1+2\epsilon)/a]$.

Proof. Given a segment with linear mapping function $f(d) = a \cdot d + b$ and error threshold ϵ, the storage positions pos_i and pos_{i+1} of objects o_i and o_{i+1} must satisfy the constraints $pos_i - \epsilon \leq f(d_i) \leq pos_i + \epsilon$ and $pos_{i+1} - \epsilon \leq f(d_{i+1}) \leq pos_{i+1} + \epsilon$. By the monotonicity of f, $pos_i - \epsilon \leq f(d_i) \leq f(d_{i+1}) \leq pos_{i+1} + \epsilon$ is satisfied, which implies $\frac{1-2\epsilon}{a} \leq d_{i+1} - d_i \leq \frac{1+2\epsilon}{a}$. Since distances from objects in e to its center $e.c$ are monotonically increasing ($d_{i+1} - d_i \geq 0$ for all $i \geq 0$), we have $\max\left\{0, \frac{1-2\epsilon}{a}\right\} \leq d_{i+1} - d_i \leq \frac{1+2\epsilon}{a}$. □

After constructing these segments, we further leverage them to evaluate whether node e needs to add one or more additional pivot(s) as follows. Let ag_{high} be $(pos_{\max} - pos_{\min} + \epsilon)/(d_{\max} - d_{\min})$, reflecting the average distribution density of distances from objects within e to the node center, f be the linear function of a segment formed by a set of objects under *ShrinkingCone*. If the slope of f is greater than ag_{high}, it means that many objects have similar distances to the node center $e.c$ in this segment, suggesting that we should add a pivot for these objects. Here, $d_{\min}$ and $d_{\max}$ represent the minimum and maximum distances from objects in e to its center. Objects are mapped to an array using piecewise linear functions based on these distances, where $pos_{\min}$ and $pos_{\max}$ indicate the first and last valid object positions in the array, respectively.

Consider the example in Fig. 2(a)-(b). Given a leaf node with 8 objects, we compute and sort them based on the distances from these 8 objects to node center o_1, which are 0, 0.7, 1.2, 4.8, 5.5, 6.1, 6.6, and 7.0, respectively. Suppose that the predefined error threshold ϵ is 1, we apply the *ShrinkingCone* segmentation algorithm to form two segments based objects within e. That is, objects o_1, o_2, o_3, and o_4 form one segment with a slope of 1.7; objects o_5 - o_9 form another segment with a slope of 3.3, which is larger than $ag_{high} = 3.0$,i.e., $ag_{high} = (8 - 0 + \epsilon)/(3 - 0) = 3.0$, as shown in Fig. 2(c). Thus, we should add a pivot for objects o_5 - o_9.

Step-2: Pivot Selection and Learning Model Optimization. The first pivot is the node center of e. We form one model for all objects located in segment(s) with slope(s) no larger than ag_{high}, which fits distance values between the node center and these kinds of objects. The slope and intercept of the learning model are computed using the least squares method [14]. We skip the details to save space.

For the other objects, given the set of objects located in the same segment with a slope larger than ag_{high}, we always choose the left endpoint of each segment as a pivot. The intuition behind it is outliers or boundary points can better partition objects, resulting in greater distance differences for most objects [4, 6, 10]. As the left endpoint lies at the boundary of these objects, it can serve as a pivot for managing objects in the corresponding segment.

Property 3. *The segments formed in Step-1 are all maximal segments, that is, when adding another object to these segments would violate the predefined error bound ϵ, which is a positive integer. Thus, the minimal number of objects covered by a maximal segment is $\epsilon + 1$ [28].*

Since we add a single pivot for each segment with a slope greater than ag_{high}, the number of pivots in node e is at most equal to the number of such segments. Moreover, PROPERTY 3 implies that each segment formed in Step-1 contains at least $\epsilon + 1$ objects. Therefore, to cover all M objects in node e, at least $\lceil M/(\epsilon+1) \rceil$ segments are needed. Consequently, the maximum number of pivot points in node e is bounded by $\lceil M/(\epsilon+1) \rceil$.

Consider the example in Fig. 2(e). We use the node center o_1 as the first pivot. It corresponds to objects located in the segment with slope smaller than $ag_{high} = 3, 0$. These objects can be fitted by the linear model $F_1 : pos = \lfloor 1.67 \cdot d + 0.12 \rfloor$. We use the object o_5 as the second pivot. Here, objects o_5-o_9 are located in the same segment with slope larger than $ag_{high} = 3, 0$. Their distances to the corresponding pivot o_5 are 0, 2.2, 3.0, 3.7, and 4.6, respectively, which can be fitted by another linear model $F_2 : pos = \lfloor 0.87 \cdot d - 0.36 \rfloor$.

Cost Analysis. Let us assume that each leaf node contains at most M objects. The cost of computing distances from all these objects to the node center is $O(M \cdot Cost(o))$, and the cost of sorting is $O(M \cdot \log M)$. Here, $Cost(o)$ denotes the cost of accessing a single object. The *ShrinkingCone* algorithm spends $O(M)$ cost for segmentation. Additionally, we spend $O(M \cdot Cost(o) + M \cdot \log M)$ cost in selecting pivots and forming learning models. In total, the cost spent by a leaf node is $O(M \log M + M \cdot Cost(o))$. Similarly, the cost spent by an inner node is also $O(M \log M + M \cdot Cost(o))$. As the number of nodes within an M-Tree is $O(|O|/M)$, the running cost of the segment-based local construction is $O(|O|/M \cdot (M \cdot \log M + M \cdot Cost(o)) = O(|O| \cdot (\log M + Cost(o))$. Here, $|O|$ is the number of objects within the dataset O.

4.1.2 The Node Merge-based LM-tree Refinement. Let T be the temporary LM-Tree formed through segment-based local construction. Following the third inspiration in Section 3.1.1, we can refine the structure by evaluating whether selected pivots (and their corresponding models) in each node e can efficiently handle elements within its neighboring nodes. If so, we merge these nodes while retaining only few pivots and models. This optimization reduces the index's node scale while maintaining strong pruning power.

Specifically, we employ a bottom-up merging approach that processes nodes layer by layer. During this hierarchical process, a node may have both inner and leaf child nodes. Since they store different information, we only merge leaf nodes with leaf nodes, merge inner nodes with inner nodes, avoiding cross-type merges. Due to space constraints, we only illustrate algorithm about merging child nodes of an inner node. Algorithm 2 shows the pseudo-code.

Let e be an inner node with child nodes $E = \{e_1, e_2, \ldots, e_M\}$. To evaluate the potential of merging each pair of nodes within E, we first construct a *Gain Table*, where each element T_{ij} within the *Gain Table* quantifies the gain when merging nodes e_i and e_j into a new composite node e_{ij}. The gain value is computed as:

$$T_{ij} = e_i.|P| + e_j.|P| - e_{ij}.|P| \tag{1}$$

The intuition behind using Eq. 1 for evaluation is: when e_i and e_j merge into e_{ij}, their combined pivot count $e_{ij}.|P|$ typically reduces compared to the original pivots ($e_i.|P| + e_j.|P|$), if $e_i.|P| + e_j.|P| - e_{ij}.|P| > 0$. This reduction decreases the number of pivots, as well as the linear models we should maintain. Here, $e_i.|P|$ refers to the number of pivots within e_i. e_{ij} is a node formed by merging e_j into e_i, its center is $e_i.c$ and radius is $d(e_i.c, e_j.c) + e_j.r$, which can bound all elements within e_i and e_j. Note, e_{ij} and e_{ji} ($i \neq j$) are not the same: the former takes e_i's center with radius $d(e_i.c, e_j.c) + e_j.r$, while the latter takes e_j's center with radius $d(e_i.c, e_j.c) + e_i.r$. Moreover, we do not form e_{ij} when constructing the *Gain Table*, where we only evaluate $e_{ij}.|P|$ in this step, without computing the distances from objects to pivots or fitting linear models.

Algorithm 2: Node Merge Algorithm

Input: An inner node e with pivots and learning models
Output: e after merging

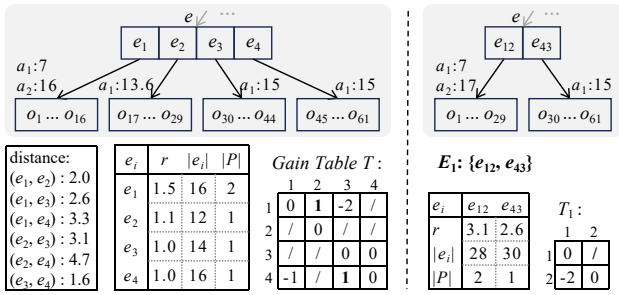
1 Init $Gain[][], E \leftarrow$ all child nodes e_i of e;
2 **for** *each node* $e_i \in E$ **do**
3 **for** *each node* $e_j \in E - e_i$ **do**
4 **if** $\sum e_i.|PDL_v| < \sum e_j.|PDL_v|$ **then**
5 $Gain[i][j] \leftarrow /$;
6 **else**
7 **if** $\frac{e_i.F(d(e_j.c,e_i.c)-e_j.r)}{d(e_j.c,e_i.c)-e_j.r}$ *outside slope range of* $e_i.F$ **then**
8 $Gain[i][j] \leftarrow /$;
9 **else**
10 $pos \leftarrow e_i.F(d(e_j.c, e_i.c) - e_j.r)$;
11 $e_{ij}.|P| \leftarrow$ *Step-I,II in Section 4.1*(e_i, e_j, pos);
12 $Gain[i][j] \leftarrow e_i.|P| + e_j.|P| - e_{ij}.|P|$;
13 **while** *no merge case* **do**
14 $e_i, e_j \leftarrow$ `getMaxGain` $(Gain[][])$;
15 $e_{ij} \leftarrow$ `mergeNode` (e_i, e_j);
16 $E_1 \leftarrow E_1 \cup e_{ij}$;
17 $E_1 \leftarrow E_1 \cup$ unmerged nodes;
18 Repeat from line 2 until E_i has no mergeable nodes;
19 return e;

A straightforward approach is to apply the algorithm in Section 4.1.1 to compute each $e_{ij}.|P|$. However, this involves accessing all objects within both e_i and e_j, which incurs a significant running cost. To address this issue, we propose three optimization strategies.

Firstly, we use Theorem 4 to avoid unnecessary evaluation via considering minimum distances among nodes. Specifically, let us assume that both nodes e_i and e_j all contain one pivot (i.e., the node center). $e_i.PDL$ stores the minimum distances from e_i's all child nodes to $e_i.c$, where all distance values in $e_i.PDL$ are fitted by a single linear function $e_i.F$. We first compute the minimum distance from e_j to e_i,i.e., $d(e_j.c, e_i.c) - e_j.r$, and then use the function F to compute $F(d(e_j.c, e_i.c) - e_j.r)$. Accordingly, points $(0, 0)$ and $(d(e_j.c, e_i.c) - e_j.r, F(d(e_j.c, e_i.c) - e_j.r))$ can form a linear function with slope $a' = \frac{F(d(e_j.c,e_i.c)-e_j.r)-0}{(d(e_j.c,e_i.c)-e_j.r)-0}$. According to Theorem 4, if this slope a' falls outside the slope range $[F.a - \epsilon/(dp_{\max} - 0), F.a + \epsilon/(dp_{\max} - 0)]$ defined by F, e_j cannot be merged with e_i, and we need not to calculate T_{ij} (lines 7-8). Here, $dp_{\max}$ denotes the maximum distance value in $e_i.PDL$. Note, the idea of Theorem 4 can be naturally extended to the case where e_i has multiple pivots; however, we omit it due to space constraints.

Theorem 4. *Assuming e_i and e_j only have one pivot, e_j cannot be merged with e_i if $\frac{e_i.F(d(e_j.c,e_i.c)-e_j.r)-0}{(d(e_j.c,e_i.c)-e_j.r)-0}$ falls outside range $[F.a - \frac{\epsilon}{dp_{\max}}, F.a + \frac{\epsilon}{dp_{\max}}]$, with F being the linear model F of e_i.*

Proof. According to the *ShrinkingCone* strategy, if the slope $a' = \frac{F(d(e_j.c,e_i.c)-e_j.r)-0}{(d(e_j.c,e_i.c)-e_j.r)-0}$ falls outside the slope range $[F.a - \frac{\epsilon}{dp_{\max}}, F.a + \frac{\epsilon}{dp_{\max}}]$, we should form at least additional one segment (also pivot). Since e_j only has a single pivot, merging e_i and e_j in this case would result in at least 2



(a) intialization (b) e after mering

Fig. 3. An example of node merging ($\epsilon = 1$, / means no merge cost calculation needed).

pivots. Due to we only perform the merge when $e_i.|P| + e_j.|P| - e_{ij}.|P| > 0$, e_j could not merge with e_i. □

As shown in Fig. 3, the slope range of e_4 is computed as $[e_4.a - \frac{\epsilon}{dp_{\max}}, e_4.a + \frac{\epsilon}{dp_{\max}}] = [14.0, 16.0]$, where $dp_{\max} = e_4.r$. The minimum distance from the node e_2 to e_4 is $d(e_2.c, e_4.c) - e_2.r = 3.6$, and the corresponding predicted position is $F(3.6) = 16$. This gives a slope of $a' = 16/3.6 = 4.4$, which falls outside the range $[14.0, 16.0]$. Therefore, e_2 cannot be merged with e_4 according to Theorem 4.

Secondly, if Theorem 4 fails to avoid computing $e_{ij}.|P|$, we leverage the linear model(s) within e_i to optimize the computation. Here, we also assume that e_i only use its center as pivot. However, this approach also could be applied to the case where $e_i.|P| > 1$ and $e_j.|P| > 1$, we omit its description due to space constraints. Specifically, due the the natural property of segmentation algorithm discussed in Section 4.1.1, each segment is maximal and inserting a new object into $PDL[pos]$ will not affect the previous partition result $PDL[0, pos - 1]$. Thus, we need not to re-segment all elements in e_i and e_j. Instead, we only apply the algorithm discussed in Section 4.1.1 to process elements within e_j, as well as those in $e_i.PDL[pos, M]$. Here, pos refers to the position in $e_i.PDL$ of $d(e_j.c, e_i.c) - e_j.r)$, i.e., $pos = e_i.F(d(e_j.c, e_i.c) - e_j.r)$ (lines 9-12).

Thirdly, the running cost for determining $e_{ij}.|P|$ depends on the size of e_j, while the running cost for determining $e_{ji}.|P|$ depends on the size of e_i. To balance merge quality (measured by mergeable node count) with computational efficiency, we selectively evaluate either $e_{ij}.|P|$ or $e_{ji}.|P|$ based on their size.i.e., we compute $e_{ij}.|P|$ if e_i has more children, otherwise $e_{ji}.|P|$. (lines 4–5).

Above all, for each pair e_i and e_j within E, we use the above three optimization strategies to avoid or speed up the $e_{ji}.|P|$ calculation. After the *Gain Table* is formed, we iteratively select the pair of nodes with the highest merge gain, and merge them into one node if these two nodes have not merged with other nodes. This process continues until no more nodes can merge with the others. At that moment, we form a new node set E_1, which contains newly merged nodes, along with nodes that are not involved in any merge. We then repeat the *Gain Table* initialization and node merging based on E_1. From then on, we repeat the above operations until the merge gain between any pair of nodes in E_i all equals 0. At that moment, E_i is considered as the final node set (lines 13-18).

Returning to the example in Fig. 3, after constructing the *Gain Table* based on E, we select e_1 and e_2, as well as e_4 and e_3, for merging because both pairs yield a positive gain. After merging, no node could be further merged, we obtain $E_1 = \{e_{12}, e_{43}\}$. We then construct a new *Gain Table* based on E_1 as shown in Fig. 3(c), and since no further merge is possible, E_1 is regarded the final node set, i.e., node e contains the following two child nodes: e_{12} and e_{43}.

Algorithm 3: Incremental Maintenance Algorithm

Input: LM-Tree and an object o_{in}/o_{de} to be inserted/deleted
Output: Updated LM-Tree

1 leaf node $e_L \leftarrow$ `findLeafNode` (o_{in}/o_{de});
2 $p_v, F_v, PDL_v \leftarrow$ `findSegment` $(o_{in}/o_{de}, e_L)$;
3 $pos \leftarrow F_v(d(o_{in}/o_{de}, p_v))$;
4 **Function** `handleLeafInsert`(o_{in})**:**
5 insert o_{in} into $PDL_v[pos]$;
6 **if** $|PDL_v| \geq 2 \cdot$ *not empty* $|PDL_v|$ **then**
7 $e \leftarrow$ `ShrinkingCone` (PDL_v);
8 **if** $e_i.|P| \geq 2 \cdot$ *initial* $e_i.|P|$ **then**
9 `splitNode` (e, o_{in});
10 **if** *#nodes in* e_L*'parent* $\geq 2 \cdot$ *initial count* **then**
11 `mergeNode` ($e'_L s$ parent node);
12 **Function** `handleLeafDelete`(o_{de})**:**
13 delete o_{de} from $PDL_v[pos]$;
14 **if** $|PDL_v| \leq 1/2 \cdot$ *initial* $|PDL_v|$ **then**
15 $e \leftarrow$ `ShrinkingCone` (PDL_v, B_v);
16 return LM-Tree;

Cost Analysis. Let us assume that a node e within the initial LM-Tree contains at most M child nodes or objects, leaf nodes are located at level 0, and the root node is located at the highest level. Thus, an inner node e at level i ($i \geq 1$) contains at most $O(M^i)$ objects. For a single inner node e at level i, initializing each gain table entry T_{ij} requires $O(M^i \cdot Cost(o) \cdot \log err)$ operations due to the optimized distance computations, yielding $O(M^i \cdot Cost(o) \cdot M^2 \cdot \log err)$ total cost for the *Gain Table* initialization since there are $O(M^2)$ possible pairs. The merging process iterates up to $M-1$ times (reducing M nodes to 1), where each iteration performs one merge and updates $O(M)$ table entries, with the cost being $O(M^i \cdot Cost(o) \cdot M^3 \cdot \log err)$. Furthermore, there are $\frac{|O|}{M^{i+1}}$ nodes at level i, total computational cost for processing these node is bounded by $O(\frac{|O|}{M^{i+1}} \cdot M^i \cdot Cost(o) \cdot M^3 \cdot \log err)$, which is $O(|O| \cdot Cost(o) \cdot M^2 \cdot \log err)$. Thus, the overall node merge cost is bounded by $O(|O| \cdot M^2 \cdot Cost(o) \cdot \log err \cdot \log |O|)$, with $\log |O|$ being the height of the initial LM-Tree.

4.2 The Incremental Maintenance Algorithms

In this section, we propose an efficient incremental maintenance algorithm for LM-Tree, which allows both insertions and deletions of objects. Algorithm 3 illustrates the detailed procedure.

Insert Operation. Given an object o_{in} to be inserted, we first find a proper leaf node for insertion (line 1). The logic for reaching the proper leaf node is similar with the range query algorithm described in Section 3. One difference is that, if o_{in} is bounded by the boundary of multiple child nodes, the one with the smallest radius is chosen. If no child node can contain o_{in}, the child node that requires the smallest increase in its covering radius is selected.

When a leaf node e_L is reached, we first find the tuple (p_v, dpc_v, F_v, PDL_v) such that $dpc_v \leq d(o_{in}, e.c)$ and $dpc_{v+1} > d(o_{in}, e.c)$. If no suitable tuple is found, the last tuple in e_L is returned (line 2). Then we calculate the distance $d(o_{in}, p_v)$, insert o_{in} into the suitable storage position $PDL_v[pos]$ of PDL_v based on the corresponding linear function F_v (line 5). Note that, a single position can store multiple objects. If multiple objects are contained in

Table 2. Datasets Used in the Experiments.

Dataset	Size	Dim.	Metric
LA	1,073,727	2	L_2-norm
Words	354,984	1-34	Edit distance
Color	1,281,167	32	L_1-norm
Synthetic	1,000,000	20	L_∞-norm

$PDL_v[pos]$, it indicates that new pivot(s) (or new linear model(s)) should be added. To avoid the high computational overhead caused by frequently re-selecting pivots, we propose a deferred method.

Specially, after insertion, if the number of objects in PDL_v exceeds twice the number of non-empty positions in PDL_v, we apply the algorithm discussed in Section 4.1.1 to process objects within PDL_v, including new pivot selection, linear models reforming (lines 6-7). Note, re-segmenting one segment does not affect other segments, since each segment is already a maximal segment as stated in PROPERTY 3. After re-selecting pivots, if the number of pivots in e_L exceeds 2 times initial number of pivots, e_L is split into two nodes e_L^1 and e_L^2 following the logic of node splitting under M-Tree. (lines 8-9). After splitting, if the parent node of e_L^1 and e_L^2, denoted as e, exceeds 2 times initial number of its child nodes, we use Algorithm 2 for merging e's child nodes (lines 10-11). Similarly, when an inner node is split, we perform the same operation on its parent node. When the root node is split, the tree height increases by one.

Delete Operation. Given an object o_{de} to be deleted, we first remove it from its corresponding leaf node e_L (line 13). Note that, if o_{de} is a pivot, i.e., $o_{de} = p_v$, we do not update p_v, only remove o_{de} from $PDL_v[0]$. After deletion, if the number of objects in PDL_v is half of initial number of objects in PDL_v, we adopt the *ShrinkingCone* segmentation algorithm discussed in FITing-Tree [28] to form one (or a group of) segment(s) based on objects within PDL_v. For each new segment, if it is a high-slope segment, a new pivot is selected and a learning model is constructed for it. Otherwise, the corresponding objects will be managed by the node center $e_L.c$, where these objects will be orderly inserted into PDL_0 (lines 15-16);

Cost Analysis. To insert or delete an object o, the algorithm traverses $O(\log|O|)$ nodes. Each node e spends $Cost(o)$ in computing the distance from o to the node center. If e contains o, the algorithm scans pivots within e to find the pivot p_v that maintains o, then uses the linear model of p_v, the cost is bounded by $O(\log|P| + Cost(o) + \log err)$, where $|P|$ denotes the number of pivots in e and err represents the predicted error of linear model. For subsequent operations, since the cost of re-segmentation, node merge, and split are amortized across a linear number of operations, the overall complexity remains consistent with the initial construction cost analysis. Accordingly, the amortized cost of an insertion and a deletion is bounded by $O(\log|O| \cdot (\log|P| + Cost(o) + \log err) + \log|O| + M^2 \cdot \log err)$ and $O(\log|O| \cdot (\log|P| + Cost(o) + \log err) + \log|O|)$, respectively.

5 Performance Evaluation

This section presents the results of our extensive experiments that are designed to show the performance of our proposed algorithms.

5.1 Experiment Settings

Datasets. Following [1], we conduct our experiments using four datasets, including three real datasets and one synthetic dataset. (i) LA consists of geographical locations in Los Angeles. The L_2-norm is utilized to measure similarity; (ii) Words contains proper nouns, acronyms, and compound words taken from the Moby project. The *edit distance* is used to measure similarity; (iii) Color consists of standard MPEG-7 image features extracted from Flickr. The L_1-norm is utilized to

Table 3. Parameters and their settings.

Parameters	Settings
data cardinality $\|O\|$	200K, 400K, **600K**, 800K, 1000K
range query selectivity r_q	2%, 4%, **8%**, 16%, 32%
kNN query value k	20, 50, **100**, 150, 200
segmentation error ϵ	1, 2, 4, **5**, 6, 8, 10, 12, 16, 32
dimensionality Dim	2, 10, 20, 30, 40, 50, 60

measure similarity; (iv) Synthetic is a synthetic dataset, which contains 1M 20-dimensional data. Objects within this dataset follow a uniform distribution, and the L_∞-norm is employed. Table 2 summarizes statistics of datasets used in the experiments.

Performance Metrics. Following a previous study [1], we use the following four performance metrics: (i) *Overall running time* measures the algorithm's performance in handling queries and updates. We report the average running time required to handle 400K updates and 1000 metric range queries. (ii) *Build and update time* measures the time required for the algorithm to construct the index and the average update time for processing 400K updates, indicating how fast the algorithms respond to build and update. (iii) *Query time* measures the average query time of answering 1000 queries under different algorithms, indicating how fast the algorithms respond to metric range and kNN queries. (iv) *Index size* records the size of the index under different data cardinality, reflecting the memory usage of the index.

Parameters. Following [1], we evaluate the performance of different algorithms by considering parameters $|O|$, r_q, k, ϵ, and Dim. Here, $|O|$ denotes the number of objects maintained by each index. r_q refers to the query radius of a range query, and we use the selectivity to set the search radius r_q. In particular, the selectivity is a hyperparameter and denotes the expected percentage of query result objects within the dataset of a range query. Given a query point q and a selectivity of 8%, the search radius r_q is defined as the distance from q to its $(8\% \times |O|)$-th nearest neighbor in the dataset. This is because different datasets have different distance ranges, making it difficult to use the same fixed value across datasets. The parameter k refers to the k value of a kNN query, which varies from 20 to 200. In our algorithm, a smaller ϵ leads to more pivots, and thus more distance computations, which can affect indexing performance. Therefore, we vary ϵ from 1 to 32 to test the impact of different numbers of pivots. Dim refers to the dimensionality of the data object. In real implementation, we scale the *Synthetic* dataset to achieve this task. We vary Dim from 2 to 60 to test the impact of dimensionality on index performance. Table 3 shows the parameter settings, with bold indicating the default values.

Competitors. In addition to LM-Tree, we implement M-Tree, MVP-Tree, PM-Tree, EGNAT, LIMS, Flood, and Qd-Tree as competitors. (i) M-Tree [5] organizes objects in a tree structure based on distance comparisons, using covering radii for pruning. Specifically, in our experiments, we use an optimized version of the M-Tree, which minimizes overlapping between nodes and maximizes memory efficiency; (ii) MVP-Tree [3] organizes objects in a tree structure, where each node is split into m-ary using multiple pivots, and these pivots are employed for pruning during queries; (iii) PM-Tree [6] is an extension of M-Tree that improves pruning power by adding a set of fixed global pivots; (iv) EGNAT [25] is a dynamic m-ary extension of GHT that partitions the dataset via the generalized hyperplane method, employs pivot-based cut regions to accelerate similarity searches; (v) LIMS [4] is a state-of-the-art learned metric index. It converts metric data into one-dimensional data, thereby accelerating metric similarity search by applying a one-dimensional learned index; (vi) Flood [35] is a grid-based spatial index whose main idea is to optimize the index structure and data storage layout by leveraging both data and query distributions; (vii) Qd-Tree [36] can be regarded

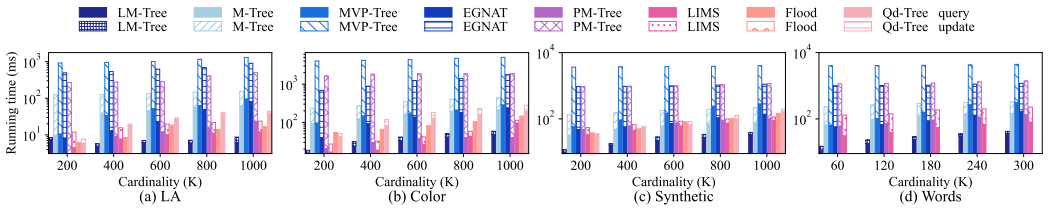


Fig. 4. Scalability of LM-Tree vs. Cardinality of Dataset (r_q = 8%, and ϵ = 5).

as a workload-aware learned index inspired by Kd-Tree. All algorithms are implemented with C++, and all experiments are conducted on a 64-bit Ubuntu 24.04 LTS machine equipped with an AMD 7900X CPU, 128GB RAM, and an NVIDIA 4070 12GB GPU. Since most recent algorithms (including the majority of our competitors [3]) operate in memory, we also conduct our experiments under the same in-memory conditions.

5.2 Algorithm Comparisons

In this section, we evaluate the performance of different algorithms. Since Flood and Qd-Tree are spatial indices, their performance is only evaluated using the *LA*, *Color*, and *Synthetic* datasets.

5.2.1 Overall Performance Comparisons. Fig. 4 shows the average running time per update and query when handling 400K updates and 1000 range queries (r_q = 8%). Here, 1 update operation consists of 1 insertion and 1 deletion, the solid bar represents the average query time, and the hollow bar represents the average update time. We find that LM-Tree outperforms all competitors across all datasets in general. For example, under the *LA* dataset, it is on average 22.3X, 174.2X, 105.9X, 56.3X, 3.7X, 2.6X, and 4.1X faster than M-Tree, MVP-Tree, EGNAT, PM-Tree, LIMS, Flood, and Qd-Tree, respectively. We mainly have the following 3 findings.

Firstly, LM-Tree spends the lowest cost in supporting range queries in most cases. The main reason is that LM-Tree can self-adaptively select pivots based on the data distribution. Each pivot in LM-Tree is responsible for managing a distinct, non-overlapping subset of child nodes or objects. Leveraging this property, the pivot-based pruning strategy can prune multitudes of nodes or objects outside the query range. Furthermore, LM-Tree employs lightweight linear models to accelerate search processing.

In contrast, M-Tree relies on a single pivot (i.e., node center) for pruning, which results in less pruning power. MVP-Tree, EGNAT, and PM-Tree introduce multiple pivots to improve pruning performance; this also increases the computational overhead of accessing each node or object. Additionally, since they cannot effectively update pivots, these pivots will become less effective as data updates. LIMS enhances pruning power through a combination of multiple pivots and learned indices. However, as data updates, these selected pivots will become less effective. Additionally, since it transforms objects in metric spaces into 1-dimensional values, differences of these 1-dimensional values cannot effectively reflect distance relationships among objects in metric spaces, leading to the index not providing powerful pruning ability for supporting searches. Flood and Qd-Tree are formed based on the initial data and query distributions. As data continuously updates, the actual distribution gradually deviates from the initial data and query distributions, making index structures not well aligned with the query workload.

Secondly, we observe that LM-Tree spends the lowest updating cost among all metric space-based indices. One reason is that LM-Tree has a smaller index node scale due to its node merging strategy, which helps avoid accessing deeper tree structures during updates. Another reason is that LM-Tree

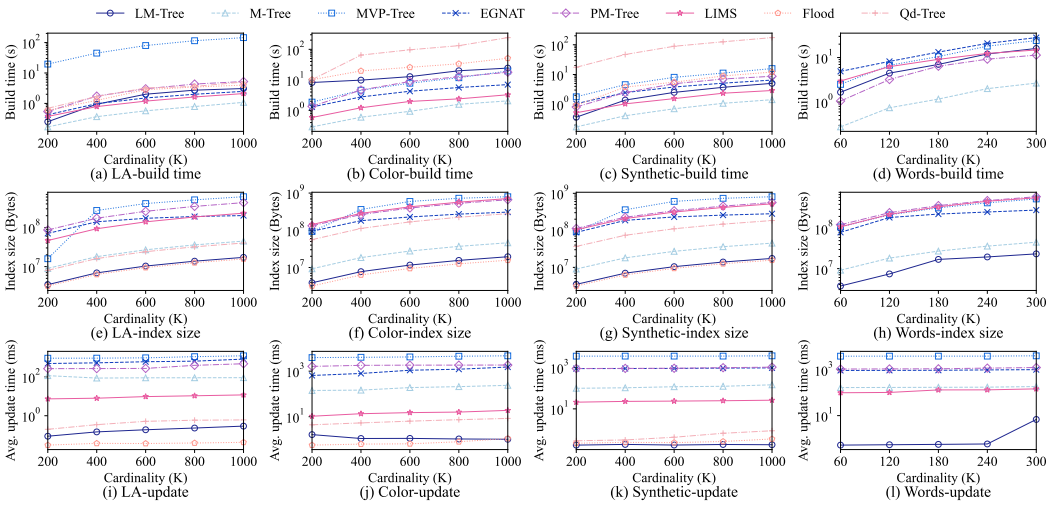


Fig. 5. Scalability of LM-Tree vs. Cardinality of Dataset ($r_q = 8\%$, $k = 100$, and $\epsilon = 5$).

organizes objects using pivots and linear models, which helps us efficiently locate the insertion position of an object. Additionally, the algorithm maintains low frequencies for pivot updates, linear model reforms, and node restructuring according to data distribution. In contrast, M-Tree's high update cost results from extensive node traversals and the inability to utilize pivots/learning models for update acceleration. MVP-Tree and EGNAT use multiple pivots in each node to improve pruning efficiency. As data updates, the pruning power of initial pivots degrades, necessitating access to more nodes and objects during updates. Furthermore, they must consider distance relationships among many objects when updating pivots, which will incur a high running cost. Similarly, the update cost of PM-Tree and LIMS, which use multiple global pivots, is still high.

For spatial-based learned indices, both Flood and Qd-Tree demonstrate efficient processing of insertions and deletions. Additionally, LM-Tree's update cost is slightly higher than Flood on some datasets. However, we want to highlight that both Flood and Qd-Tree cannot provide an effective mechanism for addressing the issue caused by changes in data distribution. In contrast, LM-Tree adaptively adjusts its pivots and learning models to changes in data distribution, thereby maintaining a powerful pruning ability and achieving competitive update performance. This trade-off is justified by its substantial superiority in query time. Additionally, both Flood and Qd-Tree can only support similarity search on spatial data, while LM-Tree can be applied to a broader range of domains (e.g., similarity search based on edit distance).

Thirdly, we observe that the running time of LM-Tree is stable and not sensitive to data distribution and cardinality. One main reason is that LM-Tree could adaptively identify regions where pruning is difficult and enhance pruning efficiency by introducing additional pivots. Furthermore, our node merging strategy can reduce the node count based on the data distribution, which reduces the number of nodes that should be accessed during the search and update.

5.2.2 Building, Storage, and Update Performance Comparisons. In this section, we evaluate the cost of building, storage, and updates for all algorithms across different datasets and data cardinalities.
Build Time vs. Data Cardinality. As shown in Fig. 5(a)-(d), the building time difference between LM-Tree and the others is not large. Besides, we want to highlight two points. Firstly, LM-Tree always incurs a longer build time than M-Tree as it is built on top of an M-Tree, upon which additional operations, such as pivot selection, linear model forms, and node merging, introduce

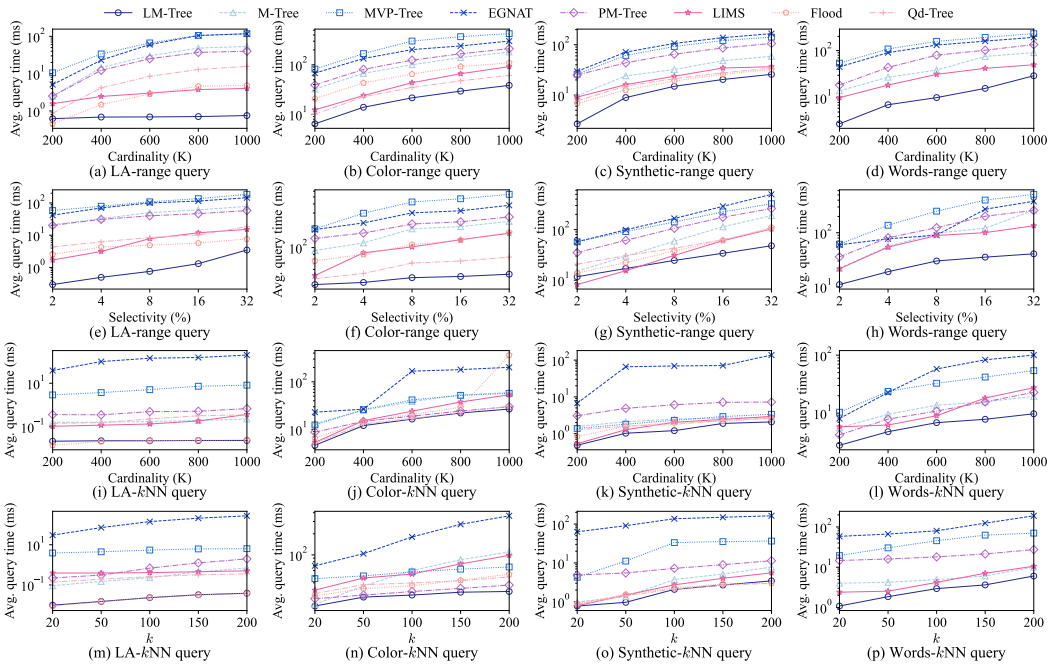


Fig. 6. Effect of parameters vs. $|O|$, r_q and k ($|O| = 1000K, r_q = 8\%, k = 100$, and $\epsilon = 5$).

extra build overhead. However, as the running cost of pivot selection, linear model optimization, and node merging is not high, the overall build cost is not high. Secondly, Flood and Qd-Tree are built based on both historical data and query distributions. During the build process, multiple queries are required to determine the partition boundaries along each dimension, which incurs high running costs.

Index Size vs. Data Cardinality. As shown in Fig. 5(e)-(h), LM-Tree achieves a lower index size than all metric space-based indices and Qd-Tree, while maintaining a comparable index size to Flood. For example, under the *LA* dataset, the index size of LM-Tree is reduced by 61.6%, 93.5%, 93.7%, 96.0%, 92.6%, and 57.1% compared to M-Tree, MVP-Tree, EGNAT, PM-Tree, LIMS, and Qd-Tree, respectively, while it incurs a 9.3% increase compared with Flood. Note that the LM-Tree incurs a relatively low space cost, as its node merging strategy effectively reduces the total tree size.

Update Performance vs. Data Cardinality. Fig. 5(i)–(l) show the average update time of executing 400K updates under different data cardinalities. Our findings indicate that LM-Tree once again outperforms metric space-based indices and Qd-Tree. This is because that LM-Tree employs a node merging strategy to reduce the number of nodes, uses pivots and linear models to determine insertion positions efficiently. The above methods ensure that LM-Tree can efficiently support updates. Although LM-Tree's update cost is slightly higher than Flood's on the *LA* and *Color* datasets, its self-adaptive structure confers consistently powerful pruning capabilities, making this a worthwhile trade-off for its substantial query performance gains.

5.2.3 Query Performance Comparisons. In this section, we evaluate query performance across varying data cardinalities, query selectivities, and values of k.

Query Performance vs. Data Cardinality. Fig. 6(a)-(d) and Fig. 6(i) –(l) report the average query time of range and kNN queries across various data cardinality, respectively. Undoubtedly, LM-Tree performs the best in most cases. MVP-Tree and EGNAT incur high costs because they use multiple

Table 4. #dists under different r_q and k ($|O| = 1000K$, unit: K).

r_q / k	2%	4%	8%	16%	32%	20	50	100	150	200
LM-Tree	**8.3**	**15.4**	**86.1**	**223.9**	**463.8**	**0.3**	**0.4**	0.6	**0.8**	**1.0**
M-Tree	183.9	280.4	364.8	428.3	539.3	0.4	**0.4**	**0.5**	0.9	1.1
MVP-Tree	185.5	252.1	312.7	370.0	456.4	4.2	5.0	8.1	10.1	10.2
EGNAT	185.0	281.7	366.4	429.8	540.0	303.0	303.6	312.9	318.4	320.7
PM-Tree	182.9	279.5	364.3	428.2	539.5	1.5	1.7	2.0	2.1	2.3
LIMS	59.4	109.4	278.0	426.3	537.2	14.9	16.0	16.7	16.9	18.3
Flood	255.7	381.8	503.8	731.7	738.1	22.5	23.6	24.6	26.3	26.8
Qd-Tree	256.6	339.6	417.5	490.7	616.2	26.5	27.5	27.9	28.8	29.1

Table 5. Query performance vs. ϵ ($1000K, r_q = 8\%$, unit: ms).

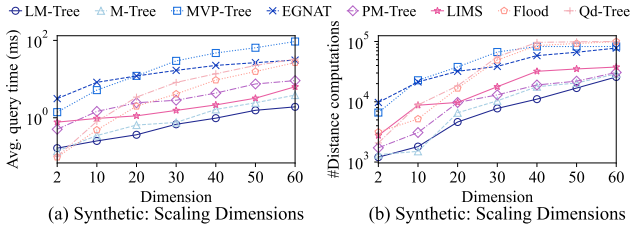
Dataset / ϵ	1	2	4	5	6	8	10	12	16	32
LA	3.93	1.88	0.84	**0.78**	0.85	1.56	1.60	1.48	1.51	1.55
Color	48.32	44.99	42.17	**38.04**	40.48	40.44	39.77	39.48	39.78	39.82
Synthetic	41.25	38.23	27.43	**24.38**	25.86	26.54	34.62	37.81	36.61	38.22
Words	47.60	41.76	38.89	36.69	**35.54**	37.04	37.30	37.46	36.89	37.78

pivots within each node to partition data. During each query, distances between the query object q and all pivots in overlapping nodes must be computed, leading to a high running cost. Although LIMS leverages a learned model to accelerate search, it still organizes objects using multiple global pivots. Additionally, it transforms objects in metric spaces into 1-dimensional values, which fail to preserve the original distance relationships. Consequently, the index lacks consistently powerful pruning ability. Flood and Qd-Tree incur higher query costs because neither of them adapts to changes in the query and data distributions. As data continuously updates, the actual distribution gradually deviates from the initial data and query distributions, leading to significant computational overhead during query processing. For example, queries may frequently access nodes within Qd-Tree (or cells within Flood) that contain a multitude of objects. When accessing these nodes (or cells), the algorithm must scan objects within them, leading to high running costs. In contrast, LM-Tree assigns each pivot to manage a disjoint subset of objects (or child nodes) within a node, meaning that only a small number of pivots, nodes, and objects need to be scanned during the search. In other words, it effectively prunes the search space using pivots and linear models.

Query Performance vs. r_q and k. Fig. 6(e)-(h) and (m)-(p) report the average metric range and kNN query performance across four datasets, showing how query performance varies with increasing query selectivity r_q and value k. As expected, the query cost for all methods increases as r_q or k grows, LM-Tree generally achieves the best performance in terms of query times. For example, on the *LA* dataset, the range query performance of LM-Tree outperforms M-Tree, MVP-Tree, EGNAT, PM-Tree, LIMS, Flood, and Qd-Tree by an average of $54.7X$, $131.5X$, $109.6X$, $47.2X$, $7.8X$, $6.1X$, and $10.3X$, respectively. LM-Tree uses fewer pivots and models to enable accurate distance pruning, thus avoiding lots of unnecessary calculations. Take distance computation amount in the *LA* dataset as an example in Table 4. LM-Tree achieves the lowest number of distance computations across almost all settings of r_q and k.

5.2.4 Effect of ϵ. Table 5 shows how query performance varies with different values of ϵ. It can be observed that as ϵ increases, the query time initially decreases. This is an expected result, as more pivots provide stronger pruning power as mentioned above. However, beyond a certain value, as ϵ continues to increase, the query time increases. One reason is that a larger ϵ leads to inaccurate identification of high-slope regions (i.e., where objects have similar distances to the center). As a result, many high-slope regions are not effectively recognized and still rely on pruning based on the center, requiring more nodes or objects to be scanned during the search. Another reason is that larger values of ϵ increase the model's prediction error, leading to more distance comparisons, which in turn reduces the indexing query performance.

5.2.5 Effect of Dim. Following previous work [6], we scale the *Synthetic* dataset up to 60 dimensions and use 100K data objects for evaluation. For performance metrics, we use the average query time and the number of distance computations of kNN queries to evaluate how the index performance changes as the dimensionality increases. Fig. 7 shows how the index performance changes as the data dimensionality increases. It can be observed that the average query time and the number of distance computations of all methods increase with dimensionality, but LM-Tree generally achieves the best performance among all methods, which demonstrates the effectiveness of our adaptive pivot selection strategy for pruning. The performance of coordinate-based index methods, such as Flood and Qd-Tree, degrades rapidly in higher dimensions because their filtering cost increases exponentially with dimensionality.



(a) Synthetic: Scaling Dimensions (b) Synthetic: Scaling Dimensions

Fig. 7. Effect of Dim. ($|O| = 100K$, $k = 100$, and $\epsilon = 5$).

5.2.6 Stability Analysis. To reflect the stability of our LM-Tree, we first repeatedly test the average query time for 10 runs on the *Color* dataset, all parameters are default values. In each run, all methods use the same query set to execute 1,000 range queries, and the average query time is recorded. Finally, we compute the ratio of the standard error to the mean of these ten runs to examine the performance variability across multiple runs. Table 6 shows the ratio of standard error to average query time for 10 runs under different indices. We find that LM-Tree is generally more stable than most competitors. Although MVP-Tree is more stable than ours, it has a higher running cost in supporting similarity search and incremental maintenance.

Table 6. Standard Error of 10 Runs / Avg. query time ($r_q = 8\%$).

Index	LM-Tree	M-Tree	MVP-Tree	EGNAT	PM-Tree	LIMS	Flood	Qd-Tree
200K	0.61%	1.83%	0.41%	6.15%	4.79%	1.05%	0.83%	0.43%
400K	0.35%	2.31%	0.35%	2.63%	1.85%	1.13%	2.80%	0.33%
600K	0.44%	6.65%	0.15%	5.90%	1.92%	8.85%	1.61%	0.51%
800K	0.57%	7.11%	0.29%	3.32%	1.87%	3.31%	0.91%	1.03%
1000K	0.84%	3.12%	0.54%	2.63%	2.22%	2.57%	0.99%	1.19%

Next, we evaluate the cost variation of our method under update sequences on *Color* dataset. We first form an LM-Tree based on 1,000K objects and measure the per-operation processing time for the following three types of operations: (i) 400K insertions only; (ii) 400K deletions only; and (iii) 400K insertions and 400K deletions executed simultaneously. Fig. 8 shows the processing time per operation for handling types (i)–(iii). It can be observed that LM-Tree remains generally stable under insertion, deletion, and update operations. Specifically, we highlight the following two points. First, as objects are continuously inserted, the number of objects to be maintained in the index increases, leading to a higher insertion cost; conversely, as objects are continuously deleted, the number of maintained objects decreases, resulting in a lower deletion cost. Second, during insertions and deletions, operations such as node splits and merges may be triggered, which explains the occurrence of higher latency points. However, such cases happen infrequently. As shown in Fig. 8(c), the number of node merge operations is 160, i.e., only 0.02% of all update operations.

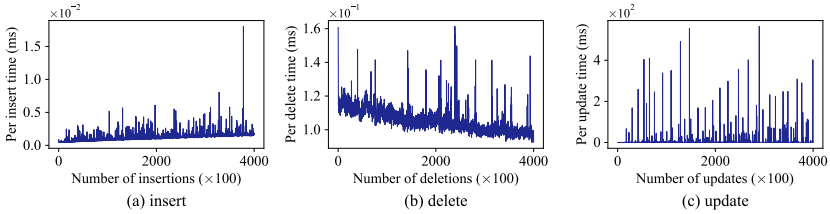


Fig. 8. Cost Variation of LM-Tree under Update Sequences.

5.2.7 Ablation Study. In this section, we conduct an ablation study to evaluate the effect of our proposed pivot selection and node merging strategy on index performance. Table 7 reports the average query time, number of distance computations, and number of nodes for 1,000 range queries with a selectivity of 8%. We observe that W/O NM (short for LM-Tree without Node Merge) achieves faster query times than M-Tree, MVP-Tree, PM-Tree, EGNAT, LIMS, Flood, and Qd-Tree, due to our proposed pivot selection and learning model optimization strategies. Compared to W/O NM, LM-Tree improves query performance by an additional 33.7%, with a significant reduction in both node count and distance computations. The main reason is that LM-Tree maintains fewer nodes than that of W/O NM, and thus achieves the best query performance.

Table 7. Average query time, distance computations, and #nodes under different datasets ($|O| = 1000K$, $r_q = 8\%, \epsilon = 5$)

Dataset	Color			Synthetic		
	time(ms)	#dists(K)	#nodes(K)	time(ms)	#dists(K)	#nodes(K)
M-Tree	181.24	757.94	56.82	59.04	348.02	46.13
MVP-Tree	426.07	753.94	61.56	141.28	206.44	64.49
EGNAT	301.09	516.87	39.84	166.22	228.75	46.73
PM-Tree	211.47	677.80	2.78	106.05	184.45	1.69
LIMS	102.09	1039.41	–	40.93	786.56	–
Flood	115.19	1158.52	4.86	38.34	273.06	3.70
Qd-Tree	95.96	999.08	6.12	42.90	389.48	2.60
W/O NM	38.44	839.13	7.39	37.11	473.37	2.18
LM-Tree	32.31	814.70	5.90	24.59	467.28	2.16

6 Conclusion

In this paper, we presented LM-Tree, a hybrid learned index that combines the strengths of pivots and learning models to enhance the performance of similarity search in metric spaces. Compared with traditional metric access methods, LM-Tree dynamically maintains a variable number of pivots per node, each manages a disjoint subset of data. By integrating lightweight learning models to approximate distance distributions, LM-Tree achieves stronger pruning power and faster query performance. Additionally, our proposed maintenance algorithm ensures efficient updates, including pivot update, model reform, node split, and node merge. Experimental results on real and synthetic datasets demonstrate that LM-Tree significantly outperforms state-of-the-art efforts.

Acknowledgments

The work is partially supported by the National Key Research and Development Program of China (No. 2024YFF0617702), the National Natural Science Foundation of China (No. U23A20309, U22A2025, 62472293, 62232007), the Dameng Database Research Fund (No. 2025021900057, 2025021900058), and the 111 Project (No.B16009).

References

[1] Lu Chen, Yunjun Gao, Xuan Song, Zheng Li, Yifan Zhu, Xiaoye Miao, and Christian S. Jensen. Indexing metric spaces for exact similarity search. *ACM Comput. Surv.*, 55(6):128:1–128:39, 2023.

[2] Caetano Traina Jr., Agma J. M. Traina, Bernhard Seeger, and Christos Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, Proceedings*, volume 1777, pages 51–65, 2000.

[3] Tolga Bozkaya and Z. Meral Özsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 357–368, 1997.

[4] Yao Tian, Tingyun Yan, Xi Zhao, Kai Huang, and Xiaofang Zhou. A learned index for exact similarity search in metric spaces. *IEEE Trans. Knowl. Data Eng.*, 35(8):7624–7638, 2023.

[5] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 426–435. Morgan Kaufmann, 1997.

[6] Tomás Skopal, Jaroslav Pokorný, and Václav Snásel. Pm-tree: Pivoting metric tree for similarity search in multimedia databases. In *Advances in Databases and Information Systems, 8th East European Conference, ADBIS 2004, Budapest, Hungary, September 22-25, 2004, Local Proceeding*, 2004.

[7] Marcos R. Vieira, Caetano Traina Jr., Fabio Jun Takada Chino, and Agma J. M. Traina. Dbm-tree: A dynamic metric access method sensitive to local density data. *J. Inf. Data Manag.*, 1(1):111–128, 2010.

[8] Xiangmin Zhou, Guoren Wang, Jeffrey Xu Yu, and Ge Yu. M+-tree : A new dynamical multidimensional index for metric spaces. In *Database Technologies 2003, Proceedings of the 14th Australasian Database Conference, ADC 2003, Adelaide, South Australia, February 2003*, volume 17, pages 161–168, 2003.

[9] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 311–321. ACM/SIAM, 1993.

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

[11] Lu Chen, Yunjun Gao, Xinhan Li, Christian S. Jensen, and Gang Chen. Efficient metric indexing for similarity search. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 591–602. IEEE Computer Society, 2015.

[12] Caetano Traina Jr., Agma J. M. Traina, Roberto F. Santos Filho, and Christos Faloutsos. How to improve the pruning ability of dynamic metric access methods. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 4-9, 2002*, pages 219–226. ACM, 2002.

[13] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504, 2018.

[14] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 969–984. ACM, 2020.

[15] Tomás Skopal, Jaroslav Pokorný, Michal Krátký, and Václav Snásel. Revisiting m-tree building principles. In *Advances in Databases and Information Systems, 7th East European Conference, ADBIS 2003, Dresden, Germany, September 3-6, 2003, Proceedings*, volume 2798, pages 148–162, 2003.

[16] Shichao Jin, Okhee Kim, and Wenya Feng. Mx-tree: A double hierarchical metric index with overlap reduction. In *Computational Science and Its Applications - ICCSA 2013 - 13th International Conference, Ho Chi Minh City, Vietnam, June 24-27, 2013, Proceedings, Part V*, volume 7975 of *Lecture Notes in Computer Science*, pages 574–589. Springer, 2013.

[17] Ives Rene Venturini Pola, Caetano Traina Jr., and Agma J. M. Traina. The mm-tree: A memory-based metric tree without overlap between nodes. In *Advances in Databases and Information Systems, 11th East European Conference, ADBIS 2007, Varna, Bulgaria, September 29-October 3, 2007, Proceedings*, volume 4690, pages 157–171, 2007.

[18] Xiangmin Zhou, Guoren Wang, Xiaofang Zhou, and Ge Yu. Bm^+-tree: A hyperplane-based index method for high-dimensional metric spaces. In *Database Systems for Advanced Applications, 10th International Conference, DASFAA 2005, Beijing, China, April 17-20, 2005, Proceedings*, volume 3453, pages 398–409, 2005.

[19] Lior Aronovich and Israel Spiegler. Cm-tree: A dynamic clustered index for similarity search in metric databases. *Data Knowl. Eng.*, 63(3):919–946, 2007.

[20] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[21] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[22] Lu Chen, Yunjun Gao, Baihua Zheng, Christian S. Jensen, Hanyu Yang, and Keyu Yang. Pivot-based metric indexing. *Proc. VLDB Endow.*, 10(10):1058–1069, 2017.

[23] David Novak, Michal Batko, and Pavel Zezula. Metric index: An efficient and scalable solution for precise and approximate similarity search. *Inf. Syst.*, 36(4):721–733, 2011.

[24] Sergey Brin. Near neighbor search in large metric spaces. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 574–584. Morgan Kaufmann, 1995.

[25] Mauricio Marín, Roberto Uribe, and Ricardo J. Barrientos. Searching and updating metric space databases using the parallel EGNAT. In *Computational Science - ICCS 2007, 7th International Conference Beijing, China, May 27-30, 2007, Proceedings, Part I*, volume 4487 of *Lecture Notes in Computer Science*, pages 229–236. Springer, 2007.

[26] Jiaoyi Zhang and Yihan Gao. CARMI: A cache-aware learned index with a cost-based construction algorithm. *Proc. VLDB Endow.*, 15(11):2679–2691, 2022.

[27] Na Guo, Yaqi Wang, Wenli Sun, Yu Gu, Jianzhong Qi, Zhenghao Liu, Xiufeng Xia, and Ge Yu. Chameleon: Towards update-efficient learned indexing for locally skewed data. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*, pages 4316–4328. IEEE, 2024.

[28] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. FITing-Tree: A data-aware index structure. In *SIGMOD*, pages 1189–1206, 2019.

[29] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *Proc. VLDB Endow.*, 14(2):74–86, 2020.

[30] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. SPRIG: A learned spatial index for range and knn queries. In *Proceedings of the 17th International Symposium on Spatial and Temporal Databases, SSTD 2021, Virtual Event, USA, August 23-25, 2021*, pages 96–105. ACM, 2021.

[31] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. Learned index for spatial queries. In *20th IEEE International Conference on Mobile Data Management, MDM 2019, Hong Kong, SAR, China, June 10-13, 2019*, pages 569–574. IEEE, 2019.

[32] Jack A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2-4, 1984, Waterloo, Ontario, Canada*, pages 181–190. ACM, 1984.

[33] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. Effectively learning spatial indices. *Proc. VLDB Endow.*, 13(11):2341–2354, 2020.

[34] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2119–2133. ACM, 2020.

[35] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 985–1000. ACM, 2020.

[36] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 193–208. ACM, 2020.

[37] Beng Chin Ooi. Spatial kd-tree: A data structure for geographic database. In Hans-Jörg Schek and Gunter Schlageter, editors, *Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung, Darmstadt, 1.-3. April 1987, Proceedings*, volume 136 of *Informatik-Fachberichte*, pages 247–258. Springer, 1987.

[38] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. The rlr-tree: A reinforcement learning based r-tree for spatial data. *Proc. ACM Manag. Data*, 1(1):63:1–63:26, 2023.

[39] Dalsu Choi, Hyunsik Yoon, Hyubjin Lee, and Yon Dohn Chung. Waffle: In-memory grid index for moving objects with reinforcement learning-based configuration tuning system. *Proc. VLDB Endow.*, 15(11):2375–2388, 2022.

[40] Yufan Sheng, Xin Cao, Yixiang Fang, Kaiqi Zhao, Jianzhong Qi, Gao Cong, and Wenjie Zhang. WISK: A workload-aware learned index for spatial keyword queries. *Proc. ACM Manag. Data*, 1(2):187:1–187:27, 2023.

Received July 2025; revised October 2025; accepted November 2025