A Learned Index for Exact Similarity Search in Metric Spaces

Yao Tian, Tingyun Yan, Xi Zhao, Kai Huang, and Xiaofang Zhou, *Fellow, IEEE*

Abstract—Indexing is an effective way to support efficient query processing in large databases. Recently the concept of *learned index*, which replaces or complements traditional index structures with machine learning models, has been actively explored to reduce storage and search costs. However, accurate and efficient similarity query processing in high-dimensional metric spaces remains to be an open challenge. In this paper, we propose a novel indexing approach called LIMS that uses data clustering, pivot-based data transformation techniques and learned indexes to support efficient similarity query processing in metric spaces. In LIMS, the underlying data is partitioned into clusters such that each cluster follows a relatively uniform data distribution. Data redistribution is achieved by utilizing a small number of pivots for each cluster. Similar data are mapped into compact regions and the mapped values are totally ordinal. Machine learning models are developed to approximate the position of each data record on disk. Efficient algorithms are designed for processing range queries and nearest neighbor queries based on LIMS, and for index maintenance with dynamic updates. Extensive experiments on real-world and synthetic datasets demonstrate the superiority of LIMS compared with traditional indexes and state-of-the-art learned indexes.

Index Terms—Learned Index, Multi-dimension, Metric Space.

1 Introduction

Similarity search is one of the fundamental operations in the era of big data. It finds objects from a large database within a distance threshold to a given query object (called *range queries*) or the top-k most similar to the query object (called k *nearest neighbor queries*, or kNN *queries*), based on certain similarity measures or distance functions. For example, in spatial databases, a similarity query can be used to find all restaurants within a given range in terms of the *Euclidean distance*. In image databases, a similarity query can be used to find the top 10 most similar images to a given image in terms of the *Earth mover's distance* [1]. To accommodate a wide range of data types and distance functions, we consider similarity search in the context of metric spaces in this paper. A metric space is a generic space that makes no requirement of any particular data representation, but only a distance function that satisfies the four properties, namely non-negativity, identity, symmetry and triangle inequality (Definition 1, Section 3). A number of metric-space indexing methods have been proposed in the literature to accelerate similarity query processing [2], [3], [4], [5], [6]. However, these indexing methods that are based on tree-like structures are increasingly challenged by the rapidly growing volume and complexity of data. On the one hand, query processing with such indexes requires traversing many index nodes (*i.e.*, nodes on the path from the root node to a leaf node) in the tree structure, which can be time-consuming. On the other hand, tree-like indexes impose non-negligible storage pressure on datasets that store complex and large objects, such as image and audio feature data.

In recent years, the concept of *learned index* [7] has been developed to provide a new perspective on indexing. By enhancing or even replacing traditional index structures with machine learning models that can reflect the intrinsic patterns of data, a learned index can look up a key quickly and save a lot of memory space required by traditional index structures at the same time. The original idea is limited to the one-dimensional case where data is sorted in an in-memory dense array. Directly adapting this idea for a multi-dimensional case is unattractive, since multi-dimensional data has no natural sort order. Several multi-dimensional learned index structures have been proposed to address this issue [8], [9], [10], [11], [12], [13], [14], [15] (detailed discussions refer to Section 2). Despite the significant success of these learned indexes compared with traditional indexing methods, they still have some limitations. First, the existing learned index structures do not support similarity search in metric spaces. The metric space has neither coordinate structure nor dimension information (Remark 1, Section 3), so the numbering rules (*e.g.*, z-order [16]) and specific pruning strategies designed for vector spaces are not applicable. The triangle inequality is the only property we can utilize to reduce the search space. The generality of metric space provides an opportunity to develop unified indexing methods, while it also presents a significant challenge to develop an efficient learned indexing method. Second, the existing learned multi-dimensional index structures suffer from the phenomenon called *curse of dimensionality*. By integrating machine learning models into traditional multi-dimensional indexes, these learned indexes are restricted to certain types of data space partition-

- *Y. Tian, X. Zhao, K. Huang and X.F. Zhou are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: ytianbc@cse.ust.hk, {xizhao, ustkhuang}@ust.hk, zxf@cse.ust.hk.*
- *T.Y. Yan is with the Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou, China. E-mail: tingyun_yan@e.gzhu.edu.cn.*

Manuscript received December 28, 2021; revised 5 April 2022; published online xx xx xx.

ing (*e.g.,* grid partitioning), which inevitably leads to rapid performance degradation when the number of dimensions grows. Third, the time to train a machine learning model that can well approximate complex data distributions is typically very long, which makes learned indexes difficult to adapt to frequent insertion/deletion operations and query pattern changes. Finally, some existing learned indexes [10] can only return approximate query results, *i.e.,* there may exist false negatives in the result set, because of errors caused by machine learning models.

To address the aforementioned limitations, we develop a novel disk-based learned index structure for metric spaces, called LIMS, to facilitate exact similarity queries (*i.e.,* point, range and kNN queries). In contrast to the coordinate-based data partitioning, LIMS adopts a distance-based clustering strategy to group the underlying data into a number of subsets so as to decompose complex and potentially correlated data into clusters with simple and relatively uniform distributions. LIMS selects a small set of pivots for each cluster and utilizes the distances to the pivots to perform data redistribution. This reduces the dimensionality of the data to the number of pivots adopted. By using a proper pivot-based mapping, LIMS organizes similar objects into compact regions and imposes a total order over the data. Such an organization can significantly reduce the number of distance computations and page accesses during query processing. In order to further boost the search performance, LIMS follows the idea of *learned index*, using several simple polynomial regression models to quickly locate data records that might match the query filtering conditions. Furthermore, LIMS can be partially reconstructed quickly due to its independent index structure for each cluster, which makes LIMS adaptable to changes. As we will show later, LIMS significantly outperforms other multi-dimensional learned indexes and traditional indexes in terms of the average query time and the number of page accesses, especially when processing high dimensional data.

The main contributions of this paper include:

- We design LIMS, the first learned index structure for metric spaces, to facilitate exact similarity search.
- Efficient algorithms for processing point, range and kNN queries are proposed, enabling a unified solution for searching complex data in a representation-agnostic way. An update strategy is also proposed for LIMS.
- To the best of our knowledge, no experiment evaluation between different learned indexes has been performed. In this paper, we compare four multi-dimensional learned indexes. Extensive experiments on real-world and synthetic data demonstrate the superiority of LIMS.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 introduces the basic concepts and formulates the research problem. Section 4 describes the details of LIMS. LIMS-based similarity query algorithms are discussed in Section 5. Section 6 reports the experimental results. Section 7 concludes the paper.

2 Related Work

We focus on reviewing learned multi-dimensional indexes here. Good surveys of various traditional metric-space indexing methods can be found in [2], [3], [4], [5], [6].

The idea of *learned index* is that indexes can be regarded as models which take a key as the input and output the position of the corresponding record. If such a "black-box" model can be learned from data, a query can be processed by a function invocation in $O(1)$ time instead of traversing a tree structure in $O(\log n)$ time. RMI is the first to explore how to enhance or replace classic index structures with machine learning models [7]. It assumes that data is sorted and kept in an in-memory dense array. In light of this, a machine learning model essentially is to learn a cumulative distribution function (CDF). RMI consists of a hierarchy of models, where internal nodes in the hierarchy are the models responsible for predicting the child model to use, and a leaf model predicts the position of record. Since RMI utilizes the distribution of data and requires no comparison in each node, it provides significant benefits in terms of storage consumption and query processing time. For the sake of quickly correcting errors caused by machine learning models and supporting range queries, RMI is limited to index key-sorted datasets, which makes a direct application of RMI to multi-dimensional data infeasible because there is no obvious ordering of points. Even if these points are embedded into an ordered space, guaranteeing the correctness and efficiency of a query (*e.g.,* range query and kNN query) remains a challenging task.

ZM index is the first effort to apply the idea of *learned index* to multi-dimensional spaces [8]. It adopts the z-order space filling curve [16] to establish the ordering relationship for all points and then invokes RMI to support point and range queries. The correctness is guaranteed by a nice geometric property of the z-order curve, *i.e.,* monotonic ordering. However, ZM index needs to check many irrelevant points during the refinement phase, which would get worse for high dimensional spaces. It does not support kNN queries and index updates.

Recursive spatial model index (RSMI) builds on RMI and ZM [10]. It develops a recursive partitioning strategy to partition the original space, and then groups data according to predictions. This results in a learned point grouping, which is different from RMI that fixes the data layout first and trains a model to estimate positions. For each partition, RSMI first maps points into the rank space and then invokes ZM to support point, range and kNN queries. However, the correctness of range and kNN queries can not be guaranteed. In addition, since RSMI is still based on space filling curves, the good performance of RSMI is confined to low dimension spaces.

LISA [9], a learned index structure for spatial data can effectively reduces the number of false positives compared with ZM, by 1) partitioning the original space into grid cells based on the data distribution; 2) ordering data with a partially monotonic mapping function and rearranging data layout according to mapped values; 3) decomposing a large query range into multiple small ones. LISA has a dynamic data layout as RSMI, but the correctness of range and kNN queries can be guaranteed by the monotonicity of the models. However, its advantage in low scan overhead comes with the costly checking procedure and high index construction time. And the grid-based partitioning strategy makes it unsuitable for high dimensional spaces. Besides, LISA-based kNN query processing suffers from

many repeated page accesses due to doing range queries with increasing radius from the scratch.

Similar to LISA, Flood [13] also partitions data space into grid cells along dimensions such that for each dimension, the number of points in each partition is approximately the same. Flood assumes a known query workload and utilizes sample queries to learn an optimal combination of indexing dimensions and the number of partitions. Once these are learned, Flood maintains a table to record the position of the first point in each cell. At query time, Flood invokes RMI for each dimension to identify the cells intersecting the query and looks up the cell table to locate the corresponding records. However, it cannot efficiently adapt to correlated data distribution and skewed query workloads. Tsunami [14] extends Flood by utilizing query skew to partition data space into some regions, and then further dividing each region based on data correlations. However, simply choosing a subset of dimensions could degrade the performance as dimensionality increases. These studies are not discussed further as we do not assume a known query workload.

A multi-dimensional learned (ML) index [11] combines the idea of iDistance [17] and RMI. It first partitions data into clusters, and then identifies the cluster center as the reference point. After all data points are represented in a one-dimensional space based on the distances to the reference point, RMI can be applied. Different from iDistance, ML uses a scaling value rather than a constant to stretch the data range. However, points along a fixed radius have the same value after the transformation, leading to many irrelevant points to be checked. ML does not support data updates. Note that ML cannot be directly applied in metric spaces since reference points selection is realized by the KMeans algorithm [18]. The reference point that is the mean of points in the clusters may not be in the dataset. It is not always possible to create "artificial" objects in metric datasets [19].

Different from finding a sort order over multi-dimensional data and then learning the CDF, a reinforcement learning based R-tree for spatial data (RLR-Tree) [20] uses machine learning techniques to improve on the classic R-tree index. Instead of relying on hand-crafted heuristic rules, RLR-Tree models two basic operations in R-tree, *i.e.,* choosing a subtree for insertion and splitting a node, as *Markov decision process* [21], so reinforcement learning models can be applied. Because it does not need to modify the basic structure of the R-tree and query processing algorithms, it is easier to be deployed in the current databases systems than the learned indexes. However, due to the curse of dimensionality, the minimum bounding rectangle (MBR) for a leaf node (even in an optimal R-tree) can be nearly as large as the entire data space, such that the R-tree becomes ineffective. Similar to RLR-Tree, Qd-Tree [15] uses the reinforcement learning to optimize the data partitioning strategy of kd-tree based on a given query workload, and suffers from the same problem. These studies are not discussed further since they are out of our scope.

3 Background

In this section, we first introduce some basic concepts and then the formal definition of learned index for exact similarity search in metric spaces is presented. Table 1 lists the key notations and acronyms used in this paper.

TABLE 1: List of key notations

Notation	Description
P	The dataset
$\mathbb{U}, dist, d$	The data space, distance metric, dimensionality
p, q	A data point, a query point
r	Query radius
k	The number of nearest neighbors
K	The number of clusters
m	The number of pivots
N	The number of super rings
C_i	The ith cluster
$O_j^{(i)}$	The jth pivots in ith cluster
$dist\_max_j^{(i)}$	The distance of the furthest object in ith cluster from the jth pivot
$dist\_min_j^{(i)}$	The distance of the nearest object in ith cluster from the jth pivot

Definition 1 (Metric space)**.** *A metric space is a pair* $(\mathbb{U}, dist)$*, where* $\mathbb{U}$ *is a set of objects and* $dist : \mathbb{U} \times \mathbb{U} \rightarrow [0, \infty)$ *is a function so that* $\forall p_1, p_2, p_3 \in \mathbb{U}$*, the following holds:*

- *non-negativity:* $dist(p_1, p_2) \geq 0$;
- *identity:* $dist(p_1, p_2) = 0$ *iff* $p_1 = p_2$;
- *symmetry:* $dist(p_1, p_2) = dist(p_2, p_1)$;
- *triangle inequality:* $dist(p_1, p_3) \leq dist(p_1, p_2) + dist(p_2, p_3)$.

Remark 1. *Metric space is generic because it only requires the distance function satisfying the above properties. Vector space* $\mathbb{R}^d$ *with the Euclidean distance is a special metric space, where additional properties,* e.g., *the dimension and coordinates, are specified, which can be used to accelerate the search.*

In this paper, we consider three types of exact similarity queries in metric spaces: the range query, point query, and kNN query.

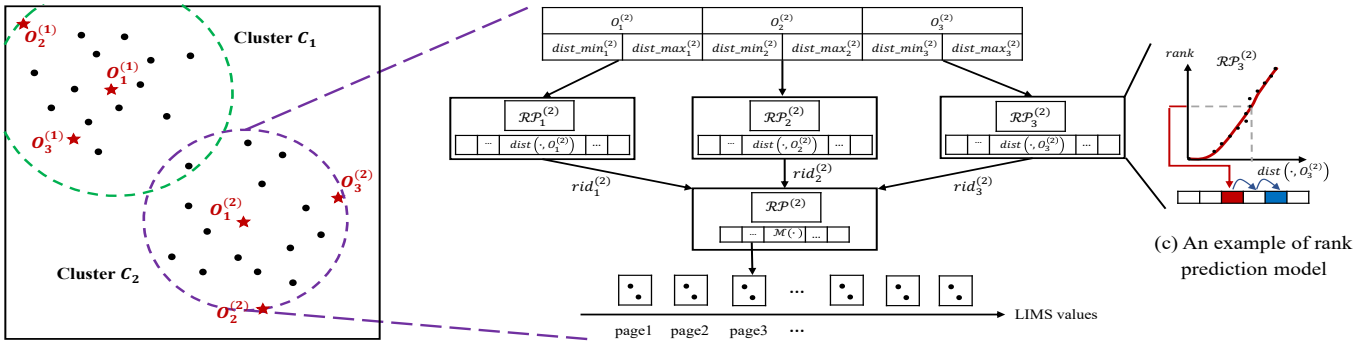
Definition 2 (Range query)**.** *Given a set* $P \subseteq \mathbb{U}$*, a query object* $q \in \mathbb{U}$*, and a query radius* $r \geq 0$*, a range query returns all objects in* P *within the distance* r *of* q*,* i.e., $range(q, r) = \{p \in P | dist(p, q) \leq r\}$.

Remark 2. *Point query is a special case of range query with* $r = 0$ *and an arbitrary metric. In this case, we say* $p = q$.

Definition 3 (kNN query)**.** *Given a set* $P \subseteq \mathbb{U}$*, a query object* $q \in \mathbb{U}$*, and a positive integer* k*, a kNN query returns a set of* k *objects, denoted as* $k\text{NN}(q, k)$*, such that* $\forall p \in k\text{NN}(q, k), p' \in P \setminus k\text{NN}(q, k), dist(q, p) \leq dist(q, p')$.

Example 1. *Consider a word dataset* $P = \{$"$fame$", "$gain$", "aim", "ACM"$\}$ *associated with the edit distance [22]. A range query* $range($"$game$"$, 2)$ *returns all words in* P *within the edit distance 2 to* "$game$", i.e., $\{$"$fame$", "$gain$"$\}$. *The* 1*NN query* $kNN($"$game$"$, 1)$ *returns the nearest neighbor of* "$game$", i.e., $\{$"$fame$"$\}$.

Problem Statement. Let $(\mathbb{U}, dist)$ be a metric space and $P = \{p_1, p_2, \ldots, p_n\} \subseteq \mathbb{U}$ be a set of objects. The **learned index for exact similarity search in metric spaces** is to learn an index structure for P so that point query, range query and kNN query can be processed accurately and efficiently. In addition, the index structure is supposed to support insertion and deletion operations.



(a) Data clustering and pivots in the metric space

(b) Index structure for cluster C_2

(c) An example of rank prediction model

Fig. 1: LIMS index structure

4 LIMS

In this section, we first give an overview of the index structure of LIMS and then present everything needed to build LIMS. LIMS-based query processing will be discussed in Section 5.

4.1 Overview

LIMS consists of three parts: the data clustering and pivot selection, the pivot-based mapping function and associated binary relationship, as well as rank prediction models. Fig. 1 gives an overview of LIMS index structure in a metric space associated with the Euclidean distance, although other metric spaces also apply for LIMS. LIMS first partitions the underlying data into a set of clusters, *e.g.,* 2 clusters in Fig. 1(a), so that each of them follows a relatively uniform data distribution, and then a set of data-dependent pivots for each cluster are picked, *e.g.,* 3 pivots $O_1^{(1)}, O_2^{(1)}$ and $O_3^{(1)}$ for cluster C_1 (detailed discussions refer to Section 4.3). Then, LIMS maintains a learned index for each cluster separately. Since the index structure is same for each cluster, we take cluster C_2 for example. LIMS computes the distances from each object in the cluster to the well-chosen pivots, *e.g.,* $dist(\cdot, O_3^{(2)})$ in Fig. 1(b). The maximum and minimum distances from each pivot to the corresponding objects, *e.g.,* $dist\_max_3^{(2)}$ and $dist\_min_3^{(2)}$ are stored so as to support efficient queries. Since objects are sorted by distance values, LIMS can learn a series of rank prediction models, *e.g.,* $\mathcal{RP}_3^{(2)}$ in Fig. 1(c), for quick computation of the rank of an object given its distance to the pivot. After that, a well-defined pivot-based mapping function $\mathcal{M}$ is called to transform each object into an ordered set (Definition 6, 7, Section 4.2). We call elements in this set *LIMS values*. Finally, we physically maintain all data objects sequentially on disk in ascending order of their LIMS values and the relationship between LIMS values and the addresses of data objects in disk pages can be learned by another rank prediction model, *e.g.,* $\mathcal{RP}^{(2)}$ (detailed discussions refer to Section 4.2).

In what follows, we first focus on the specific learned index structure for each cluster, and then turn back to clustering and pivot selection methods. In other words, we assume that the data space has been partitioned, and the pivots in each cluster have been determined.

4.2 Index Structure

Suppose that K clusters, say $\{C_1, C_2, \ldots, C_K\}$, and m pivots for each cluster, say $\{O_1^{(i)}, O_2^{(i)}, \ldots, O_m^{(i)}\}$, $i = 1, \ldots, K$, have been determined. Then, for each cluster $C_i, i = 1, \ldots, K$ and pivot $O_j^{(i)}, j = 1, \ldots, m$, all data objects (unique identifiers) are sorted in ascending order of their distances to the pivot. Based on m sorted lists in C_i, LIMS learns m rank prediction models. For model reuse, we define it formally as follows:

Definition 4 (Rank)**.** *Let A be a finite multiset drawn from an ordered set B. For any element $x \in A$, we define the rank of x as the number of elements smaller than x,* i.e.,

$$rank(x) = |\{x' \in A | x' < x\}|. \tag{1}$$

Example 2. *Let $A = \{1.5, 1.5, 1.8, 1.8, 2.0\}$ be a multiset of distance values to a given pivot sorted in ascending order. Then, $rank(1.5) = 0, rank(1.8) = 2, rank(2.0) = 4$.*

Definition 5 (Rank Prediction Model)**.** *Let A be a finite multiset drawn from an ordered set B. Rank prediction model $\mathcal{RP} : B \to [0, +\infty)$ is a function learned from $\{(x, rank(x))\}_{x \in A}$, so that it can predict the rank for any element $x \in B$,* i.e.,

$$rank(x) \approx \mathcal{RP}(x). \tag{2}$$

Remark 3. *In the strict sense, there is no definition of rank for element $x \in B \backslash A$. What we want to express here is the number of elements in A smaller than x. Without confusion, we still use rank for simplicity.*

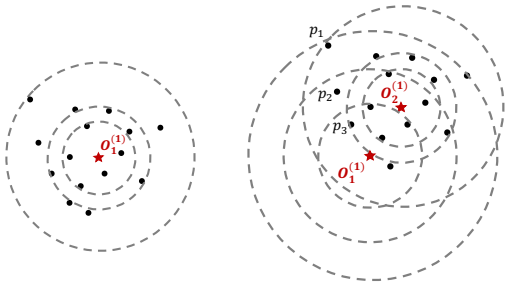
A series of rank prediction models $\mathcal{RP}_j^{(i)}$ (*a.k.a.* one-dimensional learned indexes) can be trained as follows: let $D_j^{(i)} = \{dist(p, O_j^{(i)})\}_{p \in C_i}$, then the training set is $\tilde{D}_j^{(i)} = \{(x, rank(x))\}_{x \in D_j^{(i)}}$. Let $\mathcal{RP}_j^{(i)}$ be a polynomial function of x, and the loss function $\mathcal{L}$ be the *squared error* as follows:

$$\mathcal{L} = \sum_{\tilde{D}_j^{(i)}} \left(\mathcal{RP}_j^{(i)}(x) - rank(x)\right)^2. \tag{3}$$

Then, the rank prediction models can be determined by minimizing $\mathcal{L}$ loss using the *gradient descent*. For example, if the degree of polynomial is 2, then $\mathcal{RP}_j^{(i)}(x) = ax^2 + bx + c$, where a, b, c are parameters to be learned. After $\mathcal{RP}_j^{(i)}$ is trained, we can get the approximate rank for any object $p \in \mathbb{U}$ with the distance in $D_j^{(i)}$ and the error can be easily

corrected by exponential search in $O(\log err)$ time, where err is the difference between the estimated rank and the correct rank. For object $p \in \mathbb{U}$ with the distance not in $D_j^{(i)}$, we can also use $\mathcal{RP}_j^{(i)}$ and exponential search to find its corresponding rank, *i.e.,* the rank of the first element larger than $dist(p, O_j^{(i)})$ in $D_j^{(i)}$ with the same time complexity.

In order to accelerate query processing, to be introduced in Section 5, LIMS divides ranks of data objects into N equal parts, *i.e.,* makes the data in the cluster covered by N super rings as evenly as possible. Fig. 2 gives two examples in a metric space with the Euclidean distance. In Fig. 2(a), the number of pivots and rings are specified as 1 and 3, respectively, so we partition all data objects in the cluster into 3 rings *w.r.t.* the pivot $O_1^{(1)}$ such that each ring includes about 5 data objects. In Fig. 2(b), the number of pivots and rings are specified as 2 and 3, respectively, so we partition all data objects into 6 rings, where 3 *w.r.t.* the pivot $O_1^{(1)}$ and 3 *w.r.t.* $O_2^{(1)}$. In this way, LIMS effectively avoids that lots of objects whose ring IDs defined in Equation (4) are same, while a few objects have different ring IDs.



(a) $m = 1, N = 3$ (b) $m = 2, N = 3$

Fig. 2: Examples of data partitioning

The ring ID, *i.e.,* which ring the data object is located, denoted as $rid_j^{(i)}$, can be computed by:

$$rid_j^{(i)}(p) = \left\lfloor \frac{rank\left(dist\left(p, O_j^{(i)}\right)\right)}{\left\lceil |D_j^{(i)}|/N \right\rceil} \right\rfloor. \quad (4)$$

When the above steps are completed, each data object is equipped with m ring IDs. Then, LIMS designs a novel pivot-based mapping function $\mathcal{M}$ to transform all data objects in the metric space into ordered sets with an associated binary relation $\leq$. The formal definitions are as follows.

Definition 6 (Pivot-based mapping function)**.** *Given a cluster C_i, $i = 1, \ldots, K$, and its corresponding pivots $O_j^{(i)}$, $j = 1, \ldots, m$, let $\{rid_1^{(i)}, \ldots, rid_m^{(i)}\}$ be a set of m ring ID functions. Then, for any data object $p \in \mathbb{U}$, we define a pivot-based mapping function $\mathcal{M}$ as follows:*

$$\mathcal{M}(p) = rid_1^{(i)}(p) \oplus rid_2^{(i)}(p) \oplus \cdots \oplus rid_m^{(i)}(p). \quad (5)$$

We call $\mathcal{M}(p)$ LIMS value of p.

Example 3. *Consider the cluster in Fig. 2(b). The ring IDs of p_1, p_2 and p_3 are $rid_1(p_1) = 2$ and $rid_2(p_1) = 2$, $rid_1(p_2) = 1$ and $rid_2(p_2) = 2$, $rid_1(p_3) = 0$ and $rid_2(p_3) = 1$, respectively. The ring IDs of other data objects can be computed similarly. According to the definition of pivot-based mapping function, $\mathcal{M}(p_1) = 2 \oplus 2, \mathcal{M}(p_2) = 1 \oplus 2$ and $\mathcal{M}(p_3) = 0 \oplus 1$.*

In order to build a learned index structure on LIMS values, we need to impose a binary relationship $\leq$ between LIMS values as follows:

Definition 7 (Binary Relation $\leq$)**.** *Let S be a multiset of LIMS values in cluster C_i with m pivots, $i = 1, \ldots, K$. Then, S can be ordered as follows:*

$$rid_1^{(i)}(p) \oplus \cdots \oplus rid_m^{(i)}(p) \leq rid_1^{(i)}(p') \oplus \cdots \oplus rid_m^{(i)}(p') \quad (6)$$

if and only if condition (1) or (2) is satisfied.

1) $\forall j \in \{1, \ldots, m\}$

$$rid_j^{(i)}(p) = rid_j^{(i)}(p'); \quad (7)$$

2) $\exists k \in \{1, \ldots, m\}$ such that

$$rid_j^{(i)}(p) = rid_j^{(i)}(p') \text{ for } j < k \text{ and } rid_k^{(i)}(p) < rid_k^{(i)}(p'), \quad (8)$$

where " $=$ " and " $<$ " in conditions is the order of natural numbers.

It's straightforward to prove that the binary relation $\leq$ in Definition 7 is well-defined, *i.e.,* it is reflexive, antisymmetric and transitive [23], so $(S, \leq)$ is an ordered set. In our implementation, we use the concatenation of ring IDs as LIMS value, which satisfies conditions in Definition 7 obviously.

Example 4. *Reconsider Example 3, LIMS values of p_1, p_2 and p_3 are sorted as follows: $\mathcal{M}(p_3) < \mathcal{M}(p_2) < \mathcal{M}(p_1)$.*

Now that all objects in the metric space are transformed into the corresponding ordered sets, we can sort them sequentially in ascending order of LIMS values, and store data in a number of disk pages with each page fully utilized. To quickly locate the addresses of data objects, LIMS learns a rank prediction model. Specifically, let $D^{(i)} = \{\mathcal{M}(p)\}_{p \in C_i}$ be a multiset of LIMS values and $\tilde{D}^{(i)} = \{(x, rank(x))\}_{x \in D^{(i)}}$, then a rank prediction model $\mathcal{RP}^{(i)}$ can be learned based on $\tilde{D}^{(i)}$. Similar to the loss function in Equation (3), we still try to minimize the squared error. After $\mathcal{RP}^{(i)}$ is trained, we can get the approximate rank (address) for any object $p \in \mathbb{U}$ with the LIMS value $\mathcal{M}(p) \in D^{(i)}$. The error can be easily corrected by exponential search that stops when the first occurrence of $\mathcal{M}(p)$ is found. For object $p \in \mathbb{U}$ with the LIMS value not in $D^{(i)}$, we can also use $\mathcal{RP}^{(i)}$ and exponential search to find its corresponding rank, *i.e.,* the position where the first occurrence of the element larger than $\mathcal{M}(p)$ in $D^{(i)}$.

4.3 Data Clustering and Pivot Selection

As mentioned in [7], one challenge of replacing traditional tree-like indexes with learned indexes is that it is difficult to approximate complex data distributions with a single model. If we construct a learned index on the whole dataset directly, the function to be learned would be very steep in regions where the data objects are dense, while very gentle for sparse regions. Such a complicated relationship can be fit by a neural network, but it will incur expensive query costs in practice. Based on the observation that real-life data are usually clustered and correlated [24], LIMS groups the underlying data into a number of clusters and maintains a learned index for each cluster instead. This

strategy gives two advantages: first, the data distribution of each cluster becomes simpler, which simplifies the model to be learned (*e.g.,* a polynomial function); second, simple models have lower query and (re-)construction costs. In this paper, we simply adopt the k-center algorithm [25], a simple yet effective algorithm that guarantees to return a 2-approximate optimal centroid set. We can follow the same steps to build LIMS on top of other clustering algorithms such as the kMeans [18], which can potentially further improve our approach. Different from a general clustering problem where the number of clusters can be flexible, the number of clusters used for indexing affects the index structure and search performance. Therefore, we propose a statistic to determine the number of clusters, to be discussed in detail in Section 5.4. For now, we assume that the number of clusters has been determined.

Once the clusters are obtained, LIMS picks a few data objects, named pivots [26], for each cluster and computes distances from each data object in the cluster to pivots. This strategy gives three advantages: first, we can use the triangle inequality on these pre-computed distances to prune the search space; second, learned indexes can be built naturally on these 1-dimensional distance values; third, re-distributing data with reference to well-chosen pivots may effectively ease the curse of dimensionality because metric search performance depends critically on the intrinsic dimension, a property relying on the data distribution itself, as opposed to the dimension where the data is represented [2], [19], [27], [28]. For example, the intrinsic dimension of a plane is two no matter if it is embedded in a high dimensional space. While LIMS is not dependent on the underlying pivot selection method, the number and locations of pivots have an influence on the retrieval performance. The more high-quality pivots there are, the more information they provide and the higher pruning power for query processing. However, the time taken for checking pruning conditions also increases (For LIMS, it mainly refers to the cost of generating search intervals, to be discussed in detail in Section 5). Extensive methods for pre-defining an optimal set of pivots have been proposed. A good survey can be found in [29]. In our implementation, we adopt the farthest-first-traversal (FFT) algorithm [25] because of its linear time and space complexity.

Algorithm 1: Range Query

Input: q: a query object; r: a query radius
Output: $\mathcal{S}$: objects in P satisfying $dist(p,q) \leq r$

1 Let $flag[K]$ be an array of all $TRUE$, $rid\_min[K][m], rid\_max[K][m]$ be arrays of all 0;
2 **for** *each cluster* C_i **do**
3 **for** *each pivot* $O_j^{(i)}$ **do**
4 **if** $dist(O_j^{(i)},q) > dist\_max_j^{(i)} + r$ *OR* $dist(O_j^{(i)},q) < dist\_min_j^{(i)} - r$ **then**
5 $flag[i] = FALSE$;
6 *break*;
7 **for** *each TRUE cluster* C_i **do**
8 **for** *each pivot* $O_j^{(i)}$ **do**
9 $r_{min} \leftarrow \max\{dist(O_j^{(i)},q) - r, dist\_min_j^{(i)}\}$;
10 $r_{max} \leftarrow \min\{dist(O_j^{(i)},q) + r, dist\_max_j^{(i)}\}$;
11 $rank'_{min}, rank'_{max} \leftarrow \mathcal{RP}_j^{(i)}(r_{min}), \mathcal{RP}_j^{(i)}(r_{max})$;
12 $rank_{min} \leftarrow ExpSearch(rank'_{min}, r_{min})$;
13 $rank_{max} \leftarrow ExpSearch(rank'_{max}, r_{max})$; /*assume $rank_{min} < rank_{max}$, otherwise discard C_i */;
14 $rid\_min[i][j] \leftarrow rid_j^{(i)}(rank_{min})$;
15 **if** $D_j^{(i)}[rank_{max}] = r_{max}$ **then**
16 $rid\_max[i][j] \leftarrow rid_j^{(i)}(rank_{max})$;
17 **else**
18 $rid\_max[i][j] \leftarrow rid_j^{(i)}(rank_{max} - 1)$;
19 generate LIMS-value ranges $\mathcal{R}$ based on rid_min and rid_max;
20 **for** *each range* $I \in R$ **do**
21 $lbound', ubound' \leftarrow \mathcal{RP}^{(i)}(I.left), \mathcal{RP}^{(i)}(I.right)$;
22 $lbound \leftarrow ExpSearch(lbound', I.left)$;
23 $ubound \leftarrow ExpSearch(ubound', I.right)$;
24 **if** $D^{(i)}[ubound] = I.right$ **then**
25 $ubound \leftarrow ExpSearch2(ubound, I.right)$;
26 **else**
27 $ubound \leftarrow ubound - 1$;
28 /* assume $lbound < ubound$, otherwise discard I */;
29 add to $\mathcal{P}$ all unvisited pages from $\lfloor lbound/\Omega \rfloor$ to $\lfloor ubound/\Omega \rfloor$;
30 add to $\mathcal{S}$ all objects saved in $\mathcal{P}$ satisfying $dist(p,q) \leq r$;
31 **return** $\mathcal{S}$

5 LIMS-based Query Processing

In this section, we proceed to present query processing algorithms for point, range and kNN queries using LIMS. Section 5.3 explains dynamic updates. Section 5.4 discusses the choice of K. Without loss of generality, we assume there is no two objects in P are totally same.

5.1 Range Query

Given a query object q, query radius r and dataset P, a range query is to retrieve all objects in P within the distance r of q, *i.e.,* $range(q,r) = \{p \in P | dist(p,q) \leq r\}$. Algorithm 1 outlines the range query processing, which consists of 4 sub-procedures. *TriPrune (Line 1-6):* prune irrelevant clusters by triangle inequality; *AreaLocate (Line 7-18):* determine affected areas of relevant clusters with the help of rank prediction functions $\mathcal{RP}_j^{(i)}$; *IntervalGen (Line 19):* generate search intervals on LIMS values; *PosLocate (Line 20-31):* locate positions of records in the disk by rank prediction functions $\mathcal{RP}^{(i)}$.

TriPrune (Line 1-6). The algorithm starts by computing the distances between the query q and the pivots, and then utilizes triangle inequality property in the metric space to prune a number of irrelevant clusters and thus accelerate the search. Specifically, according to triangle inequality, an object p in cluster C_i may fall into the query range, it must satisfy the following: $\forall j \in \{1,\ldots,m\}$,

$$dist(O_j^{(i)},q) - r \leq dist(O_j^{(i)},p) \leq dist(O_j^{(i)},q) + r. \quad (9)$$

Recall that LIMS also maintains both maximum distance $dist\_max_j^{(i)}$ and minimum distance $dist\_min_j^{(i)}$ of the cluster, hence, for any object p in the cluster, we must have: $\forall j \in \{1,\ldots,m\}$,

$$dist\_min_j^{(i)} \leq dist(O_j^{(i)},p) \leq dist\_max_j^{(i)}. \quad (10)$$

Combing Equations (9) and (10), we can derive that a cluster $C_i, i = 1,\ldots,K$ is needed to be further checked if and only if the following condition (Line 4) is satisfied:

$$\bigwedge_{j=1}^{m} [dist(O_j^{(i)}, q) \leq dist\_max_j^{(i)} + r \\ \wedge\ dist(O_j^{(i)}, q) \geq dist\_min_j^{(i)} - r]. \quad (11)$$

AreaLocate (Line 7-18). For a cluster C_i that needs to be searched, LIMS first determines the affected areas in the metric space according to the following equations:

$$r_{min}{}_j^{(i)} = \max\{dist(O_j^{(i)}, q) - r, dist\_min_j^{(i)}\}; \quad (12)$$

$$r_{max}{}_j^{(i)} = \min\{dist(O_j^{(i)}, q) + r, dist\_max_j^{(i)}\}, \quad (13)$$

where $j = 1, \ldots, m$. Then, LIMS invokes corresponding rank prediction models $\mathcal{RP}_j^{(i)}$ to predict the min and max ranks of affected areas and the error is fixed via exponential search (Line 11-13). Min ring ID can be easily calculated by calling $rid_j^{(i)}$ function, while the max ring ID is divided to 2 cases in order to narrow down search range as much as possible (Line 14-18).

IntervalGen (Line 19). Instead of doing intersection of several candidate sets in the metric space via costly distance computations, LIMS reduces the search space by doing intersection of LIMS-value intervals directly. Specifically, for a relevant cluster $C_i, i = 1, \ldots, K$, let $L_j^{(i)} = \{rid\_min[i][j], rid\_min[i][j] + 1, \ldots, rid\_max[i][j]\}$, $j = 1, \ldots, m-1$ and $L_m^{(i)} = \{rid\_min[i][m], rid\_max[i][m]\}$. Depth-first search (DFS) is run on a directed acyclic graph (DAG) composed of vertexes $\cup_{j=1}^{m} L_j^{(i)}$ and fully connected edges from $L_j^{(i)}$ to $L_{j+1}^{(i)}$, $j = 1, \ldots, m-1$, to find all paths from $L_1^{(i)}$ to $L_m^{(i)}$. These paths form a total of $\prod_{j=1}^{m-1} |L_j^{(i)}|$ LIMS-value search ranges. It is through *IntervalGen* that the number of data objects to be accessed is significantly reduced, while the pruning cost remains low. Here is an example of this step.

Example 5. *Consider a cluster with $m = 3$ pivots and $N = 10$ rings. Suppose it is relevant to a range query such that the minimum and maximum ring IDs of affected region w.r.t. the 1st, 2nd and 3rd pivots are $rid\_min_1 = 2$, $rid\_max_1 = 4$, $rid\_min_2 = 6$, $rid\_max_2 = 8$, $rid\_min_3 = 1$, $rid\_max_3 = 5$, respectively, i.e., $L_1 = \{2, 3, 4\}, L_2 = \{6, 7, 8\}$ and $L_3 = \{1, 5\}$. Then, LIMS-value search ranges can be computed by running DFS on the DAG shown in Figure 3. The final search ranges are the union of* $[2 \oplus 6 \oplus 1, 2 \oplus 6 \oplus 5]$, $[2 \oplus 7 \oplus 1, 2 \oplus 7 \oplus 5]$, $[2 \oplus 8 \oplus 1, 2 \oplus 8 \oplus 5]$, $[3 \oplus 6 \oplus 1, 3 \oplus 6 \oplus 5]$, $[3 \oplus 7 \oplus 1, 3 \oplus 7 \oplus 5]$, $[3 \oplus 8 \oplus 1, 3 \oplus 8 \oplus 5]$, $[4 \oplus 6 \oplus 1, 4 \oplus 6 \oplus 5]$, $[4 \oplus 7 \oplus 1, 4 \oplus 7 \oplus 5]$ *and* $[4 \oplus 8 \oplus 1, 4 \oplus 8 \oplus 5]$.

PosLocate (Line 20-31). For each search range, the position of lower bound in the disk can be easily located via rank prediction model $\mathcal{RP}^{(i)}$ and exponential search. Ω is the maximum number of objects each page can hold. The upper bound is divided into 2 cases to make sure the results correct. For example, it's possible that different objects have the same LMIS value, so we find the last occurrence of the LIMS value via another exponential search, denoted as $ExpSearch2$, to guarantee no objects missed (*i.e.,* no false negatives). Finally, all retrieved objects are further refined in the refinement step (Line 30), where exact distance computations are performed.

Correctness. To prove that Algorithm 1 offers exact answers for a range query, we need to show that 1) all data objects in the result set satisfy $dist(p, q) \leq r$, *i.e.,* no false positives; and 2) no objects satisfying $dist(p, q) \leq r$ are missed *i.e.,* no false negatives. Obviously, no false positives can be returned because a final refinement step is applied, where the exact distance computations are performed to guarantee all data objects in the result set satisfy $dist(p, q) \leq r$. We prove there are no false negatives by contradiction. Assume that there exists an object $p \in C_i$ that satisfies $dist(p, q) \leq r$ but is not returned. According to the triangle inequality, we know that $\forall j \in \{1, \ldots, m\}$, $dist(O_j^{(i)}, q) \leq dist(O_j^{(i)}, p) + dist(p, q) \leq dist\_max_j^{(i)} + r$. Similarly, we can derive $dist(O_j^{(i)}, q) \geq dist\_min_j^{(i)} - r$. It indicates that the cluster C_i (and thus p) will not be pruned in *TriPrune* step (Equation (11)). Since $dist(O_j^{(i)}, p) \leq dist(O_j^{(i)}, q) + r$ and $dist(O_j^{(i)}, p) \leq dist\_max_j^{(i)}$, we know that $dist(O_j^{(i)}, p) \leq \min\{dist(O_j^{(i)}, q) + r, dist\_max_j^{(i)}\} = r_{max}{}_j^{(i)}$ (Equation (13)). Similarly, we can derive $dist(O_j^{(i)}, p) \geq r_{min}{}_j^{(i)}$ (Equation (12)). Although the rank prediction models $\mathcal{RP}_j^{(i)}$ have an error, the error is fixed by exponential search. Thus, we have $rid\_min[i][j] \leq rid_j^{(i)}(p) \leq rid\_max[i][j]$ after *AreaLocate* step. Denote by $l_j^{(i)}$ the value in $L_j^{(i)}$ satisfying $l_j^{(i)} = rid_j^{(i)}(p)$, then $\mathcal{M}(p)$ can be written as $l_1^{(i)} \oplus \cdots \oplus l_m^{(i)}$. According to the procedure of generating LIMS-value search ranges in *IntervalGen* step, there must exist a range $[l_1^{(i)} \oplus \cdots \oplus l_{m-1}^{(i)} \oplus rid\_min[i][m], l_1^{(i)} \oplus \cdots \oplus l_{m-1}^{(i)} \oplus rid\_max[i][m]]$. Based on the binary relation in Definition 7, we know that $\mathcal{M}(p)$ falls in the above search range, and thus in the union of all search ranges after *IntervalGen* step. The rank prediction model $\mathcal{RP}^{(i)}$ in *PosLocate* step may have an error, but the error is fixed by exponential search. Hence, exact distance computations between all objects in the search ranges and the query object are performed. p will be returned, which leads to a contraction. Therefore, Algorithm 1 can answer the range query correctly.

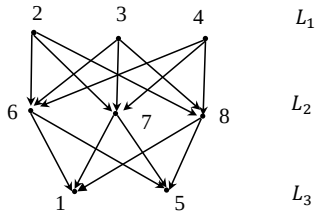


Fig. 3: An illustration of finding LIMS-value search ranges

Query Cost. *TriPrune* takes $O(mKD)$ time, where D represents the cost of distance computation. The cost of *AreaLocate* depends on rank prediction models. We use $O(\mathcal{RP})$ and $O(\log err)$ to denote the prediction cost of $\mathcal{RP}_j^{(i)}$ or $\mathcal{RP}^{(i)}$, and the cost of fixing error incurred by models via exponential search, respectively. Hence, we need $O(mK(\mathcal{RP} + \log err))$ time to locate affected areas of relevant clusters. The cost of *IntervalGen* is from running DFS on DAG, which takes $O(|V| + |E|)$ time, where $|V|$ is the number of vertexes and $|E|$ is the number of edges. Similar to *AreaLocate* subprocedure, *PosLocate* takes $O(|\mathcal{R}|(\mathcal{RP} + \log err))$ time, where $|\mathcal{R}|$ is the number of LIMS-value search ranges. Generally, $|\mathcal{R}| > |E| > |V|$. In addition, we need to access disk pages in $\mathcal{P}$ to refine and retrieve finial result, which

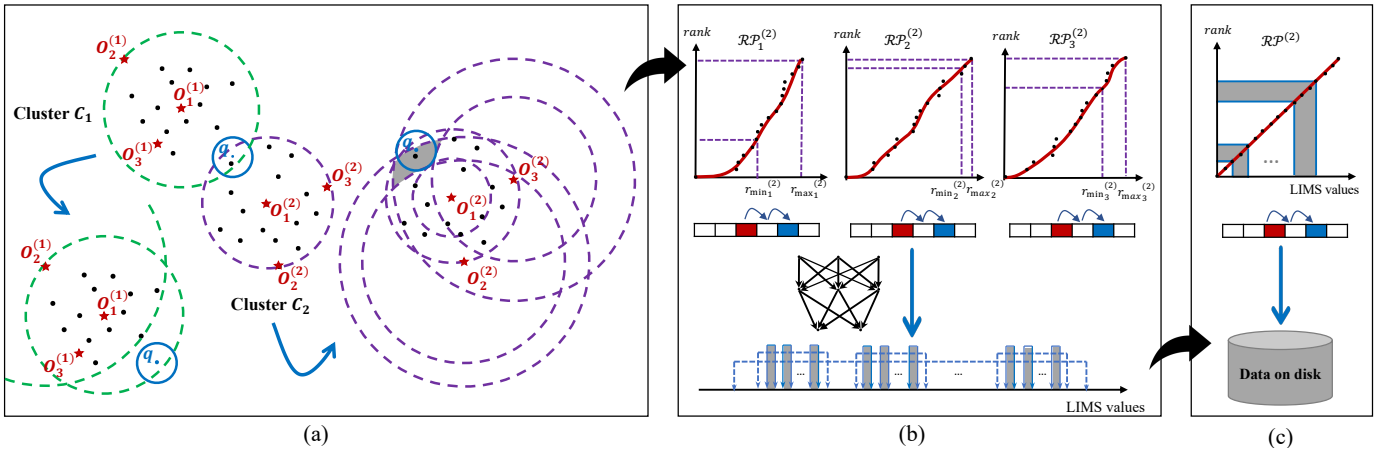


Fig. 4: An example of range query based on LIMS

takes $O(|\mathcal{P}|\Omega D)$ time. Therefore, the overall query time is $O((mK + |\mathcal{R}|)(\mathcal{RP} + \log err) + (mK + |\mathcal{P}|\Omega)D)$.

Example 6. *Figure 4 is an example of LIMS-based range query. It is straightforward to see that cluster C_1 does not satisfy $dist(O_2^{(1)}, q) \leq dist\_max_2^{(1)} + r$, and thus can be discarded from further processing directly, while cluster C_2 cannot be discarded. As for C_2, LIMS first inputs the min and max boundaries* (i.e., *purple dashed lines in Figure 4(a)) of affected areas into corresponding rank prediction models $\mathcal{RP}_j^{(2)}$ and ring ID functions $rid_j^{(2)}$, $j = 1, 2, 3$. Then, LIMS runs DFS on DAG to transform the intersection of several candidate sets in the metric space (the grey region in Figure 4(a)) to the intersection of intervals (the grey region in Figure 4(b)), to significantly reduce the number of distance computations and page accesses. Next, positions of objects on disk are estimated by the rank prediction function $\mathcal{RP}^{(2)}$. Finally, candidate objects are retrieved and a refinement step is applied (Figure 4(c)).*

5.2 *k*NN Query

In LIMS, a kNN query is processed by conducting a series of range queries with increasing search radius $r = \Delta r$, $2 \cdot \Delta r$, $3 \cdot \Delta r, \ldots$ until k nearest neighbors are found, where Δr is a given small initial radius. Algorithm 2 outlines the LIMS-based kNN query. Given a query q and k, a range query with a radius $r = \Delta r$ is issued at the beginning. To answer this query, LIMS invokes Algorithm 1 (Line 6). $\mathcal{Q}$ is a max priority queue to record the candidate k nearest neighbors. If the distance from the retrieved object p to the query object q is smaller than the current furthest distance in $\mathcal{Q}$, LIMS will extract the object with the maximum distance value from $\mathcal{Q}$ and insert p into $\mathcal{Q}$ (Line 11-14). The search stops if and only if the furthest object in the current priority queue falls within the current query range and further expansion of the query radius does not change the answer set (Line 4-8). Otherwise, it enlarges the query radius by Δr (Line 3) and invokes Algorithm 1 till the termination conditions are satisfied. LIMS also maintains an array to record whether a page has been processed. If so, it will be skipped during the next range query to avoid repeated accesses (Line 10). The number of range query calls depends on Δr and the distance between q and the k-th NN in the dataset, which can be determined by $\lceil \frac{dist(\mathcal{Q}[1], q)}{\Delta r} \rceil + 1$.

Algorithm 2: kNN Query

Input: q: a query point; k: a positive integer; Δr: a positive number
Output: $\mathcal{Q}$: top k nearest neighbors to q

1 $r \leftarrow 0$, $flag \leftarrow FALSE$, $\mathcal{Q}$ is a max priority queue on k objects initialized to ∞;
2 **while** $flag = FALSE$ **do**
3 $\quad r \leftarrow r + \Delta r$;
4 $\quad$ **if** $dist(\mathcal{Q}[1], q) < r$ **then**
5 $\quad\quad flag \leftarrow TRUE$;
6 $\quad$ call $range(q, r)$ to get pages set $\mathcal{P}$;
7 $\quad$ call $check(\mathcal{P})$;
8 $\quad$ **return** $\mathcal{Q}$;
9 **procedure** $check(\mathcal{P})$:
10 **for** *each unvisited page* $P \in \mathcal{P}$ **do**
11 $\quad$ **for** *each point* $p \in P$ **do**
12 $\quad\quad$ **if** $dist(\mathcal{Q}[1], q) > dist(p, q)$ **then**
13 $\quad\quad\quad$ Extract-Max($\mathcal{Q}$);
14 $\quad\quad\quad$ Insert($\mathcal{Q}, p$);

Remark 4. *Intuitively, the initial radius Δr affects the number of range query calls, and thus the query efficiency. We observed that a small Δr would not degrade the query performance severely. As we know, the extra cost of using range queries with increasing radius to answer a kNN query mainly comes form 1) traversing the index multiple times and 2) accessing the same pages multiple times. In LIMS, a query is processed by a function invocation in $O(1)$ time instead of $O(\log n)$ time in tree-like indexes, which makes the cost of multiple traverses negligible. Besides, LIMS would not access visited pages. However, a too large initial radius can degrade the query performance. Therefore, we recommend a small initial search radius, which can be simply estimated based on distances between pairs of data objects sampled from the dataset.*

5.3 Updates

LIMS allows both insertions and deletions of data. To support the efficient insertion, LIMS maintains a sorted array for each cluster in ascending order of distance values to the centroid. Given an object p to be inserted, LIMS first runs a point query to find if there is a page containing p. If so, the algorithm terminates immediately. Otherwise, LIMS finds the cluster closest to p and inserts p into the sorted array of this cluster. All newly inserted data are arranged in a number of pages with each page fully utilized.

During query processing, LIMS uses the triangle inequality and exponential search to retrieve these inserted objects matching query filters. Given an object p to be deleted, LIMS first runs a point query for p. If p is found, it is marked as 'deleted'. Then, LIMS updates maximum and minimum distances to each pivot of the cluster p belongs to. Due to the space limitation, we omit the pseudocode here. Such an easy update strategy is effective and efficient because 1) LMIS partitions the data space into many clusters that amortize the additional search time on the sorted arrays; 2) LIMS maintains an index for each cluster separately, which allows for partially rebuilding the index, *i.e.,* retraining rank prediction models for certain clusters, especially if deletions invalidate a cluster or insertions result in too much overlapping between some clusters. The procedure of retraining rank prediction models is the same as that of training them; 3) the short index construction (reconstruction) time of LIMS ensures the feasibility of this strategy in practice (Section 6).

5.4 Last Piece

The last piece of LIMS is to determine the number of clusters. The optimal number is related not only to the data distribution but also to the query workload. However, the query workload is not available during data clustering and we do not assume a known query workload in this paper. Therefore, an alternative should be developed to predefine the number of clustering parameter K. Recall that the goal of clustering in LIMS is to decompose complex and potentially correlated data into a few clusters. Ideally, these clusters are independent and each can be accurately fit by a linear function. However, in most real-world use cases, the data do not follow such a perfect pattern. On the one hand, the overlapping between clusters may incur extra pruning and refinement costs. On the other hand, uneven intra-cluster distribution incurs more arithmetic and comparison operations. In order to pick a K to avoid or reduce such overhead, we introduce *overlap rate* (OR) and *mean absolute error* (MAE) to evaluate the goodness of clustering. OR quantifies the extent of overlapping among clusters. It can be computed as:

$$OR = \frac{1}{K(K-1)} \sum_{i=1}^{K} \sum_{i' \neq i} \frac{r^{(i,i')}}{dist\_max_1^{(i)}}. \quad (14)$$

Without confusion, we use $dist\_max_1^{(i)}$ to represent the distance of the furthest object in ith cluster from the centroid, and $r^{(i,i')}$ is the length of the overlapping area computed as:

$$r^{(i,i')} = \min\{dist(O_1^{(i)}, O_1^{(i')}) + dist\_max_1^{(i')}, dist\_max_1^{(i)}\} - \max\{(dist(O_1^{(i)}, O_1^{(i')}) - dist\_max_1^{(i')}, dist\_min_1^{(i)}\}. \quad (15)$$

MAE quantifies the quality of the linear regression fit for each cluster. It can be computed as:

$$MAE = \frac{1}{m|P|} \sum_{i=1}^{K} \sum_{j=1}^{m} \sum_{\tilde{D}_j^{(i)}} |a_j^{(i)} x + b_j^{(i)} - rank(x)|, \quad (16)$$

where $|P|$ is the cardinality of dataset, $\mathcal{RP}_j^{(i)}(x) = a_j^{(i)} x + b_j^{(i)}$ are linear rank prediction models learned from $\tilde{D}_j^{(i)}$.

TABLE 2: Summary of datasets

Datasets	Cardinality	Dim.	Ins.	Metric
Color	1,281,167	32	4.2	L_2-norm
Forest	565,892	6	1.5	L_2-norm
GaussMix	5,10,20,40,60,80M	2,4,8,12,16	6.2	L_2-norm
Skewed	10M	2,4,8,12,16	5.6	L_1-norm
Signature	100K	65	36	*Edit distance*

We model the overhead as $OR + \lambda MAE$, where $\lambda > 0$ is a user-defined weight. Inspired by the elbow method [30], we choose the *elbow* or *knee* of a curve as the clustering number to use, *i.e.,* a point where adding more clusters will not give much better modeling of the data. The number estimated by the techniques described above turns out to be very close to the optimal number of clusters observed in practice, as shown in Section 6.

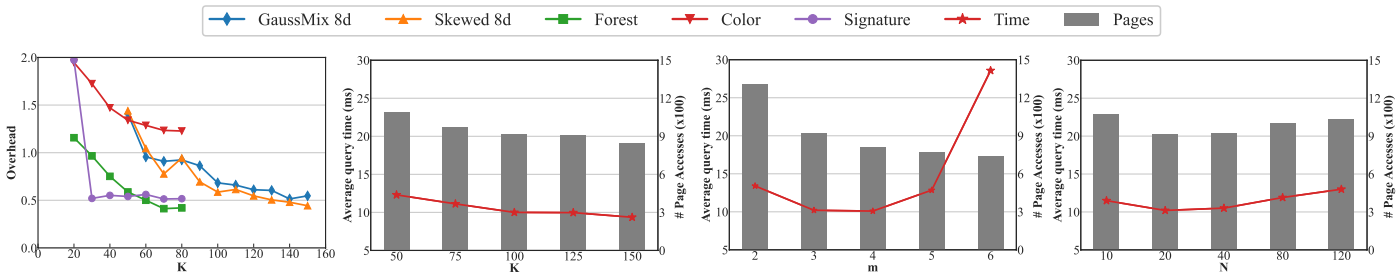
6 Experiments

In this section, we present the results of an in-depth experimental study on LIMS. We implement LIMS [1] and associated similarity search algorithms in C++. All experiments are conducted on a computer running 64-bit Ubuntu 20.04 with a 2.30 GHz Intel(R) Xeon(R) Gold 5218 CPU, 254 GB RAM, and an 8.2 TB hard disk.

6.1 Experimental Settings

Datasets. We employ two real-world datasets, namely, *Color Histogram*[2] and *Forest Cover Type*[3] following the experimental settings of ML index [11]. *Color Histogram* contains 1,281,167 32-dimensional image features extracted from the ImageNet[4] dataset. *Forest Cover Type* is collected by US Geological Survey and US Forest Service. It includes 565,892 records, each of which has 12 cartographic variables. We extract 6 quantitative variables of them as our data object. Following the experimental settings of iDistance [17], we generate 2, 4, 8, 12, 16-dimensional *GaussMix* datasets. Every dataset contains up to 80 million points (5.36GB in size) sampled from 150 normal distributions with the standard deviation of 0.05 and randomly determined means. Default settings are underlined. Without loss of generality, L_2 norm is utilized to the above datasets. Following the experimental settings of RSMI [10], we create 2, 4, 8, 12, 16-dimensional *Skewed* datasets. They are generated from uniform data by raising the values in each dimension to their powers, *i.e.,* a randomly generated data point are converted from $(x_1, x_2, \ldots, x_d)$ to $(x_1, x_2^2, \ldots, x_d^d)$. The size of each dataset is 10 million and L_1 norm is employed. Without loss of generality, all the data values of the above datasets are normalized to the range $[0, 1]$. Following the experimental settings in [31], we also generate a *Signature* dataset, where each object is a string with 65 English letters. We first obtain 25 'anchor signatures' whose letters are randomly chosen from the alphabet. Then, each anchor produces a cluster with 4,000 objects, each of which is obtained by randomly

1. https://github.com/learned-index/LIMS
2. https://image-net.org/download-images
3. https://www.kaggle.com/c/forest-cover-type-prediction/data
4. http://image- net.org/download- images



(a) Estimated effect of K (b) Effect of K on range query (c) Effect of m on range query (d) Effect of N on range query

Fig. 5: Effect of parameters

changing x positions in the corresponding anchor signature to other random letters, where x is uniformly distributed in the range $[1, 30]$. The *edit distance* is used to compute the distance between two signatures. Table 2 summarizes the statistics of the datasets.

Competitors. We compare LIMS with three representative multi-dimensional learned indexes as mentioned in Section 2, *i.e.,* ZM [8], ML [11], LISA [9], and three traditional indexes, *i.e.,* R*-tree [32], M-tree [33] and SPB-tree [34] [35]. ZM and ML are in-memory indexes, so we adapt them to the disk by storing data in ascending order of their z-order/mapped values in a number of pages with each page fully utilized. For LISA, no open-source C++ code is available, so we implement it following Python version implementation. For R*-tree, M-tree and SPB-tree, we use the original implementations. In addition, to study the effectiveness of learning components in LIMS, we design a method called non-learned index for metric spaces (N-LIMS) by replacing rank prediction models in LIMS with the traditional B$^+$-trees [36]. All competitors are configured to use a fixed disk page size of 4KB.

Evaluation Metrics. Four metrics are used to evaluate the performance of indexes: the average number of page accesses, the average query time, indexing time and index size. We randomly select 200 objects from each dataset and repeat each experiment 20 times to get average results.

6.2 Effect of Parameters

We first study the effect of parameters, including the number of clusters K, the number of pivots m and the number of rings N, to optimize LIMS-based similarity search, as summarized in Fig. 5. Only one parameter varies whereas the others are fixed to their default values in every experiment. By default, the selectivity of range query, *i.e.,* the fraction of objects within the query range from the total number of objects, is set to 0.01%. The k of kNN query is 5. Degrees of $\mathcal{RP}_j^{(i)}$ and $\mathcal{RP}^{(i)}$ are 20 and 1. m and N are 3 and 20. K is determined according to the method described in Section 5.4. We set $K = 100$ for 10M 8d *GaussMix* and 8d *Skewed*, $K = 50$ for *Color Histogram*, *Forest Cover Type* and *Signature*.

6.2.1 Effect of K

Fig. 5(a) plots the criterion $OR + \lambda MAE$ versus K on both real and synthetic data, where $K = 20, 30, \ldots, 150$ and λ is set to $1/\max\{MAE(K)\}$. We can see that different datasets have different elbow points. In order to show the estimation is close to the actual optimal number of clusters, we also plot the actual average query time and the number of page accesses for range query by varying K. Due to the space limitation, we only report the performance on 10M 8d *GaussMix* dataset in Fig. 5(b). It can be observed that query time decreases slowly after $K = 100$, which is consistent with the recommended choice of K in Fig. 5(a). Therefore, we set $K = 100$ for this dataset.

6.2.2 Effect of m

Fig. 5(c) reports the query performance on 10M 8d *GaussMix* dataset by varying the number of pivots m. We can see that increasing the number of pivots always reduces (or at least does not increase) the number of page accesses. This is expected because the intersection of metric regions defined by more pivots is always smaller than (or at least equal to) the intersection of metric regions defined by fewer pivots. As discussed in Section 4.3, the more pivots, the stronger pruning ability. This observation can also be proven using Equation (11). However, the average query time decreases when using up to four pivots, and then increases progressively. That is because the cost for filtering unqualified objects grows as well with more pivots. The best number of pivots for a metric index is a trade-off between filter cost and scanning cost. Therefore, the default value for m is set to 3 unless otherwise stated.

6.2.3 Effect of N

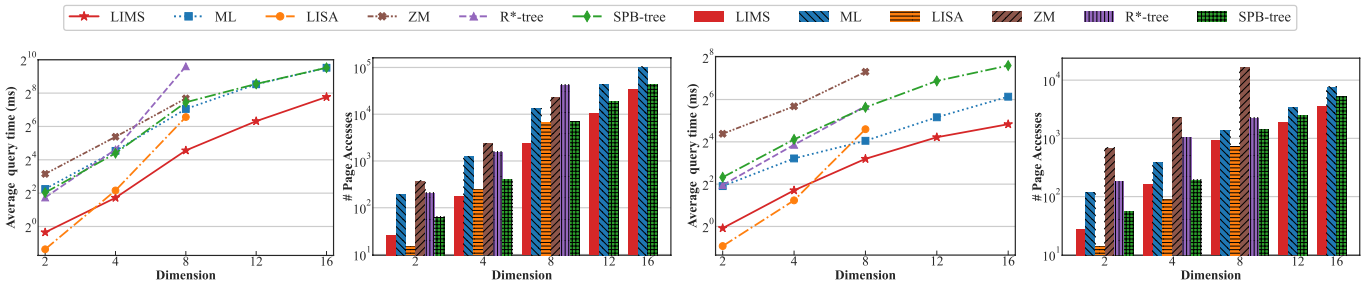
Fig. 5(d) reports the query performance on 10M 8d *GaussMix* dataset by varying the number of rings N. For the same reason as above, the average query time presents a down and up trend whereas the lowest value turns out at $N = 20$.

6.3 Range Query Performance

In this subsection, we study the performance of LIMS, ML, LISA, ZM, R*-tree, M-tree and SPB-tree on range query from different angles, as summarized in Fig. 6, 7 and 8.

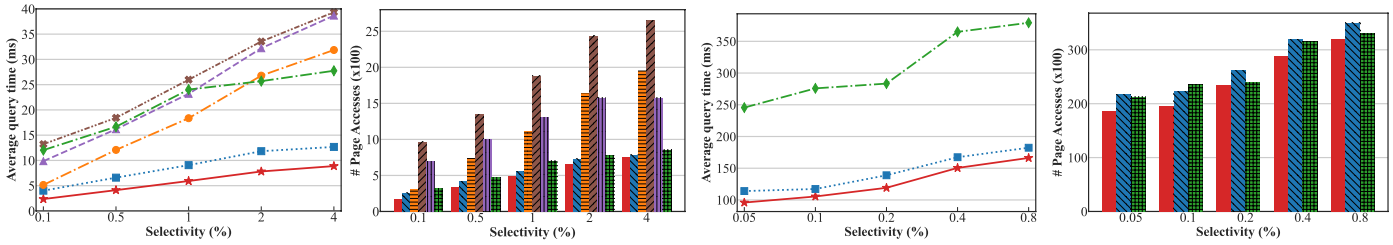
6.3.1 Performance with dimensionality

The first set of experiments studies the average query time and the number of page accesses under different dimensionalities. Fig. 6(a)(b) and Fig. 6(c)(d) report the results on *Skewed* and *GaussMix* datasets, respectively. From the figures, we have the following observations: 1) The average query time of all methods increases with dimensionality, but LIMS, ML and SPB-tree grow much slower than others, suggesting that data clustering and pivot-based data transformation techniques are effective in alleviating the curse of dimensionality. The coordinate-based methods, *i.e.,* LISA, ZM and R*-tree, degrade rapidly with dimensionality and even do not work, hence we do not report their results



(a) Query time on *Skewed* (b) # Page accesses on *Skewed* (c) Query time on *GaussMix* (d) # Page accesses on *GaussMix*

Fig. 6: Range query performance with dimensionality



(a) Query time on *Forest Cover Type* (b) # Page accesses on *Forest* (c) Query time on *Color* (d) # Page accesses on *Color*

Fig. 7: Range query performance with selectivity

after $8d$. 2) LIMS is slightly slower than LISA when $2d$ and when $4d$ on *GaussMix*, but LIMS offers the best performance in a higher range of dimensions on both datasets. The reason is that metric-space indexes can only use the four distance properties to prune the search space, while LISA can easily locate the cells that overlap with the query range by coordinates. Fewer assumptions about the data result in poorer pruning and slightly larger query costs in the low-dimensional case. However, the filtering cost of LISA increases exponentially with dimensionality and it does not work when the dimension is greater than 8. Note that on $8d$ *GaussMix*, even though LIMS has higher numbers of page accesses than LISA, it is still faster due to the low filter cost. On *Skewed*, LIMS begins to have a lead advantage since $4d$ because LIMS can be applied to a metric space with any distance metrics naturally (*e.g.*, L_1 norm here) and guarantees a few false positives and fast query response. 3) LIMS is always better than ML in both the query time and page accesses. This is intuitive since ML transforms different objects with equi-distance from the pivot into the same 1-dimensional value, while LIMS integrates the pruning abilities from multiple pivots. In addition, with the help of a well-defined pivots-based mapping, LIMS maps nearby data into the compact region, which further reduces the search region, and thus fewer objects to be accessed in the refinement step. 4) LIMS outperforms the learned index, ZM, by over an order of magnitude. This is because LIMS uses clusters and LIMS values to organize objects into compact regions, while ZM uses z-order curve to organize data, which incurs too many false positives, and thus large page accesses and distance computations. 5) LIMS is always better than traditional indexes. The main reason is that query processing with a traditional index requires traversing many tree nodes multiple times, which is time-consuming. M-tree is omitted since it is considerably worse than others. 6) The performance of all indexes on *GaussMix* is better than that on *Skewed* due to the simple distribution.

6.3.2 *Performance with selectivity*

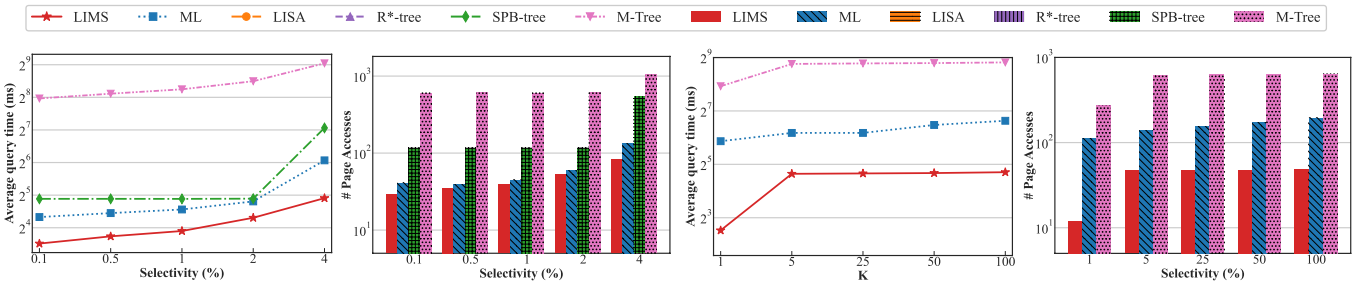
The second set of experiments studies the average query time and the number of page accesses with different selectivity. Fig. 7(a)(b) show the results on *Forest* dataset by varying selectivity from 0.1% to 4%. Fig. 7(c)(d) show the results on *Color* dataset by varying the selectivity from 0.05% to 0.8%. N on these small datasets is cut in half. From the figures, we have the following observations: 1) both query time and page accesses among indexes grow with the selectivity since more objects are queried. 2) LIMS achieves better performance under all selectivities by at least 1.7X and up to 4.4X faster. Consistent high efficiency of LIMS shows the robustness for range queries in varying settings. 3) On *Color Histogram*, the advantage in fewer page accesses shows the potential advantages that LIMS can achieve over ML for processing similarity search with costly distance metrics.

6.3.3 *Performance in metric spaces*

The third set of experiments studies the average query time and the number of page accesses in a metric space. We compare LIMS with SPB-tree and M-tree. We also extend ML to the metric dataset by replacing the KMeans in ML with the k-center algorithm. Other competitors are omitted since they are not applicable for metric spaces. Fig. 8(a)(b) report the results on *Signature* dataset. Clearly, LIMS has a decided advantage over all competitors under all selectivities, where LIMS is around 20X faster than M-tree and the page accesses are at least 12X fewer. This is expected because of expensive and unavoidable costs to traverse the tree structure in traditional index structures, and poor pruning powers in ML.

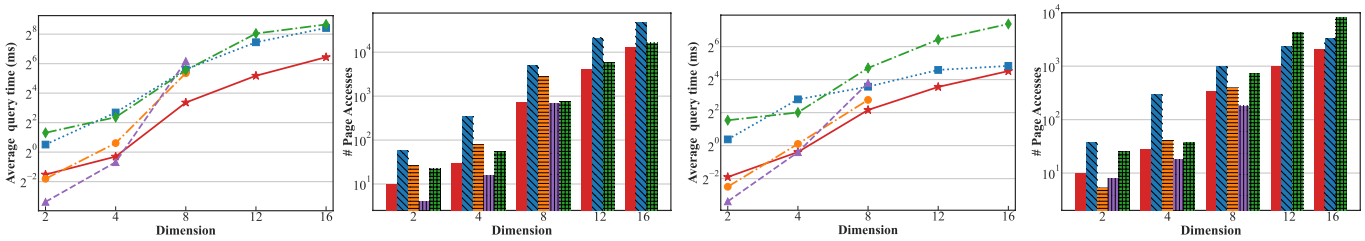
6.4 *k*NN Query Performance

In this subsection, we study the performance of LIMS, ML, LISA, R*-tree, M-tree and SPB-tree on kNN query, as summarized in Fig. 8, 9, and 10. ZM is excluded because it does not support kNN query.



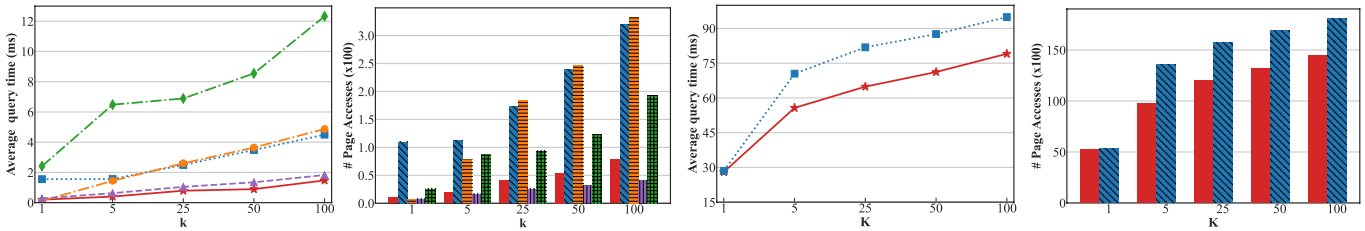
(a) Time of range queries (b) # Page accesses of range queries (c) Time of kNN queries (d) # Page accesses of kNN queries

Fig. 8: Range and kNN query performance on *Signature*



(a) Query time on *Skewed* (b) # Page accesses on *Skewed* (c) Query time on *GaussMix* (d) # Page accesses on *GaussMix*

Fig. 9: kNN query performance with dimensionality



(a) Query time on *Forest* (b) # Page accesses on *Forest* (c) Query time on *Color* (d) # Page accesses on *Color*

Fig. 10: kNN query performance with k

6.4.1 Performance with dimensionality

Fig. 9(a)(b) and Fig. 9(c)(d) report the average query time and the number of page accesses on *Skewed* and *GaussMix* datasets, respectively. From the figures, we have the following observations: 1) R*-tree works best in low dimensions, but it is less effective than LIMS with increased dimensionality. 2) The relative performance of LIMS and ML on kNN queries is similar to their performance on range queries, since both techniques follow a similar search region expansion paradigm. 3) Even though LISA uses a model learned from the data distribution to estimate the initial radius, it still suffers from too many unnecessary page accesses. The reason is that once the number of retrieved objects is smaller than k, LISA will issue a range query with a larger radius from the scratch, leading to the same page accessed repeatedly. When the estimated initial radius is large, it also incurs many page accesses.

6.4.2 Performance with k

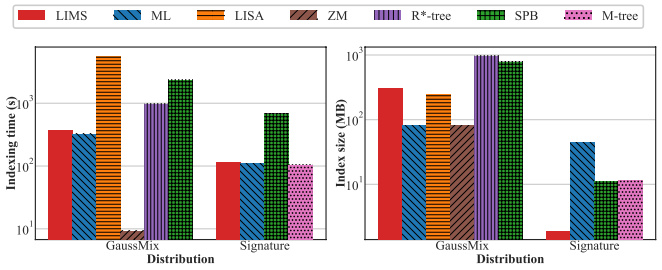
Fig. 10(a)(b) and Fig. 10(c)(d) show the performance when varying the value of k in $\{1, 5, 25, 50, 100\}$ on *Forest* and *Color* datasets, respectively. LIMS is the fastest except for 1NN on *Forest*, where LISA is slightly faster owing to a proper radius estimation. If setting Δr in LIMS to the value recommended by LISA. LIMS can achieve better performance ($0.11ms$ v.s. $0.12ms$). However, the training time of model used to estimate Δr in LISA is long, which is not favorable for frequent insertion/deletion operations and query pattern changes. As k grows, LISA becomes not comparable due to the aforementioned reasons. The results of SPB-tree on *Color* is omitted because it is drastically slower. Consistent high efficiency of LIMS indicates that it scales to large k values.

6.4.3 Performance in metric spaces

Fig. 8(c)(d) present the performance on *Signature* dataset. As expected, LIMS again yields the fastest query time and fewest page accesses. The curve becomes gentle after $k = 5$ because many signatures share the same edit distance to a given signature.

6.5 Indexing Time and Index Size

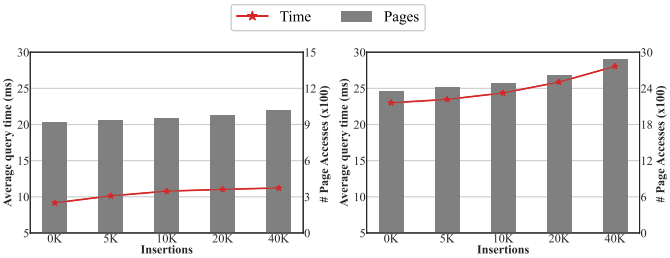
In this subsection, we report the indexing time and index size. Due to the space limitation, we only report the results on $8d$ 10M *GaussMix* and *Signature*. As Fig. 11(a) shown, LIMS does not suffer from long training time, a common dilemma faced by learned indexes. On 10M $8d$ *GaussMix*, LIMS is 15.5X faster than LISA ($368s$ v.s. $1.6h$), which makes LIMS easy to rebuild and ensures simple update operations effective in practice. As described in Section 5.3, when the query performance does not degrade severely, we can partially rebuild LIMS by retraining rank prediction models



(a) Indexing time (b) Index size

Fig. 11: Indexing time and index size

for some clusters. Retraining the index of a cluster only takes an average of $0.5s$. LIMS has a longer construction time than ZM because of expensive distance computations in metric spaces. However, LIMS has a decided advantage over ZM regardless of the dataset or dimension. Fig. 11(b) shows the index size. The index size of LIMS is only 1/3 that of R*-tree on *GaussMix* and 1/23 that of ML on *Signature*, respectively, since traditional indexes have to store a large number of internal nodes and ML have to store multiple stages of learned models. LIMS has a slightly larger index size than other learned indexes on *GaussMix* because LIMS is a metric-space index that needs to store many pre-computed distances between pivots and data objects. However, it is accepted because a few extra distance values are relatively insignificant compared to the complex and large data, such as images and audio. We expect the indexing time and index size to be smaller when using fewer pivots.



(a) *GaussMix* (b) *Skewed*

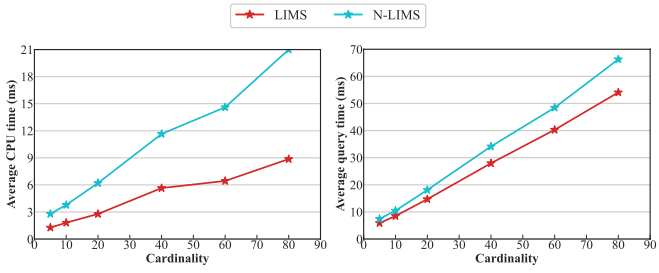
Fig. 12: Range query performance after insertions

6.6 Updates

In this subsection, we examine the impact of data updates. Without loss of generality, we assume that the distribution of underlying data does not change greatly over time. Fig. 12(a)(b) report the average query time and the number of page accesses for range queries after inserting 5K, 10K, 20K and 40K objects into 10M 8d *GaussMix* and *Skewed* datasets, respectively. As expected, insertions cause query time increase since there are more points to query, and the index becomes less optimal. However, the rate of performance degradation is slow and steady. For example, on *Skewed*, after 40K insertions, LIMS still achieves 3.4X speedup compared to the second best index ($28.05ms$ v.s. $94.68ms$). Retraining can be triggered when the query performance deteriorates beyond a user-specified threshold. We also studied the impact of deletion operations, but omit those due to negligible influence on query performance and space constraints.

6.7 Ablation Study

Finally, we make a comparison of LIMS and N-LIMS to show the effectiveness of learning components in LIMS. Since the only difference between two methods is whether to use B$^+$-trees or the rank prediction models and exponential search to locate the start and end of a range query, both have the same number of page accesses (I/O cost). Fig. 13(a) reports the average CPU time of range query by varying the cardinality of 8d *GaussMix* dataset. From the figure, we have the following observations: 1) LIMS outperforms N-LIMS on all data sizes. The reason is that rank prediction models in LIMS achieve a good grasp of the data distribution so that a query can be processed by a function invocation in $O(1)$ time instead of traversing B$^+$-trees in $O(\log n)$ time, which implies the effectiveness of the machine learning model. 2) The advantage of LIMS becomes more obvious when the data size increases. This is because the query cost of LIMS does not directly depend on the cardinality, which allows LIMS scalable to very large data sets. To present the whole picture for a fair comparison, we also report the average query time (CPU time + I/O time) of LIMS and N-LIMS in Fig. 13(b). As expected, LIMS again offers the best query performance. Furthermore, N-LIMS is still faster than the second best competitor ($10.45ms$ v.s. $12.21ms$), which further confirms the superiority of LIMS index structure.



(a) CPU time (b) Query time

Fig. 13: Comparison of LIMS and N-LIMS

7 Conclusions

As a universal abstraction of various data types, metric spaces and associated similarity search play an important role in many real-life applications. In this paper, we have developed LIMS, a novel learned index structure, for efficient exact similarity search query processing in metric spaces. LIMS takes advantage of compact-partitioning methods, pivot-based techniques and the idea of *learned index* to organize and access multi-dimensional data. Our extensive experimental results illustrate that LIMS can respond significantly better to the problem of 'curse of dimensionality' compared with other learned index structures. It also has a clear advantage over traditional indexes. In the future, we plan to take this work further by considering query workload information such that workload-aware optimization can be made.

Acknowledgments

This research is partially supported by Natural Science Foundation of China (Grant# 62072125) and is conducted in the JC STEM Lab of Data Science Foundations funded by The Hong Kong Jockey Club Charities Trust.

REFERENCES

[1] Y. Rubner, C. Tomasi, and L. J. Guibas, "A metric for distributions with applications to image databases," in *ICCV*, pp. 59–66, 1998.

[2] E. Chávez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 273–321, 2001.

[3] M. L. Hetland, "The basic principles of metric indexing," in *Swarm intelligence for multi-objective problems in data mining*, pp. 199–232, 2009.

[4] D. Rachkovskij, "Distance-based index structures for fast similarity search," *Cybernetics and Systems Analysis*, vol. 53, no. 4, pp. 636–658, 2017.

[5] P. Zezula, G. Amato, V. Dohnal, and M. Batko, *Similarity Search - The Metric Space Approach*, vol. 32 of *Advances in Database Systems*. Kluwer, 2006.

[6] L. Chen, Y. Gao, X. Song, Z. Li, X. Miao, and C. S. Jensen, "Indexing metric spaces for exact similarity search," *CoRR*, vol. abs/2005.03468, 2020.

[7] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *SIGMOD*, pp. 489–504, 2018.

[8] H. Wang, X. Fu, J. Xu, and H. Lu, "Learned index for spatial queries," in *MDM*, pp. 569–574, 2019.

[9] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "LISA: A learned index structure for spatial data," in *SIGMOD*, pp. 2119–2133, 2020.

[10] J. Qi, G. Liu, C. S. Jensen, and L. Kulik, "Effectively learning spatial indices," *PVLDB*, vol. 13, no. 11, pp. 2341–2354, 2020.

[11] A. Davitkova, E. Milchevski, and S. Michel, "The ml-index: A multidimensional, learned index for point, range, and nearest-neighbor queries," in *EDBT*, pp. 407–410, 2020.

[12] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan, "Sagedb: A learned database system," in *CIDR*, 2019.

[13] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *SIGMOD*, pp. 985–1000, 2020.

[14] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: A learned multi-dimensional index for correlated data and skewed workloads," *PVLDB*, vol. 14, no. 2, pp. 74–86, 2020.

[15] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P. Larson, D. Kossmann, and R. Acharya, "Qd-tree: Learning data layouts for big data analytics," in *SIGMOD*, pp. 193–208, 2020.

[16] J. A. Orenstein and T. H. Merrett, "A class of data structures for associative searching," in *PODS*, pp. 181–190, 1984.

[17] H. V. Jagadish, B. C. Ooi, K. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search," *TODS*, vol. 30, no. 2, pp. 364–397, 2005.

[18] R. Xu and D. C. W. II, "Survey of clustering algorithms," *IEEE Trans. Neural Networks*, vol. 16, no. 3, pp. 645–678, 2005.

[19] C. T. Jr., R. F. S. Filho, A. J. M. Traina, M. R. Vieira, and C. Faloutsos, "The omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient," *VLDBJ*, vol. 16, no. 4, pp. 483–505, 2007.

[20] T. Gu, K. Feng, G. Cong, C. Long, Z. Wang, and S. Wang, "The rlr-tree: A reinforcement learning based r-tree for spatial data," *CoRR*, vol. abs/2103.04541, 2021.

[21] A. Feinberg, "Markov decision processes: Discrete stochastic dynamic programming (martin l. puterman)," *SIAM*, vol. 38, no. 4, p. 689, 1996.

[22] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, 2001.

[23] B. SCHROEDER, *Ordered Sets: An Introduction with Connections from Combinatorics to Topology*. BIRKHAUSER, 2018.

[24] K. Chakrabarti and S. Mehrotra, "Local dimensionality reduction: A new approach to indexing high dimensional spaces," in *VLDB*, pp. 89–100, 2000.

[25] D. S. Hochbaum and D. B. Shmoys, "A best possible heuristic for the *k*-center problem," *Math. Oper. Res.*, vol. 10, no. 2, pp. 180–184, 1985.

[26] E. Chávez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 273–321, 2001.

[27] B. Pagel, F. Korn, and C. Faloutsos, "Deflating the dimensionality curse using multiple fractal dimensions," in *ICDE*, pp. 589–598, 2000.

[28] K. L. Clarkson *et al.*, "Nearest-neighbor searching and metric space dimensions," *Nearest-neighbor methods for learning and vision: theory and practice*, pp. 15–59, 2006.

[29] Y. Zhu, L. Chen, Y. Gao, and C. S. Jensen, "Pivot selection algorithms in metric spaces: a survey and experimental study," *VLDBJ*, vol. 31, no. 1, pp. 23–47, 2022.

[30] R. L. Thorndike, "Who belongs in the family?," *Psychometrika*, vol. 18, no. 4, pp. 267–276, 1953.

[31] Y. Tao, M. L. Yiu, and N. Mamoulis, "Reverse nearest neighbor search in metric spaces," *TKDE*, vol. 18, no. 9, pp. 1239–1252, 2006.

[32] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *SIGMOD*, pp. 322–331, 1990.

[33] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB*, pp. 426–435, 1997.

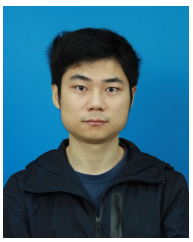
[34] L. Chen, Y. Gao, X. Li, C. S. Jensen, and G. Chen, "Efficient metric indexing for similarity search," in *ICDE*, pp. 591–602, IEEE Computer Society, 2015.

[35] L. Chen, Y. Gao, X. Li, C. S. Jensen, and G. Chen, "Efficient metric indexing for similarity search and similarity joins," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 3, pp. 556–571, 2017.

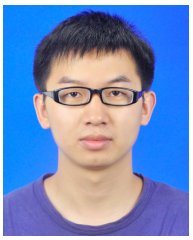
[36] T. Bingmann, "TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers," 2018. https://panthema.net/tlx, retrieved Oct. 7, 2020.

Yao Tian is currently a PhD student with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (HKUST), supervised by Prof. Xiaofang Zhou. She received her MSc degree in School of Mathematical Science from Zhejiang University in 2020. Her research interests include the learned index and approximate query processing in high dimensional spaces.

Tingyun Yan is currently a postgraduate student in Cyberspace Institute of Advanced Technology, Guangzhou University, supervised by Prof. Xiaofang Zhou. He received the Bachelor degree in Software College from Northeastern University in 2020. His research interests include spatial index and learned index.



Xi Zhao is currently a research assistant at HKUST, under the supervision of Prof. Xiaofang Zhou. He received the Master degree in Computer Science from Huazhong University of Science and Technology, China, in 2021. His research interests include the approximate query processing in high dimensional spaces, exact trajectory similarity search and exact textual similarity search.



Kai Huang is a Postdoc at the Department of Computer Science and Engineering, HKUST, under the supervision of Prof. Xiaofang Zhou. He received his PhD degree in School of Computer Science from Fudan University in 2020, and BEng degree in Software Engineering from East China Normal University in 2014. His research interests include graph database and privacy-aware data management.

Xiaofang Zhou is Otto Poon Professor of Engineering and Chair Professor of Computer Science and Engineering at the Hong Kong University of Science and Technology. He received his Bachelor and Master degrees in Computer Science from Nanjing University, in 1984 and 1987 respectively, and his PhD degree in Computer Science from the University of Queensland in 1994. His research is focused on finding effective and efficient solutions for managing, integrating, and analysing very large amount of complex data for business, scientific and personal applications. His research interests include spatial and multimedia databases, high performance query processing, web information systems, data mining, data quality management, and machine learning. He is a Fellow of IEEE.