LISA: A Learned Index Structure for Spatial Data

Pengfei Li
Zhejiang University, China
pfl@zju.edu.cn

Hua Lu
Roskilde University, Denmark
luhua@ruc.dk

Qian Zheng
NTU, Singapore
zhengqian@ntu.edu.sg

Long Yang
Zhejiang University, China
yanglong@zju.edu.cn

Gang Pan
Zhejiang University, China
gpan@zju.edu.cn

ABSTRACT

In spatial query processing, the popular index R-tree may incur large storage consumption and high IO cost. Inspired by the recent *learned index* [17] that replaces B-tree with machine learning models, we study an analogy problem for spatial data. We propose a novel Learned Index structure for Spatial dAta (**LISA** for short). Its core idea is to use machine learning models, through several steps, to generate searchable data layout in disk pages for an arbitrary spatial dataset. In particular, LISA consists of a mapping function that maps spatial keys (points) into 1-dimensional mapped values, a learned shard prediction function that partitions the mapped space into shards, and a series of local models that organize shards into pages. Based on LISA, a range query algorithm is designed, followed by a lattice regression model that enables us to convert a *K*NN query to range queries. Algorithms are also designed for LISA to handle data updates. Extensive experiments demonstrate that LISA clearly outperforms R-tree and other alternatives in terms of storage consumption and IO cost for queries. Moreover, LISA can handle data insertions and deletions efficiently.

KEYWORDS

Database; Spatial index; Learned Index

ACM Reference Format:
Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389703

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389703

1 INTRODUCTION

To support efficient queries, spatial databases have for decades relied on delicate indexes. Among others, R-tree [13] is the most popular spatial index that prunes irrelevant data for queries, *i.e.*, avoiding accessing data chunks without desired data. However, this classical index-centric paradigm is increasingly being challenged in the era of big spatial data.

On the one hand, the ever-increasing data volume entails large R-trees. Sometimes, R-trees are larger than their underlying datasets, especially if the data features more spatial dimensions, there are few non-spatial attributes, and/or the R-tree's disk page utilization ratio [21] is small. This imposes severe storage pressure on spatial databases, and slows down search algorithms by too many tree node visits. On the other hand, the velocity of big spatial data, *e.g.*, fast updates of locations, causes R-trees out of date very frequently. To ensure the freshness of an R-tree, we have to perform frequent and efficient updates on the tree. This makes it difficult to implement and maintain R-trees, and the impact is even worse when velocity and volume issues co-exist. Consequently, R-tree and its variants [4, 15] fall short in indexing big spatial data today. Spatial databases call for outside-the-box, paradigm-changing innovations that can replace R-trees to meet the demands of big spatial data.

A recent work [17] replaces B-tree indexing 1-dimensional data with a recursive model index (RMI) that consists of a number of machine learning models staged into a hierarchy. RMI assumes that the data is sorted and kept in an in-memory dense array. Given a search key x, RMI predicts, with some error bound, where x's data is positioned in the array. The position prediction is equivalent to approximating the cumulative distribution function (CDF) of all search keys. RMI is able to process point and range queries in the 1-dimensional space. With a learned CDF $F(x)$, to get all keys in $[x_1, x_2]$, RMI calculates $F(x_1)$ and $F(x_2)$. The keys stored between positions $F(x_1) - \epsilon$ and $F(x_2) + \epsilon$ form the predicted answer for the range query, where ϵ is the error bound.

However, the idea of RMI does not fly in the context of spatial data. First of all, spatial data invalidates the 1-dimensional data assumption required by RMI. Although it is possible to learn multi-dimensional CDFs, such CDFs cannot provide

for a range query elegant lower/upper bounds which require only two searches via model prediction. In contrast, such CDFs will result in searching local regions qualified on one dimension but not all dimensions. Moreover, when the data is disk-resident, even a small error bound ϵ means accessing several extra pages for locating a search key not in the predicted page. When processing a spatial query that needs to load a number of pages, RMI will have to access many pages, incurring considerable IO cost.

The recent Z-order Model (ZM) [29] combines the Z-order space filling curve [25] and RMI to index spatial data. However, it needs to check many irrelevant keys for range queries and does not support KNN queries or data updates.

In this paper, we propose a novel Learned Index structure for Spatial dAta (LISA). Its core idea is to use machine learning models, through several well designed steps, to generate searchable data layout in disk pages for an arbitrary spatial dataset. It adopts an easy yet effective strategy—partitioning space into a series of grid cells based on the data distribution along a sequence of axes and numbering the cells also along these axes. LISA builds a partially monotonic function $\mathcal{M}$ according to the borders of cells to map the data from $\mathbb{R}^d$ into $\mathbb{R}$. If two spatial points, *i.e.*, keys, $\boldsymbol{x_0}$ and $\boldsymbol{x_1}$ are in the cells C_i and C_j, respectively, and $i < j$, then $\mathcal{M}(\boldsymbol{x_0}) < \mathcal{M}(\boldsymbol{x_1})$ is guaranteed to hold. Furthermore, based on the keys' mapped values, LISA learns a monotonic function $\mathcal{SP}$ composed of a series of piecewise linear functions in order to assign a shard id to each mapped value. By tuning $\mathcal{SP}$'s parameters, each shard contains similar numbers of keys. By building a local model that manages the keys for each shard, the keys that fall in the same shard are stored in one or more consecutive disk pages, and two keys with different shard ids are stored in two different pages. In general, local models are responsible for allocating pages to store keys and splitting or merging pages to update data. Therefore, our LISA is also able to handle data updates including insertion and deletion. Unlike RMI [17] that fixes the data layout first and then learns the model to approximate the layout, LISA uses the learned model to generate the data layout (shards).

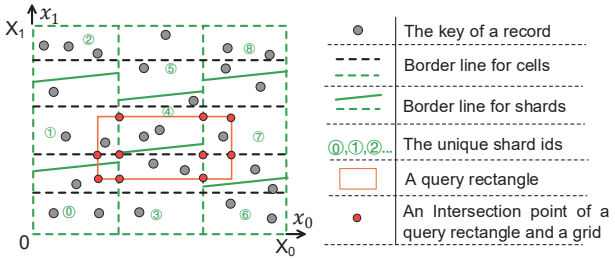

Figure 1: An example of data layout with shards

Fig. 1 gives an example of shard layout in LISA. To access all keys falling in a range query's rectangle qr, we only need to search all cells that overlap with this rectangle. For each such cell C_i, suppose $qr_i = qr \cap C_i = [l_0, u_0) \times \cdots \times [l_{d-1}, u_{d-1})]$. We calculate $m_l = \mathcal{M}\big((l_0, \ldots, l_{d-1})\big)$ and $m_u = \mathcal{M}\big((u_0, \ldots, u_{d-1})\big)$, use $\mathcal{SP}$ to obtain the relevant shards that intersect with $\mathcal{M}^{-1}\big([m_l, m_u)\big)$, and use the local models to access the data of the keys falling in qr_i.

In LISA, a KNN query is converted to range queries. We propose a lattice regression model $\mathcal{LR}$ to estimate a range query's rectangle size for a given query point and K. Subsequently, a range query with that estimated rectangle size is executed. If the range query returns less than K nearest neighbors, the range will be augmented for another range query until sufficient neighbors are found.

The storage consumption of LISA is considerably smaller than R-tree that has to materialize a tree with all nodes and entries based on MBRs and parent-children relationships. In contrast, LISA only keeps the parameters of $\mathcal{M}$, $\mathcal{SP}$, the local models, and $\mathcal{LR}$ for KNN query. Specifically, $\mathcal{M}$'s parameters contain several numbers and a small list only, and $\mathcal{SP}$ is composed of a series of piecewise linear functions whose parameters are a number of coefficients. A local model's parameters are usually several numbers as well, and storing a lattice regression model requires little space.

Overall, compared to the traditional R-tree, LISA makes significant improvements in multiple aspects. First, LISA's storage consumption is considerably smaller. Second, LISA incurs much less IO cost when processing range and KNN queries. Third, LISA supports data updates, *i.e.*, insertion and deletion, efficiently.

Our contributions in this paper are summarized as follows.

- We design LISA, a novel learned index structure, for disk-resident spatial data. To the best of our knowledge, it is the first full-fledged learned index for spatial data. (Section 3)
- We design an efficient algorithm to process range query using LISA, followed by a lattice regression model that enables us to process a KNN query as a series of range queries in LISA. (Section 4)
- We devise efficient algorithms to update LISA for key insertions and deletions. (Section 5)
- We conduct extensive performance evaluation using real and synthetic data. The results demonstrate that LISA outperforms alternative methods in terms of storage consumption and IO cost for range and KNN queries. Also, LISA is able to handle data updates efficiently. (Section 6)

In addition, Section 2 formulates the research problem and introduces our solution framework, Section 7 reviews the related work, and Section 8 concludes the paper and points to future research directions.

2 PRELIMINARIES

This section presents the definitions, a baseline method, and an overview of LISA. Table 1 lists the important notations.

Table 1: Notations

d	The dimensionality of data keys
$\boldsymbol{V}$	the whole space with $\boldsymbol{V} = [0, X_0) \times \cdots \times [0, X_{d-1})$
C_i	The ith cell
S_i	The ith shard
$\mathcal{M}$	Mapping function
$\mathcal{SP}$	Shard prediction function
$\mathcal{L}_i$	The shard S_i's corresponding local model
Ω	The maximum number of keys stored in a page
Ψ	The average number of keys in the initial dataset falling in a shard
σ	The number of break points in $\mathcal{SP}$'s each piecewise linear function

2.1 Definitions

Without loss of generality, we work on spatial data in a d-dimensional space $\boldsymbol{V} = [0, X_0) \times \cdots \times [0, X_{d-1}) \subseteq \mathbb{R}^d$.

Definition 1 (Key). A key $\boldsymbol{k}$ is a unique identifier for a data record with $\boldsymbol{k} = (x_0, ..., x_{d-1}) \in \mathbb{R}^d$.

We have $0 \le x_i < X_i$ for $0 \le i < d$. A key $(x_0, ..., x_{d-1})$ is also a *point* in $\boldsymbol{V}$. We use a grid to partition $\boldsymbol{V}$ into cells.

Definition 2 (Cell). A grid cell *cell* is a (hyper)rectangle in $\boldsymbol{V}$ whose lower and upper corners are points $(l_0, \ldots, l_{d-1})$ and $(u_0, \ldots, u_{d-1})$, *i.e.*, $cell = [l_0, u_0) \times \cdots \times [l_{d-1}, u_{d-1})$.

In addition to the grid, we map each d-dimensional point into a sequential order using a mapping function.

Definition 3 (Mapping Function). A mapping function $\mathcal{M}$ is a partially monotonic function on the domain $\boldsymbol{V}$ to the non-negative range, *i.e.*, $\mathcal{M} : [0, X_0) \times \cdots \times [0, X_{d-1}) \to [0, +\infty)$, such that $\mathcal{M}\big((x_0, ..., x_{d-1})\big) \le \mathcal{M}\big(((y_0, \ldots, y_{d-1})\big)$ when $x_0 \le y_0, \ldots, x_{d-1} \le y_{d-1}$.

In our setting, the mapped values are used to organize spatial keys in sequential disk pages in a spatial database. For a page P, $P.keys$ denotes the set of keys stored in P, and $\mathcal{M}(P)$ denotes the set of $P.keys$' mapping results, *i.e.*, $\mathcal{M}(P) \triangleq \mathcal{M}(P.keys)$. For a page P and a key $\boldsymbol{k} = (x_0, \ldots, x_{d-1})$, if $\inf \mathcal{M}(P) \le \mathcal{M}(\boldsymbol{k}) \le \sup \mathcal{M}(P)$[1], key $\boldsymbol{k}$ must be stored in page P in our setting. In this case, we say page P **contains** key $\boldsymbol{k}$.

With these basic definitions, we design a baseline method.

2.2 Baseline Method

Given a mapping function $\mathcal{M}$, we can extend the learned index method for range query on scalar values [17] to spatial data. This baseline method works as follows. We sort all keys according to their mapped values, store them in a number of pages with each page fully utilized, and store the addresses of these pages in a dense array. Suppose $d = 2$. If a point (x, y)'s mapped value is larger than those of the keys stored in the first j pages, *i.e.*, $\mathcal{M}\big((x, y)\big) > \sup \bigcup_{i=0}^{j-1} \mathcal{M}(P_i)$, we store (x, y) in page P_j, *i.e.*, the page address index of (x, y) is j. Here, the first j pages' addresses are exactly the first j items in the dense array. Subsequently, for a query rectangle $qr = [l_0, u_0) \times [l_1, u_1)$, we only need to predict i_1 and i_2, the indices of (l_0, l_1) and (u_0, u_1), respectively, load the $i_2 - i_1 + 1$ pages, and scan those pages to find those keys that fall in the query rectangle qr.

However, the baseline suffers from a severe problem—many pages irrelevant to the query rectangle may be loaded, incurring considerable unnecessary IOs. Fig. 2 shows an example where the mapping function is $\mathcal{M}\big((x, y)\big) = x + y$. As a result, the dataset is partitioned into three parts. The query rectangle fully falls inside the second part. This results in many, namely 11, irrelevant points accessed for the range query that only contains three relevant keys.

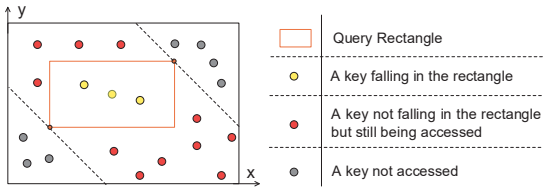

Figure 2: Baseline method pitfall

Let $G = \mathcal{M}^{-1}([\mathcal{M}\big((l_0, l_1)\big), \mathcal{M}\big((u_1, u_1)\big)])$. In the baseline method, $G \setminus qr$ may be large and contain many keys, which increases the IO cost of a range query. The location and shape of a query rectangle are not fixed in advance. When fixing a mapping function $\mathcal{M}$, the Lebesgue measure [24] $\mu(G \setminus qr)$ is large for most rectangles, which causes many keys to be loaded unnecessarily. To address this problem, we design a general learned indexing method **LISA** that works for arbitrary spatial datasets.

2.3 LISA Overview

Our LISA goes beyond the cells generated by the mapping function $\mathcal{M}$. Conversely, we are also interested in the key set that maps to a known interval in the range $[0, +\infty)$.

Definition 4 (Shard). A shard S is the preimage of an interval $[a, b) \subseteq [0, +\infty)$ under the mapping function $\mathcal{M}$, i.e., $S = \mathcal{M}^{-1}\big([a, b)\big)$.

Given an initial dataset, we build a mapping function $\mathcal{M}$ based on the data distribution. After getting the mapped values of all keys, a monotonic shard prediction function $\mathcal{SP}$ is learned to partition the keys into different shards. Suppose $m_i = \inf \mathcal{SP}^{-1}\big([i, i+1)\big)$ and $m_{i+1} = \inf \mathcal{SP}^{-1}\big([i+1, i+2)\big)$ for mapped values m_i and m_{i+1}. The ith shard $S_i \triangleq \mathcal{M}^{-1}\big([m_{i-1}, m_i)\big)$.

In our setting, all shards exhibit a total order with respect to their corresponding intervals in the mapped range. In addition, all shards are disjoint with each other and the union of them is contained in $\boldsymbol{V}$. According to the definitions of the mapping function and shard, it is apparent that

$$\inf \mathcal{M}(S_i) > \sup \mathcal{M}(S_j) \text{ when } i > j \tag{1}$$

[1] $\inf(S)$ and $\sup(S)$ denote a set S's lower and upper bounds, respectively.

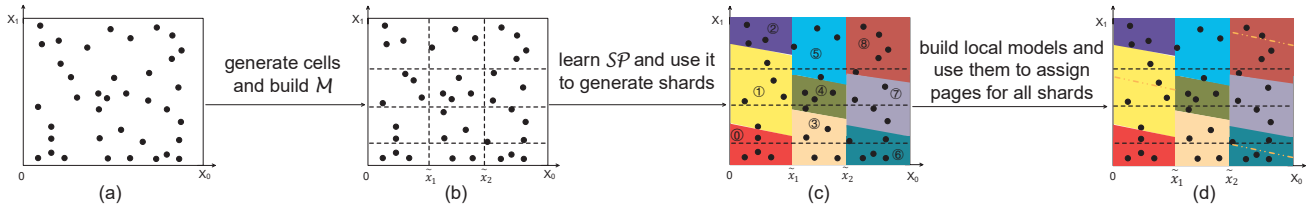

Figure 3: LISA framework

Further, the keys in a shard are stored in a number of disk pages, whereas the keys in one page must come from the same shard. In contrast, the keys in a grid cell may correspond to a number of pages, and vice versa. In other words, we have a one-to-many relationship between shards and pages, and a many-to-many relationship between cells and pages. If a page P **overlaps** with a grid cell $C \subseteq \boldsymbol{V}$, there are points stored in P that fall in C. If shard $S \subseteq \boldsymbol{V}$ **contains** a page P, any key stored in P falls in S. We use local models to determine the page addresses for the keys in a shard.

Definition 5 (Local Model). A local model $\mathcal{L}_i$ is a model that processes operations within a shard S_i. It keeps dynamic structures such as the addresses of pages contained by S_i.

Given a spatial dataset, we generate the mapping function $\mathcal{M}$, the shard prediction function $\mathcal{SP}$ and a series of local models. Based on them, we build our index structure, LISA, to process range query, KNN query and data updates. LISA aims to reduce the storage consumption and IO cost compared to existing indexes such as R-Tree.

Essentially, LISA consists of four parts: the representation of grid cells (Section 3.1), the mapping function $\mathcal{M}$ (Section 3.2), the shard prediction function $\mathcal{SP}$ (Section 3.3), and the local models for all shards (Section 3.5). As illustrated in Fig. 3, the procedure of building LISA is composed of the steps to build the four parts.

To get all pages that overlap with a query rectangle qr, we first decompose qr into a number of smaller rectangles each intersecting only one cell. For each such cell C_i, we use $\mathcal{SP}$ to select the shards in C_i that overlap with qr. Finally, we use local models to get the addresses of the pages overlapping with qr. The details are given in Section 4.1. In LISA, a KNN query is converted into a series of range queries. We build a lattice regression model to estimate an appropriate query range for a query point and K. The query range is augmented if less than K neighbors are found in a range query. The details are given in Section 4.2.

To insert or delete a key $\boldsymbol{k}$, the first step is to calculate $\boldsymbol{k}$'s mapped value. Next, $\boldsymbol{k}$'s shard is obtained through $\mathcal{SP}$. Subsequently, the local model will locate the corresponding disk page within the corresponding shard. For an insertion (Section 5.1), if the page is full, the local model will split it. For a deletion (Section 5.2), the local model will count the keys stored in the page from which $\boldsymbol{k}$ is to be deleted. Different strategies may be executed, *e.g.*, consecutive pages may be merged if they each contain too few keys.

3 DESIGN AND TRAINING OF LISA

3.1 Generating Grid Cells

On each dimension, we partition the space $\boldsymbol{V}$ along the x_i-axis into T_i parts such that the initial dataset's keys are evenly covered by every part. We use $\boldsymbol{\Theta_i} = [\theta_1^{(i)}, ..., \theta_{T_i}^{(i)}]$ to denote the border points generated by the partition operation for the x_i-axis. The whole space $\boldsymbol{V}$ can be represented by the union of a series of disjoint cells, *i.e.* $\boldsymbol{V} = \bigcup_{t=0}^{T_0 \times \cdots \times T_{d-1}-1} C_t$ where

$$C_t = [\theta_{j_0}^{(0)}, \theta_{j_0+1}^{(0)}) \times \cdots \times [\theta_{j_{d-1}}^{(d-1)}, \theta_{j_{d-1}+1}^{(d-1)}), \text{with}$$
$$t = \left(\left(\left(j_0 \times T_1 + j_1\right) \times T_2 + j_2\right) \times \cdots\right) \times T_{d-1} + j_{d-1}$$

Note that every partition operation for a part along the x_i-axis can also be performed using a 1-dimensional monotonic regression model. The model's input is merely the x_i components of the keys falling in the part. Its output range is $[0, T_i]$. When T_i is large, saving the model may need less storage space than keeping $\boldsymbol{\Theta_i}$. In practice, each T_i is set to a small number (less than 300). In this paper, we only keep the lists of those border points.

Given a query rectangle $qr \subseteq \mathbb{R}^d$, we can easily get the grid cells that intersect qr. Let those cells be C_{i_0},...,C_{i_Q}. Then, qr can be rewritten as $qr = \bigcup_{j=0}^{Q} qr_j$ with $qr_j = qr \cap C_{i_j}$. To query all keys that fall in qr, we only need to search $Q+1$ cells. This way reduces the search space considerably.

In some extreme cases, the keys fall in only a few cells. Fig. 4.(a) shows such an example, where only 3 out of 9 cells contain keys. In this case, the aforementioned original partitioning strategy needs to be modified. First, we partition the space $\boldsymbol{V}$ along the x_0-axis into T_0 parts such that the initial dataset's keys are evenly covered by every part. Next, each part is partitioned along the x_1-axis to generate T_1 smaller parts. Further, the smaller parts are partitioned along the x_2-axis. This process is repeated until the whole space is partitioned along all the d axes. Fig. 4.(b) illustrates the procedure for the case of $d = 2$. Note that applying the modified strategy requires us to keep more parameters. In practice, we can choose to apply the modified and original strategies along different axes. We use a simple strategy to detect the extreme cases. For each axis, we calculate the variance of the number of keys falling in a part. If the variance is larger than a pre-defined threshold, LISA will perform the modified strategy for this axis. Usually, if the data distribution is

complicated, we apply the modified strategy along the first $d-1$ axes, and the original strategy along the last axis.

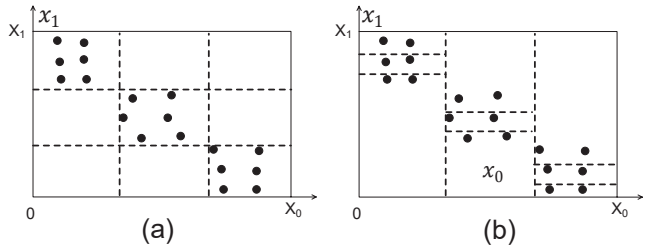

Figure 4: Original and modified partitioning strategies

3.2 Mapping Function $\mathcal{M}$

The mapping function $\mathcal{M}$ should satisfy the condition

$$\mathcal{M}(\boldsymbol{x_i} \in \boldsymbol{V}) < \mathcal{M}(\boldsymbol{x_j} \in \boldsymbol{V}) \text{ when } i < j, \text{ where}$$
$$\boldsymbol{x_i} \in C_i \text{ and } \boldsymbol{x_j} \in C_j \tag{2}$$

A naive way to guarantee it is to set $\mathcal{M}(\boldsymbol{x}) = i$ if $\boldsymbol{x} \in C_i$. However, a cell may contain multiple keys, and two keys' mapped values may be different. To avoid this, we give a more proper definition of $\mathcal{M}$. Suppose $\boldsymbol{x} = (x_0, ..., x_{d-1})$ and $\boldsymbol{x} \in C_i = [\theta^{(0)}_{j_0}, \theta^{(0)}_{j_0+1}) \times \cdots \times [\theta^{(d-1)}_{j_{d-1}}, \theta^{(d-1)}_{j_{d-1}+1})$ then

$$\mathcal{M}(\boldsymbol{x}) = i + \frac{\mu(H_i)}{\mu(C_i)}, \text{ where } H_i = [\theta^{(0)}_{j_0}, x_0) \times \cdots \times [\theta^{(d-1)}_{j_{d-1}}, x_{d-1})$$

Above, μ is the Lebesgue measure [24] on $\mathbb{R}^d$. It is easy to check $\mathcal{M}$ is well-defined.

3.3 Shard Prediction Function $\mathcal{SP}$

The RMI [17] model is based on a simplifying assumption that 1-dimensional data is sorted by keys and stored in an in-memory dense array. Given a search key, RMI predicts its index in the array, with an error bound within which it is guaranteed the key's data record will be found. When the idea of RMI is applied to data stored in disk pages, the error bound can result in considerable page accesses before the key's data is found in a particular page. In other words, the RMI can incur considerable IO cost for search in disk pages. To reduce IO cost in the context of disk pages, we need to design a learned model different than RMI.

In essence, every key in RMI has a unique array index and RMI predicts a key's array index with an error as small as possible. In other words, the data layout is already fixed before the RMI is trained. However, it is difficult to train a regression model with zero loss, especially when the keys are stored in disk pages instead of a dense array. We consider this problem from a different angle. Rather than fixing the data layout using a regression model that can predict a key's position, we design and train a model that directly arranges the data layout in disk pages.

The shard prediction function $\mathcal{SP}$ attempts to do so. Specifically, $\mathcal{SP} : \mathbb{R} \rightarrow [0, +\infty)$ whose input is a key's mapped value determines which keys should be assigned with the same shard id and stored together. Note that if we directly extend the idea in [17], the shard ids of all keys should be fixed in the beginning and we need to approximate the mapping between the keys and fixed shard ids.

Basically, $\mathcal{SP}$ is a regression model. First, we sort all keys' mapped values and save them in a dense array. The mapped values and their array indices are the training input of $\mathcal{SP}$. After $\mathcal{SP}$ is trained, for a point $\boldsymbol{x} \in \mathbb{R}^d$ with mapped value m, its shard id is decided to be $\lfloor \mathcal{SP}(m) \rfloor$. $\mathcal{SP}$ must be constrained to be monotonic such that the condition in Formula (1) can be guaranteed.

In theory, $\mathcal{SP}$ can be any monotonic regression model. However, considering the diversity of data distributions in real-life applications, it is difficult to use a single regression model to predict the indices of all mapped values accurately unless the model size is sufficiently large. Existing index structures like B-Tree or R-Tree all partition the data into a hierarchy, narrowing the search space and reducing search errors for key finding. This strategy is effective on general datasets. Likewise, we introduce a simple yet effective way to build and train $\mathcal{SP}$ according to a similar strategy.

Like RMI [17], $\mathcal{SP}$ is also a staged model. Suppose $\mathcal{SP}$ contains U small regression models at the bottom level. We simply find a list of numbers $\boldsymbol{M_p} = [\tilde{m}_1, ..., \tilde{m}_U]$ to evenly partition the mapped values, *i.e.*, to make the numbers of mapped values in $[\tilde{m}_{i-1}, \tilde{m}_i)$ almost the same for $0 \leq i < U$. Because the mapped values are sorted, performing this operation is fast. For each part, we build a monotonic regression model $\mathcal{F}_i$ whose domain is $[\tilde{m}_{i-1}, \tilde{m}_i)$. If $\mathcal{F}_i$ needs to process too many mapped values, $\mathcal{F}_i$ itself can be a staged model. In our experiments, each $\mathcal{F}_i$ handles less than 50,000 mapped values and it is not staged.

RMI [17] adopts neural networks [27] as the regression models. However, for a neural network to guarantee the monotonicity and have enough expressive power, its size must be large. For example, to constrain a general fully connected neural network to be monotonic, a general way is to set the neurons in the fully-connected layers non-negative. In this case, many activation functions (*e.g.*, ReLU) will not work and the fully-connected network almost becomes a linear regression function. Besides, it is difficult for a neural network to approximate a non-smooth function in practice.

Fig. 5 gives an example of a real dataset for which we randomly select a small regression model $\mathcal{F}_i$ and obtain the list of sorted mapped values to be processed. The figure illustrates the relationship between the

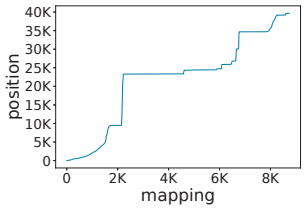

Figure 5: Example of keys' mapped values vs. indices

mapped values of keys and their indexes in the list. It is difficult for a neural network to approximate this complex relationship, especially if the network is not large. To capture the relationship, we use monotonic piecewise linear functions instead. This gives three advantages: 1) It is much easier to constrain a piecewise linear function to be monotone. 2) To achieve the same precision, the parameter size of a piecewise linear function is smaller than that of a neural network. 3) It takes much less time to train piecewise linear functions. We proceed to detail how to train $\mathcal{SP}$.

3.4 Training of $\mathcal{SP}$

Let $\boldsymbol{M_p} = [\tilde{m}_1, ..., \tilde{m}_U]$ denote the number list that partitions the mapped values (Section 3.3). Suppose $V+1$ is the number of mapped values that each $\mathcal{F}_i$ needs to process and Ψ is the estimated average number of keys falling in a shard. Thus, each $\mathcal{F}_i$ generates $D = \lceil \frac{V+1}{\Psi} \rceil$ shards. The shard prediction function $\mathcal{SP}$ with input $x \in \mathbb{R}$ is expressed as follows.

$$\mathcal{SP}(x) = \mathcal{F}_i(x) + i \times D, \text{ where } i = \text{binary-search}(\boldsymbol{M_p}, x)$$

Let $\boldsymbol{x} = (x_0, \cdots, x_V)$ be the keys' mapped values that fall in $[\tilde{m}_{i-1}, \tilde{m}_i)$. Without loss of generality, suppose $\boldsymbol{x}$ is sorted, *i.e.*, $x_i \le x_j, \forall 0 \le i < j \le V$. Let $\boldsymbol{y} = (0, \cdots, V)$. We first build a piecewise linear regression function f_i with inputs $\boldsymbol{x}$ and $\boldsymbol{y}$. For a given point with mapped value $m \in [\tilde{m}_{i-1}, \tilde{m}_i)$, its shard id is $\lfloor \frac{f_i(m)}{\Psi} \rfloor + i \times D$, *i.e.*, $\mathcal{F}_i(x) = \frac{f_i(m)}{\Psi}$.

Next, we describe how to build and train a general monotone piecewise linear function f. Given $V+1$ sorted mapped values $\boldsymbol{x} = (x_0, \cdots, x_V)$ and their indices $\boldsymbol{y} = (0, \cdots, V)$, each f_i is built and trained with the same procedure.

A piecewise linear function can be described as follows,

$$f(x) = \begin{cases} b_0 + a_0(x - \beta_0) & \beta_0 \le x < \beta_1 \\ b_1 + a_1(x - \beta_1) & \beta_1 \le x < \beta_2 \\ \vdots & \vdots \\ b_\sigma + a_\sigma(x - \beta_\sigma) & \beta_\sigma \le x \end{cases} \tag{3}$$

Here, $\boldsymbol{\beta} = (\beta_0, \beta_1, \cdots, \beta_\sigma)$ is the set of $\sigma + 1$ breakpoints. Without loss of generality, this formulation assumes that the breakpoints are ordered as $\beta_0 < \beta_1 < \cdots < \beta_\sigma$. It is hard to guarantee Eq. (3) to be monotone even if we force every $a_i \ge 0$ because this function may be not continuous. The monotonicity condition can be easily satisfied if the piecewise linear functions are C^0 continuous over the domain. In this case, the slopes and intercepts of each linear region depend on previous values. Let $\bar{\alpha} = b_0$, Eq. (3) reduces to

$$f(x) = \begin{cases} \bar{\alpha} + \alpha_0(x - \beta_0) & \beta_0 \le x < \beta_1 \\ \bar{\alpha} + \alpha_0(x - \beta_0) + \alpha_1(x - \beta_1) & \beta_1 \le x < \beta_2 \\ \cdots & \cdots \\ \bar{\alpha} + \alpha_0(x - \beta_0) + \alpha_1(x - \beta_1) & \\ + \cdots + \alpha_\sigma(x - \beta_\sigma) & \beta_\sigma \le x \end{cases} \tag{4}$$

To make Eq. (4) monotonically increasing, we only need to guarantee

$$\sum_{i=0}^{\eta} \alpha_i \ge 0, \ \forall\, 0 \le \eta \le \sigma. \tag{5}$$

Let $\boldsymbol{\alpha} = (\bar{\alpha}, \alpha_0, \alpha_1, \cdots, \alpha_\sigma)$. The square loss function $L(\boldsymbol{\alpha}, \boldsymbol{\beta})$ is defined as follows.

$$L(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \sum_{i=0}^{V} \big(f(x_i) - y_i\big)^2 \tag{6}$$

It is difficult to directly optimize the parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ together. We propose a simple approach which adopts the idea of the optimal control to optimize $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ iteratively.

Fixing $\boldsymbol{\beta} = \hat{\boldsymbol{\beta}} = (\hat{\beta}_0, \hat{\beta}_1, \cdots, \hat{\beta}_\sigma)$, $\boldsymbol{\alpha}$ can be regarded as the least square solution of the linear equation $\boldsymbol{A\alpha} = \boldsymbol{y}$, where

$$\boldsymbol{A} = \begin{bmatrix} 1 & x_0 - \hat{\beta}_0 & (x_0 - \hat{\beta}_1)\mathbf{1}_{x_0 \ge \hat{\beta}_1} & \cdots & (x_0 - \hat{\beta}_\sigma)\mathbf{1}_{x_0 \ge \hat{\beta}_\sigma} \\ 1 & x_1 - \hat{\beta}_0 & (x_1 - \hat{\beta}_1)\mathbf{1}_{x_1 \ge \hat{\beta}_1} & \cdots & (x_1 - \hat{\beta}_\sigma)\mathbf{1}_{x_1 \ge \hat{\beta}_\sigma} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N - \hat{\beta}_0 & (x_V - \hat{\beta}_1)\mathbf{1}_{x_V \ge \hat{\beta}_2} & \cdots & (x_V - \hat{\beta}_\sigma)\mathbf{1}_{x_V \ge \hat{\beta}_\sigma} \end{bmatrix}$$

The loss function is a quadratic function of $\boldsymbol{\alpha}$. The solution of $\boldsymbol{\alpha}$ can be computed by solving the linear equation

$$\frac{\partial E(\boldsymbol{\alpha}, \hat{\boldsymbol{\beta}})}{\partial \boldsymbol{\alpha}} = 2\boldsymbol{A}^\intercal \boldsymbol{r} = 0, \text{where } \boldsymbol{r} = \boldsymbol{A\alpha} - \boldsymbol{y} \tag{7}$$

$$\Rightarrow \boldsymbol{\alpha} = (\boldsymbol{A}^\intercal \boldsymbol{A})^{-1} \boldsymbol{A}^\intercal \boldsymbol{y} \tag{8}$$

Since matrix $\boldsymbol{A}^\intercal \boldsymbol{A}$ is symmetric, Eq. (7) can be solved by the efficient Cholesky algorithm [28]. Clearly, different breakpoints $\boldsymbol{\beta}$ give rise to different optimal parameters. The next step is to choose an optimal breakpoint. Let $\boldsymbol{\alpha}^\star(\boldsymbol{\beta})$ be the optimal $\boldsymbol{\alpha}$ for particular breakpoints $\boldsymbol{\beta}$. The problem becomes to find $\boldsymbol{\beta}^\star$ such that

$$L(\boldsymbol{\alpha}^\star(\boldsymbol{\beta}^\star), \boldsymbol{\beta}^\star) = \min\{L(\boldsymbol{\alpha}^\star(\boldsymbol{\beta}), \boldsymbol{\beta}) | \boldsymbol{\beta} \in \mathbb{R}^{\sigma+1}\}$$

We define

$$\boldsymbol{g} = \frac{\partial E(\boldsymbol{\alpha}, \boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = 2\boldsymbol{KGr}, \ \boldsymbol{Y} = \frac{\partial g}{\partial \boldsymbol{\beta}} = 2\boldsymbol{KGG}^\intercal \boldsymbol{K}^\intercal$$

where

$$\boldsymbol{K} = \text{diag}(\bar{\alpha}, \alpha_0, \cdots, \alpha_\sigma), \ \boldsymbol{G} = \begin{bmatrix} -1 & -1 & \cdots & -1 \\ p_0^{(0)} & p_0^{(1)} & \cdots & p_0^{(V)} \\ p_1^{(0)} & p_1^{(1)} & \cdots & p_1^{(V)} \\ \vdots & \vdots & \ddots & \vdots \\ p_\sigma^{(0)} & p_\sigma^{(1)} & \cdots & p_\sigma^{(V)} \end{bmatrix}$$

and

$$p_i^{(l)} = -\mathbf{1}_{x_l \ge \beta_i}.$$

Since $\boldsymbol{g} = \nabla_{\boldsymbol{\beta}} L$, $-\boldsymbol{g}$ specifies the steepest descent direction of $\boldsymbol{\beta}$ for L. However, in practice, we do not use $-\boldsymbol{g}$ as the direction for $\boldsymbol{\beta}$'s update—the convergence rate of $-\boldsymbol{g}$ is low

since it does not consider the second-order derivative of L *w.r.t.* to $\boldsymbol{\beta}$. To address this problem, we perform the update along the direction of $\boldsymbol{s} = -\boldsymbol{Y}^{-1}\boldsymbol{g}$. It is clear that $\boldsymbol{Y}$ is positive definite. Thus, the update is guaranteed to be performed in the descent direction. Let

$$\boldsymbol{g}^{(k)} = \boldsymbol{g}|_{\boldsymbol{z}=\boldsymbol{z}^{(k)}},\ \boldsymbol{Y}^{(k)} = \boldsymbol{Y}|_{\boldsymbol{z}=\boldsymbol{z}^{(k)}},\ \boldsymbol{s}^{(k)} = \boldsymbol{s}|_{\boldsymbol{z}=\boldsymbol{z}^{(k)}}$$

where

$$\boldsymbol{z} = \begin{bmatrix}\boldsymbol{\alpha}\\ \boldsymbol{\beta}\end{bmatrix}, \boldsymbol{z}^{(k)} = \begin{bmatrix}\boldsymbol{\alpha^{(k)}}\\ \boldsymbol{\beta^{(k)}}\end{bmatrix}$$

In the beginning, we set $\beta_0^{(0)} = x_0$ and $\beta_i^{(0)} = x_{\lfloor i\times\frac{V}{\Psi}\rfloor}$ for $1 \le i \le \sigma$. We obtain $\boldsymbol{\alpha}^{(0)}$ by solving Eq. (7). At iteration k, we perform a cell search along the direction $\boldsymbol{s}^{(s)}$ to find the update step $lr^{(k)}$ such that

$$\begin{aligned} &L\big(\boldsymbol{\alpha}^{\star}(\boldsymbol{\beta^{(k)}} + lr^{(k)}\boldsymbol{s}^{(k)}), \boldsymbol{\beta^{(k)}} + lr^{(k)}\boldsymbol{s}^{(k)}\big)\\ &= \min\big\{L\big(\boldsymbol{\alpha}^{\star}(\boldsymbol{\beta^{(k)}} + lr\boldsymbol{s}^{(k)}), \boldsymbol{\beta^{(k)}} + lr\boldsymbol{s}^{(k)}\big)|\\ &\quad lr \ge 0 \text{ and } \boldsymbol{\alpha}^{\star}(\boldsymbol{\beta^{(k)}} + lr\boldsymbol{s}^{(k)}) \text{ satisfies Eq. (5)}\ \big\}\end{aligned}$$

At the next iteration, we increment k by one and set

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} + lr^{(k)}\boldsymbol{s^{(k)}}.$$

The iteration continues until L converges.

3.5 Local Models for Shards

For each shard S_i, we can easily get the keys that fall in it and their mapped values with the help of the mapping function $\mathcal{M}$ and the shard prediction function $\mathcal{SP}$. If S_i contains more than Ω keys, we need to decide how to partition them and save them in more than one page. The local model $\mathcal{L}_i$ is created for this and future operations.

Simply speaking, the parameters of $\mathcal{L}_i$ consists of two parts: PA is a list which stores the addresses of the pages that overlap with S_i, and PM is the sequence of sorted mapped values to partition the keys in S_i.

Suppose $\boldsymbol{I_i} = (\boldsymbol{k}_0, \cdots, \boldsymbol{k}_{\Gamma-1})$ are the Γ keys which fall in S_i and are sorted along their mapped values. If $\Gamma \le \Omega$, $\boldsymbol{I_i}$ can be stored in one page. We only need to append the page's address in $\mathcal{L}_i.PA$ and set $\mathcal{L}_i.PM$ to be empty. If $\Gamma > \Omega$, we partition $\boldsymbol{I_i}$ along their mapped values into several pieces and make each piece contain less than Ω keys. The keys in each piece are stored in one page. The addresses of these pages are appended to $\mathcal{L}_i.PA$, and $\mathcal{L}_i.PM$ is the sequence of the mapped values to partition $\boldsymbol{I}_i$. Algorithm 1 builds $\mathcal{M}$ and $\mathcal{SP}$, generates shards and stores keys on disk. Lines 7-22 in Algorithm 1 detail how to generate local models for shards.

The local model $\mathcal{L}_i$ provides two different search functions: *lbound* and *ubound* such that

$$\mathcal{L}_i.lbound(m) = j, \text{ where } \mathcal{L}_i.PM[j-1] < m \le \mathcal{L}_i.PM[j]$$
$$\mathcal{L}_i.ubound(m) = j, \text{ where } \mathcal{L}_i.PM[j-1] \le m < \mathcal{L}_i.PM[j]$$

When $\mathcal{L}_i.PM$ is empty, both $\mathcal{L}_i.lbound(m)$ and $\mathcal{L}_i.ubound(m)$ are equal to 0. Suppose the size of $\mathcal{L}_i.PM$ is J and $m_{max} = \mathcal{L}_i.PM[J-1]$. If $m > m_{max}$, $\mathcal{L}_i.lbound(m) = J-1$ and $\mathcal{L}_i.ubound(m) = J-1$ when $m \ge m_{max}$.

The parameters of the mapping function $\mathcal{M}$ and shard prediction function $\mathcal{SP}$ will not change once they are built and trained. On the contrary, $\mathcal{L}_i.PA$ and $\mathcal{L}_i.PM$ may be changed due to data updates in shard S_i.

Algorithm 1 `BuildLISA`

Input: $\boldsymbol{I}$: keys in the initial dataset
Ω: predefined page size
Output: $\mathcal{M}$: monotone mapping function
$\mathcal{SP}$: shard prediction function
$\boldsymbol{\mathcal{L}}$: local models for all shards

1: **for all** $i \leftarrow 0$ to $d-1$ **do**
2: partition $\boldsymbol{V}$ along the x_i-axis and generate Θ_i
3: generate all grid cells based on all Θ_is
4: generate mapping function $\mathcal{M}$
5: calculate the mapped values of all keys in $\boldsymbol{I}$ using $\mathcal{M}$
6: build $\mathcal{SP}$ and use it to partition $\boldsymbol{V}$ and generate shards
7: **for each** shard S_i **do**
8: get $\boldsymbol{I_i} = (\boldsymbol{k}_0, \cdots, \boldsymbol{k}_{\Gamma-1})$, the keys in $\boldsymbol{I}$ that fall in S_i and $\boldsymbol{m}_i = (m_0, \cdots, m_{\Gamma-1})$, the mapped values of $\boldsymbol{I}_i$
9: initialize $\mathcal{L}_i$ by setting $\mathcal{L}_i.PA$ and $\mathcal{L}_i.PM$ to be empty
10: **if** $\Gamma \le \Omega$ **then**
11: allocate a new page P on disk and store $\boldsymbol{I}_i$ in P
12: append the address of P to $\mathcal{L}_i.PA$
13: **else**
14: $W \leftarrow \lceil\frac{\Gamma}{\Omega}\rceil$, $\Delta \leftarrow \lceil\frac{\Gamma}{W}\rceil$
15: **for** $j \leftarrow 1$ to $V-1$ **do**
16: allocate a new page P on disk
17: store $\boldsymbol{I}_i^{(j)} = (\boldsymbol{k}_{(i-1)\times\Delta}, \cdots, \boldsymbol{k}_{i\times\Delta-1})$ in P
18: append the address of P to $\mathcal{L}_i.PA$
19: append $m_{i\times\Delta}$ to $\mathcal{L}_i.PM$
20: store $\boldsymbol{I}_i^{(W)} = (\boldsymbol{k}_{(W-1)\times\Delta}, \cdots, \boldsymbol{k}_{\Gamma-1})$ in P
21: append the address of P to $\mathcal{L}_i.PA$
22: append $\mathcal{L}^i$ to $\boldsymbol{\mathcal{L}}$
23: **return** $\mathcal{M}, \mathcal{SP}, \boldsymbol{\mathcal{L}}$

4 LISA-BASED QUERY PROCESSING

4.1 Range Query

Given a query rectangle $qr = [l_0, u_0) \times \cdots \times [l_{d-1}, u_{d-1})$, we first get the cells that overlap with qr and decompose qr into the union of smaller query rectangles each of which intersects one and only one cell. Suppose $qr = \bigcup_{j=0}^{Q'} qr'_j$ with $qr'_j = [l'_{j_0}, u'_{j_0}) \times \cdots \times [l'_{j_{d-1}}, u'_{j_{d-1}}) \subseteq C_{i_j}$. When T_i, d or qr is large, Q' may also be large. In this case, we need to search many small rectangles. To make it more efficient, we merge *consecutive* small rectangles. If $C_{i_j}, ..., C_{i_{j+n}}$ satisfy $i_{j+m} - i_{j+m-1} = 1$ and $\bigcup_{m=0}^{n} C_{i_{j+m}}$ is (path) connected, they are merged to form a big cell, namely $\tilde{G}_i$. As a result, qr can be represented as $\bigcup_{j=0}^{Q} qr_j$ with $qr_j = qr \cap \tilde{G}_j = [l_{j_0}, u_{j_0}) \times$

$\cdots \times [l_{j_{d-1}}, u_{j_{d-1}})$. This operation can be performed easily in a recursive manner.

Next, we calculate the mapped values of all qr_j's vertices. We use $m_l^{(0)}, m_u^{(0)}, \cdots, m_l^{(Q)}, m_u^{(Q)}$ to denote $\mathcal{M}((l_{0_0}, ..., l_{0_{d-1}}))$, $\mathcal{M}((u_{0_0}, ..., u_{0_{d-1}}))$, $\cdots$, $\mathcal{M}((l_{Q_0}, ..., l_{Q_{d-1}}))$ and $\mathcal{M}((u_{Q_0}, ..., u_{Q_{d-1}}))$, respectively. After calculating $\mathcal{SP}(m_l^{(0)})$, $\mathcal{SP}(m_u^{(0)})$, $\cdots$, $\mathcal{SP}(m_l^{(Q)})$ and $\mathcal{SP}(m_u^{(Q)})$, it is easy to select the pages that overlap with qr using the corresponding local models.

Algorithm 2 accesses the keys that fall in the query rectangle qr. In lines 6 and 11, we use different binary search functions of the local models. This is because m_l^j and m_u^j may be equal to some items in $\mathcal{L}_{k_l^{(j)}}.PA$ or $\mathcal{L}_{k_u^{(j)}}.PA$. Moreover, it is possible that the mapped values of different keys are equal. In some extreme cases, these keys may be stored in different pages, which means $\mathcal{L}_j.PM$ includes the shared mapped value for some j. By using *lbound* and *ubound* in different situations, it is guaranteed that the returned *PageAddrs* does not miss any pages that overlap with qr.

Algorithm 2 `RangeQuery`

Input: qr: the query rectangle
Output: $\boldsymbol{R}$: the keys falling in qr

1: decompose qr such that $qr = \bigcup_{j=0}^{Q} qr_j$ with $qr_j = [l_{j_0}, u_{j_0}) \times \cdots \times [l_{j_{d-1}}, u_{j_{d-1}})$.
2: set *PageAddrs* to be an empty set
3: **for** $j \leftarrow 0$ to Q **do**
4: $\quad m_l^{(j)} \leftarrow \mathcal{M}((l_{j_0}, .., l_{j_{d-1}}))$, $m_u^{(j)} \leftarrow \mathcal{M}((u_{j_0}, .., u_{j_{d-1}}))$
5: $\quad k_l^{(j)} \leftarrow \lfloor \mathcal{SP}(m_l^j) \rfloor$, $k_u^{(j)} \leftarrow \lfloor \mathcal{SP}(m_u^j) \rfloor$
6: $\quad l \leftarrow \mathcal{L}_{k_l^{(j)}}.lbound(m_l^j)$
7: $\quad$ **for** $s \leftarrow l$ to $\mathcal{L}_{k_l^{(j)}}.PA.size() - 1$ **do**
8: $\quad\quad PageAddrs.add(\mathcal{L}_{k_l^{(j)}}.PA[s])$
9: $\quad$ **for** $t \leftarrow k_l^{(j)} + 1$ to $k_u^{(j)} - 1$ **do**
10: $\quad\quad PageAddrs.addall(\mathcal{L}_t.PA)$
11: $\quad u \leftarrow \mathcal{L}_{k_u^{(j)}}.ubound(m_u^j)$
12: $\quad$ **for** $s \leftarrow 0$ to u **do**
13: $\quad\quad PageAddrs.add(\mathcal{L}_{k_u^{(j)}}.PA[s])$
14: **for each** $P \in PageAddrs$ **do**
15: $\quad$ add to $\boldsymbol{R}$ all the keys that fall in qr and are saved in P
16: **return** $\boldsymbol{R}$

4.2 KNN Query

As we use a machine learning model $\mathcal{SP}$ to generate shards, we do not know the analytical representation of the shards. Thus, it is difficult to apply the traditional pruning strategies for KNN query in R-trees to our LISA. Instead, we combine a lattice regression model [12] and the range query (Algorithm 2) to solve the KNN query problem.

Given a query point $q_{knn} = \boldsymbol{x} = (x_0, ..., x_{d-1})$ and a distance value $\delta > 0$, let $\mathcal{Q}(\boldsymbol{x}, e) \triangleq [x_0 - \delta, x_0 + \delta) \times \cdots \times [x_{d-1} - \delta, x_{d-1} + \delta)$ and $\mathcal{B}(\boldsymbol{x}, \delta) \triangleq \{\boldsymbol{p} \in \boldsymbol{V} \mid \|\boldsymbol{x} - \boldsymbol{p}\|_2 \leq \delta\}$. Suppose $\boldsymbol{x'} \in \boldsymbol{V}$ is the Kth nearest key to $\boldsymbol{x}$ in the database. Let $\delta = \|\boldsymbol{x'} - \boldsymbol{x}\|_2$. We can create a query rectangle $qr = \mathcal{Q}(\boldsymbol{x}, \delta + \epsilon)$ where $\epsilon \to 0$. Clearly, the K nearest keys to $\boldsymbol{x}$ are all in $\mathcal{B}(\boldsymbol{x}, \delta)$ and thus in qr, as shown in Fig. 6. If we can estimate an appropriate distance bound δ for every query point, the KNN query can be solved using the range query.

It is still a regression problem to estimate the distance bounds that can apply to arbitrary query points. We can directly apply a neural network regression model. However, except the coordinates, we do not have other features of the points in a dataset. Also, the regression model's size should not be too large and training it cannot cost too much time. Besides, we hope the model to have good interpretability. Thus, we use a lattice regression model $\mathcal{LR}$ to estimate the distance bounds.

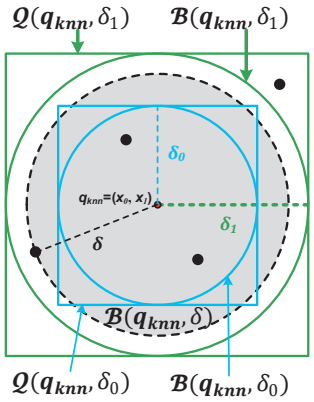

Figure 6: Example of a KNN query conversion ($K = 3$)

Firstly, n points from the space $[0, X_0) \times \cdots \times [0, X_{d-1})$ are randomly sampled. For each point $\boldsymbol{x_i}$, we access its Kth nearest key from the dataset. To do this, we can set a large distance bound δ_i and perform a range query with rectangle $\mathcal{Q}(\boldsymbol{x_i}, \delta_i)$. After that, we select all keys falling in $\mathcal{B}(\boldsymbol{x_i}, \delta_i)$. If the number of selected keys is smaller than K, we augment δ_i until we can get the Kth nearest key $\boldsymbol{x_i'}$. Let $y_i = \|\boldsymbol{x_i'} - \boldsymbol{x_i}\|_2 + \epsilon$, the matrix $\boldsymbol{X} = [\boldsymbol{x_0}, ..., \boldsymbol{x_{n-1}}]$ and the vector $\boldsymbol{y} = [y_0, ..., y_{n-1}]$ are the training input and output, respectively.

Consider a lattice consisting of $m = \tau^d$ nodes where τ is the number of nodes along every dimension. Each node consists of an input-output pair ($\boldsymbol{a_i} \in \mathbb{R}^{\boldsymbol{d}}$, $b_i \in \mathbb{R}$) and the inputs $\{\boldsymbol{a_i}\}$ form a grid cell that contains the input space $\mathbb{D}$ within its convex hull. Let $\boldsymbol{A}$ be a $d \times m$ matrix $\boldsymbol{A} = [\boldsymbol{a_0}, ..., \boldsymbol{a_{m-1}}]$ and $\boldsymbol{b}$ be a vector $\boldsymbol{b} = [b_0, \ldots, b_{m-1}]$. For any $\boldsymbol{x} \in \mathbb{D}$, there are $q = 2^d$ nodes in the lattice that form a shard. $\mathcal{LR}(\boldsymbol{x})$ is linearly interpolated from these nodes' outputs, *i.e.*, $\mathcal{LR}(\boldsymbol{x}) = \sum_{i=0}^{q-1} w_i(\boldsymbol{x}) b_{c_i(\boldsymbol{x})}$, where $c_0(\boldsymbol{x}), \ldots, c_{q-1}(\boldsymbol{x})$ denote the indices of these nodes and $w_i(\boldsymbol{x})$ is the weight corresponding to the c_ith node. We restrict the sum of all $w_i(\boldsymbol{x})$ to be 1 and each $w_i(\boldsymbol{x})$ is inversely proportional to $\|\boldsymbol{x} - \boldsymbol{a_{c_i(\boldsymbol{x})}}\|_2$. Let $\boldsymbol{w}(\boldsymbol{x})$ be the sparse $m \times 1$ vector with $c_j(\boldsymbol{x})$th entry $w_j(\boldsymbol{x})$ for $j = 1, \ldots, q-1$ and zeros elsewhere. For the training inputs $\{\boldsymbol{x_0}, \ldots, \boldsymbol{x_{n-1}}\}$, let $\boldsymbol{W}$ be the $m \times n$ matrix $\boldsymbol{W} = [\boldsymbol{w}(\boldsymbol{x_1}), \ldots, \boldsymbol{w}(\boldsymbol{x_{n-1}})]$. The nest lattice outputs $\boldsymbol{b}^*$ that

minimize total squared-ℓ_2 loss with graph Laplacian regularization [3] are

$$\boldsymbol{b}^* = \arg\min_{\boldsymbol{b}} \frac{1}{n}(\boldsymbol{b}\boldsymbol{W} - \boldsymbol{y})(\boldsymbol{b}\boldsymbol{W} - \boldsymbol{y})^\intercal + \lambda \boldsymbol{b}\boldsymbol{L}\boldsymbol{b}^\intercal \tag{9}$$

where $\boldsymbol{L} = \frac{2\left(\mathrm{diag}(\mathbf{1}^\intercal \boldsymbol{E}) - \boldsymbol{E}\right)}{\mathbf{1}^\intercal \boldsymbol{E} \mathbf{1}}$ and $\boldsymbol{E}$ is the $m \times m$ sparse lattice adjacency matrix with $\boldsymbol{E}_{ij} = 1$ for nodes directly adjacent to one another only. The optimization problem above has the closed form solution $\boldsymbol{b}^* = \frac{1}{n}\boldsymbol{y}\boldsymbol{W}^\intercal(\frac{1}{n}\boldsymbol{W}\boldsymbol{W}^\intercal + \lambda \boldsymbol{L})^{-1}$, where $\frac{1}{n}\boldsymbol{W}\boldsymbol{W}^\intercal + \lambda \boldsymbol{L}$ is a sparse positive definite matrix. Eq. (9) can be solved using Cholesky algorithm [28].

Now the KNN query can be converted to a series of range queries. Given a point $\boldsymbol{q}_{knn}$ and K, we estimate the distance bound as $\delta_0 = \mathcal{LR}(\boldsymbol{q}_{knn})$. We access the keys in $\mathcal{Q}(\boldsymbol{q}_{knn}, \delta_0)$ and filter out those keys outside $\mathcal{B}(\boldsymbol{q}_{knn}, \delta_0)$. If the number of remaining keys is smaller than K, we augment δ_0 and issue a range query repeatedly until we get K nearest keys. More specifically, suppose there are only $K' < K$ keys in $\mathcal{B}(\boldsymbol{q}_{knn}, \delta_0)$ and we want to augment δ_0 to δ_1. Here, we make a reasonable assumption that the density of keys is almost the same in a small area. Accordingly, the Lebesgue measure of $\mathcal{B}(\boldsymbol{q}_{knn}, \delta_1)$ is $\frac{K}{K'}$ times of that of $\mathcal{B}(\boldsymbol{q}_{knn}, \delta_1)$. Thus, it is easy to infer that δ_1 should equal to $\sqrt[d]{\frac{K}{K'}}\delta_0$. For example, in Fig. 6, there are only two keys in $\mathcal{B}(\boldsymbol{q}_{knn}, \delta_0)$. So, we enlarge the query range by augmenting δ_0 to $\delta_1 = \sqrt{\frac{3}{2}}\delta_0$. The new query range $\mathcal{Q}(\boldsymbol{q}_{knn}, \delta_0)$ contains $K = 3$ keys falling in the KNN region $\mathcal{B}(\boldsymbol{q}_{knn}, \delta)$. Algorithm 3 gives the details of LISA based KNN query processing.

Algorithm 3 `KNNQuery`

Input: $\boldsymbol{x}$: a point in $\boldsymbol{V}$ K: a positive integer
Output: $\boldsymbol{R}$: the K nearest keys to $\boldsymbol{x}$

$\delta \leftarrow \mathcal{LR}(\boldsymbol{x})$
while true do
  set $\boldsymbol{R'}$ to be an empty list
  $qr \leftarrow \mathcal{Q}(\boldsymbol{x}, \delta)$
  $\tilde{\boldsymbol{R}} \leftarrow$ the keys falling in qr (Algorithm 2)
  for each key $\boldsymbol{k}$ in $\boldsymbol{R}$ **do**
    if $\boldsymbol{k} \in \mathcal{B}(\boldsymbol{x}, \delta)$ **then**
      add $\boldsymbol{k}$ to $\boldsymbol{R'}$
  if $|\boldsymbol{R'}| < K$ **then**
    $\eta \leftarrow 2$
    if $|\boldsymbol{R'}| > 0$ **then**
      $\eta \leftarrow \sqrt[d]{\frac{K}{|\boldsymbol{R'}|}}$
    $\delta \leftarrow \delta \times \eta$
  else
    break
sort $\boldsymbol{R'}$ along the distance from every key to $\boldsymbol{x}$
add the first K keys in $\boldsymbol{R'}$ to $\boldsymbol{R}$
return $\boldsymbol{R}$

5 DATA UPDATE IN LISA

5.1 Insertion

It is easy to insert a key using LISA. Given a key $\boldsymbol{k}$ to be inserted, we first calculate its mapped value $\mathcal{M}(\boldsymbol{k})$. By calculating $i = \lfloor \mathcal{SP}(\mathcal{M}(\boldsymbol{k})) \rfloor$, we can easily know which shard $\boldsymbol{k}$ falls in. Next, the local model $\mathcal{L}_i$ will find an existing page P that contains $\boldsymbol{k}$. If P is full, a new page P' will be allocated. The keys stored in P and $\boldsymbol{k}$ will be evenly split into two parts and stored in P and P', respectively. Algorithm 4 describes key insertion in LISA.

Note that if P is not full, we simply append $\boldsymbol{k}$ to P without sorting the keys in P. The sorting occurs only when P must be split, because we only need to guarantee that for any two consecutive pages P_1 and P_2, $\inf \mathcal{M}(P_2) \geq \sup \mathcal{M}(P_1)$. Here 'consecutive' means the addresses of P_1 and P_2 are consecutively stored in a $\mathcal{L}_i.PA$ for some i.

Algorithm 4 `Insertion`

Input: $\boldsymbol{k}$: the key to be inserted

1: $m \leftarrow \mathcal{M}(\boldsymbol{k})$, $i \leftarrow \lfloor \mathcal{SP}(m) \rfloor$
2: **if** $\mathcal{L}_i.PA$ is empty **then**
3: allocate a new page P on disk and store $\boldsymbol{k}$ in P
4: append the address of P in $\mathcal{L}_i.PA$
5: **else**
6: $u \leftarrow \mathcal{L}_i.ubound(m)$
7: get the page P whose address is $\mathcal{L}_i.PA[u]$
8: **if** P is not full **then**
9: append $\boldsymbol{k}$ to P
10: **else**
11: load all keys in P into a list L and append r in L
12: sort L by the mapped values of keys in L
13: clear P and save the first $\frac{\Omega}{2} + 1$ keys in L to P
14: allocate a new page P' and save the last $\frac{\Omega}{2}$ keys to P'
15: insert the address of P' to $\mathcal{L}_i.PA$ at index $u + 1$
16: $m' \leftarrow \mathcal{M}(L[\frac{\Omega}{2} + 1])$
17: insert m' to $\mathcal{L}_i.PM$ at index u

5.2 Deletion

Similar to the insertion operation, to delete a given key $\boldsymbol{k}$ with LISA, we first calculate $\mathcal{M}(\boldsymbol{k})$ and $\mathcal{SP}(\mathcal{M}(\boldsymbol{k}))$. Then, we use the corresponding local model to find if there is a page that contains $\boldsymbol{k}$. If so, the key will be removed from the page. A page is freed only when it does not store any keys. To increase the utilization ratio of pages, we can merge two consecutive pages overlapping with the same shard if the numbers of keys stored in both pages are small. Suppose P_0 and P_1 are two pages whose addresses are $\mathcal{L}_i.PA[j]$ and $\mathcal{L}_i.PA[j+1]$, respectively, for some i and j; n_0 and n_1 are the numbers of keys stored in P_0 and P_1, respectively. If $n_0 + n_1 \leq \Omega$, P_0 and P_1 can be merged. Meanwhile, $\mathcal{L}_i.PA$ and $\mathcal{L}_i.PM$ need to be modified. The deletion algorithm with merging operations is shown in Algorithm 5 where

$\#\mathcal{L}_i.PA[j]$ denotes the number of keys stored in the page whose address is $\mathcal{L}_i.PA[j]$.

Algorithm 5 `DeletionWithMerge`

Input: $\boldsymbol{k}$: the key to be deleted

$m \leftarrow \mathcal{M}(\boldsymbol{k})$, $i \leftarrow \lfloor \mathcal{SP}(m) \rfloor$
$l \leftarrow \mathcal{L}_i.lbound(m)$, $u \leftarrow \mathcal{L}_i.ubound(m)$
$pid \leftarrow -1$, $T \leftarrow \mathcal{L}_i.PA.size()$
for $j \leftarrow l$ to u **do**
  get the page P whose address is $\mathcal{L}_i.PA[j]$
  if P contains $\boldsymbol{k}$ **then**
    delete $\boldsymbol{k}$ from P, $pid \leftarrow j$
    if P is empty **then**
      free P, erase $\mathcal{L}_i.PA[j]$ and $\mathcal{L}_i.RN[j]$
      if $\mathcal{L}_i.PM.size() > 0$ **then**
        $j' \leftarrow \min\left(\mathcal{L}_i.PM.size() - 1, \max(0, j-1)\right)$
        delete $\mathcal{L}_i.PM$'s j'th item
      return
$l \leftarrow -1$
if $pid \geq 0$ and $T > 1$ **then**
  if $pid = 0$ **then**
    if $\#\mathcal{L}_i.PA[0] + \#\mathcal{L}_i.PA[1] \leq \Omega$ **then**
      $l \leftarrow 0$
  else if $pid = T-1$ **then**
    if $\#\mathcal{L}_i.PA[T-2] + \#\mathcal{L}_i.PA[T-1] \leq \Omega$ **then**
      $l \leftarrow T-2$
  else
    if $\#\mathcal{L}_i.PA[j-1] + \#\mathcal{L}_i.PA[j] \leq \Omega$ **then**
      $l \leftarrow j-1$
    else if $\#\mathcal{L}_i.PA[j] + \#\mathcal{L}_i.PA[j+1] \leq \Omega$ **then**
      $l \leftarrow j$
if $l \geq 0$ **then**
  get pages P_0 and P_1 whose addresses are $\mathcal{L}_i.PA[l]$ and $\mathcal{L}_i.PA[l+1]$, respectively
  append all keys in P_1 to P_0
  free P_1 and erase $\mathcal{L}_i.PA[l+1]$

6 EXPERIMENTS

In this section, we report on the experimental studies that compare LISA with selected alternative methods.

6.1 Experimental Settings

Datasets. We use the following spatial datasets.

• **imis-3months**[2] is collected by IMIS Hellas S.A., a company focusing on AIS technology and Public Information Systems. The dataset includes 168,2420,595 records each having a point captured in longitude and latitude. After duplicate removal, 98,170,016 records remain.

• **ImageNet** is generated using the original images dataset[3] in the ImageNet database [8]. Firstly, we resize all images to the resolution of 450×600. For each resized image, we get the 3-channel values of all pixels indexed by $(50i + 25, 50j + 25)$, where $0 \leq i < 9$ and $0 \leq j < 12$. For each pair of pixels indexed by $(50i_0 + 25, 50j_0 + 25)$ and $(50i_1 + 25, 50j_1 + 25)$ satisfying $12(i_0 + i_1) + j_0 + j_1 = 53$, we concatenate the 3-channel values of both pixels to form a 6-dimensional vector. Thus, 54 vectors are generated from an image. After collecting the vectors from all resized pages and removing the duplicates, 72,891,949 vectors remain.

• **Uniform (2d-6d)** are 5 synthetic datasets in $\mathbb{R}^2$ to $\mathbb{R}^6$, respectively. Every dataset contains 100M points that are randomly sampled from a uniform distribution.

• **Zipf (2d-6d)** are 5 synthetic datasets in $\mathbb{R}^2$ to $\mathbb{R}^6$, respectively. Every dataset contains 100M points that are randomly sampled from a Zipf distribution.

Competitors. We compare LISA with **Baseline** (the baseline method described in Section 2.2) and four existing methods: **R-tree** [13], **R*-tree** [4], **KD-tree** [5] and **ZM** [29]. The node capacities of R-tree and R*-tree are set to be Ω. We use the BFS (Breadth-first search) algorithm to traverse and number nodes in a KD-tree. When saving a KD-tree to disk pages, the nodes numbering from $i \times \Omega$ to $(i+1) \times \Omega - 1$ are saved in the same page. For ZM/Baseline, the keys in the initial dataset are sorted according to their Z-order/mapped values. Similarly, the keys whose Z-order/mapped values numbering from $i \times \Omega$ to $(i+1) \times \Omega - 1$ are saved in the same page.

Evaluation Metrics. For each dataset, we randomly select 50% of the points as the initial dataset $\boldsymbol{I}$. The other 50% form the extra dataset $\boldsymbol{E}$ to be inserted into the databases. From $\boldsymbol{I} \cup \boldsymbol{E}$, 50% of the points are randomly selected as the dataset $\boldsymbol{D}$ to be deleted. We conduct experiments on three configurations. First, we build all five methods using $\boldsymbol{I}$ in configuration Init. Next, we apply insertions of keys in $\boldsymbol{E}$ in configuration AI. Finally, we delete keys in $\boldsymbol{D}$ from the data in configuration AD. Four metrics on three configurations are used to evaluate the performance of the methods: **Size** indicates the disk storage space that a method consumes. **IO** is the average number of pages to be loaded for a range/KNN query. In order to have clear comparison in some cases, we also use metrics **IO Ratio** and **Size Ratio**. They are the ratio of a method's cost to R-tree's cost.

Parameter settings. Table 2 shows the parameter settings. Every number in a key is of double type in our datasets.

Table 2: Parameter settings

Parameter	Setting
Disk page size (PS)	4096 bytes
MBR size (MS)	$8 \times 2 \times d = 16d$ bytes
Page address size (AS)	4 bytes
Ω	$\frac{\text{PS}}{\text{MS + AS}}$
Ψ	Ω
σ	100

When generating grid cells, the space $\boldsymbol{V}$ is partitioned along every axis into the same number of parts, *i.e.*, $T_0 = T_1 =$

[2] http://chorochronos.datastories.org/?q=content/imis-3months
[3] http://image-net.org/download-images

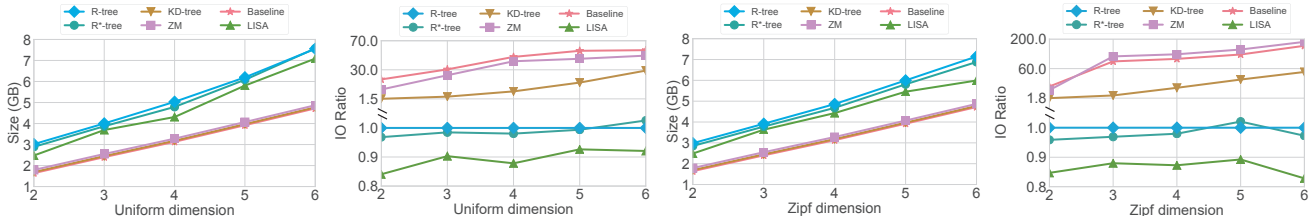

Figure 7: Performance on configuration Init

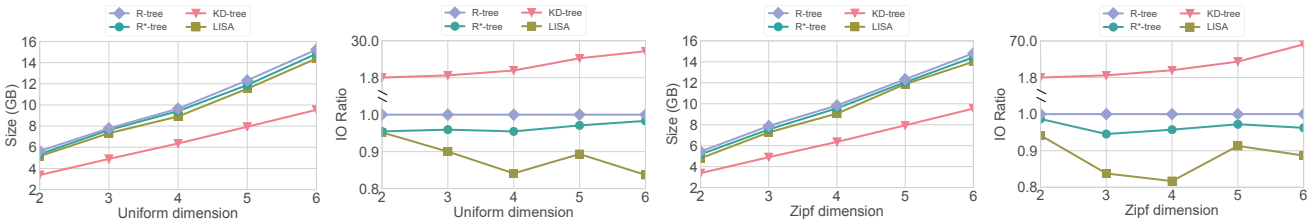

Figure 8: Performance on configuration AI

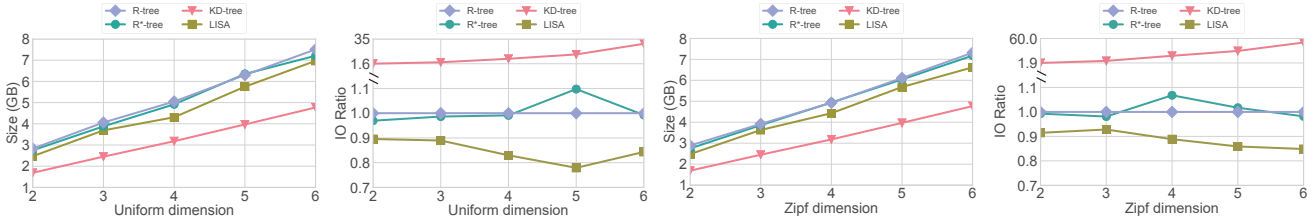

Figure 9: Performance on configuration AD

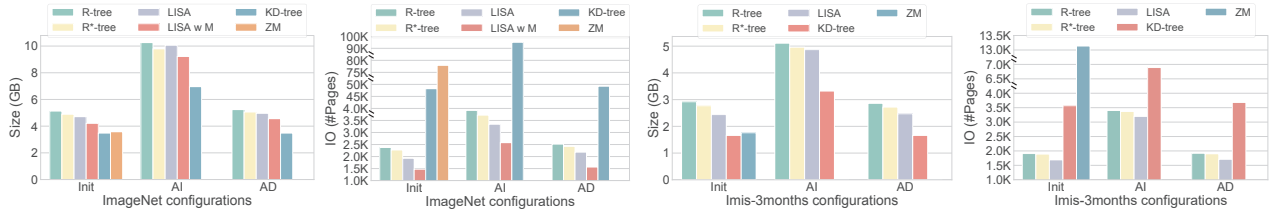

Figure 10: Performance on ImageNet (Left) and Imis-3month (Right) datasets

$\cdots = T_{d-1} = T$. We set T to be 240, 90, 32, 18, 12 when $d = 2, 3, 4, 5, 6$, respectively. For a Zipf dataset with $d > 2$, we use the modified strategy to partition the space (Section 3.1). For each dataset, we randomly generate 10,000 query rectangles and 10,000 points for KNN queries. In a query rectangle, each side length is a random value in $(0, \frac{1}{4}\Delta_i)$, where Δ_i is the distance between the min and max values on the ith axis.

In our experiments, the parameters of the mapping function $\mathcal{M}$, the shard prediction function $\mathcal{SP}$, all local models and the lattice regression model $\mathcal{LR}$ are loaded into the main memory before any query is executed. Suppose N is the number of disk pages to store those parameters. Let H be an integer value such that an R-tree's top H highest levels contain at least N internal nodes while $H - 1$ highest levels contain less than N internal nodes. All nodes in the top H levels are also loaded into the main memory before any query is executed. Such N (or more) IOs are excluded in the IO costs for subsequent queries in our measurements. Table 3 compares the initial memory consumptions of LISA's parameters and R-tree's pre-loaded internal nodes for every Zipf/Uniform initial dataset. Apparently, LISA's parameters consume considerably less memory than R-tree.

Table 3: Initial memory consumption (MB)

Method		$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 6$
Uniform	R-tree	24.2	48.1	79.7	122.2	174.9
	LISA	**9.1**	**12.4**	**14.2**	**17.8**	**25.5**
Zipf	R-tree	23.9	47.1	78.0	120.1	172.2
	LISA	**9.3**	**12.6**	**14.5**	**18.6**	**26.1**

6.2 Range Query Performance

Fig. 7, 8, 9 and 10 report the overall performance results of LISA, R-tree, R*-tree, KD-tree, ZM and Baseline on range query. We compare them on three configurations. On the configuration Init, we build LISA and ZM using the dataset $\boldsymbol{I}$ and insert $\boldsymbol{I}$ into R-tree, R*-tree and KD-tree. Then, the dataset $\boldsymbol{E}$ is inserted into LISA using Algorithm 4 on the configuration AI. Meanwhile, we insert $\boldsymbol{E}$ into the three trees. Finally, on the configuration AD, we delete $\boldsymbol{D}$ from LISA and the three trees.

Clearly, LISA achieves better performance on the IO consumption of range queries. In most scenarios, LISA saves more than 10-20% IO consumption compared to R-tree and R*-tree. Also, LISA needs 5% to 10% less disk storage than R-tree and R*-tree on average. LISA has clear advantages over KD-tree, ZM and Baseline. Although the size of LISA is larger than those of KD-tree, ZM and Baseline, they perform considerably less efficiently on range queries. In most cases, they need to read 10 times more pages than LISA. For simplicity, we exclude Baseline in the following experiments.

The key distribution of 'ImageNet' is complicated. To investigate if the modified partitioning strategy (Section 3.1) works, we build two instances of LISA: one using the modified strategy (LISA w M) and the other using the original strategy (LISA) for this dataset. As shown in the left part of Fig. 10, LISA w M processes range queries fastest among the six methods.

6.3 Effect of Dataset Cardinality

In order to investigate how the dataset cardinality affects the performance of range queries, we design an experiment as follows. For the dataset of 3d-Uniform and 3d-Zipf, we randomly pick up 10M, 20M, 30M, 40M, 50M keys from the initial dataset $\boldsymbol{I}$ and use them to build LISA and other methods. The comparative results are reported in Fig. 11.

Clearly, all five methods' sizes increase almost linearly as the dataset size increases. In terms of IO cost, KD-tree and ZM still perform worst. The number of pages that LISA reads is only about 85% to 90% of that of R-tree and R*-tree.

6.4 LISA Under Many Insertions

We also conduct experiments to analyze how LISA and the other methods perform if we insert many new keys. We first build R-tree, R*-tree, KD-tree and LISA using the 3d-uniform and 3d-Zipf initial datasets. Then, we repeatedly insert 50M keys into all methods and observe their sizes and performance on range queries. Fig. 12 shows the performance results.

All methods' sizes increase linearly as more keys are inserted. As shown in Fig. 12, for 3d-Uniform dataset, the storage cost of LISA is less than that of R-tree and R*-tree no matter how many keys are inserted. R-tree or R*-tree perform slightly better than LISA only when we insert at least 150M 3d-Zipf keys, three times the size of the initial dataset.

Now considering the metric of IO, KD-tree still performs much worse than the others. LISA has the biggest advantage over R-tree and R*-tree if there is no data update operations. Thus a static database will benefit greatly from LISA. When keys are inserted, the IO consumption of LISA for range queries increases. Still, LISA performs better than R-tree in all scenarios. When the number of keys inserted is four times the cardinality of the initial dataset, the advantage of LISA on the metric of IO is not so clear. In this case, we can choose to re-generate the cells and build the mapping function $\mathcal{M}$ and the shard prediction function $\mathcal{SP}$. In our experiments, generating cells and building $\mathcal{M}$ using the initial dataset with a cardinality of 50M takes only about 10 minutes, and training $\mathcal{SP}$ takes a few hours in a single machine. Considering the fact that LISA outperforms R-tree and R*-tree in terms of storage consumption and IO cost, we are confident that LISA has great potential in spatial data processing.

6.5 Response Time Comparison

We also compare LISA with the alternatives in terms of the average response time, which equals CPU time + IO time, on a single range query. We issue and process 10,000 range queries using LISA and the other four methods on the 'ImageNet' initial dataset. Table 4 shows the average CPU time and the response time per query for the five methods. In the computer used to run the experiments, recovering a disk page takes about 15 μs.

Clearly, for all methods but KD-tree, CPU time is only a small part in the response time. Compared to R-tree and R*-tree, LISA has no advantage in terms of CPU time. However, its IO cost is much smaller and thus it can get faster response for range queries.

Table 4: Performance of response time

Method	3D Uniform		ImageNet	
	CPU time (ms)	Response time (ms)	CPU time (ms)	Response time (ms)
R-tree	1.34	11.26	3.85	39.50
R*-tree	1.31	11.08	3.61	37.79
KD-tree	729	765.5	5,655	6,378
ZM	1.97	246.4	2.32	1,173
LISA	1.43	**10.07**	5.08	**27.22**

6.6 KNN Query Performance

We also compare our LISA with R-tree, R*-tree and KD-tree on KNN query on the 3d-Zipf and 3d-Uniform datasets. We randomly generate 10,000 points and vary K from 1 to 10.

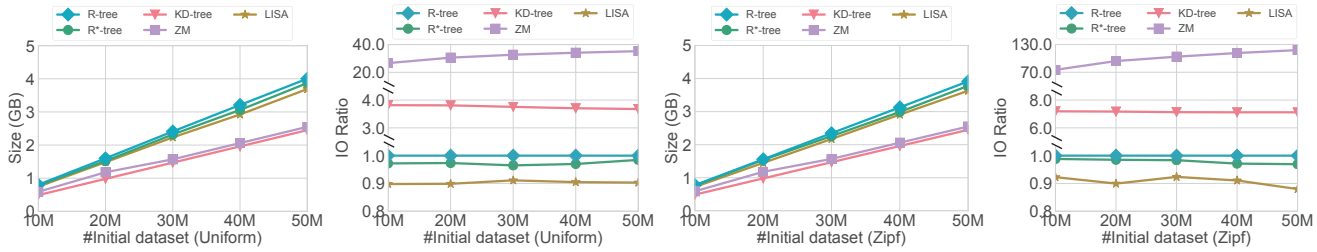

Figure 11: Performance with different dataset cardinalities

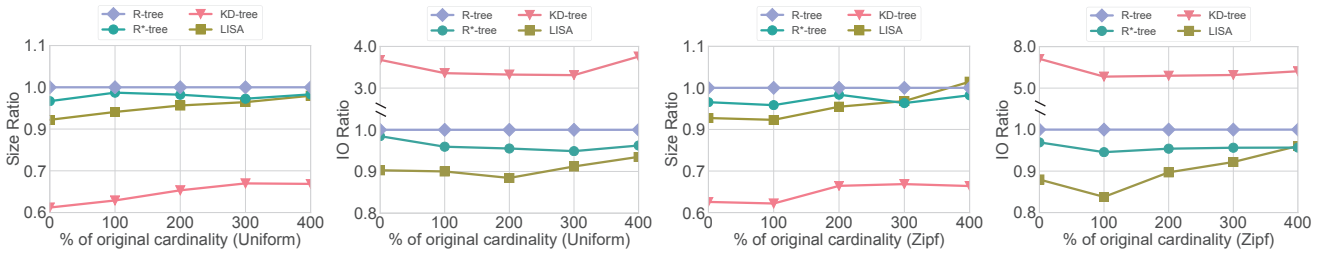

Figure 12: Performance after inserting different amounts of keys

We implement KD-tree's KNN algorithm [10] and two R-tree based KNN algorithms, BFS (best first search) [14] and DFS (depth first search) [23]. Also, for each point and K, we use R-tree's BFS algorithm to calculate the distance between Kth nearest keys to this point in advance. This distance is the ideal distance bound for a KNN algorithm that converts KNN queries into range queries. This method is named as 'Ideal'. The IO cost and CPU time for KNN queries of the five methods with varying K are reported in Fig. 13 and Table 5, respectively. The performance gap between R-tree based BFS/DFS and R*-tree based BFS/DFS is very small. Therefore, we exclude R*-tree in Fig. 13 to make the comparison easy to observe.

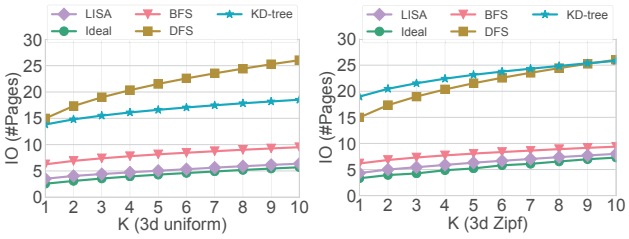

Figure 13: IO cost on KNN query

Referring to Fig. 13, LISA and Ideal perform closely and considerably better than others in terms of IO cost. Compared to BFS, LISA and Ideal only need to load less than 80% pages for each KNN query. LISA and Ideal have clearer advantages over DFS and KD-tree. Ideal achieves slightly better performance than LISA with the lattice regression model $\mathcal{LR}$. The performance gap between LISA and Ideal is negligible, which shows our $\mathcal{LR}$ is able to estimate a distance bound δ sufficiently close to the ideal distance bound. It is also evident from Table 5 that building $\mathcal{LR}$ takes about 22 minutes only. Both steps of building $\mathcal{LR}$ need to be executed only once. Also, the average CPU time of LISA on KNN queries is much smaller than that of the alternatives.

Table 5: CPU time of KNN tasks

Tasks		Time
Building $\mathcal{LR}$	Sampling training data	165 s
	$\mathcal{LR}$ training	22.15min
KNN with $\mathcal{LR}$	$\mathcal{A}$pproximating distance bound	0.002 ms
	$\mathcal{R}$ange Query	0.32 ms
Algorithm on single query ($K = 10$)	LISA ($\mathcal{A}$ + $\mathcal{R}$)	**0.322** ms
	R-tree BFS	1.05 ms
	R-tree DFS	0.61 ms
	KD-tree	0.68 ms

7 RELATED WORK

Spatial indexes. Traditional spatial index structures generally fall into three categories. The first category partitions the space into regions and then index those regions. Typical examples are KD-tree [5], quadtree [10] and octree [18]. A KD-tree is a special BSP tree (binary space partitioning tree) [11] which organizes points in a K-dimensional space. In a quadtree often used for 2D data, each internal node has exactly four quadrants as children. As a three-dimensional analog of quadtrees, an octree node has eight children.

The second category partitions the dataset into different subsets and then indexes those subsets. Typical indexes are

R-tree [13] and its variants such as R*-tree [4], Hilbert R-tree [15] and R+-tree [26]. Specifically, spatial data objects are approximated using the minimum bounding rectangle (MBR), multiple small MBRs are grouped together into a larger MBR, and the process recurs until a single largest MBR is formed. All MBRs correspond to R-tree nodes at different levels from root (the single largest MBR) level to the leaf level (all object MBRs). Each node contains a number of index entries each having an MBR and a pointer to its corresponding child node (at an internal level) or object (at the leaf level). The parent-child relationship between nodes is determined by that between a larger MBR and those smaller MBRs that together form the larger one. R-tree and R*-tree differ in how data updates (insertion and deletion) are handled, whereas R+-tree splits an object into multiple parts and MBRs in different leaves to avoid MBR overlap. As a generalization of R-tree, M-tree [6] features an overall structure and construction process similar to the counterparts in R-tree. However, unlike R-tree that uses MBRs to organize objects and nodes, M-tree employs a distance metric and the triangle inequality to organize objects into nodes and the nodes into their parent nodes.

Range query processing via R-tree follows the filter-and-refinement paradigm. The search starts from the root and goes to a child node only if the child node's MBR overlaps with the query range. The search goes deeply in a recursive manner, possibly covering multiple paths to the leaf level, until all objects whose MBR overlaps with the query range are fetched through the leaf nodes. Those nodes that are not searched are filtered, whereas those fetched objects are further checked (refined) to see if their concrete geometry satisfies the query condition. *K*NN query processing via R-tree takes the depth-first search (DFS) [23] or the best-first search (BFS) [14]. DFS prunes unqualified tree nodes (and data objects) using distance metrics between a query object and MBRs. BFS prioritizes the search such that those tree nodes that are closer to the query object are always visited first.

In the third category, a multi-dimensional space is transformed to 1-dimensional and the data objects in turn fall into regions that are ordered sequentially. Subsequently, the regions (thus the objects) are indexed by a B+-tree [7]. Query processing requires the same transformation plus extra handling to avoid false negatives in the results. As a typical example, Microsoft SQL Server [1] uses B+-trees and Hilbert space filling curves [25] to build its spatial indexes. In MongoDB [2], spatial indexes are built with B+-trees and GeoHash [20]. Moreover, UB-tree [22] is basically a B+-tree [9] with records stored according to Z-order [25].

Learned indexes. Kraska *et al.* [17] propose the idea of learned indexes, which applies machine learning models to data access in databases. By capturing the relationship between search keys and their positions in the database through CDFs, the learned index replaces the traditional B-tree with a recursive model index (RMI) that consists of a number of simple models staged into a hierarchy. Given a search key, the RMI predicts where the corresponding data record is positioned, and it guarantees that the record is found around the predicted position within a known error bound. However, the RMI only works for scalar values and falls short in spatial data. The RMI idea is extended to index multi-dimensional data [16] through a model that transforms multi-dimensional data into 1-dimensional data. Such an idea does not perform well because it heavily relies on the performance of the transformation model which however is very hard to design for arbitrary datasets. Mitzenmacher [19] proposes a variation of learned Bloom filter [17] that features two layers of Bloom filters surrounding the learned function. Wang *et al.* [29] propose a learned Z-order Model (ZM) index which combines the Z-order space filling curve and the staged learning model to process point and range queries. However, the ZM index does not support data updates or *K*NN queries.

8 CONCLUSION AND FUTURE WORK

In this work, we propose LISA—a novel learned index structure for spatial data. LISA consists of four parts: 1) the representation of grid cells, 2) a partial monotone mapping function $\mathcal{M}$ that maps spatial keys to 1-dimensional mapped values, 3) a monotone shard prediction function $\mathcal{SP}$ that predicts the shard id for a given mapped value and partitions the mapped space into shards, and 4) local models that carry out intra-shard operations with respect to disk pages. We conduct extensive experiments using real and synthetic datasets. The experimental results demonstrate that LISA outperforms traditional spatial indexes in terms of storage and IO costs for range and *K*NN queries. Moreover, LISA supports data insertion and deletion operations efficiently.

This work opens several directions for future research on learned indexes for spatial data. First, it is interesting to design more efficient *K*NN query algorithms using LISA. Second, it is relevant to investigate other possible forms for the functions used in LISA. Third, it makes sense to study other query types (*e.g.*, spatial joins and closest pairs) using LISA. Last but not least, it is interesting to investigate how LISA can be adapted or modified to index massive trajectories.

ACKNOWLEDGMENTS

We thank all anonymous reviewers. This research is supported by Natural Science Foundation of China (No. 61925603, 61772460). Hua Lu's work was conducted when he was employed at Aalborg University, Denmark. Hua Lu and Gang Pan are corresponding authors.

REFERENCES

[1] Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server/default.aspx. Accessed April 2020.

[2] MongoDB. https://www.mongodb.com/. Accessed April 2020.

[3] Rie Kubota Ando and Tong Zhang. 2006. Learning on Graph with Laplacian Regularization. In *NIPS*. 25–32.

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

[5] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

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

[7] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. 2009. ImageNet: A large-scale hierarchical image database. In *CVPR*. 248–255.

[9] Ramez Elmasri and Sham Navathe. 2017. *Fundamentals of database systems*. Pearson.

[10] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.* 4 (1974), 1–9.

[11] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. 1980. On visible surface generation by a priori tree structures. In *SIGGRAPH*. 124–133.

[12] Eric K. Garcia and Maya R. Gupta. 2009. Lattice Regression. In *NIPS*. 594–602.

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

[14] Gísli R. Hjaltason and Hanan Samet. 1999. Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.* 24, 2 (1999), 265–318.

[15] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB*. 500–509.

[16] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *CIDR*.

[17] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504.

[18] Donald Meagher. 1982. Geometric modeling using octree encoding. *Computer Graphics and Image Processing* 19, 2 (1982), 129–147.

[19] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *NeurIPS*. 462–471.

[20] Gustavo Niemeyer. 2008. Geohash. *Retrieved June* 6 (2008), 2018.

[21] Michel Scholl Philippe Rigaux and Agnes Voisard. 2002. *Spatial Databases: With Application to GIS*. Morgan Kaufmann Publishers, Chapter 6 Spatial Access Methods.

[22] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. 2000. Integrating the UB-Tree into a Database System Kernel. In *VLDB*. 263–272.

[23] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. 1995. Nearest Neighbor Queries. In *SIGMOD*. 71–79.

[24] Halsey Lawrence Royden and Patrick Fitzpatrick. 1988. *Real analysis*. Vol. 32. Macmillan New York.

[25] Hans Sagan. 2012. *Space-filling curves*. Springer Science & Business Media.

[26] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB*. 507–518.

[27] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *ICLR*.

[28] Gilbert W Stewart. 1973. *Introduction to matrix computations*. Elsevier.

[29] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *MDM*. 569–574.