Efficient Learned Spatial Index with Interpolation Function based Learned Model

Songnian Zhang, Suprio Ray, *Member, IEEE,* Rongxing Lu, *Fellow, IEEE,* Yandong Zheng

Abstract—Recently, researchers have demonstrated that learned index can improve query performance while reducing the storage overhead. It potentially offers an opportunity to address the spatial query processing challenges caused by the surge in location-based services. Although several learned indexes have been proposed to process spatial data, the main idea behind these approaches is to utilize the existing one-dimensional learned models, which requires either converting the spatial data into one-dimensional data or applying the learned model on individual dimensions separately. As a result, these approaches cannot fully leverage or take advantage of the information regarding the spatial distribution of the original spatial data. To this end, in our previous work, we proposed a spatial (multi-dimensional) interpolation function based learned model to develop a spatial learned index and designed efficient range and kNN query strategies over it. However, there are some limitations in the proposed learned model, such as the prediction accuracy and index building time. In this paper, we address the limitations of our previous work and propose a new spatial learned model by employing the characteristics of the spatial interpolation functions and a novel dynamic encoding technique. Detailed experiments are conducted with real-world datasets. The results indicate that our new proposed learned model is better than our previous one in terms of building time, prediction accuracy, and storage overhead simultaneously, and the new learned spatial index is better than the existing learned spatial indexes in query execution time and index building time.

Index Terms—Learned Model, Spatial Interpolation Function, Hybrid Tree, Learned Spatial Index.

1 Introduction

As location-based services (LBS) have been widely deployed and have become highly popular, spatial query processing has attracted considerable interest in the research community. Although several spatial indexes, such as R-tree and k-d tree, have been proposed to facilitate spatial query execution, it is still challenging to process the spatial queries efficiently due to the rapidly growing volume of spatial data. Recently, Kraska et al. [1] suggested substituting the traditional indexes with machine learning based indexes (also called *learned index*). Since then, several follow-up research projects [2], [3], [4], [5], [6] have shown that the learned index can indeed improve query performance by learning data distribution and query workload patterns. In addition to this advantage, some learned index research [3], [6] revealed that, compared to the traditional indexes, the learned index can speed up query processing with a smaller index storage overhead. This can free up expensive main memory for in-memory data processing.

Typically, there are two main aspects involving a learned index, namely, a learned model and a local search. The former is trained and used to quickly locate the approximate position of a search key, while the latter is responsible for refining the accurate position. Since the learned model can greatly reduce the range of local search, in the current study, the local search is usually achieved by performing a local binary or exponential search. This strategy can ensure query performance without introducing additional index storage. However, as the fundamental aspect, finding a reasonable learned model and further employing it in the learned index is still challenging. Currently, most of the learned indexes are constructed based on mainly one of two categories of learned models: machine learning [1] and piecewise linear functions [3]. However, to the best of our knowledge, both of these learned models can be only applied to single dimensional data. As a result, the existing spatial learned indexes either transform multi-dimensional data into one-dimensional data before introducing the learned model as a foundation [7], [8] or apply a learned model on every single dimension [6] separately. Note that although the recently proposed multi-dimensional learned index *Tsunami* [9] studied the correlation over two dimensions to build a learned model for improving its efficiency, it mainly focused on two-dimensional data with strong correlation patterns. Unfortunately, the spatial data (usually consists of the latitude and longitude data) does not have obvious correlation patterns. For this reason, the question then arises, *"Is there a learned model that can be directly applied to any spatial (two-dimensional) data and achieve better performance?"*

Aiming to address the above-mentioned question, in our previous work [10], we explored how to utilize spatial (two-dimensional) interpolation functions as the learned model to directly predict the position of a spatial search key. Based on this idea, we proposed a SPatial inteRpolation functIon based Grid index (SPRIG) to support range and kNN queries over spatial data. The main contributions of our previous work were: i) designed and implemented a spatial learned index SPRIG, which offered a new learned model for developing the spatial learned index; ii) proposed two query execution strategies to support range and kNN queries, respectively. The experimental results showed that our learned index SPRIG achieved up to an order of magnitude better performance than *ZM-index* in range queries

• *S. Zhang, S. Ray, R. Lu, and Y. Zheng are with the Faculty of Computer Science, University of New Brunswick, Fredericton, NB E3B 5A3, Canada (e-mail: szhang17@unb.ca, sray@unb.ca, rlu1@unb.ca, yzheng8@unb.ca).*

and was about 2.7×, 3×, and 9× faster than the multi-dimensional learned index Flood in terms of index building, range queries, and kNN queries, respectively [10]. From these contributions, we can see that our previous work mainly focused on developing a spatial interpolation based index SPRIG, i.e., building a SPRIG system and designing efficient query strategies over it, and the efficiency of the learned model that underpins the SPRIG system was left for future research. Consequently, in this paper, we focus on improving the spatial learned model employed in our SPRIG system, which can be directly applied to any spatial data and is entirely different from the existing spatial learned models that can be only applied to single-dimensional data. There are two limitations with our original spatial learned model: i) the model building time, especially the time cost of training a grid layout, is a bit large; ii) the prediction error may be large in some cases. In the worst case, the prediction error in one dimension may be equal to the grid length in that dimension, leading to a larger local search range. To address the first limitation, we propose a new approach to employ the spatial interpolation function in our learned model, which can reduce building time and storage overhead simultaneously. Regarding the second limitation, it is actually an open problem for all spatial learned models that adopt the space-filling curve techniques. To address it, in this paper, we propose a dynamic encoding technique to keep the prediction error within an upper bound. Additionally, in our previous work, we employed a cost model to obtain an optimal grid layout for the best query performance. However, this cost model needed to be calibrated by executing query workloads in order to obtain the optimal grid parameters. Thanks to our dynamic encoding technique, we introduce a new cost model in this paper, which can obtain an approximately optimal grid layout without requiring a prior execution of query workloads. Specifically, compared to our previous work, we make the following new contributions in this paper:

- First, we elaborate on the details of our spatial learned model: an interpolation function based hybrid tree, denoted as IH-tree, and analyze its limitations.
- Second, we propose a new learned model IH-tree+ by employing the characteristics of spatial interpolation functions and a novel dynamic encoding technique. Compared to IH-tree, IH-tree+ can improve building time, prediction accuracy, and cell locating time, while reducing the storage overhead. Meanwhile, the dynamic encoding technique enables a new cost model, which avoids the need for a prior execution of the query workload and is much more efficient than the cost model used in our previous work [10].
- Third, by compressing IH-tree+ into bit vectors, we propose a compact version of IH-tree+, denoted as C-IH-tree+, for storage-limited scenarios. Compared to IH-tree+, C-IH-tree+ has almost an order of magnitude improvement in storage overhead.
- Finally, we conduct extensive experiments to evaluate our proposed learned models and compare them in terms of building time, prediction accuracy, cell predicting time, and storage overhead. The results show that: i) our IH-tree+ is around 1.8× faster than IH-tree in building time; ii) IH-tree+ outperforms IH-tree by at least 3× in terms of prediction accuracy. In the best case, IH-tree+ achieves 83× improvement in prediction accuracy; iii) IH-tree+ reduces at least 20% storage overhead compared to IH-tree. In addition, we employ IH-tree as the learned model in SPRIG and construct a new learned spatial index, denoted as SPRIG+, by replacing IH-tree with IH-tree+. Furthermore, we compare SPRIG+ against our previous learned index SPRIG [10], a state-of-the-art learned index Flood [6], and a learned spatial index ZM-index [7]. The results show that SPRIG+: i) is better than Flood and ZM-index in query execution time; ii) has the best performance in index building time; iii) is better than SPRIG in storage overhead.

The remainder of this paper is organized as follows. In Section 2, we discuss the related work. Then, we introduce the spatial interpolation function and our SPRIG scheme in Section 3. After that, we present our learned models in Section 4 and cost model in Section 5, followed by performance evaluation in Section 6. Finally, we draw our conclusion in Section 7.

2 Related Work

Kraska et al. [1] presented the idea of the learned index, which is based on learning the relationship between keys and their positions in a sorted array. They adopted a machine learning based technique as the learned model and built a recursive model index (RMI), which predicts the position of a search key within a known error bound. Since then, a variety of learned indexes was proposed to handle one-dimensional data. Recently, Tang et al. [2] proposed a scalable learned index *XIndex* based on RMI, which focuses on handling concurrent writes without affecting the query performance. Very differently, Galakatos et al. [3] exploited the piecewise linear function as the learned model to build a data-aware index *FITing-tree* that replaces leaf nodes of B+-tree with the learned piecewise linear functions. Unlike *FITing-tree*, Ferragina et al. [4] introduced a pure learned index *PGM-index* that does not mix the traditional data structure and learned model. However, their work still focuses on one-dimensional data and uses the existing linear learned model.

Naturally, the idea of the learned index has been extended to spatial and multi-dimensional data. Wang et al. [7] proposed a learned index *ZM-index* for spatial queries. In that work, the authors utilized the Z-order curve to convert two-dimensional data into one-dimensional values, and then applied a machine learning model to predict a key's position on one-dimensional data. Qi et al. [5] refined the idea of *ZM-index* and built a recursive spatial model index (RSMI). Before applying Z-order curve, their work adopts a rank space-based transformation technique to mitigate the uneven-gap problem. Although this work is to deal with range and kNN queries over spatial data, it is for approximate queries rather than accurate queries that are our targets, and it still adopts the one-dimensional learned model. *LISA* [11] is a disk-based spatial learned index that achieves low storage consumption and I/O cost. In this work, the authors used a mapping function to map spatial keys into one-dimensional values and a monotone shard prediction function, which is similar to the piecewise linear functions, to predict the shard id for a given mapped value. Extending to multi-dimensional data, *ML-index* [8] is an RMI based learned index. It first converts the multi-dimensional data into one dimension by employing the i-Distance technique [12]. Based on the one-dimensional data, *ML-index* uses the RMI to estimate the approximate position of a search key. Recently proposed index Flood [6] can also support multi-dimensional data and is very relevant to our work. It adopts the existing learned model as the building block to predict the key's position on a single dimension. By integrating $d-1$ dimensions' positions, where d is the number of dimensions, Flood can locate the cell that covers the search key. *Tsunami* [9] is also a multi-dimensional learned index. This work can address the limitations of Flood by introducing the space-partitioning decision tree and capturing correlations. However, it cannot be applied to spatial data with complex correlation patterns. We refer the readers to [13] for more related work about learned multi-dimensional indexes.

3 Background

Before delving into the details of our learned models, in this section, we introduce the spatial interpolation function and the SPRIG scheme proposed in our previous work.

3.1 Spatial Interpolation Function

Given a set of 2-dimensional sample points $\{(x_i, y_j) | 1 \le i, j \le n\}$ and their corresponding values $\{v_{ij} = f(x_i, y_j) \mid 1 \le i, j \le n\}$, one can construct a spatial (two-dimensional) interpolation function $f(x, y)$ that passes through all these sample points [14]. Afterward, given any point (x, y), it is easy to estimate the value of $f(x, y)$ with the interpolation function. Borrowing the idea from the learned index [1], if we treat v_{ij} as the position of the point (x_i, y_j), we can use $f(x, y)$ to quickly estimate the position of any given point. Moreover, if the sample points are not random but can represent the distribution of the original spatial dataset, we can use fewer sample points to fit the spatial interpolation function for estimating positions. It indicates that the spatial interpolation function can learn the spatial position distribution with a lower storage overhead, which fits well with the goal of the learned index. Therefore, it is feasible and promising to exploit the spatial interpolation function as the learned model.

In the literature, there are many spatial interpolation functions, such as *bilinear interpolation*, *bicubic interpolation*, and *radial based function (RBF) interpolation* [15], [16]. For ease of description, hereinafter we use the *bilinear interpolation* function for our learned model. Note that it is easy to replace the bilinear interpolation function with other interpolation functions in our learned model.

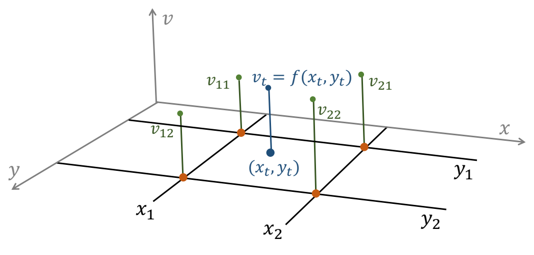


Fig. 1. *Bilinear interpolation* function.

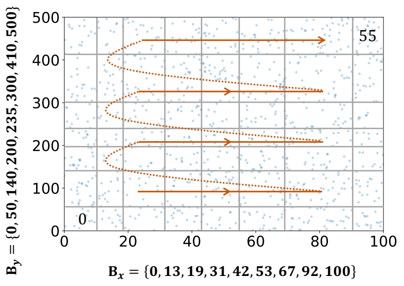


Fig. 2. Adaptive grid $G_{n\times m}$ with space-filling curve, where $n = 8, m = 7$. Cell ids encoded from 0 to $55 = 8 \cdot 7 - 1$ along the orange line.

***Bilinear interpolation* function [17].** Assume there are four points $\{(x_1, y_1), (x_1, y_2), (x_2, y_1), (x_2, y_2)\}$, which form a spatial domain, and the corresponding values $v_{ij} = f(x_i, y_j), i, j \in \{1, 2\}$, as shown in Fig. 1. With these values, we can fit a bilinear interpolation function $f(x, y)$. The intuitive approach of fitting the bilinear interpolation function is to perform the linear interpolation twice. One is along the x dimension, and the other is along the y dimension.

- Linear interpolation in the x dimension:

$$\begin{cases} f(x, y_1) = \frac{x_2 - x}{x_2 - x_1} v_{11} + \frac{x - x_1}{x_2 - x_1} v_{21} \\ f(x, y_2) = \frac{x_2 - x}{x_2 - x_1} v_{12} + \frac{x - x_1}{x_2 - x_1} v_{22}. \end{cases} \tag{1}$$

- Linear interpolation in the y dimension with Eq. (1):

$$\begin{aligned} f(x, y) &= \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) \\ &= \frac{1}{(x_2 - x_1)(y_2 - y_1)} [x_2 - x, x - x_1] \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix} \end{aligned} \tag{2}$$

Given any point $\{(x_t, y_t) \mid x_1 < x_t < x_2, y_1 < y_t < y_2\}$ in the area, we can obtain the value $v_t = f(x_t, y_t)$. Although it is simple to obtain unknown values by the interpolated spatial functions, it is still challenging to build an efficient learned model with the spatial interpolation function. In this paper, we enable the spatial interpolation functions to build our spatial learned index.

3.2 Our SPRIG System

In our previous work [10], we proposed a learned spatial index, SPRIG, to deal with range and kNN queries on an adaptive grid. Here, we simply recall index building and query processing.

Index Building. Our SPRIG consists of three components: 1) An $n \times m$ grid layout $G_{n\times m}$, where n is the number of columns along x dimension, while m is for y dimension; 2) A table $\mathcal{T}$; and 3) The spatial interpolation function based learned model M. The main idea of building $G_{n\times m}$ is to find boundary values for each dimension. These boundary values can divide the space into several columns along one dimension and ensure each column has an almost equal number of data records. We define x and y dimensions' boundaries as set B_x and B_y, respectively, where $|B_x| = n + 1$ and $|B_y| = m + 1$. Totally, there are $n \times m$ cells for the grid, and we have $G_{n\times m} = (B_x, B_y)$. Fig. 2 illustrates an example of the adaptive grid, in which $B_x = \{0, 13, 19, 31, 42, 53, 67, 92, 100\}$ and $B_y = \{0, 50, 140, 200, 235, 300, 410, 500\}$. See Algorithm 1 in [10] for calculating boundary values.

Next, we employ a simple space-filling curve to index these cells. First, we allocate integers in the range [0, $n\times m-$

1] as cell ids along x dimension and define a 2-dimensional array $\mathsf{C_{id}}$ to store these cell ids: $\mathsf{C_{id}}[i][j] = j \cdot n + i, 0 \leq i < n$ and $0 \leq j < m$. Afterward, we can build a table $\mathcal{T}$ to map the cell id to the covered records, in which the key is the cell id and the value is a pair $(\mathit{firstAddress}, \mathit{size})$ indicating the pointer to the first record and the number of records in the cell. Based on $\mathsf{G}_{n\times m}$ and $\mathsf{C_{id}}$, we can fit the spatial interpolation function $\mathsf{C_{id}} \leftarrow \mathsf{M}(\mathsf{B_x}, \mathsf{B_y})$ as our learned model.

Query processing. The principle behind processing queries is to locate the cell that the query point falls in. Then, based on the located cell, the range and kNN query strategies can be adopted to filter and refine the desired data points. Specifically, there are two steps to locate the cell that covers the query point:

- *Step-1*. Predict cell id with learned model M. Given a query point (x_q, y_q), we can use the learned model to obtain a predicted cell id $\mathit{pid} = \mathsf{M}(x_q, y_q)$. Afterward, it is simple to calculate the locations of *pid* in set $\mathsf{B_x}$ and $\mathsf{B_y}$. That is, $l_{px} = \mathit{pid} \mod n$, and $l_{py} = \mathit{pid} \ / \ n$.
- *Step-2*. Locate the real cell id with local binary search. Given a pair of error guarantee (eg_x, eg_y), by applying local binary searches on $\mathsf{B_x}$ and $\mathsf{B_y}$, we can obtain the real x and y locations of the query point, denoted as l_{rx} and l_{ry}. The search range on $\mathsf{B_x}$ set is $[l_{px} - eg_x, \ l_{px} + eg_x]$, while it is $[l_{py} - eg_y, \ l_{py} + eg_y]$ on $\mathsf{B_y}$ set. Finally, we can obtain the real cell id of (x_q, y_q), namely, $\mathit{rid} = l_{ry} \cdot n + l_{rx}$.

Take $(x_q, y_q) = (38, 420)$ in Fig. 2 as an example, which should fall in the cell with id = 51, i.e., $\mathit{rid} = 51$. Suppose the predicted cell id $pid = \mathsf{M}(x_q, y_q) = 44$ and the $(eg_x, eg_y) = (1, 2)$. In *Step-1*, $l_{px} = \mathit{pid} \mod n = 44 \mod 8 = 4$ and $l_{py} = \mathit{pid} \ / \ n = 44/8 = 5$. In *Step-2*, since $eg_x = 1$, the binary search range on $\mathsf{B_x}$ is $[3, 5]$, and it is $[3, 7]$ on $\mathsf{B_y}$ due to $eg_y = 2$. Since $\mathsf{B_x}[3] = 31 < 38 < \mathsf{B_x}[4] = 42$, $l_{rx} = 3$. Similarly, as $\mathsf{B_y}[6] = 410 < 420 < \mathsf{B_y}[7] = 500$, we can get $l_{ry} = 6$. As a result, $\mathit{rid} = l_{ry} \cdot n + l_{rx} = 6 \cdot 8 + 3 = 51$.

After obtaining the real cell id *rid* of the query point, one can perform our range and kNN strategies starting from the cell to collect query results. Since the range and kNN query strategies are not the focus of this paper, we will not re-describe them here. See details in our previous work [10].

4 Our Proposed Learned Models

From Section 3.2, we can see that the key idea behind our learned index SPRIG is to locate the real cell id of a query point. Although SPRIG is specifically suitable for spatial data, it still follows the basic principle of the most of existing learned indexes to locate a search key, i.e.,

predict position with learned model + refine with the local search

As shown in Fig. 3, given a key, the learned model is first used to predict the approximate position of the key, denoted as *pos*. Then, the local search is performed in the range of [*pos* - *eg*, *pos* + *eg*] to obtain the real position of the search key, where *eg* is the error guarantee or prediction error [1]. Since the local search is usually achieved by performing a binary or exponential search, naturally, a series of questions about the learned model arises: how to build

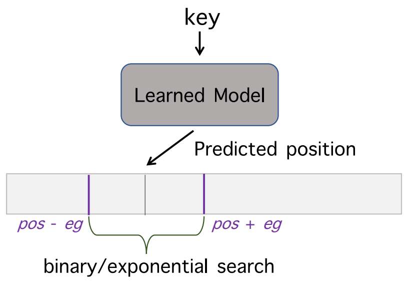


Fig. 3. Learned model + Local search. pos is the predicted position, and eg is error guarantee or prediction error.

the learned model and how does the learned model work? In this section, we first introduce the details of our learned model, and then propose an optimized version in terms of building costs, storage overhead, and prediction accuracy. In addition, we propose a compressed learned model for the situations that concern about the storage overhead.

4.1 Interpolation-based Learned Model

As discussed in Section 3.2, our learned model is built with the boundary sets: $\mathsf{B_x}$, $\mathsf{B_y}$, and the cell id array $\mathsf{C_{id}}$. The main idea is to recursively divide the 2-dimensional region into four sub-regions over the boundary sets $\mathsf{B_x}$ and $\mathsf{B_y}$, and each leaf node will be represented by a fitted spatial interpolation function. Obviously, our learned model is an interpolation-based hybrid tree structure. To ease the description,we denote it as IH-tree, i.e., M=IH-tree.

4.1.1 Model Building

Given the tree height h $(1 < h < \log \min(n, m))$, which is trained with the cost model introduced in [10], we show the model building process as follows.

Internal node. When recursively partitioning the 2-dimensional region, we calculate the center point of the corresponding (sub)region as an internal node. Since we divide the region over the generated grid, i.e., $\mathsf{G}_{n\times m} = (\mathsf{B_x}, \mathsf{B_y})$, here the center point is the middle position projecting to $\mathsf{B_x}$ and $\mathsf{B_y}$. For example, the root node is $(x = \lfloor n/2 \rfloor, y = \lfloor m/2 \rfloor)$ and represents the whole region. Each internal node has four children (sub-regions), which are encoded from 0 to 3 along 'Σ' shape. The center point of the children nodes can be calculated with the following equations:

$$\mathit{Child\ 0} : \begin{cases} x = \lfloor (\mathit{parent}.x - \mathit{parent}.\mathit{bl}.x)/2 \rfloor + \mathit{parent}.\mathit{bl}.x \\ y = \lfloor (\mathit{parent}.y - \mathit{parent}.\mathit{bl}.y)/2 \rfloor + \mathit{parent}.\mathit{bl}.y, \end{cases} \tag{3}$$

$$\mathit{Child\ 1} : \begin{cases} x = \lfloor (\mathit{parent}.\mathit{tr}.x - \mathit{parent}.x)/2 \rfloor + \mathit{parent}.x \\ y = \lfloor (\mathit{parent}.y - \mathit{parent}.\mathit{bl}.y)/2 \rfloor + \mathit{parent}.\mathit{bl}.y, \end{cases} \tag{4}$$

$$\mathit{Child\ 2} : \begin{cases} x = \lfloor (\mathit{parent}.x - \mathit{parent}.\mathit{bl}.x)/2 \rfloor + \mathit{parent}.\mathit{bl}.x \\ y = \lfloor (\mathit{parent}.\mathit{tr}.y - \mathit{parent}.y)/2 \rfloor + \mathit{parent}.y, \end{cases} \tag{5}$$

$$\mathit{Child\ 3} : \begin{cases} x = \lfloor (\mathit{parent}.\mathit{tr}.x - \mathit{parent}.x)/2 \rfloor + \mathit{parent}.x \\ y = \lfloor (\mathit{parent}.\mathit{tr}.y - \mathit{parent}.y)/2 \rfloor + \mathit{parent}.y, \end{cases} \tag{6}$$

where $(\mathit{parent}.x, \mathit{parent}.y)$ is the center point of the parent node, and $\{\mathit{parent}.\mathit{bl}, \mathit{parent}.\mathit{rt}\}$ are the bottom left and top right points of the parent' region, respectively. When the height of the IH-tree reaches to $h-1$, we will stop partitioning the sub-regions and turn to generate leaf nodes.

Leaf node. Each leaf node represents a sub-region and stores the fitted spatial interpolation function of the sub-region. Taking the bilinear interpolation function as an example, we can have the following function:

$$f(x, y) \simeq \phi_1 + \phi_2 x + \phi_3 y + \phi_4 xy. \tag{7}$$

Fitting such a function for a sub-region indicates calculating the four coefficients $\{\phi_i \mid i \in [1,4]\}$ so that we can compute $f(x, y)$ when x, y are given. If we have a formula expansion of Eq. (2), we can obtain the following equation to calculate the coefficients $\{\phi_i \mid i \in [1,4]\}$:

$$\begin{bmatrix}\phi_1\\ \phi_2\\ \phi_3\\ \phi_4\end{bmatrix} = \frac{1}{(x_2-x_1)(y_2-y_1)}\begin{bmatrix} x_2y_2 & -x_2y_1 & -x_1y_2 & x_1y_1\\ -y_2 & y_1 & y_2 & -y_1\\ -x_2 & x_2 & x_1 & -x_1\\ 1 & -1 & -1 & 1\end{bmatrix}\begin{bmatrix}v_{11}\\ v_{12}\\ v_{21}\\ v_{22}\end{bmatrix}. \tag{8}$$

In this paper, $\{(x_1, y_1), (x_1, y_2), (x_2, y_1), (x_2, y_2)\}$ are the intersection points of the grid $\mathsf{G}_{n\times m}$, and $v_{ij} = f(x_i, y_j), i, j \in \{1, 2\}$, are the corresponding cell ids retrieved from the 2-dimensional array C_{id}. After calculating these four coefficients $\{\phi_i \mid i \in [1,4]\}$, the leaf node can store them to represent the sub-region. Note that we can replace the bilinear interpolation functions with other spatial interpolation functions, for example bicubic interpolation function. In this case, there are 16 coefficients for the interpolation function [17].

From the above process, we know that our IH-tree is well balanced. This is because our IH-tree is built on the grid layout $\mathsf{G}_{n\times m} = (\mathsf{B_x}, \mathsf{B_y})$ by recursively dividing $\mathsf{B_x}$ and $\mathsf{B_y}$ into two parts, respectively. In particular, the middle values of $\mathsf{B_x}$ and $\mathsf{B_y}$ are selected as the center point to partition the spatial region into 4 sub-regions. For each sub-region, we will treat it as a new spatial region and divide it into 4 new sub-regions. In addition, we ensure that the tree height h is in the range of $1 < h < \log \min(n, m)$. Therefore, each internal node of the IH-tree will have 4 children, and our IH-tree is balanced.

Fig. 4 illustrates an example of IH-tree, in which we only show the leaf nodes of (2, 5) and omit others for the space consideration. Totally, there should be 16 leaf nodes. The root node (4, 3) represents the whole area while the internal node (2, 5) indicates the blue area. Taking fitting function $f_3(x, y)$ under the internal node (2, 5) as an example, we can calculate $\{\phi_{3i} \mid i \in [1,4]\}$ with Eq. (8), where $x_1 = 19, x_2 = 42, y_1 = 300, y_2 = 500$ are retrieved from $\mathsf{B_x}$ and $\mathsf{B_y}$, and $v_{11} = 42, v_{12} = 50, v_{21} = 43, v_{22} = 51$ are cell ids.

4.1.2 *Predict Cell with Learned Model*

Given a query point (x_q, y_q), we can use the built model M, i.e., IH-tree, to predict the cell that (x_q, y_q) falls in, denoted as $pid = \mathsf{M}(x_q, y_q)$. First, we need to navigate the tree by comparing $(x_q, \mathsf{B_x}[node.x])$ and $(y_q, \mathsf{B_y}[node.y])$. For example, if $x_q < \mathsf{B_x}[node.x]$ and $y_q < \mathsf{B_y}[node.y]$, *child 0* will be selected as the next level node. When reaching a leaf node, it means the query point (x_q, y_q) falls in the sub-region represented by the leaf node. Generally, there are four leaf nodes for an internal node, and their spatial interpolation functions are defined as $f_i(x, y)$, where $i \in [0, 3]$. Then, by computing $f_i(x_q, y_q)$ with Eq. (7), we can obtain the predicted cell id, i.e., $pid = f_i(x_q, y_q)$.

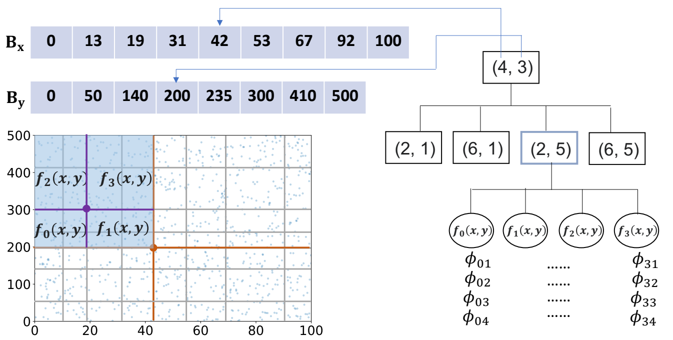


Fig. 4. IH-tree. To save space, we only show the leaf nodes of (2,5) and omit the leaf nodes of (2,1), (6,1), and (6,5).

Here, we present an example of query point $(38, 420)$ in the region of Fig. 4. First, the query point will be compared with the root node (4, 3). Since $38 < \mathsf{B_x}[4] = 42$ and $420 > \mathsf{B_y}[3] = 200$, the internal node (2, 5), i.e., *child* 2, is navigated. Further, as $38 > \mathsf{B_x}[2] = 19$ and $420 > \mathsf{B_y}[5] = 300$, $f_3(x, y)$ is selected to compute the predict cell id, namely, $pid = f_3(38, 420)$ with Eq. (7).

With *pid* and error guarantees, we can get the real cell id by performing the local search, with details in Section 3.2. Furthermore, range and kNN queries can be responded to starting from the real cell in which the query point falls.

4.2 Optimized Learned Model

In this section, we design an optimized learned model IH-tree+ by employing two optimization techniques to IH-tree. First, in order to reduce the model building time and storage overhead, we introduce a novel approach that can use the spatial interpolation function to predict cell id without fitting. Second, we propose a dynamic encoding technique to improve prediction accuracy.

4.2.1 *Learned Model without Fitting*

Although the proposed learned model M=IH-tree can quickly predict the cell id, we observed that the leaf nodes of our IH-tree comprise the majority of nodes: $\sum_{i=0}^{h-1} 4^i = \frac{4^h-1}{3}$ internal nodes and 4^h leaf nodes. Moreover, when building the IH-tree, each leaf node needs to fit a spatial interpolation function and store 4 coefficients (suppose we adopt the bilinear interpolation function here) that are usually *doubles* (while the internal node stores *integers*). Consequently, it negatively affects the model building time and storage overhead. Aiming at the above issues, we leverage the property of bilinear interpolation function to reduce the model building time and storage overhead simultaneously.

In IH-tree, each leaf node represents a sub-region that is expressed as a fitted spatial interpolation function. Without loss of generality, each sub-region can be formed by two boundary values in each dimension:

$$\begin{aligned} x_1 \le x \le x_2,\\ y_1 \le y \le y_2. \end{aligned} \tag{9}$$

If we normalize the sub-region of the leaf node with the following equation:

$$\begin{aligned} x' &= (x - x_1)/(x_2 - x_1)\\ y' &= (y - y_1)/(y_2 - y_1), \end{aligned} \tag{10}$$

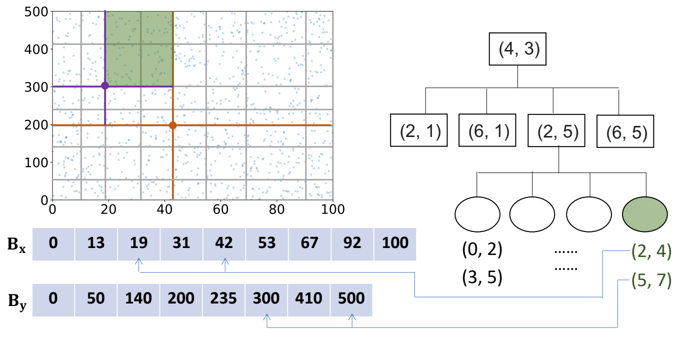

(4, 3)
(2, 1)
(6, 1)
(2, 5)
(6, 5)
(0, 2)
(3, 5)
(2, 4)
(5, 7)
Bx 0 13 19 31 42 53 67 92 100
By 0 50 140 200 235 300 410 500

Fig. 5. IH-tree+. The leaf node stores the positions of the covered subspace. The light blue area is represented as (2, 4) and (5, 7) that are the positions of $(x_1 = 19, x_2 = 42)$ and $y_1 = 300, y_2 = 500)$ over $\mathsf{B_x}$ and $\mathsf{B_y}$, respectively.

we can obtain a new bilinear interpolation function by simply plugging Eq. (10) into Eq. (2):

$$\begin{aligned} f(x,y) &= \frac{1}{(x_2-x_1)(y_2-y_1)}[x_2-x, x-x_1]\begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22}\end{bmatrix}\begin{bmatrix} y_2-y \\ y-y_1\end{bmatrix} \\ &= (1-x')(1-y')v_{11} + (1-x')y'v_{12} \\ &\quad + x'(1-y')v_{21} + x'y'v_{22}. \end{aligned} \tag{11}$$

With Eq. (11), we can only store four integers, i.e., the positions of (x_1, x_2) on $\mathsf{B_x}$ and (y_1, y_2) on $\mathsf{B_y}$, for each leaf node. We denote these positions as $x_1.pos, x_2.pos, y_1.pos$ and $y_2.pos$. Regarding $\{v_{ij} \mid i,j \in \{1,2\}\}$, we can retrieve them from the two dimensional array $\mathsf{C_{id}}$ if we know $\{x_1.pos, x_2.pos, y_1.pos, y_2.pos\}$. Alternatively, we can also calculate $\{v_{ij} \mid i,j \in \{1,2\}\}$ if we know these positions, for example, $v_{11} = y_1.pos \cdot n + x_1.pos$. Fig. 5 shows our IH-tree+, in which four integers (positions) are stored in leaf nodes. Similar to the example of IH-tree in Fig. 4, here we only show the leaf nodes of (2, 5) and omit the leaf nodes of (2, 1), (6, 1), and (6, 5). In Fig. 5, the light green area is determined by $(x_1 = 19, x_2 = 42)$ and $(y_1 = 300, y_2 = 500)$, which is also the fourth leaf node under the internal node (2, 5). Different from IH-tree, here we only store the positions of (x_1, x_2) and (y_1, y_2), instead of the fitted interpolation function of the sub-region. Consequently, we store $(x_1.pos = 2, x_2.pos = 4)$ and $(y_1.pos = 5, y_2.pos = 7)$ in this leaf node.

If we would like to predict the cell id of a query point (x_q, y_q), the approach is the same as the IH-tree before navigating to leaf node. When reaching to leaf node, since we store $(x_1.pos, x_2.pos)$ and $(y_1.pos, y_2.pos)$, we can first normalize (x_q, y_q) with Eq. (10), i.e.,

$$\begin{aligned} x'_q &= (x_q - \mathsf{B_x}[x_1.pos])/(\mathsf{B_x}[x_2.pos] - \mathsf{B_x}[x_1.pos]) \\ y'_q &= (y_q - \mathsf{B_y}[y_1.pos])/(\mathsf{B_y}[y_2.pos] - \mathsf{B_y}[y_1.pos]). \end{aligned} \tag{12}$$

After that, the cell id can be predicted using Eq. (11) as follows:

$$\begin{aligned} f(x_q,y_q) =& (1-x'_q)(1-y'_q)(y_1.pos \cdot n + x_1.pos) \\ &+ (1-x'_q)y'_q((y_2.pos-1)\cdot n + x_1.pos) \\ &+ x'_q(1-y'_q)(y_1.pos\cdot n + x_2.pos - 1) \\ &+ x'_q y'_q((y_2.pos-1)\cdot n + x_2.pos - 1). \end{aligned} \tag{13}$$

Analysis. In the leaf node of IH-tree, we need to store four coefficients of the bilinear interpolation function, which are *double* type (8 bytes), as $\{x_i, y_i | i \in \{1,2\}\}$ in Eq. (8) are usually real geographic location data. However, in the leaf node of IH-tree+, we store four integers (each integer has 4 bytes). Furthermore, the number of leaf nodes accounts for almost 75% of total nodes in our tree structure. Thus, we can significantly reduce the storage overhead. In addition, regarding the model building, we need to fit a bilinear interpolation function for each leaf node in IH-tree with Eq. (8). However, for building IH-tree+, we can get rid of this step and directly store sub-region's positions for the corresponding leaf node. This will reduce the time cost of model building. Although IH-tree+ will increase the time cost of predicting cell id due to the normalization, it is a negligible cost compared to the incurred benefits, which will be evaluated in Section 6.1.3.

4.2.2 *Dynamic Encoding Technique*

Recall the last step of predicting cell, i.e., calculating predicted cell id. Regardless of whether Eq. (7) in IH-tree or the new approach with Eq. (11) is used, the cell ids $\{v_{ij} \mid i,j \in \{1,2\}\}$ are encoded globally. For a leaf node, the predicted cell id will fall in the range of $[y_1.pos \cdot n + x_1.pos, (y_2.pos - 1)\cdot n + x_2.pos - 1]$. In Fig. 6(a), the predicted range for the light green area is [42, 51], leading to the predicted cell lying outside the area that the query point should be. In the worst case, the prediction error in the x dimension is n, i.e., $eg_x = n$. From Fig. 3, we know that the larger prediction error entails the larger range of the local search, which obviously deteriorates the performance in locating real cell id. Ideally, we hope to limit the predicted cell to the area covered by the corresponding leaf node. In this way, the range of the subsequent local search will be small and controllable. However, since the cell ids are globally indexed by the space-filling curve technique (as shown in Fig. 2), it is possible that the predicted cell id falls outside the leaf node area.

Note that employing other space-filling curve techniques, such as the Z-order curve and Hilbert curve, also cannot ensure that the predicted cell is within the desired area. In fact, it is an open problem for all learned models that adopt the space-filling curve techniques, for example, the false positive problem in the ZM-index [7] incurred by the Z-order curve. This is because the space-filling curve techniques cannot ensure that the closer data points in the two-dimensional space are closer in the encoded one-dimensional values.

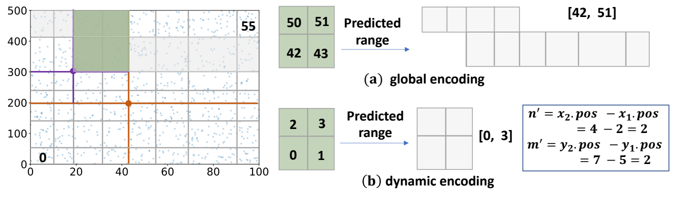


Fig. 6. Global and dynamic encoding. The dynamic encoding is over a $n' \times m'$ grid. Regarding the light blue area, $n' = 2$ and $m' = 2$.

To address this problem, we design a dynamic encoding technique to ensure the predicted cell that falls within the area covered by the leaf node, which can significantly improve the prediction accuracy. The key idea is to dynamically encode the cell ids in the leaf node area instead of

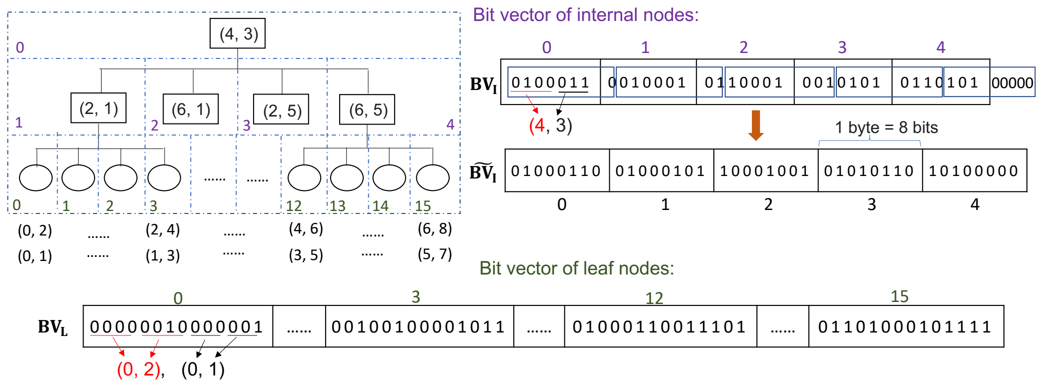


Fig. 7. Compressed IH-tree+. Two bit vectors. One is for internal nods, and the other is for leaf nodes.

using the globally encoded cell ids. The cells covered by a leaf node can be encoded from 0 to $n' \times m' - 1$ with the same shape as the global encoding, where $n' = x_2.pos - x_1.pos$ and $m' = y_2.pos - y_1.pos$. In this way, we have values for $\{v_{ij} \mid i, j \in \{1, 2\}\}$ as follows:

$$\begin{cases} v_{11} = 0, \\ v_{12} = (m' - 1) \cdot n', \\ v_{21} = n' - 1, \\ v_{22} = m' \cdot n' - 1. \end{cases} \tag{14}$$

Given a query point (x_q, y_q), when a leaf node is selected, we can normalize the point with Eq. (12) and calculate the predicted cell id with the following equation.

$$\begin{aligned} f'(x_q, y_q) =& (1 - x'_q)y'_q((m' - 1) \cdot n') \\ &+ x'_q(1 - y'_q)(n' - 1) \\ &+ x'_q y'_q(m' \cdot n' - 1). \end{aligned} \tag{15}$$

In our dynamic encoding technique, since v_{11} is always 0, Eq. (15) only needs to calculate three terms while Eq. (13) needs four terms.

Since the predicted cell id with Eq. (15) is the local cell id, we need to convert it to global cell id with the following equation:

$$\begin{aligned} pid =& (f'(x_q, y_q)/n' + y_1.pos) \cdot n \\ &+ (f'(x_q, y_q) \mod n' + x_1.pos). \end{aligned} \tag{16}$$

Analysis. Since we employ the dynamic encoding technique, the predicted range will be from 0 to $n' \times m' - 1$. As a result, the prediction cell is limited to the leaf node area. Fig. 6(b) depicts an example of the dynamic encoding technique. We will evaluate and compare the prediction accuracy in Section 6.1.2.

Additionally, this technique can make the prediction error controllable. That is, we can ensure the prediction error should be no larger than the position range of the leaf node area. In our previous work [10], since the prediction error is unpredictable, we use the maximum prediction error as error guarantee, which was obtained by executing the whole query workload. However, thanks to the dynamic encoding technique, our IH-tree+ can ensure the real prediction error in x dimension to be no larger than $x_2.pos - x_1.pos - 1$ and $y_2.pos - y_1.pos - 1$ for the y dimension. As a result, we can set the error guarantees in x dimension (eg_x) and y dimension (eg_y) as follows.

$$\begin{aligned} eg_x &= x_2.pos - x_1.pos - 1, \\ eg_y &= y_2.pos - y_1.pos - 1. \end{aligned} \tag{17}$$

It means that we do not need to obtain the error guarantees by performing the query workload. Under this setting, the tree height of our IH-tree+ can be calculated with $h = \log(\max(\frac{n}{eg_x}, \frac{m}{eg_y}))$.

4.3 Compressed IH-tree+

Generally, the learned model will be deployed in the main-memory to speed up the query processing. In order to reduce the consumption of precious memory, we present a compressed version of our proposed learned model, IH-tree+. Since our IH-tree+ is well balanced, it is easy to compress the tree structure into bit vectors. Motivated by [18], our compressed IH-tree+ is shown in Fig. 7.

- The internal nodes are compressed into a bit vector, denoted as $\mathrm{BV_I}$, and each node has $\lfloor \log n \rfloor + 1 + \lfloor \log m \rfloor + 1$ bits, where n and m are the number of the columns in x and y dimensions, respectively.
- The leaf nodes are compressed into another bit vector, denoted as $\mathrm{BV_L}$, and each node has $2(\lfloor \log n \rfloor + 1 + \lfloor \log m \rfloor + 1)$ bits.

For ease of implementation, we reorganize each bit vector with bytes. If the remaining bits are less than 8, we will complement it with 0s. In Fig. 7, we only show the reorganized bit vector of internal nodes and omit the reorganized bit vector of leaf nodes for simplicity. The reorganized bit vectors are denoted as $\widetilde{\mathrm{BV}}_\mathrm{I}$ and $\widetilde{\mathrm{BV}}_\mathrm{L}$, respectively. Note that, if we know a node's position in $\mathrm{BV_I}$ and $\mathrm{BV_L}$, it is easy to access the corresponding bits in $\widetilde{\mathrm{BV}}_\mathrm{I}$ and $\widetilde{\mathrm{BV}}_\mathrm{L}$. Here, we take mapping $\mathrm{BV_I}$ to $\widetilde{\mathrm{BV}}_\mathrm{I}$ as an example.

$$\begin{aligned} &\widetilde{\mathrm{BV}}_\mathrm{I}.loc = \mathrm{BV_I}.pos \cdot (\lfloor \log n \rfloor + 1 + \lfloor \log m \rfloor + 1)/8 \\ &offset = \widetilde{\mathrm{BV}}_\mathrm{I}.pos \cdot (\lfloor \log n \rfloor + 1 + \lfloor \log m \rfloor + 1) \mod 8, \end{aligned} \tag{18}$$

where $\mathrm{BV_I}.pos$ is the node position in $\mathrm{BV_I}$, $\widetilde{\mathrm{BV}}_\mathrm{I}.loc$ is the byte location in $\widetilde{\mathrm{BV}}_\mathrm{I}$, and *offset* is the offset in the byte. In Fig. 7, the internal node (6, 5) has the node position $\mathrm{BV_I}.pos = 4$. With Eq. (18), we can access its bits in the fourth byte, i.e., $\widetilde{\mathrm{BV}}_\mathrm{I}.loc = (4 \cdot 7)/8 = 3$ (encoding from 0), and *offset* is $(4 \cdot 7) \mod 8 = 4$. Similarly, we can access the leaf node's bits from

Algorithm 1 Predicting cell id over compressed IH-tree+

```
Input: Two bit vectors, BV_I and BV_L. Query point $(x_q, y_q)$. The tree height, $h$.
Output: Predicted cell id, pid.
1: node ← BV_I[0];   $\alpha \leftarrow 0$;   $\beta \leftarrow 0$;   $\delta \leftarrow 0$;
2: while $\alpha < h$ do
3:     $\alpha \leftarrow \alpha + 1$;
4:     if $x_q <$ B_x[node.x] then
5:         $\beta \leftarrow y_q <$ B_y[node.y] ? 0 : 2
6:     else
7:         $\beta \leftarrow y_q <$ B_y[node.y] ? 1 : 3
8:     if $\alpha < h - 1$ then
9:         BV_I.pos ← $(\delta \cdot 4 + \beta) + (4^{\alpha} - 1)/3$;
10:        node ← BV_I[BV_I.pos];
11:        $\delta \leftarrow \delta \cdot 4 + \beta$;
12: BV_L.pos ← $\delta \cdot 4 + \beta$;
13: leaf node ← BV_L[BV_L.pos];
14: return pid ← Eq. (15) and Eq. (16) with leaf node;
```

$\widetilde{\mathsf{BV}_\mathsf{L}}$ by mapping $\mathsf{BV}_\mathsf{L}.pos$ to $\{\widetilde{\mathsf{BV}_\mathsf{L}}.loc, offset\}$. As a result, hereafter, we only show how to calculate the node positions in BV_I and BV_L.

When predicting the cell id of a query point (x_q, y_q), traversing the IH-tree+ for selecting the leaf node is converted to searching over BV_I for obtaining the leaf node's corresponding position in BV_L. Typically, there are three steps to predict cell id with the compressed IH-tree+.

Step-1: Search over BV_I. First, we define α as the level number of the IH-tree+, where $\alpha \in [0, h-1]$. Next, the first position in BV_I is accessed since it must be the root node. After comparing (x_q, y_q) with $(\mathsf{B_x}[node.x], \mathsf{B_y}[node.y])$, we can determine which *child* is selected. We define the child number as β, where $\beta = \{0, 1, 2, 3\}$. Suppose $(x_q, y_q) = (60, 240)$. It will be compared with $(\mathsf{B_x}[4], \mathsf{B_y}[3]) = (42, 200)$ first. Since $x_q > 42, y_q > 200$, *child 3* will be selected, i.e., $\beta = 3$. Consequently, if we define δ as the parent's position in its level, the child node's position in BV_I can be calculated with the following equation:

$$\mathsf{BV}_\mathsf{I}.pos = (\delta \cdot 4 + \beta) + \frac{4^{\alpha} - 1}{3}, \tag{19}$$

where $\delta \cdot 4 + \beta$ is the child node's position in its level. Following the above example, we have $\mathsf{BV}_\mathsf{I}.pos = (0 \cdot 4 + 3) + (4^1 - 1)/3 = 4$. It is worth noting that the node (6, 5)'s position in its level is $0 \cdot 4 + 3 = 3$, which can be used to calculate its children's position in BV_I.

Step-2: Retrieve leaf node from BV_L. When reaching to the last level's internal node, after determining β, the corresponding leaf node's position in BV_L can be easily calculated with $\delta \cdot 4 + \beta$. Here δ is the internal node's position in the last level. In our example, the last level's internal node is (6, 5). Since $x_q < \mathsf{B_x}[5] = 67, y_q < \mathsf{B_y}[5] = 300$, $\beta = 0$. Therefore, $\mathsf{BV}_\mathsf{L}.pos = 3 \cdot 4 + 0 = 12$. Thus, we can retrieve and recover (4, 6) and (3, 5) from BV_L.

Step-3: Calculate the predicted cell id. After obtaining (4, 6) and (3, 5), the final step is to calculate the predicted cell id with Eq. (15) and Eq. (16), which is the same as the uncompressed IH-tree+.

Algorithm 1 formally depicts the process of predicting a cell id with the compressed IH-tree+.

Analysis. Our IH-tree+ has $\sum_{i=0}^{h-1} 4^i = \frac{4^h - 1}{3}$ internal nodes and 4^h leaf nodes, where h is the tree height. Theoretically, the storage footprint of our IH-tree+ is as least $\frac{4^h-1}{3} \cdot 4 \cdot 2 \cdot 8 + 4^h \cdot 4 \cdot 4 \cdot 8$ bits since each internal node stores two integers, while the leaf node has four integers. However, the storage overhead of the compressed IH-tree+ is around $\frac{4^h-1}{3} \cdot (\lfloor \log n \rfloor + 1 + \lfloor \log m \rfloor + 1) + 4^h \cdot 2 \cdot (\lfloor \log n \rfloor + 1 + \lfloor \log m \rfloor + 1)$ bits. Since $\log n$ and $\log m$ are usually less than 15, and the compressed IH-tree+ does not have pointers that exist in the real-world tree-structure, the compressed IH-tree+ will appreciably reduce the storage consumption. Negatively, the compressed IH-tree will increase the time cost when predicting cell id. It is because we need to recover bits to the corresponding integers and calculate the node's position in BV_I. We will evaluate the IH-tree+ and its compressed version in Section 6 and shift the decision to users.

TABLE 1
Notations in cost model

Notation	Definition
$\mathsf{T}(\mathsf{F}_{in}^{n\times m})$	the execution time of the spatial interpolation function
$\mathsf{T}(\mathsf{B}^{n\times m})$	the execution time of local binary search
T_r	average time of retrieving a cell
T_s	average time of scanning a data point
N_i	the number of cells intersected with query window
N_c	the number of cells contained within query window
N_p	the data points in the intersected cells

5 Cost Model

In our previous work [10], we use the following cost model to determine the number of columns in x dimension and y dimension, i.e., the value of n and m:

$$\mathit{Time} = \mathsf{T}(\mathsf{F}_{in}^{n\times m}) + \mathsf{T}(\mathsf{B}^{n\times m}) + \mathsf{T}_r \cdot (N_i + N_c) + \mathsf{T}_s \cdot N_p.$$

The notations used in the above model are listed in Table 1. Given a dataset $\mathcal{D}$ and a query workload $\mathcal{W}$, we obtain the best layout parameters: $n \times m$ that makes *Time* to have minimal average value. Note that the above model is an example model used for range query. Although this model allows us to obtain the best layout parameters, it is expensive since we need to perform range queries over each possible layout and select the best one. In this paper, our proposed learned model IH-tree+ makes it possible to model time cost with the number of operations, which is motivated by [19]. Similar to our previous work, we take the range query as an example to show the new approach in building a cost model.

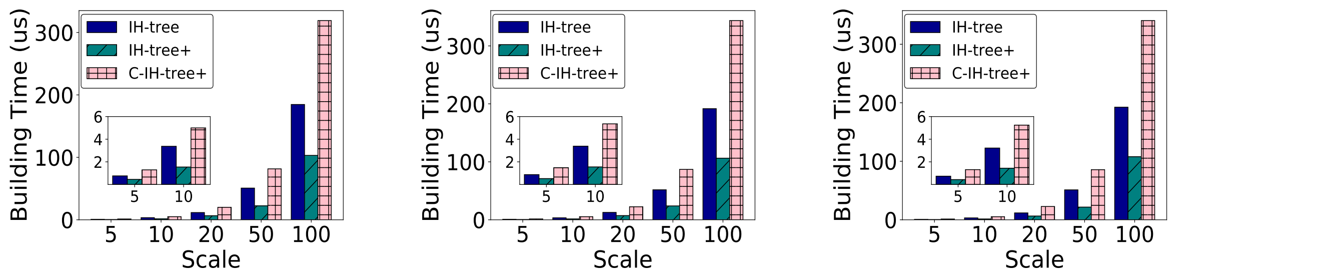
First, we define T_c as the average time of comparing two double values and T_e as the average time of refining a data point. Then, the time cost of performing a range query can be roughly modeled as follow:

$$\mathit{Time} = 2 \cdot \log(\max(n, m)) \cdot \mathsf{T}_c + n' \cdot m' \cdot \mathsf{T}_r + N' \cdot \mathsf{T}_e,$$

where n' and m' are the number of cells that intersected with the query window in x and y dimensions and N' is the number of data points in the intersected cells. With this model, we can obtain the layout parameters that render the minimal time cost without performing real range queries.

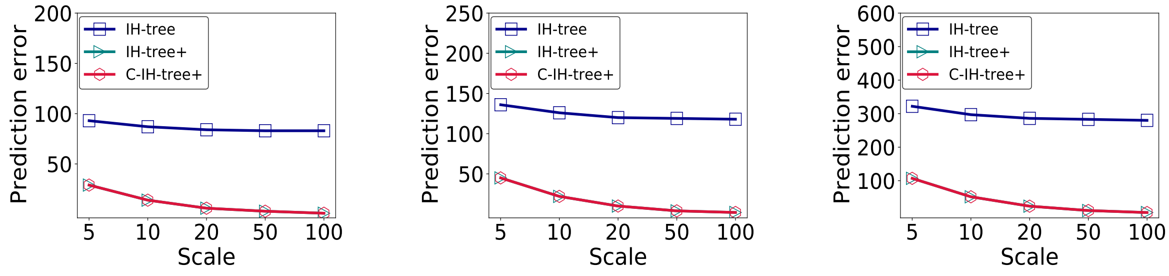
6 Evaluation

This paper aims to improve the performance of learned model that is employed as the foundation in the learned index. As a result, in Section 6.1, we experimentally evaluate and compare the performance of our proposed learned models, i.e, IH-tree, IH-tree+, and compressed IH-tree+, in terms of model building time, prediction accuracy, models' storage overhead, and cell predicting time. Besides, in our



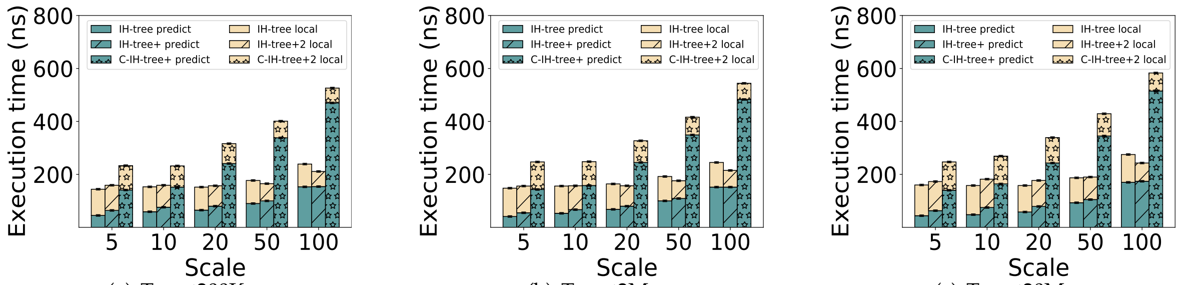
(a) Tweet200K (b) Tweet2M (c) Tweet20M

Fig. 8. Time cost of building learned models over different datasets (grid layouts) varying with scale factors from 5 to 100. Since the time cost is too small when the scale factors are 5 and 10, we use a reduced figure to zoom in them.



(a) Tweet200K (b) Tweet2M (c) Tweet20M

Fig. 9. Prediction accuracy over different datasets (grid layouts) varying with scale factors from 5 to 100.



(a) Tweet200K (b) Tweet2M (c) Tweet20M

Fig. 10. Time cost of predicting cell and local search over different datasets (grid layouts) varying with scale factors from 5 to 100.

SPRIG system, we adopt IH-tree as the learned model. By replacing IH-tree with IH-tree+ or its compressed version, we have a new learned spatial index SPRIG+. In order to show the performance of learned spatial indexes, in Section 6.2, we compare SPRIG+, SPRIG with Flood (a state-of-the-art learned index [6]) and ZM-index (a learned spatial index [7]) in terms of query execution time, index building time, and storage overhead.

All learned models and learned spatial indexes were implemented in Java and evaluated with in-memory versions. Same as our previous work, we adopt three Twitter datasets [20] consisting of tweets with their locations: *Tweet200k*, *Tweet2M*, and *Tweet20M* that have 200k, 2M, and 20M spatial points, respectively. We conduct all experiments on a machine with 16 GB memory and 3.4 GHz Intel(R) Core(TM) i7-3770 processors and running Ubuntu 16.04 OS.

6.1 Performance of Learned Models

From Section 4, we know that the performance of our learned model is related to the constructed grid, i.e., $\mathsf{G}_{n\times m} = (\mathsf{B}_x, \mathsf{B}_y)$, and the height of the proposed hybrid tree $h = \lceil \log(\max(\frac{n}{eg_x}, \frac{m}{eg_y})) \rceil$.

- Our evaluation will run over the grids with dimensions: 210×170, 300×300, and 710×690 that are the best layouts for conducting range queries over *Tweet200k*, *Tweet2M*, and *Tweet20M*, respectively, and are used in the experiments of our previous work.
- To clearly show the impact of the tree height, we will vary the position range of leaf node area by scaling down the grid layout with the scale factors: $\{5, 10, 20, 50, 100\}$. Correspondingly, the tree heights will be $\{3, 4, 5, 6, 7\}$ under the above grid layouts. For example, if the scale factor is 20, under the grid layout 210×170, the position range of leaf node area is $\lceil 210/20 \rceil \times \lceil 170/20 \rceil = 11 \times 9$. The corresponding tree height will be $h = \lceil \log(\max(\frac{210}{11}, \frac{170}{9})) \rceil = 5$.

6.1.1 Time cost of building model

Fig. 8 illustrates the time cost of building learned model varying with the scale factors from 5 to 100, in which C-IH-tree+ represents the compressed IH-tree+. We separately evaluate the building model time over *Tweet200k*, *Tweet2M*, and *Tweet20M* datasets, as shown in Fig. 8(a), Fig. 8(b), and Fig. 8(c). We can see that the building time with these three datasets has the same trend and similar values. That is because the time cost of building model is only related to the tree height h, and varying the scale factors from 5 to 100 exactly corresponds to the tree height from 3 to 7 for each dataset. Further, as the tree height increases, the building time will increase, which is in line with the expectation. From Fig. 8(a), Fig. 8(b), and Fig. 8(c), we know that our proposed IH-tree+ has the best performance in terms of model building time. Specifically, i) IH-tree+ is around $1.8\times$ faster than IH-tree. This benefit comes from the first novel optimization technique presented in Section 4.2.1. It allows us to build learned model, i.e., IH-tree+, without fitting the spatial interpolation functions for leaf nodes; ii) IH-tree+ is around $3\times$ faster than C-IH-tree+. As we discussed in Section 4.3, in C-IH-tree+, the tree structure needs to be further compressed into bit vectors. Consequently, C-IH-tree+ consumes more computational costs.

6.1.2 Prediction accuracy

As shown in Fig. 3, the prediction accuracy will impact the local search range. That is, if the prediction error is larger, the range of the binary/exponential search will be larger. In our learned index, after predicting a cell, we separately conduct the local search on x dimension and y dimension to locate the real cell (seeing details in Section 3.2). As a result, we measure the prediction accuracy with the sum of prediction errors in x and y dimensions, i.e.,

$$prediction\ error = |pid_x - rid_x| + |pid_y - rid_y|, \qquad (20)$$

where *pid* is the predicted cell id of a query point, and *rid* is the real cell id of the query point. Note that it is easy to calculate the cell's x and y positions if we know the cell id. For example, $pid_x = pid \mod n$ and $pid_y = pid/n$, where n is the number of columns in x dimension. We can use the same approach to calculate rid_x and rid_y.

In Fig. 9(a), Fig. 9(b), and Fig. 9(c), we evaluate the prediction accuracy over these three Tweet datasets by calculating the prediction error with Eq. (20). All of these figures show that: i) IH-tree+ and C-IH-tree+ have the same prediction error. That is because the only difference between these two structures is the storage organization, which is independent of the prediction error. From another prospective, this result proves the correctness of our C-IH-tree+; ii) Compared to IH-tree, our IH-tree+ and C-IH-tree+ improve the prediction accuracy at least $3\times$. With the scale factor increasing, the accuracy improvement increases and can achieve $83\times$ on *Tweet200k* dataset, $59\times$ on *Tweet2M* dataset, and $56\times$ on *Tweet20M* dataset, when the scale factor is 100. The significant benefit comes from the dynamic encoding technique presented in Section 4.2.2. It limits the prediction error within a small area covered by the leaf node. Taking *Tweet20M* dataset as an example, the corresponding gird layout is 710×690. As the scale factor increases from 5 to 100, the leaf nodes area narrows down from 142×138 to 8×7 (see Section 6 on how to calculate the leaf node area). As a result, when the scale factor is 5, the prediction error of IH-tree+ in x dimension must be less than 142 and that is less than 138 in y dimension. When the scale factor is 100, it is less than 8 and 7 in x and y dimensions, respectively. While, the prediction error of IH-tree scatters over the whole space. Our experimental results show that when the scale factor is 5, the prediction error of IH-tree+ is 107, while it is 322 for IH-tree. When the scale factor is 100, it is 5 for IH-tree+ and 280 for IH-tree. In addition, from these figures, we can see that the prediction error increases, as the grid layouts becomes larger (from 210×170 on *Tweet200k* dataset to 710×690 on *Tweet20M* dataset). Since we adopt the same scale factors, the leaf nodes area is larger for the larger grid layout, leading to a larger prediction error. In fact, we can further enlarge the scale factor for the larger grid layout to improve the prediction accuracy. It offers us guidance for ensuring prediction accuracy, namely, when the grid layout is large, we should select a larger scale factor.

6.1.3 Time cost of predicting cell id

In Fig. 10, we use the dark green bar to depict the time cost of predicting cell over *Tweet200k*, *Tweet2M*, and *Tweet20M* datasets. From this figure, we have the following observations: i) IH-tree is slightly faster than IH-tree+ in predicting cell. That is because when reaching to the leaf node, the IH-tree+ needs to first normalize the query point with Eq. (10) and then calculate the predicted cell id with Eq. (15) and Eq. (16). While for the IH-tree, it can directly calculate the predicted cell id with Eq. (7); ii) C-IH-tree+ has the worst performance in predicting cell id. That is due to the extra computational costs in calculating node position and recovering integers from compressed representation. It is reasonable since C-IH-tree+ is used for the storage-constrained environment and sacrifices the time cost of predicting cell to reduce storage overhead; iii) with the scale factor increasing, the cell predicting time increases for all three learned models. That is because, when the grid layout is given, the larger scale factor will render a larger tree height, leading to a higher search time to reach the leaf node.

In Fig. 10, we also plot the time cost of local search using the wheat color bar. Since *locating cell time = predicting cell time + local search time*, the complete bar in Fig. 10 illustrates the time cost of locating the cell that a query point falls in, which is the principle behind range and kNN queries discussed in our previous work. From these three figures, we know that, when the scale factor is larger than 20, IH-tree+ has a better performance than IH-tree in locating cell. This benefit comes from the fewer local search time. In fact, the local search time is related to the prediction accuracy discussed in Section 6.1.2. Since IH-tree+ is much more accurate than IH-tree in predicting cell (as shown in Fig. 9), its local search range will be smaller than that of IH-tree, leading to a lower local search time. When the scale factor is larger than 20, the benefit from local search will exceed the extra time cost in predicting cell. As a result, in these cases, IH-tree+ has a better performance in locating cells.

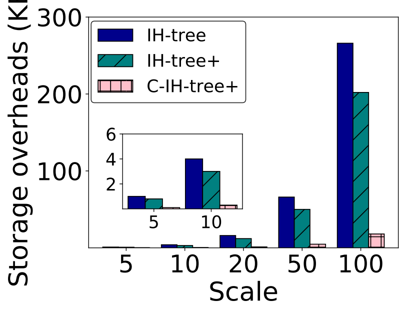


Fig. 11. Storage overhead of IH-tree, IH-tree+, and C-IH-tree+. Since the storage overhead is too small when the scale factors are 5 and 10, we use a reduced figure to zoom in them.

6.1.4 Storage overhead

In this section, we compare the storage overhead of different learned modes, i.e., IH-tree, IH-tree+, and C-IH-tree+.

Recalling Section 4, the theoretical storage footprint of these three learned models is shown as Table 2.

TABLE 2
Theoretical storage footprint of learned models

Model	Theoretical storage footprint (bits)
IH-tree	$\frac{4^h-1}{3} \cdot 4 \cdot 2 \cdot 8 + 4^h \cdot 4 \cdot 8 \cdot 8$
IH-tree+	$\frac{4^h-1}{3} \cdot 4 \cdot 2 \cdot 8 + 4^h \cdot 4 \cdot 4 \cdot 8$
C-IH-tree+	$\frac{4^h-1}{3} \cdot (\lfloor \log n \rfloor + 1 + \lfloor \log m \rfloor + 1)$ $+\ 4^h \cdot 2 \cdot (\lfloor \log n \rfloor + 1 + \lfloor \log m \rfloor + 1)$

From this table, we can see that the storage footprints of our proposed learned models are related to the tree height h and grid layout $\mathsf{G}_{n\times m}$. In our evaluation, for these three datasets, the grid layouts are 210×170, 300×300, and 710×690, respectively. Consequently, the impact of grid layout on storage is almost negligible due to the log calculation. Meanwhile, when varying the scale factors from 5 to 100, all of these grid layouts have the tree height from 3 to 7. Therefore, learned models' storage overhead is almost the same under different Tweet datasets. Thus, here we only show one figure (Fig. 11) to depict the storage overhead of our proposed learned models.

In order to obtain the practical results, we adopt the *SizeOf* class in Java to measure the storage overhead of these learned models as shown in Fig. 11. From this figure, we can see that: i) IH-tree+ can reduce at least 20% storage overhead compared to IH-tree. This benefit comes from the first optimization technique presented in Section 4.2.1, in which we store the positions (*integer*) instead of coefficients (*double*); ii) C-IH-tree+ has almost an order of magnitude improvement in storage overhead compared to IH-tree+. On the one hand, the compact bit $\lfloor \log n \rfloor + 1 + \lfloor \log m \rfloor + 1$ is usually less than $4 \cdot 2 \cdot 8 = 64$. On the other hand, there is no pointer in the compressed bit vectors, while it is intrinsic to the tree structures.

Remark. From the above evaluation, we can see that: i) IH-tree+ is better than IH-tree in terms of building time, prediction accuracy, and storage overhead. Although IH-tree is slightly faster than IH-tree+ in predicting cell id, the latter has a better performance in locating cell when the scale factor is larger than 20; ii) Compared to IH-tree+, C-IH-tree+ can significantly reduce the storage overhead although it consumes higher computational costs in building model and predicting cell, which offers a choice for the storage-limited environment. Table 3 illustrates the best learned model(s) under different measurements.

TABLE 3
Best learned model(s) in different measurements

Models	**IH-tree**	**IH-tree+**	**C-IH-tree+**
Building time		✓	
Prediction accuracy		✓	
Predicting cell time	✓		
Locating cell time	when $h \leq 5$	when $h > 5$	
Storage overhead			✓

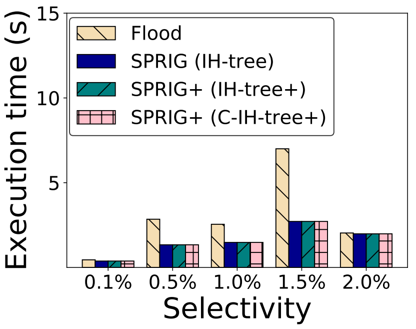


Fig. 12. Execution time of range query over *Tweet20M* varying with selectivities from 0.1% to 2.0%. Note that, the execution time of ZM-index is individually shown in Table 4 due to its significant time cost.

6.2 Performance of Learned Spatial Indexes

In this section, we evaluate the performance of SPRIG+, SPRIG, Flood, and ZM-index in terms of query execution time, index building time, and storage overhead. From Section 6.1. we can see that the learned models have the similar performance trend from *Tweet200k* to *Tweet20M*. Meanwhile, we have compared the performance of SPRIG, Flood and ZM-index over *Tweet200k*, *Tweet2M*, and *Tweet20M* in our previous work [10]. Consequently, to save space, here, we only compare these learned spatial indexes over the largest dataset: *Tweet20M*.

6.2.1 Query execution time

Fig. 12 depicts the average execution time of range query varying with selectivies from 0.1% to 2.0%. Since ZM-index is much slower than other learned indexes, we exclude it from Fig. 12 and list its performance in Table 4. We can see that our learned indexes SPRIG and SPRIG+ are better than Flood and ZM-index in all cases. Compared with SPRIG, SPRIG+ has almost the same performance. This is because: i) the *locating cell time* of IH-tree+ is close to IH-tree (seeing Section 6.1.3); ii) the query strategy accounts for a large proportion of the execution time, and both SPRIG and SPRIG+ use the same query strategy. Note that, since this paper focuses on the efficiency of the learned mode instead of query execution strategies, the execution time of kNN query remains the same as we have shown in our previous work. Thus, we omit the performance of kNN queries and only show the performance of range queries.

TABLE 4
Query execution time of ZM-index on *Tweet20M*

Selectivity	0.1%	0.5%	1.0%	1.5%	2.0%
Execution Time (s)	12	13	35	43	58

6.2.2 Index building time

Table 5 illustrates the index building time of learned spatial indexes, including ZM-index, Flood, SPRIG, and SPRIG+. For building learned spatial indexes, the process can be roughly divided into two stages: building learned model and building inverted table, in which the inverted table is used to manage the cell id and the corresponding data points in the cell. In addition, Flood, SPRIG, and SPRIG+ need to train the best layout parameters with the cost model, as shown in Section 5, before building their learned models.

TABLE 5
Index building time of learned spatial indexes

Leaned Indexes	Train grid layout	Build learned model	Build inverted table	Total
ZM-index	NA	297.6 ms	6.4 s	304 s
Flood	10.5 s	< 0.1 s	21.7 s	32.3 s
SPRIG (IH-tree)	113 s	< 0.001 s	8.7 s	121.7 s
SPRIG+ (IH-tree+)	< 0.1 s	< 0.001 s	8.7 s	8.7 s
SPRIG+ (C-IH-tree+)	< 0.1 s	< 0.001 s	8.7 s	8.7 s

TABLE 6
Storage overhead of learned spatial indexes

Leaned Indexes	Learned model	Inverted Table	Total
ZM-index	56 KB	87 KB	0.14 MB
Flood	53 KB	189 KB	0.24 MB
SPRIG (IH-tree)	266 KB	5736 KB	6.0 MB
SPRIG+ (IH-tree+)	202 KB	5736 KB	5.94MB
SPRIG+ (C-IH-tree+)	18 KB	5736 KB	5.75MB

From Table 5, we can see that SPRIG+ has the best performance in index building time: i) compared with ZM-index, SPRIG+ has less time consumption in building model. This is because ZM-index adopts the machine learning approach , i.e., feed-forward neural network, to build the learned model, while our SPRIG+ only needs to build the IH-tree+ or C-IH-tree+, which is much more efficient than the machine learning approach; ii) compared with Flood, SPRIG+ has a lower time consumption in building model and building inverted table stages. Regarding building learned model, SPRIG+ is at the microsecond level as shown in Fig. 8(c), while Flood is at the millisecond level. Regarding building inverted table, Flood is more expensive, since it needs to sort the data points in each cell; iii) compared with SPRIG, SPRIG+ significantly reduces the time cost of training layout since it does not need to execute query workload to obtain the layout parameters.

6.2.3 Storage overhead

From Section 6.2.2, we know that all of the evaluated learned spatial indexes need to build a learned mode and an inverted table. Correspondingly, there are two main components for a learned spatial index: a learned model and an inverted table. Note that our learned indexes, SPRIG+ and SPRIG, additionally contain a grid layout $\mathsf{G}_{n\times m} = (\mathsf{B}_\mathsf{x}, \mathsf{B}_\mathsf{y})$. Since it takes up very litter space, we merge it into the inverted table component. Table 6 shows the storage overhead of learned models, inverted tables, and learned spatial indexes (the *Total* column). From this table, we can see that SPRIG+ with the C-IH-tree+ has the smallest storage overhead for the learned model, which is due to the compression techniques that we used in Section 4.3. However, our learned indexes have a larger storage overhead for the inverted table, resulting in a larger storage overhead for the indexes overall. This is because, compared to ZM-index and Flood, our learned indexes have more cells to represent the entire spatial space. Although our indexes consumes more storage compared to Flood and ZM-index, we achieve significantly better query and index building performance with an acceptable storage overhead. Moreover, the overall storage requirements with our approaches are relatively small and manageable, for instance, they are about 6MB even with the largest dataset *Tweet20M* (i.e., 20 million tweets).

7 Conclusion

In this paper, we have proposed a new learned model with a novel interpolation function based hybrid tree. Specifically, we first observed that the spatial interpolation function could make it possible to predict values without fitting. Based on this observation, we improved our approach in calculating prediction values, which can reduce both the building time and storage overhead. Then, we propose a dynamic encoding technique to limit the prediction error within a given area, which can significantly improve the prediction accuracy of our spatial learned model. After that, we present a succinct version of our learned model for the storage-constrained environments. Our experimental results suggest that our proposed learned model is better than the previous one (used in our previous work [10]) in terms of building time, prediction accuracy, and storage overhead at a negligible predicting time cost. In addition, we compare SPRIG+ with our previous work and other learned indexes. The results show that SPRIG+ has the best performance in building index and is better than Flood and ZM-index in query performance. In our future work, we will consider further reducing the storage overhead of the inverted table used in our learned index and try to make our proposed learned index support updates.

Acknowledgements

This research was supported in part by NSERC Discovery Grants (04009, 03787, RGPIN-2022-03244).

References

[1] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *SIGMOD*, 2018, pp. 489–504.
[2] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. Chen, "Xindex: a scalable learned index for multicore data storage," in *SIGPLAN*, 2020, pp. 308–320.
[3] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *SIGMOD*, 2019, pp. 1189–1206.
[4] P. Ferragina and G. Vinciguerra, "The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds," *VLDB*, vol. 13, no. 8, pp. 1162–1175, 2020.
[5] J. Qi, G. Liu, C. S. Jensen, and L. Kulik, "Effectively learning spatial indices," *VLDB*, vol. 13, no. 12, pp. 2341–2354, 2020.

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

[7] H. Wang, X. Fu, J. Xu, and H. Lu, "Learned index for spatial queries," in *2019 20th IEEE MDM*. IEEE, 2019, pp. 569–574.

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

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

[10] S. Zhang, S. Ray, R. Lu, and Y. Zheng, "Sprig: A learned spatial index for range and knn queries," in *17th International Symposium on Spatial and Temporal Databases*, 2021, pp. 96–105.

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

[12] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "idistance: An adaptive b+-tree based indexing method for nearest neighbor search," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 364–397, 2005.

[13] A. Al-Mamun, H. Wu, and W. G. Aref, "A tutorial on learned multi-dimensional indexes," in *SIGSPATIAL*, 2020, pp. 1–4.

[14] L. Mitas and H. Mitasova, "Spatial interpolation," *Geographical information systems: principles, techniques, management and applications*, 1999.

[15] D. E. Myers, "Spatial interpolation: an overview," *Geoderma*, 1994.

[16] J. Li and A. D. Heap, "A review of spatial interpolation methods for environmental scientists," 2008.

[17] S. A. Teukolsky, B. P. Flannery, W. Press, and W. Vetterling, "Numerical recipes in c," *SMR*, vol. 693, no. 1, pp. 59–70, 1992.

[18] G. Jacobson, "Space-efficient static trees and graphs," in *30th annual symposium on foundations of computer science*. IEEE Computer Society, 1989, pp. 549–554.

[19] Y. Li, D. Chen, B. Ding, K. Zeng, and J. Zhou, "A pluggable learned index method via sampling and gap insertion," *arXiv preprint arXiv:2101.00808*, 2021.

[20] https://developer.twitter.com/en, 2018.

Songnian Zhang received his M.S. degree from Xidian University, China, in 2016 and he is currently pursuing his Ph.D. degree in the Faculty of Computer Science, University of New Brunswick, Canada. His research interest includes cloud computing security, big data query and query privacy.

Suprio Ray (Member, IEEE) is an Associate Professor with the Faculty of Computer Science, University of New Brunswick, Fredericton, Canada. He received a Ph.D. degree from the Department of Computer Science, University of Toronto, Canada, in 2015. His research interests include big data and database management systems, run-time systems for scalable data science, provenance and privacy issues in big data and query processing on modern hardware. E-mail: sray@unb.ca.

Rongxing Lu (Fellow, IEEE) is a Mastercard IoT Research Chair, a University Research Scholar, an associate professor at the Faculty of Computer Science (FCS), University of New Brunswick (UNB), Canada. Before that, he worked as an assistant professor at the School of Electrical and Electronic Engineering, Nanyang Technological University (NTU), Singapore from April 2013 to August 2016. Rongxing Lu worked as a Postdoctoral Fellow at the University of Waterloo from May 2012 to April 2013. He was awarded the most prestigious "Governor General's Gold Medal", when he received his PhD degree from the Department of Electrical & Computer Engineering, University of Waterloo, Canada, in 2012; and won the 8th IEEE Communications Society (ComSoc) Asia Pacific (AP) Outstanding Young Researcher Award, in 2013. Dr. Lu is an IEEE Fellow. His research interests include applied cryptography, privacy enhancing technologies, and IoT-Big Data security and privacy. He has published extensively in his areas of expertise, and was the recipient of 9 best (student) paper awards from some reputable journals and conferences. Currently, Dr. Lu serves as the Chair of IEEE ComSoc CIS-TC (Communications and Information Security Technical Committee), and the founding Co-chair of IEEE TEMS Blockchain and Distributed Ledgers Technologies Technical Committee (BDLT-TC). Dr. Lu is the Winner of 2016-17 Excellence in Teaching Award, FCS, UNB.

Yandong Zheng received her M.S. degree from the Department of Computer Science, Beihang University, China, in 2017 and she is currently pursuing her Ph.D. degree in the Faculty of Computer Science, University of New Brunswick, Canada. Her research interest includes cloud computing security, big data privacy and applied privacy.