Effectively Learning Spatial Indices

Jianzhong Qi[1], Guanli Liu[1], Christian S. Jensen[2], Lars Kulik[1]
[1]School of Computing and Information Systems, The University of Melbourne, Australia
[2]Department of Computer Science, Aalborg University, Denmark
[1]{jianzhong.qi, guanli.liu1, lkulik}@unimelb.edu.au [2]csj@cs.aau.dk

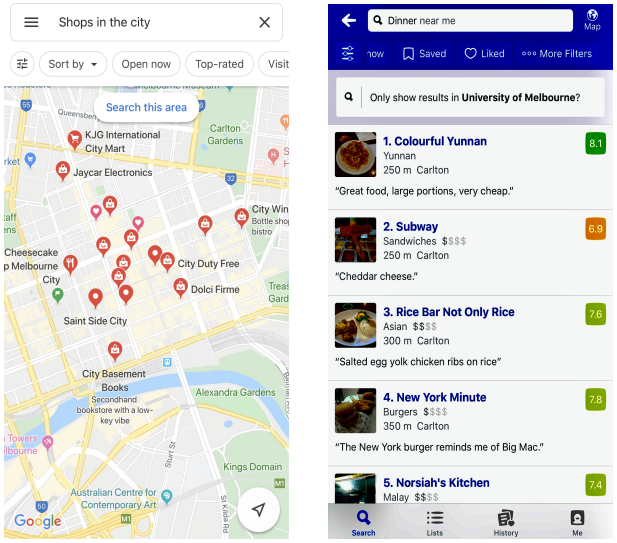
ABSTRACT

Machine learning, especially deep learning, is used increasingly to enable better solutions for data management tasks previously solved by other means, including database indexing. A recent study shows that a neural network can not only learn to predict the disk address of the data value associated with a one-dimensional search key but also outperform B-tree-based indexing, thus promises to speed up a broad range of database queries that rely on B-trees for efficient data access. We consider the problem of learning an index for two-dimensional spatial data. A direct application of a neural network is unattractive because there is no obvious ordering of spatial point data. Instead, we introduce a rank space based ordering technique to establish an ordering of point data and group the points into blocks for index learning. To enable scalability, we propose a recursive strategy that partitions a large point set and learns indices for each partition. Experiments on real and synthetic data sets with more than 100 million points show that our learned indices are highly effective and efficient. Query processing using our indices is more than an order of magnitude faster than the use of R-trees or a recently proposed learned index.

PVLDB Reference Format:
Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. Effectively Learning Spatial Indices. *PVLDB*, 13(11): 2341-2354, 2020.
DOI: https://doi.org/10.14778/3407790.3407829

1. INTRODUCTION

Spatial data and query processing are becoming ubiquitous. This is due in part to the proliferation of location-based services such as digital mapping, location-based social networking, and geo-targeted advertising. For example, Google Maps includes a large number of spatial objects such as *points of interest* (POIs). The query "Search this area (mobile window view)" in Fig. 1a is a typical example of spatial *window query*. As another example, Foursquare[1], a popular social networking app, has a "Dinner near me" (Fig. 1b) function that returns restaurants sorted by their distances to the app user. This is a k *nearest neighbor* (kNN) query, where the spatial objects are restaurants.



(a) Window query **(b)** kNN query

Figure 1: Spatial queries

To process spatial queries, indices such as *R-trees* [16], *kd-trees* [5], and *quadtrees* [12] are used. A tree traversal is often required during query processing, which may access many tree nodes and yield decreased performance, especially when the index is stored in external memory.

A recent study [26] advances the notion of a *learned index*. A database index is formulated as a function f that is learned to map a search key to the storage address where the corresponding data object is stored. Given a learned function, an object can be located by a function invocation.

Motivated by the performance benefits of learned indices for one-dimensional data [26], we learn an index for spatial data, with a focus on point data. We target applications such as those mentioned above, where queries are much more frequent than data updates. Our index is highly efficient for queries, while it also supports dynamic updates.

To learn an index, a basic step is to order the data points. Given a set of ordered data points, the learned function f maps a search key $p.key$ to the *rank* (a percentage value, denoted by $p.rank$) of the corresponding data point p. This effectively learns a *cumulative distribution function* (CDF)

[1]`https://www.foursquare.com/`

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407829

of the data set indexed. The address of p is computed as $p.addr = f(p.key) \cdot n$, where n is the data set cardinality.

To order spatial points in order to learn an index, an existing solution, the *Z-order model* [46], uses a *space-filling curve* (SFC), i.e., the *Z-curve* [35]. To apply an SFC, the underlying space is partitioned by a grid. The SFC then enumerates the cells in the grid. The visiting order gives every cell (and the points inside) a *curve value*, i.e., a *Z-value* for Z-curves. The Z-order model uses a Z-value as the search key $p.key$ and learns a function f to predict the rank of the corresponding point p among the data points sorted by their Z-values. For example, in Fig. 2a, there are eight data points $p_1, p_2, \ldots, p_8$. Their Z-values are shown in parentheses, e.g., p_3 has Z-value 6, meaning that $p_3.key = 6$. Among the eight points, points p_1 and p_2 rank ahead of p_3 by their Z-values, thus, $p_3.rank = 3/8$, which is the intended output of f. We plot the CDF to be learned in Fig. 2b.

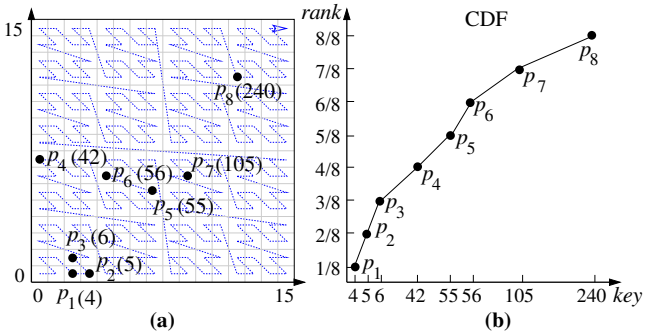


Figure 2: Index prediction with the Z-order model

To use an SFC, a grid must be imposed, but it is difficult to choose an optimal grid resolution. If the grid cells are too large (i.e., a low resolution), many points may share the same cell, which may yield false positives at query time. If we impose a limit of one point per cell (i.e., a high resolution), very small cells may be needed for dense data sets. When the data distribution is skewed, this leads to a large number of empty cells and to uneven gaps between the curve values of the data points. Meanwhile, the ranks are always continuous. This makes it difficult to learn function f that maps the curve values to ranks. Fig. 2b has large and uneven gaps between the Z-values of p_3 and p_4 ($42 - 6 = 36$), p_6 and p_7 ($105 - 56 = 49$), and p_7 and p_8 ($240 - 105 = 135$). The result is a CDF with many segments, which is difficult to approximate using a single function f. This difficulty tends to increase for more points. For the existing learned spatial index, experiments are run on up to 130,000 points, which is below the requirement of some practical applications.

To overcome this limitation, we take advantage of the state-of-the-art R-tree bulk-loading technique [37] to order the data points. This technique maps the data points into a *rank space*, orders them in the rank space using an SFC, and packs every B ordered points into a disk block (B denotes the block size) to form an R-tree. The rank space has the same dimensionality as the original (Euclidean) data space, and the coordinate of a point p in each dimension is the rank of p in the corresponding dimension of the original space. While we do not aim to bulk-load an R-tree, the rank space has the key property to guarantee that there is one point in every row/column of the grid of the SFC. This enables creating more even gaps between the curve values of the data points, which simplifies the function f to be learned.

A challenge in applying the above ordering technique is that it creates an $n \times n$ grid given n data points. When n is large, this yields a very large grid, and there might be many different gaps between the curve values, making it difficult to learn the function f. To address this challenge, we recursively partition the data set (in the original data space) until each partition allows a simple *feedforward neural network* (FFN) to learn an accurate function f. The novelty of our partitioning strategy is that it is learned by a hierarchy of partitioning functions based on the distribution of the underlying data (in contrast to based on the index order of the data [46], detailed in Section 3). A partitioning function takes the coordinates of a data point as input and outputs a partition ID for the point. Data points with the same partition ID form a partition. Thus, the partitioning function determines the partition in which a data point is indexed. This allows us to reuse the partitioning function as an indexing function to predict data locations, which translates into a highly scalable and accurate learned spatial index.

We further design query algorithms for point, window, and kNN queries that exploit this index. Our learned index and query algorithms can handle over 100 million points with high efficiency and accuracy. We also propose algorithms to handle data updates without impinging our query accuracy. In summary, the paper makes the following contributions:

1. We propose to learn a spatial index based on ordering the data points by a rank space-based transformation [37]. To scale to large data sets, we design a multi-dimensional data partitioning technique and learn an index for each partition in a recursive manner.

2. We propose algorithms for point, window, and kNN queries that exploit the learned index. In particular, our kNN algorithm is the first of its kind for a learned spatial index. We also propose update algorithms for the learned spatial index.

3. Using real and synthetic data, our extensive experiments on the proposed index and algorithms show that they can handle data sets of over 100 million points with significant query performance gains over both traditional spatial indices such as R-trees and an existing spatial adaption of the learned index technique [46].

2. RELATED WORK

We review studies on spatial indices and learned indices.

Spatial indices. Spatial indices [13] organize spatial data to enable efficient query processing. They can be classified into data partitioning based, space partitioning based, and mapping-based indices.

Data partitioning based indices partition the data based on clusters formed by the data. The *R-trees* [3, 4, 16, 42] are typical examples. An R-tree is often maintained by means of dynamic updates. In that case, the node in which a data point is placed is determined by not only the point's location (i.e., its coordinates), but also by the order in which the points are inserted. Learning an indexing function that maps the coordinates of data points to nodes is thus more difficult, and the resulting function may be less effective. An alternative approach is to pack data points into leaf nodes, thereby building, or bulk-loading, an R-tree bottom up. Most packing procedures [1, 8, 15, 23, 27, 41] rely on some ordering of the points based on their coordinates, e.g., sorting by their

x-coordinates [41]. Every B points are packed into a node according to the ordering, where B is the node capacity. This establishes a relationship between data point locations and their leaf nodes, which may be learned.

Space partitioning based indices recursively partition the space in which the data is embedded until the spatial objects in a partition fit into an index node. *Kd-trees* [5] and *quadtrees* [12] are typical examples. To learn a meaningful mapping between the spatial objects and the nodes in which they are stored, some order on the nodes must be established. Since each node corresponds to a space partition, this becomes a problem of establishing an order on the partitions, which resembles the idea of an SFC. We discuss this with the mapping-based indices. Attempts also have been made to obtain balanced indices using space partitioning [19, 39]. A typical example is the *K-D-B-tree* [39], which implements a kd-tree with a B-tree structure to support block-based storage. We compare with it empirically.

Mapping-based indices map multi-dimensional objects to one-dimensional values. The mapped values are indexed using a one-dimensional index, e.g., the B^+-tree. SFCs are often used for mapping spatial objects with low dimensionality. For two popular SFCs, the *Z-curve* [35] and the *Hilbert-curve* [10], their curve values are also called *Z-values* and *Hilbert-values*, respectively. The curve value of a cell is used as the one-dimensional value (i.e., the index key) to which a point in this cell is mapped. The curve values of the data points establish a monotonic order over the points. It is natural to store the points in this order and learn a function that maps a curve value to the rank of the corresponding point, i.e., to the address of the node that holds the point. This is done in a recent learned spatial index [46] using the Z-curve (detailed next). There are other mapping schemes, e.g., [2, 6, 21], which are designed for non-point or higher dimensional objects, and are less relevant here.

Learned indices. The intuition behind a *learned index* is that a database index can be thought of as a function f that maps a search key to the storage address of a data object. If such a function can be learned, a (point) query can be processed by a function invocation in constant time. This avoids a logarithmic-time search over a hierarchical index.

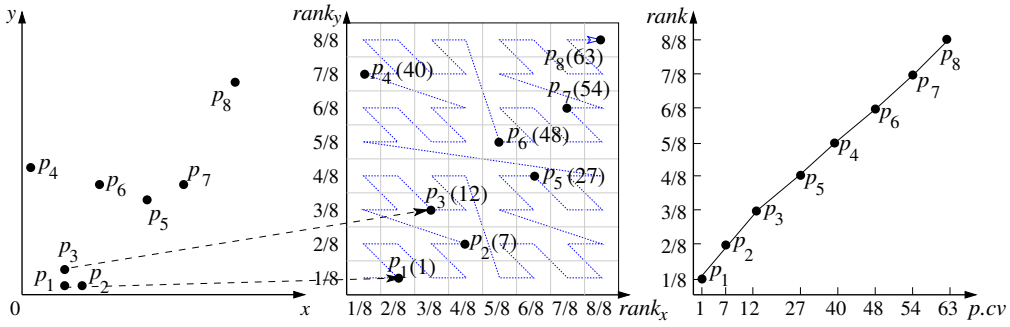
The *recursive model index* (RMI) [26] adopts this approach and learns a function f for one-dimensional data using a *feedforward neural network* (FFN). The data points are first sorted. The function f then maps a search key $p.key$ to the *rank* (a percentage value, denoted by $p.rank$) of the corresponding data point p. Function f essentially learns a *cumulative distribution function* (CDF) on the data set. The address of p is computed as $f(p.key) \cdot n$. To handle larger data sets, RMI creates a hierarchical structure. At the root level, just one function (termed "model" [26]) $f_0(p.key)$ is learned to predict $p.rank \in (0, 1]$. This may only yield a rough prediction of $p.rank$. At the next level (i.e., level 1), m_1 functions are learned, where the i-th function $f_1^{(i)}(p.key)$ is trained to predict the rank for a data point p where $f_0(p.key) \in (i/m_1, (i+1)/m_1], i \in [0, m_1)$. This means that function $f_1^{(i)}$ can focus on a subset of the data points whose ranks fall into $(i/m_1, (i+1)/m_1]$. The same process is repeated recursively to define subsequent levels of the RMI. The function learning (i.e., model training) is done iteratively starting from f_0. The L_2 loss is used to minimize the difference between the predicted ranks and the ground truth ranks of the data points allocated to each function. A number of studies (e.g., [11, 14, 18, 24, 43]) follow this idea and optimize learned indices on one-dimensional data.

The *Z-order model* [46] extends RMI to spatial data by using a Z-curve to order the data points. Function f now learns to predict the rank of point p given its Z-value as the search key. At query time, a query point is first mapped to its Z-value by interleaving the bits of its coordinates. Then, function f predicts the rank (address) given the Z-value.

SageDB [25] is a learned database system which includes a learned multi-dimensional index. To learn this index, the data points are successively sorted and partitioned along a sequence of dimensions into equal-sized cells. Then, the points are ordered by the cells in which they lie, and function f is learned to predict the rank of a point given its coordinates. The dimensions used for sorting and the partition granularity are both learned. No details are available on how this learning is done and on how the cells are ordered. Thus, it is not possible to implement and compare empirically with this technique. Another study [31, 32] partitions a d-dimensional space using a $(d-1)$-dimensional grid and learns an RMI on the d-th dimension for the data points in each grid cell. The dimension used for RMI learning, the ordering of the other $d-1$ dimensions, and the numbers of columns in those $d-1$ dimensions are learned from a sample query workload to obtain better data selectivity. Once these are learned, a *cell table* records the coordinate ranges of the grid cells, which serve to identify the cells intersected by a given window query. The RMI for each intersected cell is used to refine the data points in the cell based on their coordinates in the d-th dimension. This study is not discussed further as we do not assume a known query workload.

Four other learned multi-dimensional indices are proposed in parallel to ours. The *ML-Index* [7] adopts the *iDistance* technique [20] to map points to one-dimensional values and indexes such values with an RMI. The *LISA* [28] structure partitions the data space with a grid, numbers the grid cells with a partially monotonic function, and learns a data partitioning based on this numbering. The *Qd-tree* [47] applies reinforcement learning to optimize the data partitioning for a kd-tree-like structure according to a given query workload. *PolyFit* [29] learns polynomial approximations in order to provide efficient support for approximate multi-dimensional range aggregate queries with guaranteed error bounds.

Update handling can impact the query performance of a learned index. This is because function f may have prediction errors. A prediction *error range*, $[err_{\vdash}, err_{\dashv}]$, needs to be derived from the initial data set from which f is learned, such that the search for a point p can be constrained to $[f(p.key) \cdot n - err_{\vdash}, f(p.key) \cdot n + err_{\dashv}]$. Deletions do not impact the error range if the data points are simply flagged as "deleted." Insertions may impact the error range, and a trivial way to handle insertions is to update the error range to $[err_{\vdash} + i, err_{\dashv} + i]$ after i updates. A tighter estimation of the updated error range [17] is achieved by tracking the error range drifts of a number of *reference points*. At query time, the closest reference points on both sides of the query point are fetched, and their error range drifts are used to estimate the updated error range of the query point with a linear interpolation. The *FITing-Tree* [14] uses an additional fixed-sized buffer for each data segment (an index block) to handle insertions. Data in the buffer is kept sorted for fast search and merge operations. When a buffer is full, it is merged with its corresponding segment (where new segments may



(a) Original space **(b)** Rank space **(c)** CDF

Figure 3: Point ordering and our indexing model

Table 1: Frequently Used Symbols

Symbol	Description
P	A set of data points
d	The dimensionality of P
n	The cardinality of P
p	A data point
q	A query
$\mathcal{M}$	The index structure (mapping function)
$\mathcal{B}$	A data block
B	The capacity of a block
$\mathcal{L}_{\mathcal{M}}$	The loss function of $\mathcal{M}$
N	The partitioning threshold
h	The height of RSMI
$O(M)$	The prediction cost of a sub-model in RSMI
I	The number of new blocks created due to insertions

be created). *ALEX* [9] enhances the RMI by leaving gaps in data nodes and splits nodes to support data insertions.

3. LEARNING A SPATIAL INDEX

Given a set of n points $P = \{p_1, p_2, ..., p_n\}$ in d-dimensional Euclidean space, we aim to index P in a structure $\mathcal{M}$ for efficient query processing. We consider point, window (range), and k nearest neighbor (kNN) queries. For scalability, we consider points storing in external storage (e.g., a hard drive) in blocks of capacity B, i.e., a block holds at most B points. We present the structure in Section 3.1 and scale it to large data sets in Section 3.2. For ease of presentation, we use $d = 2$ in the discussion, although our techniques also apply to any $d \in \mathbb{N}^+$. Table 1 lists frequently used symbols.

3.1 Index Structure

The index $\mathcal{M}$ is a mapping from the coordinates of a point $p \in P$, $p.cord$, to the location in the storage that holds data related to p. This location is given by a block ID: $p.blk$.

Ordering points. We establish a relationship between $p.blk$ and $p.cord$ by packing the points into blocks based on coordinates. We exploit the state-of-the-art R-tree packing strategy [37, 38] which takes three steps (cf. Fig. 3).

(1) The points are first mapped to a *rank space*, which is an $n \times n$ grid, where each row and each column has exactly one point. To perform this mapping, the points are sorted by their x-coordinates (y-coordinates), and the rank $p.rank_x$ ($p.rank_y$) of a point p is used as its x-coordinate (y-coordinate) in the rank space, where ties are broken by the y-coordinates (x-coordinates) of the points in the original space. We assume that no two points have the same coordinates in both dimensions. In Figs. 3a and 3b, point p_1 is mapped to the second column in the rank space. Point p_3 has the same x-coordinate as p_1, but its y-coordinate is larger. Thus, p_3 is mapped to the third column.

(2) An SFC (e.g., a Z-curve) goes through the points in the rank space and maps every point p to a curve value $p.cv$. In Fig. 3b, the curve values are shown in parentheses next to the points, e.g., $p_3.cv = 12$.

(3) The points are sorted in ascending order of their curve values, and every B points are packed into a block in the sorted order. Let $p.rank$ be the rank of point p in the sorted order. The block ID for p is computed as:

$$p.blk = \lfloor p.rank \cdot n/B \rfloor \tag{1}$$

This packing strategy was used previously [37, 38] to construct R-trees with worst-case optimal window query performance. Here, we use this strategy to obtain a more uniform data distribution and more even gaps between the curve values of the data points. In Fig. 3c, our minimum and maximum gaps between the curve values of two adjacently ranked points are $12-7=5$ and $27-12=15$, while those given by the Z-ordering [46] are $5-4=1$ and $240-105=135$, respectively (cf. Fig. 2). Our variance in the curve value gaps is much smaller, leading to a simpler CDF to be learned.

Model learning. Index $\mathcal{M}$ is learned so that it approximately maps $p.cord$ to $p.blk$ for each point p:

$$p.blk \approx \mathcal{M}(p.cord) \tag{2}$$

Any regression model may be used to learn $\mathcal{M}$. We follow recent learned indices [26, 46] and use a *multilayer perceptron* (MLP, a type of feedforward neural networks), to exploit its ability to learn a non-linear function. We minimize the L_2 loss on the model predictions using standard learning procedures (*stochastic gradient descent*, SGD). The loss function $\mathcal{L}_{\mathcal{M}}$ is defined as:

$$\mathcal{L}_{\mathcal{M}} = \sum_{p \in P} (\mathcal{M}(p.cord) - p.blk)^2 \tag{3}$$

After $\mathcal{M}$ is trained, we use it to predict $p.blk$ for every point p and record the maximum errors for the cases where $\mathcal{M}(p.cord)$ is smaller than $p.blk$ and $\mathcal{M}(p.cord)$ exceeds $p.blk$, denoted by $\mathcal{M}.err_{\vdash}$ and $\mathcal{M}.err_{\dashv}$, respectively.

$$\mathcal{M}.err_{\vdash} = \max_{\forall p \in P(\mathcal{M}(p.cord) < p.blk)} \{|\mathcal{M}(p.cord) - p.blk|\} \tag{4}$$

$$\mathcal{M}.err_{\dashv} = \max_{\forall p \in P(\mathcal{M}(p.cord) > p.blk)} \{\mathcal{M}(p.cord) - p.blk\} \tag{5}$$

Discussion. The design of function $\mathcal{M}$ and its learning process resemble those of previous learned indices [26, 46].

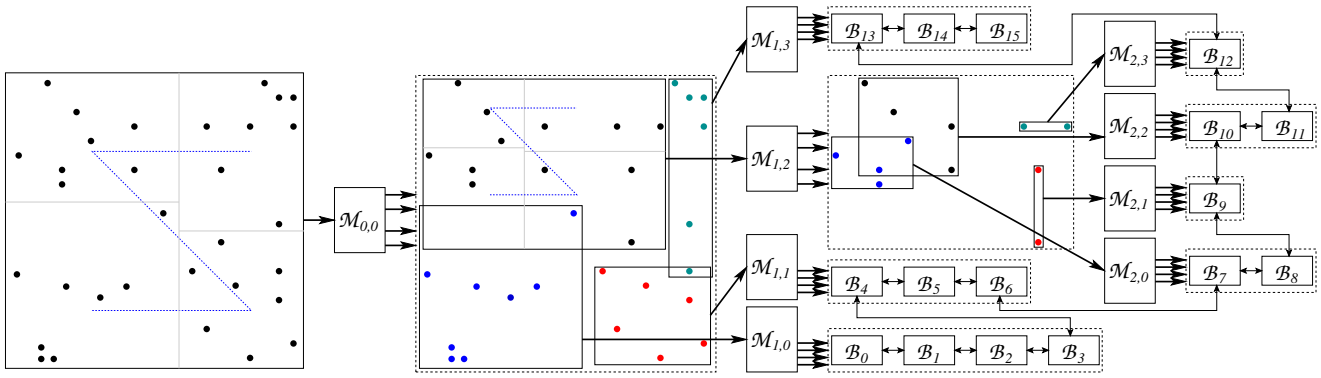


Figure 4: RSMI index structure ($N = 8$ and $B = 2$)

The advantage of $\mathcal{M}$ derives from the rank space based ordering. As discussed above, this ordering creates more even gaps between the curve values of the data points. Meanwhile, the gap between of the ranks of two adjacently ranked points is a constant $(1/n)$. This makes it easier to learn a mapping from the coordinates (which determine the curve values) to block IDs (which are determined by the ranks, cf. Fig. 3c).

3.2 Scaling to Large Data sets

Larger data sets make it more difficult for a single function to map all data points to their ranks. We propose a *recursive spatial model index* (RSMI) for such cases. Suppose that function $\mathcal{M}$ can predict block IDs of up to N points with sufficiently high accuracy. Such an N value depends on the data distribution and the model learning capacity, and it may be determined empirically. Since N points occupy $\lceil N/B \rceil$ blocks, this means that we have a function that can predict $\lceil N/B \rceil$ different block IDs with a high accuracy.

Given a data set P with cardinality $n > N$, our RSMI model recursively partitions P until each partition has at most N points. Then, for each partition, we learn an indexing model following the procedure in Section 3.1.

The partitioning strategy works as follows. We start by partitioning the full data set P into $\lceil N/B \rceil$ partitions (such that we may reuse $\mathcal{M}$ to predict $\lceil N/B \rceil$ different partition IDs later). This partitioning is done via a $2^{\lfloor \log_4 N/B \rfloor} \times 2^{\lfloor \log_4 N/B \rfloor}$ grid with $4^{\lfloor \log_4 N/B \rfloor} \leq N/B$ cells that form at most $\lceil N/B \rceil$ partitions. In Fig. 4, when $N = 8$ and $B = 2$, $2^{\lfloor \log_4 N/B \rfloor} \times 2^{\lfloor \log_4 N/B \rfloor} = 2 \times 2$ cells partition the space.

The grid is created by first cutting the data space into $2^{\lfloor \log_4 N/B \rfloor}$ columns where each column has $\lceil n/2^{\lfloor \log_4 N/B \rfloor} \rceil$ points (i.e., partitioning by x-coordinates). We further cut each column individually into $2^{\lfloor \log_4 N/B \rfloor}$ cells such that each cell has at most $\lceil n/4^{\lfloor \log_4 N/B \rfloor} \rceil$ points (i.e., partitioning by y-coordinates). The grid follows the data distribution and is non-regular, but is still a $2^{\lfloor \log_4 N/B \rfloor} \times 2^{\lfloor \log_4 N/B \rfloor}$ grid. We can apply an SFC of order $\lfloor \log_4 N/B \rfloor$ to the grid to obtain a curve value for each cell. We learn a mapping function $\mathcal{M}_{0,0}$ (i.e., the 0-th model at level 0 of RSMI) to map a point $p \in P$ to the curve value corresponding to its cell. To learn $\mathcal{M}_{0,0}$, we reuse the model structure used for $\mathcal{M}$ (since we know that $\mathcal{M}$ predicts $\lceil N/B \rceil$ different values with a high accuracy). The loss function for the learning also resembles that of $\mathcal{M}$ (Equation 3), except that now the ground truth is not $p.blk$ but the curve value of p in the grid for $\mathcal{M}_{0,0}$.

Once $\mathcal{M}_{0,0}$ is trained, we use it to predict a curve value for each point $p \in P$. Since a learned model may not be fully accurate, $\mathcal{M}_{0,0}(p.cord)$ may not yield the curve value of p. We group the points in P by the predicted curve values. This results in a learned point grouping, as illustrated by the differently colored points in Fig. 4. For the j-th group, if it still has more than N points, we repeat the above partitioning procedure to learn a function $\mathcal{M}_{1,j}$ (e.g., $\mathcal{M}_{1,2}$) that predicts the partition membership of each point p in this group. Otherwise, we follow the procedure in Section 3.1 to learn a function $\mathcal{M}_{1,j}$ to predict $p.blk$ for each point p in this group. We call such a function a *leaf model*.

When the partitioning is complete and the data points are all packed into blocks, in each block, we further store pointers (block IDs) to its preceding and subsequent blocks. In Fig. 4, $\mathcal{B}_i$ represents a block, and arrows between blocks represent the pointers. These pointers allow data scans for queries. The order of blocks under the same leaf model follows that of the data points in the blocks. The order of blocks under the different leaf models follows the order of the partition IDs (recursively) corresponding to the leaf models.

Discussion. Our RSMI model looks similar to the RMI model [26] at a first glance. However, the design strategies of RSMI and RMI are fundamentally different. In RMI, each sub-model $\mathcal{M}_{i,j}$ handles data points whose *predicted ranks* (i.e., *model output*) fall in a range $(j/m_i, (j+1)/m_i], j \in [0, m_i)$, assuming m_i sub-models at level i). Due to the nature of SFCs, points ranked adjacently can have vastly different curve values and may be quite far apart (e.g., p_7 and p_8 in Fig. 2). Thus, each sub-model may still need to handle points with very different curve values, and the mapping to ranks is still difficult to learn. In contrast, in RSMI, each sub-model $\mathcal{M}_{i,j}$ handles data points whose *coordinates* (i.e., *model input*) fall into a region. This allows each sub-model to focus on a subset of points nearby, which may facilitate better prediction accuracy, to be verified in experiments.

Also note that we have not used rank space for the higher-level sub-models in RSMI. The impact of skewed data distribution is mitigated by the use of non-regular grids and the partitioning of data points by model predictions.

For query processing, RSMI only requires a function invocation to determine the (single) sub-model to be accessed at each level. As shown in the experiments, this is more efficient than traditional hierarchical indices that require scanning and comparing entries (e.g., MBRs) in the inner nodes.

4. QUERY PROCESSING

We present algorithms to process point (Section 4.1), window (Section 4.2), and kNN (Section 4.3) queries using RSMI.

4.1 Point Queries

Algorithm 1: Point Query

Input: q: a query point.
Output: A pointer to the point indexed in our RSMI structure that has the same coordinates as q.

```
1 i ← 0; j ← 0;
2 while M_{i,j} is not a leaf model do
3     j ← M_{i,j}(q.cord); i ← i + 1;
4 j ← M_{i,j}(q.cord);
5 for j' ∈ [j − M.err_⊢, j + M.err_⊣] do
6     if q ∈ B_{j'} then
7         return a pointer to the point in B_{j'} that has the
            same coordinates as q;
8 return NULL;
```

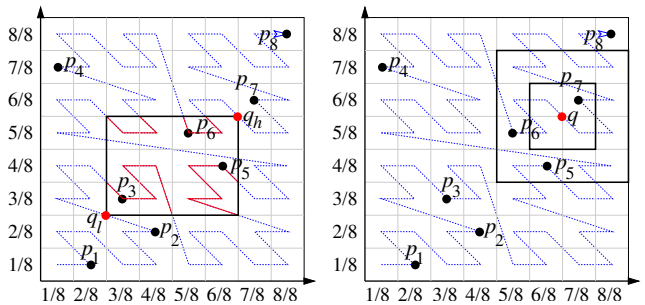
As summarized in Algorithm 1, point queries are done via a recursive call of the sub-models in the RSMI using the coordinates of a query point q as the input (Lines 1 to 3). The process starts from the root sub-model $\mathcal{M}_{0,0}$. At each level, only one sub-model needs to be visited. Let $\mathcal{M}_{i,j}$ be the sub-model visited at level i. Then, at level $i+1$, the sub-model to be visited is $\mathcal{M}_{i+1,j'}$ where $j' = \mathcal{M}_{i,j}(q.cord)$. This process continues until a leaf model is reached. The leaf model output is the predicted block ID of q, $\mathcal{M}(q.cord)$. We examine the corresponding block and its neighboring blocks as bounded by $[\mathcal{M}(q.cord) - \mathcal{M}.err_{\vdash}, \mathcal{M}(q.cord) + \mathcal{M}.err_{\dashv}]$. If q is found, we return a pointer to the found point (Lines 4 to 7). Otherwise, q is not in our structure (Line 8).

Correctness. The algorithm correctness is guaranteed by the RSMI structure. As Fig. 4 shows, if $j' = \mathcal{M}_{i,j}(p.cord)$, point p is only indexed by $\mathcal{M}_{i+1,j'}$ at the next level. Thus, if $j' = \mathcal{M}_{i,j}(q.cord)$ for q, we only need to access $\mathcal{M}_{i+1,j'}$ at the next level. At the leaf level, the model prediction may have an error, which is bounded by $\mathcal{M}.err_{\vdash}$ and $\mathcal{M}.err_{\dashv}$. Scanning the blocks in $[\mathcal{M}(q.cord) - \mathcal{M}.err_{\vdash}, \mathcal{M}(q.cord) + \mathcal{M}.err_{\dashv}]$ guarantees no false negatives.

Query cost. There are two cost components. The first is for model prediction, which consists of numerical computations. This cost depends on the prediction model. We use $O(M)$ to denote the prediction cost of a sub-model. For example, consider a simple fully connected feedforward neural network with two neurons in the input layer, m neurons in the (only) hidden layer, and a single neuron in the output layer. Then, $O(M) = O(2m)$. Given that RSMI has height h, we need $O(hM)$ time to predict a block ID for q. The second cost component is for locating the query point on disk. In the best case, we need to access the disk block with ID $\mathcal{M}(q.cord)$, which takes $O(B)$ time (scanning B points in the block). In the worst case, we need to access $\mathcal{M}.err_{\vdash} + \mathcal{M}.err_{\dashv} + 1$ blocks with $O((\mathcal{M}.err_{\vdash} + \mathcal{M}.err_{\dashv})B)$ time. Overall, a point query takes $O(hM + (\mathcal{M}.err_{\vdash} + \mathcal{M}.err_{\dashv})B)$ time.

4.2 Window Queries

Fig. 5a illustrates the window query algorithm. The solid rectangle in the middle denotes a window query q. The grid cells in q are covered by several red segments of the SFC. Processing query q means finding the points corresponding to the curve segments in q. For each such segment, if we can locate the storage location of the points corresponding to the minimum and maximum curve values of the segment then any other points on the curve segment must be stored between these two points, which form the query answer. This is because the data points are stored in curve value order.



(a) Window query **(b)** kNN query ($k = 3$)

Figure 5: Query examples (The queries are plotted in the rank space for ease of illustration. They are processed in the original space since RSMI takes point coordinates in the original space as input and outputs the predicted block ID.)

Identifying the points for the minimum and maximum curve values of the curve segments is feasible (e.g., via query window mapping [48]). However, locating such points implies issuing point queries, which may cause repetitive scanning over the same blocks (due to prediction errors).

To avoid excessive data scans, the window query algorithm only locates the points corresponding to the minimum and maximum curve values among *all* curve segments in q. We thus only need two point queries for a window query:

(1) Compute two points q_l and q_h in q with the minimum and the maximum curve values, respectively.

(2) Compute point queries for q_l and q_h and retrieve all points between them. This forms a superset of the query answer, since some of these points may not be in q (p_4 in Fig. 5a is between p_3 and p_6 but not in q).

(3) Filter the points retrieved in Step 2 with q and return the remaining points.

We focus on locating q_l and q_h in Step 1. Steps 2 and 3 are done by Algorithm 1 and a simple block scan.

The locations of q_l and q_h depend on the SFC. We discuss the cases for two commonly used SFCs, the Z-curve and the Hilbert-curve. For the Z-curve, q_l and q_h are the bottom left and the top right corners of the query window, respectively [30]. This is because a Z-curve goes through the space from the bottom left to the top right recursively. For example, in Fig. 5a, the bottom left and top right cells in q are the two with the minimum and the maximum curve values. For the Hilbert-curve, q_l and q_h must be on the boundary of q [48]. Computing q_l and q_h from the boundary in the rank space requires B-tree searches to map the point coordinates to their ranks [37, 38]. To avoid constructing and searching B-tree indices, we heuristically use the four corners of q, denoted by q_{xl}, q_{xh}, q_{yl}, and q_{yh}, for query processing.

Algorithm. Based on q_l and q_h, Algorithm 2 summarizes the window query algorithm. We first obtain q_l and q_h (Line 1). We query these two points using our point query algorithm (Line 2). If q_l (q_h) is found, we use its block ID $q_l.blk$ ($q_h.blk$) as the lower (upper) bound for data scanning. Otherwise, we use $\mathcal{M}(q_l.cord) - \mathcal{M}.err_{\vdash}$ ($\mathcal{M}(q_h.cord) + \mathcal{M}.err_{\dashv}$) to approximate the bound (Lines 3 to 10). We then scan the data blocks between the lower and upper bounds and return the points in q (Lines 11 to 15). For Hilbert-curves, we need point queries for q_{xl}, q_{xh}, q_{yl}, and q_{yh}. The lower and upper bounds for data scanning be-

come $\min\{\mathcal{M}(q_i.cord) - \mathcal{M}.err_{\vdash}\}$ and $\max\{\mathcal{M}(q_i.cord) + \mathcal{M}.err_{\dashv}\}$, respectively, where $q_i \in \{q_{xl}, q_{xh}, q_{yl}, q_{yh}\}$.

Algorithm 2: Window Query

Input: q: a window query.
Output: S: the set of data points in q.

```
1  Obtain points q_l and q_h for query window q;
2  Run point queries for q_l and q_h;
3  if q_l is found then
4      begin ← q_l.blk;
5  else
6      begin ← M(q_l.cord) − M.err_⊢;
7  if q_h is found then
8      end ← q_h.blk;
9  else
10     end ← M(q_h.cord) + M.err_⊣;
11 for i ∈ [begin, end] do
12     for p ∈ B_i do
13         if p in q then
14             S ← S ∪ {p};
15 return S;
```

Query cost. Algorithm 2 runs two (four for Hilbert-curves) point queries and a data scan. The point queries take $O(hM + (\mathcal{M}.err_{\vdash} + \mathcal{M}.err_{\dashv})B)$ time, as discussed earlier. The maximum number of blocks scanned is $(\mathcal{M}(q_h.cord) + \mathcal{M}.err_{\dashv}) - (\mathcal{M}(q_l.cord) - \mathcal{M}.err_{\vdash})$. The overall query time is $O(hM + (\mathcal{M}.err_{\vdash} + \mathcal{M}.err_{\dashv})B + (\mathcal{M}(q_h.cord) + \mathcal{M}.err_{\dashv})B - (\mathcal{M}(q_l.cord) - \mathcal{M}.err_{\vdash})B) = O(hM + (2\mathcal{M}.err_{\vdash} + 2\mathcal{M}.err_{\dashv} + \mathcal{M}(q_h.cord) - \mathcal{M}(q_l.cord))B)$. For Hilbert-curves, the query time is $O(hM + (2\mathcal{M}.err_{\vdash} + 2\mathcal{M}.err_{\dashv} + \max\{\mathcal{M}(q_i.cord)\} - \min\{\mathcal{M}(q_i.cord))B)$, where $q_i \in \{q_{xl}, q_{xh}, q_{yl}, q_{yh}\}$.

Discussion. Algorithm 2 offers approximate answers. The errors occur for two reasons. First, when Hilbert-curves are used, the four corners of q may not correspond to the exact minimum and maximum curve values bounded by q. Second, when querying q_l and q_h (or q_{xl}, q_{xh}, q_{yl}, and q_{yh}), if these points are not indexed in the RSMI, we may not return the point just next to them. Thus, the point queries may not bound the accurate blocks to be scanned for q.

Empirically, the query answer errors are small. Experiments under various settings show that the query answer accuracy (i.e., recall) is consistently above 87%. Our query answer does not have false positives (i.e., we may miss points in q but will not return any points outside q). This suits applications that need to quickly identify regions with POIs but may not require all POIs in the region, e.g., to recommend a region with many restaurants to a user.

The RSMI can also offer exact query answers with a simple modification to store a *minimum bounding rectangle* (MBR) for each sub-model in its parent model. This modification enables an R-tree-style tree traversal of the RSMI to compute exact query answers with reasonable efficiency.

4.3 K Nearest Neighbor Queries

We can use R-tree kNN algorithms (e.g., the best-first algorithm [40]) to compute kNNs accurately using the RSMI with MBRs. Below, we design a prediction-based approximate kNN algorithm to achieve high query efficiency.

Our approximate kNN algorithm follows a classical search region expansion paradigm. It starts with a small search region around the query point q and keeps expanding until k points are found. Each time the search region is expanded, a window query is run using Algorithm 2 to find points in the region. See Fig. 5b for a 3NN example. The query point q is represented by a red dot, and the window queries run are represented by the two solid rectangles enclosing q.

Algorithm. Algorithm 3 summarizes the approximate algorithm, where Q is a queue to store the found NNs prioritized by their distances to q (Line 1). The algorithm first estimates an initial search region based on the data distribution. Intuitively, if the points are uniformly distributed in a unit space, a search region of size k/n around q is expected to contain k points. We thus use a rectangular-shaped search region centered at q with width $\alpha_x\sqrt{k/n}$ and height $\alpha_y\sqrt{k/n}$ as the initial search region (Line 2). Here, α_x and α_y are *skew parameters* to adjust the search region size in both dimensions according to data skew – $\alpha_x = \alpha_y = 1$ for uniform data. We estimate α_x and α_y later.

Using the initial search region, we run a window query to find the data blocks containing points in the region (Lines 4 and 5). We go through these blocks. If a block $\mathcal{B}_i$ is closer to q (measured by the MINDIST metric [40]) than the currently found k-th NN, $Q[k]$, we go through every point $p \in \mathcal{B}_i$ and add p to Q if p is closer to q than $Q[k]$ (Lines 6 to 9). Next, if there are less than k points in Q, we double the width and the height of the search region (Lines 10 and 11). If the distance between q and $Q[k]$ ($dist(q, Q[k])$) exceeds $\sqrt{width^2 + height^2}/2$ (i.e., $Q[k]$ is outside the current search region), we also enlarge the width and height to be $2 \cdot dist(q, Q[k])$ (Lines 12 and 13). Otherwise, we terminate and return the first k points in Q (Lines 14 to 16).

Algorithm 3: kNN Query

Input: q: a query point; k: the number of targeted NNs
Output: S: the set of (approximate) kNNs of q.

```
1  Initialize an empty priority queue Q;
2  width ← α_x √(k/n); height ← α_y √(k/n);
3  while TRUE do
4      Construct window query wq with q, width, and height;
5      Query wq to obtain data block ID range [begin, end];
6      for i ∈ [begin, end] do
7          if B_i is unvisited AND
             (mindist(q, B_i) < dist(q, Q[k]) OR Q.size() < k)
           then
8              for p ∈ B_i AND (dist(q, p) < dist(q, Q[k]) OR
                 Q.size() < k) do
9                  Insert p into Q;
10     if Q.size() < k then
11         width ← 2 · width; height ← 2 · height;
12     else if dist(q, Q[k]) > √(width² + height²)/2 then
13         width ← 2 · dist(q, Q[k]); height ← 2 · dist(q, Q[k]);
14     else
15         break;
16 return S ← first k elements in Q;
```

Estimating α_x and α_y. When the data distribution is non-uniform, we estimate α_x and α_y by learning the CDF in the x- and y-dimensions. Let $CDF_x(X)$ be a function to predict the percentage of points with an x-coordinate less than or equal to X. Then, α_x is estimated by the slope of $CDF_x(X)$ at the x-coordinate $q.cord_x$ of the query q:

$$\alpha_x = \Delta / (CDF_x(q.cord_x + \Delta) - CDF_x(q.cord_x)) \quad (6)$$

Here, Δ is a system parameter (0.01 in our experiments). We estimate α_y with $CDF_y(Y)$ for the y-dimension.

Obtaining CDF is expensive. We compute a *piecewise mapping function* [48] $PMF_x(X)$ to approximate $CDF_x(X)$ (and $PMF_y(Y)$ for $CDF_y(Y)$). We partition the data set into γ partitions by their x-coordinates and compute a cumulative count $x_i.c$ for the boundary point x_i of each partition. We let $PMF_x(x_i.cord) = x_{i-1}.c/n$, where $x_i.cord$ is the x-coordinate of x_i. We compute piecewise linear functions to connect every pair of points $(x_i.cord, x_{i-1}.c/n)$ to form $PMF_x(X)$. We use $\gamma = 100$ in the experiments.

Query cost. The key factor that determines the time cost of a kNN query is the search region size. A small initial search region may incur more searches later, while a large initial search region may incur many unnecessary data block accesses. In the worst case, the number of window queries needed is $\log_2 1/(\min\{\alpha_x, \alpha_y\}\sqrt{k/n})$, which is the number of times that the search region width/height needs to be doubled to cover a unit space. Let W be the time cost of a window query derived in Section 4.2. Then, the worst-case time cost of a kNN query is $O(W \log_2 1/(\min\{\alpha_x, \alpha_y\}\sqrt{k/n}))$.

5. UPDATE HANDLING

Our index allows both insertions and deletions of data. Given a new point p to be inserted, we first run a point query for p. We insert p into the block predicted by the query. There are two cases: (1) If the predicted block has space for p (e.g., space left by a deleted point), we simply place p in the block. (2) If the predicted block is full, we create a new block $\mathcal{B}$, place p in $\mathcal{B}$, and insert $\mathcal{B}$ as the next block of the predicted block in the list of data blocks. We mark $\mathcal{B}$ as an inserted block such that it does not count towards the error bounds ($\mathcal{M}.err_{\vdash}$ and $\mathcal{M}.err_{\dashv}$) during query processing. In either case, we need to recursively update the MBRs of the ancestor models indexing p to complete the insertion. An insertion takes $O(hM + (\mathcal{M}.err_{\dashv} + \mathcal{M}.err_{\vdash})B + IB + h)$ time, where $O(hM + (\mathcal{M}.err_{\dashv} + \mathcal{M}.err_{\vdash})B)$ is the point query time to locate p, $O(IB)$ is the additional search time on the blocks created after the predicted block by previous insertions, and $O(h)$ is the MBR update time.

Given a point p to be deleted, we run a point query for p. When the data block containing p is found, we swap p with the last point in this block and mark p as "deleted." We then recursively update the MBRs of the ancestor models. We do not delete a disk block when it underflows (e.g., when becoming half empty), to ensure the validity of the error bounds. The deletion also takes an $O(hM + (\mathcal{M}.err_{\dashv} + \mathcal{M}.err_{\vdash})B + IB + h)$ time, as it also needs a point query to locate p and entails recursive MBR updates. We omit the pseudo-code of these update algorithms for succinctness.

Insertions and deletions impact the layout of the learned index and hence impact the query performance (i.e., adding an $O(IB)$ cost for each point query). A periodic rebuild may be run (e.g., overnight) to retain a high query efficiency.

6. EXPERIMENTS

We report on experimental results on the RSMI.

6.1 Experimental Setup

The experiments are done on a computer running 64-bit Ubuntu 20.04 with a 3.60 GHz Intel i9 CPU, 64 GB RAM, and a 1 TB hard disk. We use *PyTorch* 1.4 [36] and its C++ APIs to implement the learned indices based on CPU. The traditional indices are implemented using C++ (except for the RR*, which was implemented in C [4]) based on CPU.

Table 2: Parameters and Their Values

Parameter	Setting
Distribution	Uniform, Normal, **Skewed**
n (million)	1, 2, 4, 8, **16**, 32, 64, 128
Query window size (%)	0.0006, 0.0025, **0.01**, 0.04, 0.16
Query window aspect ratio	0.25, 0.5, **1**, 2, 4
k	1, 5, **25**, 125, 625
Inserted points (%)	10, 20, 30, 40, 50
Deleted points (%)	10, 20, 30, 40, 50

Competitors. We compare with the following indices:

(1) Z-order model [46] (**ZM**): This model predicts the storage address of a data point by its Z-value (cf. Section 2). We implement a recursive version of the model with three levels with 1, $\sqrt{n/B^2}$, and n/B^2 sub-models each.

(2) Grid File [33] (**Grid**): This index partitions the data space with a regular grid, assigns data points to cells based on their coordinates, and stores the data points by their cells. We use a $\sqrt{n/B} \times \sqrt{n/B}$ grid, i.e., B points (one block) per cell under a uniform distribution.

(3) K-D-B-tree [39] (**KDB**): This index implements a kd-tree [5] with a B-tree structure to support block storage.

(4) Rank space based R-tree [37, 38] (**HRR**): This is an R-tree bulk-loaded using the rank space technique (cf. Section 3.1) and a Hilbert-curve for the ordering. This R-tree offers the state-of-the-art window query performance.

(5) Revised R*-tree [4] (**RR***): This is an improvement of the R*-tree, which has shown strong query performance.

KDB, HRR, and RR* have up to five levels in the experiments (on data sets with up to 128 million points).

Implementation. We use the original implementation of HRR and RR*. For Grid, ZM, and KDB, no source code is available. We use the static component of a Grid File for moving objects [22] for Grid. We implement ZM and KDB following their papers. We run all indices and algorithms in main memory for ease of comparison (it is straightforward to place the data blocks in external memory).

For all structures, we use data blocks with a capacity of 100, i.e., $B = 100$. The R-tree and K-D-B-tree leaf nodes (blocks) and the Grid File blocks can store up to 100 points each, while the internal nodes of the tree structures store up to 100 MBRs. No buffering is assumed.

For both ZM and RSMI, each sub-model (e.g., $\mathcal{M}_{i,j}$ in RSMI) is a multilayer perceptron (MLP) with an input layer, a hidden layer, and an output layer. The number of nodes in the hidden layer equals the sum of the number of input attributes and the number of output classes divided by two, e.g., 51 in RSMI, where the input is two coordinates and the output has 100 different block ID values. We use the sigmoid activation function for the hidden layer. The MLPs are trained level by level, starting from the root sub-model (cf. Section 3.2), with a learning rate of 0.01 and 500 epochs per MLP. RSMI uses a partitioning threshold $N = 10,000$, i.e., a leaf model handles at most 10,000 data points. It learns the number of levels and the number of sub-models at each level adaptively for each data set. For ease of model training, the point coordinates and block IDs are normalized into the unit range. RSMI uses Hilbert-curves for ordering as these yield better query performance than Z-curves.

Data sets. We use two real data sets: **Tiger** and **OSM**. Tiger contains over 17 million rectangles (950 MB in size)

representing geographical features in 18 Eastern states of the USA [45]. We use the centers of the rectangles as our data points. OSM contains over 100 million points (2.2 GB in size) in the USA extracted from OpenStreetMap [34].

We generate three groups of synthetic data sets, **Uniform**, **Normal**, and **Skewed**, with up to 128 million points (2.5 GB in size). The synthetic data falls into the unit square. Uniform and Normal data sets follow uniform and normal distributions, respectively. Skewed data sets are generated from uniform data by raising the y-coordinates to their powers y^α ($\alpha = 4$ by default), following HRR [37, 38].

As summarized in Table 2, we vary the data set size n, the query window size and aspect ratio, the query parameter k, and the percentages of points inserted or deleted, respectively. Default settings are shown in boldface.

We generate queries that follow the data distribution for each set of query experiments and report the average **response time** and **number of block accesses** per query. The block accesses serve as a performance indicator for an external memory based implementation of the algorithms.

6.2 Results

We report results on the impact of N, point, window, and kNN query processing, and update handling.

6.2.1 Impact of RSMI Partition Threshold N

Table 3: Impact of N

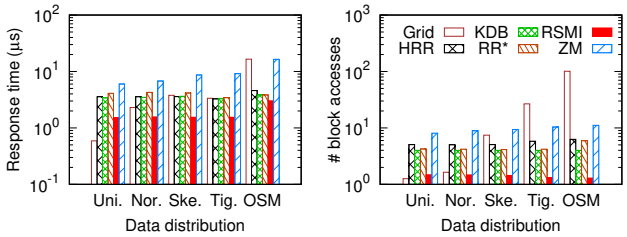
N	2,500	5,000	10,000	20,000	40,000
Construction time (s)	10,997	8,215	7,553	7,602	7,161
Height	9	5	4	4	3
Index size (MB)	488.5	425.5	405.5	398.9	391.3
Query # block accesses	1.28	1.35	1.44	1.47	1.52
Query time (μs)	1.79	1.59	1.44	1.47	1.49

We first study the impact of N to optimize RSMI. As Table 3 shows, when N increases (from 2,500 to 40,000), the index construction time, height, and index size all decrease overall. This is because a larger N means more points in each partition and fewer partitions, leading to fewer levels and sub-models, and hence shorter training times. Meanwhile, the average number of block accesses per point query increases, because the leaf models become less accurate. The point query time, however, first decreases and then increases again. This is resulted from a combined effect of fewer sub-models to compute while more data blocks to examine as N increases. The query time is the shortest when $N = 10,000$. We use this N value in the rest of the experiments.

6.2.2 Point Queries

We use all data points in each data set as the query points and report the average performance per point query.

Varying the data distribution. As Fig. 6 shows, RSMI offers the best query performance on both real (Tiger and OSM) and synthetic (Normal and Skewed) data. It improves the query time by at least 1.3 times and up to 5.5 times compared with the competing techniques, i.e., 3.0 μs vs. 3.8 μs (KDB) and 16.5 μs (Grid) on OSM. It also improves the number of block accesses by at least a factor of 5.3 (1.4 vs. 7.4 for RSMI vs. Grid on Skewed) and up to 77.5 times (1.3 vs. 100.8 for RSMI vs. Grid on OSM). Grid works better on Uniform data, as such data can be partitioned relatively evenly across the cells and take full advantage of Grid. Note



(a) Query time **(b)** # block accesses

Figure 6: Point query vs. data distribution

that Grid has much higher numbers of block accesses than the other techniques while its running time may not seem as high, because it features a simple checking procedure per block, and the blocks are stored in memory.

Table 4: Prediction Error Bounds ($\mathcal{M}.err_\vdash$, $\mathcal{M}.err_\dashv$)

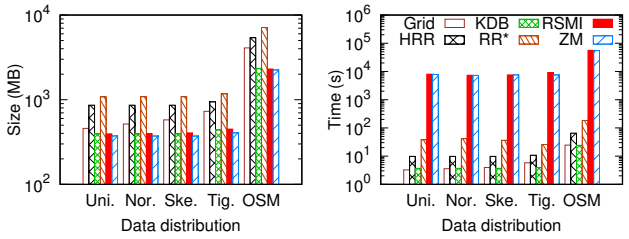
	Uni.	Nor.	Ske.	Tig.	OSM
ZM ($\times 10^4$)	(1.9, 1.9)	(1.2, 5.5)	(0.9, 3.7)	(0.7, 0.7)	(7.4, 11)
RSMI	(43, 82)	(37, 91)	(55, 78)	(70, 69)	(89, 74)

The strength of RSMI comes from its fast and accurate predictions. Table 4 summarizes the maximum prediction errors ($\mathcal{M}.err_\vdash$, $\mathcal{M}.err_\dashv$) of ZM and RSMI. For example, $\mathcal{M}.err_\vdash$ and $\mathcal{M}.err_\dashv$ of RSMI on Skewed are 55 and 78 blocks, respectively. The average number of block accesses is much lower than these bounds, e.g., 1.4 for RSMI on Skewed. ZM offers less accurate predictions due to its design. On Skewed, its prediction error can be as large as 3.7×10^4 blocks, and its average number of block accesses is 8.1 (binary search on the Z-values is used to reduce the number of block accesses). KDB, HRR, and RR* incur fewer block accesses than ZM does. However, they still need to access inner nodes. Also, due to overlapping node MBRs, the R-trees may need to access multiple nodes at each tree level.

We further report the *average depth* of RSMI, i.e., the average number of sub-models invoked to reach a data block, which are 3.11, 3.26, 3.30, 3.04, and 4.01 on Uniform, Normal, Skewed, Tiger, and OSM, respectively. RSMI only needs 3 or 4 function invocations to locate a data block. KDB, HRR, and RR* have a depth of 3 on the first four data sets and 4 on OSM (excluding the data block level). They need to scan 3 or 4 nodes (maybe more for R-trees due to overlapping MBRs) to locate a data block, which is slower. Grid and ZM have fixed depths of 1 and 3. They suffer from the number of block accesses as discussed above.

Another observation concerning ZM vs. RSMI is that ZM suffers more in terms of block accesses and less in terms of response time. This is because ZM can quickly skip a data block accessed by testing whether the Z-value of the query point belongs to the Z-value range of the block. Its processing time per block is smaller than that of RSMI.

Fig. 7a shows the index size. The learned indices are the smallest because they only store the data blocks and the (small) sub-models. In contrast, Grid stores the data blocks and a cell table that maps grid cells to the corresponding data blocks; KDB, HRR, and RR* store the data blocks (leaf nodes) and the internal nodes. RSMI has a slightly larger index size than ZM due to its slightly larger number of sub-models. This is because RSMI is constructed adaptively based on the data distribution, while the number of sub-models in ZM is determined by the number of data points

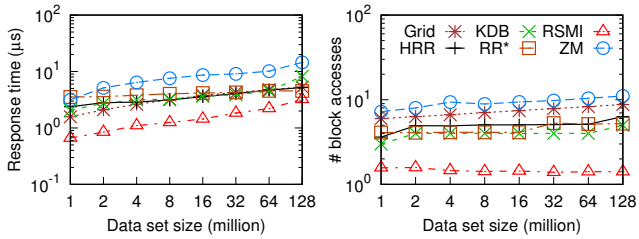


(a) Index size **(b)** Construction time

Figure 7: Index size and build time vs. data distribution

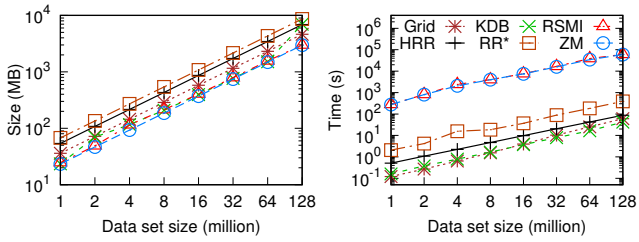
(cf. Section 6.1). For example, on OSM, RSMI has 10,445 sub-models, while ZM has 10,101 (2,303 MB vs. 2,244 MB, i.e., 2.6% larger). RR* is the largest because its nodes are less compact. HRR is also larger than RSMI because it uses two extra B-trees for its rank space mapping [37, 38].

The advantages of RSMI in query performance and index size come with a higher construction time (Fig. 7b), which is a common characteristic of learned indices. RSMI can be trained within 16 hours on OSM (over 100 million points), which is justified given its query performance. This training time assumes CPUs. If GPUs are used, we can reduce the training time by over 74%. For example, training RSMI on a Skewed data set with 128 million points takes 60,514 seconds on a CPU. This can be reduced to 15,698 seconds on an RTX 2080 Ti GPU. ZM is faster than RSMI at training, because RSMI needs to first sort and partition the data points *each* time it learns a sub-model. ZM only sorts the data points *once* for learning *all* sub-models. Among the traditional indices, HRR is bulk-loaded, which only takes a few rounds of sorting and data scans. This is faster than RR*, which is created by means of top-down insertions. Grid and KDB are the fastest due to their simple sorting-based construction.



(a) Query time **(b)** # block accesses

Figure 8: Point query vs. data set size



(a) Index size **(b)** Construction time

Figure 9: Index size and construction time vs. data set size

Varying the data set size. In Fig. 8, we vary the data set size using Skewed data. Results on the other data sets show similar patterns and are omitted due to the space limitation. The same applies in the remaining experiments.

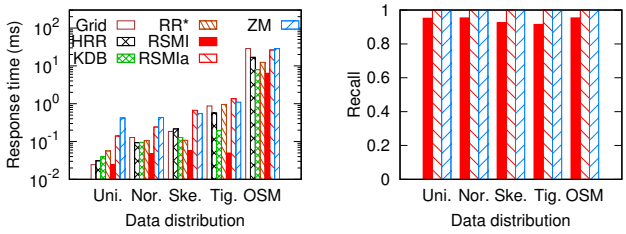
As expected, the query costs increase with the data set size. RSMI offers the lowest query costs, and the advantage is up to 5.8 times (1.3 μs vs. 7.6 μs for RSMI vs. ZM with 8 million points). We observe a slight drop in the number of block accesses for RSMI. When there are more points, RSMI may learn a structure with more levels, where the leaf models are more compact, yielding more accurate predictions and fewer block accesses. The query time still increases as there are more levels – the average depth increases from 2.49 to 4.46 for 1 to 128 million points, and the maximum depth is 10. These findings indicate that RSMI is scalable.

The index sizes and construction times also increase with the data set size, as shown in Fig. 9. RSMI is consistently small in size, while its construction time does not grow drastically (which can be further reduced using GPUs, as mentioned above). This allows it to scale to very large data sets.

6.2.3 *Window Queries*

We generate 1,000 window queries under each setting and report the average cost per query. Since the number of block accesses aligns with the query response time, we omit coverage of the number of block accesses. Also, index size and construction time are independent of the query type and are omitted hereafter. As learned indices may offer approximate window query answers (without false positives, cf. Section 4.2), we report their **recall** — the number of points returned over the cardinality of the ground truth answer.

We add one more technique called **RSMIa** to the comparison. It offers accurate query answers by performing an R-tree-like traversal by utilizing MBRs associated with the sub-models in RSMI, as described at the end of Section 4.2.



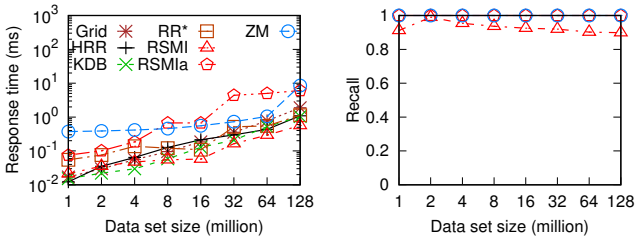
(a) Query time **(b)** Recall

Figure 10: Window query vs. data distribution

Varying the data distribution. Fig. 10 reports the window query costs across different data sets. As for point queries, RSMI is the fastest for window queries except for on Uniform data where Grid is slightly faster (0.025 ms vs. 0.024 ms). This is because Grid can easily locate the cells and hence the data blocks that overlap with the query window. On non-uniform data, Grid may have cells that only partially overlap with the query window while containing many blocks (and false positives) to be filtered for the query. On such data sets, RSMI outperforms the traditional indices by at least a factor of 1.33 (6.4 ms vs. 8.0 ms for RSMI vs. KDB on OSM) and up to 17 times (0.05 ms vs. 0.85 ms for RSMI vs. Grid on Tiger). RSMI also outperforms ZM by 4.4 times on OSM (6.4 ms vs. 28.5 ms) and by over an order of magnitude on the other four data sets. Meanwhile, RSMIa outperforms ZM on Uniform, Normal, and OSM, and its query performance is comparable to those of the R-tree indices on the real data sets Tiger and OSM. These findings show the applicability of RSMIa when accurate query answers are needed.

Next, we consider the recall of the learned indices. RSMIa has 100% recall as it uses MBRs for query processing. ZM is

more accurate than RSMI. It uses Z-curves, where the bottom left and top right corners of a query window bound the search region better than all four corners of the query window in RSMI, which uses Hibert-curves. However, RSMI has much lower query costs. It also has a consistently high recall of over 91.4% and up to 95.4% on Normal data (Fig. 10b).

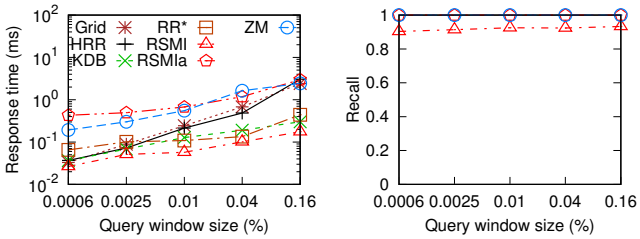


(a) Query time **(b)** Recall

Figure 11: Window query vs. data set size

Varying the data set size. When the data set size varies (Fig. 11a), RSMI is the fastest except for on data sets with fewer than 4 million points, where KDB is slightly faster. KDB creates non-overlapping data partitions, which benefits query performance. However, as the data set grows, the partitions become long and thin due to the block size limit (especially on Skewed). This leads to many tree nodes overlapping with queries and high query costs. RSMIa is faster than ZM when the data set size is below 8 million or exceeds 128 million. It queries 128 million points in just 6.1 ms.

Fig. 11b shows the recall of RSMI, which is consistently high and is over 89.8% for 128 million points. The recall drops slightly as the data set size increases, since it is more difficult to train an accurate prediction model on more points.

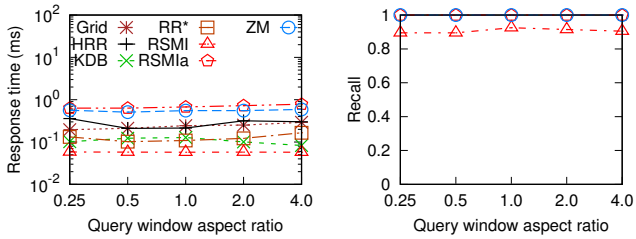


(a) Query time **(b)** Recall

Figure 12: Window query vs. query window size

Varying the query window size. We vary the query window size from 0.0006% to 0.16% of the data space size. As Fig. 12 shows, the query times grow with the query window size since more points are queried. The relative performance in both the query time and the recall among the indices is similar to that observed above. RSMI offers highly accurate answers (over 90.3%) and lowest query times, thus showing robustness for window queries in varying settings.

Varying the query window aspect ratio. We further vary the query window aspect ratio from 0.25 to 4.0. Fig. 13 shows that the aspect ratio is less impactful than the query window size. We conjecture that this is because the query costs are averaged over 1,000 queries that are positioned following the data distribution. Every set of 1,000 queries of a given aspect ratio may cover a similar set of data points overall, and hence has a similar average query cost. RSMI again outperforms all competitors and is at least 1.4 times faster (0.058 ms vs. 0.083 ms for RSMI vs. KDB when the aspect ratio is 4) than the other structures, and its recall exceeds 89.4%.

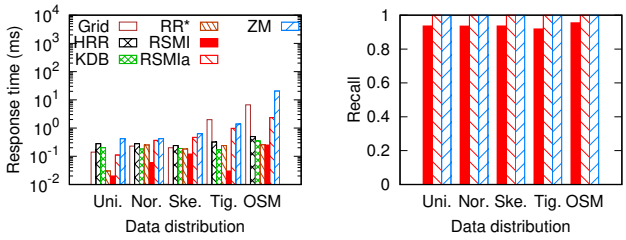


(a) Query time **(b)** Recall

Figure 13: Window query vs. query window aspect ratio

6.2.4 *KNN Queries*

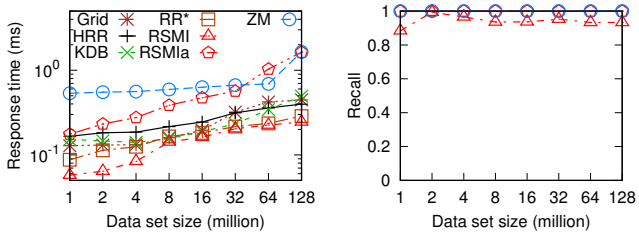
We generate 1,000 kNN queries under each setting and report the average query time and recall. Here, the recall refers to the number of true kNN points returned divided by k. This is the same as the precision. ZM does not come with a kNN algorithm, so we use our kNN algorithm for it. For the other indices, we use the best-first algorithm [40].



(a) Query time **(b)** Recall

Figure 14: KNN query vs. data distribution

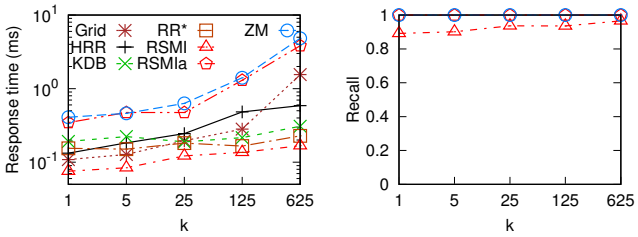
Varying the data distribution. Fig. 14 shows that RSMI is also the fastest for kNN queries. It outperforms ZM by up to 46 times (0.03 ms vs. 1.38 ms on Tiger). This is because both techniques use window queries for kNN queries, where RSMI is much faster. RSMI also outperforms the other indices. For Grid, the kNNs may spread in multiple cells which makes it uncompetitive. The other traditional indices require tree traversals and accesses to possibly many tree nodes. In terms of recall, RSMI is again very close to ZM and is over 91.8%. This shows the applicability of RSMI to kNN queries.



(a) Query time **(b)** Recall

Figure 15: KNN query vs. data set size

Varying the data set size. In Fig. 15, we vary the data set size. RSMI again yields high recall and the fastest query time, while RSMIa produces accurate answers and is faster than ZM (except when $n = 64$ million). The query times grow with the data set size. When $n = 128$ million, RSMI is over an order of magnitude faster than ZM. The recall of RSMI also decreases slightly with the data set size but it stays above 88.3%. This is in line with the observations for window queries where the data set size is varied (cf. Fig. 11).



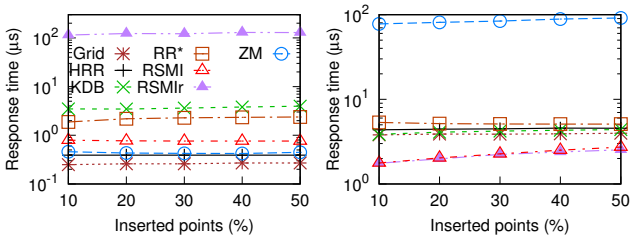
(a) Query time **(b)** Recall

Figure 16: KNN query vs. k

Varying k. In Fig. 16, we vary the query parameter k from 1 to 625. We see that the query costs increase with k, which is expected as more blocks and data points are examined. The high efficiency and recall (89.1% to 96.5%) of RSMI indicate that it scales to large k values.

6.2.5 *Update Handling*

We further examine the impact of data updates. ZM does not come with update algorithms, so we adapt ours for it. We initialize the indices with the default data set, insert (or delete) 10%n to 50%n data points, and query the updated indices with 1,000 queries. We report the average response time per insertion and the average response time per query. We also studied the impact of deletions but omit those due to space constraints. We note however, that they replicate the performance figures of insertions.



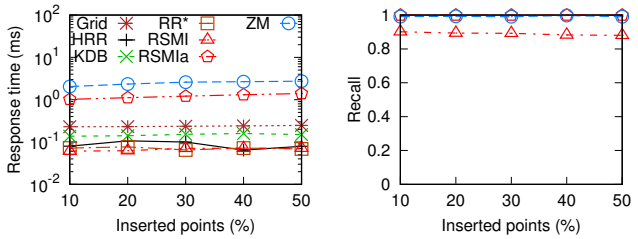
(a) Insertion time **(b)** Point query time

Figure 17: Insertion and point queries after insertions

Insertion. Fig. 17a shows slowly increasing insertion times with more insertions, as the tree index height has not increased till 50%n insertions, while the learned indices keep appending new points to the end of the predicted blocks. Grid adds a new point p to the last block in the cell enclosing p, which is the fastest. RSMI insertions cost more than those of ZM, since it has more sub-models than ZM, and it takes more time to reach a data block for the insertion.

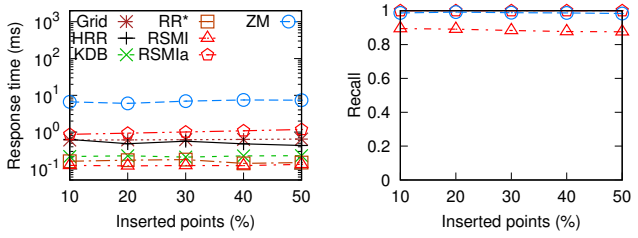
Fig. 17b shows that insertions cause the point query times to increase. There are more points to query, and the indices become less optimal. The learned indices ZM and RSMI are impacted the most as they have more blocks to scan with insertions. RSMI still yields the best query performance after 50%n insertions, i.e., 2.7 μs (RSMI) vs. 3.9 μs (Grid).

We further compare with an RSMI variant named **RSMIr**, which rebuilds the sub-models that exceeded the partition threshold after every 10%n insertions. RSMIr has an amortized insertion time of less than 130 μs, which confirms the feasibility of periodic rebuilds. It also improves the point query performance, especially when there are more insertions. A similar pattern is observed on window and kNN queries. We omit its curve in those figures for succinctness.



(a) Query time **(b)** Recall

Figure 18: Window queries after insertions



(a) Query time **(b)** Recall

Figure 19: KNN queries after insertions

For window queries (Fig. 18a), RR* keeps adjusting MBRs to reduce overlaps, while HRR checks the newly created blocks with tree traversals. They now perform similarly to RSMI. For RSMI, the created blocks are linked after blocks with the same predicted location. When the first block does not contain a query window corner point, more blocks need to be checked, which increase the query time. For kNN queries (Fig. 19a), as there are more points, the size of the initial search region of RSMI decreases. This helps RSMI retain the fastest query time. The recall of RSMI for both window and kNN queries consistently exceeds 87.5% (Fig. 18b and Fig. 19b). These results indicate the effectiveness of the update algorithm at maintaining the performance of RSMI.

7. CONCLUSIONS

Exploiting recent advances in machine learning, we propose algorithms to learn indices for spatial data. We order the data points using a rank space based technique. This ordering simplifies the indexing functions to be learned that map spatial coordinates (search keys) to disk block IDs (location). To scale to large data sets, we propose a recursive strategy that partitions a large data set and learns indices for each partition. We also provide algorithms based on the learned indices to compute point, window, and kNN queries, as well as update algorithms. Experimental results with both real and synthetic data show that the proposed learned indices and query algorithms are highly effective and efficient. Query processing using our indices is more than an order of magnitude faster than R-trees and a recent baseline learned index, while our window and kNN query results are highly accurate, i.e., over 87% across a variety of settings.

Our learned indices may be applied to spatial objects with non-zero extent using *query expansion* [44, 48], although this impacts query accuracy and efficiency. We leave in-depth studies of learned indices for spatial objects with non-zero extent for future work. Further, it is important to attempt to establish theoretical performance bounds for the learned spatial indices including the height, query accuracy, and curve value gaps, and to improve the efficiency for highly dynamic data updates, e.g., in moving object databases.

8. REFERENCES

[1] L. Arge, M. D. Berg, H. Haverkort, and K. Yi. The Priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Transactions on Algorithms*, 4(1):9:1–9:30, 2008.

[2] R. Bayer. The universal B-tree for multidimensional indexing: General concepts. In *International Conference on Worldwide Computing and Its Applications*, pages 198–209, 1997.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] N. Beckmann and B. Seeger. A revised R*-tree in comparison with related index structures. In *SIGMOD*, pages 799–812, 2009.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[6] S. Berchtold, C. Böhm, and H.-P. Kriegel. The Pyramid-technique: Towards breaking the curse of dimensionality. In *SIGMOD*, pages 142–153, 1998.

[7] A. Davitkova, E. Milchevski, and S. Michel. The ML-Index: A multidimensional, learned index for point, range, and nearest-neighbor queries. In *EDBT*, pages 407–410, 2020.

[8] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-server paradise. In *VLDB*, pages 558–569, 1994.

[9] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. Lomet, and T. Kraska. ALEX: An updatable adaptive learned index. In *SIGMOD*, pages 969–984, 2020.

[10] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *PODS*, pages 247–252, 1989.

[11] P. Ferragina and G. Vinciguerra. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, 13(8):1162–1175, 2020.

[12] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[13] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[14] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. FITing-Tree: A data-aware index structure. In *SIGMOD*, pages 1189–1206, 2019.

[15] Y. J. García R, M. A. López, and S. T. Leutenegger. A greedy algorithm for bulk loading R-trees. In *ACM GIS*, pages 163–164, 1998.

[16] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[17] A. Hadian and T. Heinis. Considerations for handling updates in learned index structures. In *International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 3:1–3:4, 2019.

[18] A. Hadian and T. Heinis. Interpolation-friendly B-trees: Bridging the gap between algorithmic and learned indexes. In *EDBT*, pages 710–713, 2019.

[19] A. Henrich, H.-W. Six, and P. Widmayer. The LSD tree: Spatial access to multidimensional point and non-point objects. In *VLDB*, pages 45–54, 1989.

[20] H. 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*, 30(2):364–397, 2005.

[21] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B^+-tree based indexing of moving objects. In *VLDB*, pages 768–779, 2004.

[22] D. V. Kalashnikov, S. Prabhakar, and S. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distributed and Parallel Databases*, 15(2):117–135, 2004.

[23] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *VLDB*, pages 500–509, 1994.

[24] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. RadixSpline: A single-pass learned index. In *International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 5:1–5:5, 2020.

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

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

[27] S. T. Leutenegger, J. M. Edgington, and M. A. López. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*, pages 497–506, 1997.

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

[29] Z. Li, T. N. Chan, M. L. Yiu, and C. S. Jensen. PolyFit: Polynomial-based indexing approach for fast approximate range aggregate queries. *CoRR*, abs/2003.08031, 2020.

[30] V. Markl. Mistral: Processing relational queries using a multidimensional access technique. In *Ph.D. Thesis, Technische Universität München*. 1999.

[31] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *NeurIPS 2019 Workshop on Machine Learning for Systems*, 2019.

[32] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *SIGMOD*, pages 985–1000, 2020.

[33] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[34] OpenStreetMap US Northeast data dump. `https://download.geofabrik.de/`, 2018. Accessed: 2020-06-10.

[35] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS*, pages 181–190, 1984.

[36] PyTorch. `https://pytorch.org`, 2016. Accessed: 2020-06-10.

[37] J. Qi, Y. Tao, Y. Chang, and R. Zhang. Theoretically optimal and empirically efficient R-trees with strong

parallelizability. *PVLDB*, 11(5):621–634, 2018.

[38] J. Qi, Y. Tao, Y. Chang, and R. Zhang. Packing R-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability. *ACM Transactions on Database Systems*, accepted to appear in 2020.

[39] J. T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *SIGMOD*, pages 10–18, 1981.

[40] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[41] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *SIGMOD*, pages 17–31, 1985.

[42] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[43] N. F. Setiawan, B. I. P. Rubinstein, and R. Borovica-Gajic. Function interpolation for learned index structures. In *Australasian Database Conference*, pages 68–80, 2020.

[44] E. Stefanakis, T. Theodoridis, T. K. Sellis, and Y.-C. Lee. Point representation of spatial objects and query window extension: A new technique for spatial access methods. *International Journal of Geographical Information Science*, 11(6):529–554, 1997.

[45] TIGER/Line Shapefiles. `https://www.census.gov/geo/maps-data/data/tiger-line.html`, 2006. Accessed: 2020-06-10.

[46] H. Wang, X. Fu, J. Xu, and H. Lu. Learned index for spatial queries. In *MDM*, pages 569–574, 2019.

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

[48] R. Zhang, J. Qi, M. Stradling, and J. Huang. Towards a painless index for spatial objects. *ACM Transactions on Database Systems*, 39(3):19:1–19:42, 2014.