Survey on Learned Index for Spatial Data

DENG Nan
20881163

MA Xiaolei
20887791

ZHANG Hexiao
20932780

ZHU Yuzhang
20881929

SHU Yi
20928806

1 INTRODUCTION

Indexing is an essential tool for high-performance database queries, with commonly used index structures including B-tree, Hash, and Bitmaps. In essence, an index can be considered as a model, such as a B-Tree-Index, which maps a key to the position of a record within a sorted array[5]. Similarly, machine learning is to learn the appropriate mapping functions between two sets. Researchers have proposed various methods to replace traditional index structures with machine learning approaches.

Learn-to-Index (LTI) is a method for constructing data indexing structures with machine learning algorithms to learn efficient mappings between the search keys and data locations. A learned index can search for a key quickly while requiring less memory space compared with traditional index structures. In our project, we will focus on muti-dimensional data indexing using LTI.

A practical application of learned indexes in spatial data is in ride-sharing platforms like Uber or Lyft, where managing location data is crucial for tasks such as ride-matching and route calculation. Learned indexes can improve efficiency by providing faster and more accurate query responses, reducing computational overhead, and enhancing user experience for drivers and passengers.

Another example is in real estate platforms, which require efficient handling of property location data for tasks like searching available listings within a specific area or distance from a point of interest. Learned indexes can optimize search performance and provide more accurate results, improving the platform's usability and effectiveness for users seeking properties.

In this report, we primarily focus on three methods for constructing spatial indexes using machine learning: RSMI[8], LISA[6], and Flood[7]. The structure of the report is as follows: In Chapter 2, we provide an overview of the prior work related to these three methods. In Chapter 3, we discuss each method in detail. In Chapter 4, we analyze the experimental techniques and results presented in the respective papers. Finally, we explore future research directions and potential issues in this domain.

2 RELATED WORK

In recent work[5], a hierarchy of machine learning models is utilized as a Recursive Model Index (RMI) for indexing 1-dimensional data. The index is modeled as a function that maps search keys to storage addresses containing the corresponding data. Given a learned function, an object can be located through a single function invocation. This method assumes data is stored and maintained in an in-memory dense array. However, RMI is not suitable for spatial data.

For spatial data and queries, traditional indexing models such as R-trees [4], kd-trees[2], and quadtrees [3] are commonly used. Query processing in these models requires tree traversal, which can lead to accessing numerous tree nodes and decreasing performance, especially when the index is stored externally.

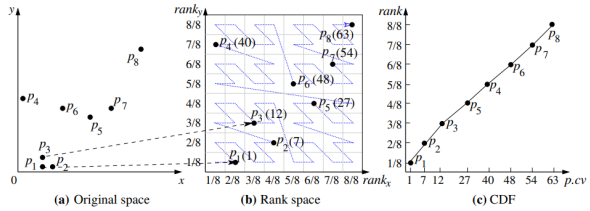

Figure 3: Point ordering and our indexing model

Figure 1: Point ordering for RSMI indexing model

A recent Z-order Model (ZM) [11] combines the Z-order space-filling curve [9] with RMI to index spatial data. However, this approach does not support k-nearest neighbor (KNN) queries or data updates, and it requires checking many irrelevant keys for range queries

Despite the previous work on solving the indexing problems, there are some outstanding research papers, and we picked three of them to learn the details.

3 RSMI: EFFECTIVELY LEARNING SPATIAL INDICES

This paper[8] focuses on point data. We want to train a model M that maps the coordinates of the input samples to the block where the samples are located, i.e.

$$p.blk \approx M\,(p.cord) \tag{1}$$

Points are ordered by their space-filling curve values (eg. Z-values) and their locations are determined by their ranks. Assuming that each block contains B records,

$$p.blk = [p.rank/B] \tag{2}$$

3.1 Ordering points

Prior to training the model, we need to preprocess the input data. Points are mapped to a rank space, where each row and column contains exactly one point. As shown in the figure, after being mapped, the coordinate of a point along a given axis represents the ranking of its original coordinate value within the dataset. This helps to create a more uniform distribution of the data, which can lead to improved accuracy when using deep learning models.

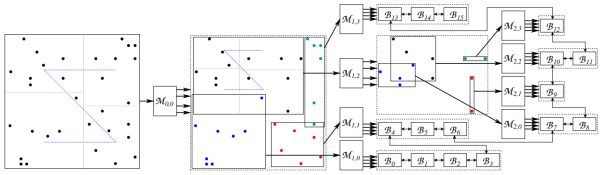

Figure 4: RSMI index structure ($N = 8$ and $B = 2$)

Figure 2: RSMI index structure

3.2 Model Learning

After preprocessing, an FNN M is trained with L2-loss. We record the maximum errors for the cases where M ($p.cord$) is smaller than $p.blk$ and M ($p.cord$) exceeds $p.blk$, which will be used in query algorithms.

3.3 RSMI: Scale to Large Datasets

A single model may not be accurate enough. Suppose the error is acceptable when the number of points does not exceed N, we recursively divide the space into multiple subspaces until the number of samples in each subspace is no more than N. This strategy is called the recursive spatial model index (RSMI).

Initially, we distribute the data into equal-sized partitions. Then, we use a Space Filling Curve (SFC) to assign Ids to partitions and learn the partition Ids using a model $M_{0,0}$. We rearrange the data based on the prediction of $M_{0,0}$. This step makes it unnecessary to consider the error of the non-leaf node model in the query. Partitioning is done recursively until each partition can be handled with a simple model according to the methods given in previous sections.

3.4 Insertion and Deletion

Deletion is handled with lazy tags and in-page rearrangement, which flags the deleted items without immediately removing them and reorganize the remaining data in place.

Inserted items can be recorded into overflow pages, ensuring that the blocks to which existing records belong are not affected.

It's important to note that indexes cannot be updated online in this system. As a result, periodic rebuilding of indexes is necessary to keep them up to date and maintain optimal performance.

3.5 Query Algorithms

The paper proposes point query, window query, and KNN query algorithms for point data. Query algorithms do not have too many complex details. It's worthwhile to note that window queries and KNN queries are not exact because we have no control over the error bounds of the model for unseen data. Experiments show that recalls are over 87% across a variety of settings.

4 LISA: LEARNED INDEX STRUCTURE FOR SPATIAL DATA

LISA [6] proposed a learned index, concept of which is to use the machine learning method to create a searchable data layout for any spatial dataset. The novel part of this article is that is choose a

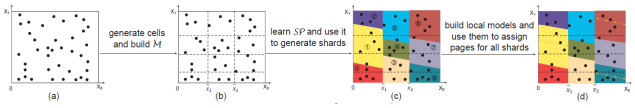

Figure 3: LISA framework

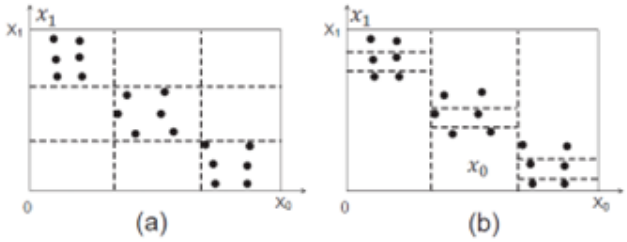

Figure 4: Original and modified partitioning strategies

monotonic function to ensure the order of the mapped value got from the mapping function. LISA framework is shown in Figure 3.

4.1 Generating Grid Cells

LISA first redefines a presentation of the gird cell to provide a more reasonable partition of the data distribution. The space V is partitioned into T_i parts according to the x_i-axis. They use $\Theta_i = \left[\theta_1^{(i)}, \ldots, \theta_{T_i}^{(i)}\right]$ to denote the points on the border generated by the operation mentioned above. In this case, they could use a series of cells to represent the space V. Specifically, they use a 1-dimensional monotonic regression model for partition. Compared to the previous partition function, this method could make an even distribution of the initial dataset in Figure 4. And the partition would keep separate the parts got from the process resulted by the previous axis partition until the space V is partitioned by all the axes.

4.2 Mapping Function M

LISA implement a partially monotonic function to map the data from $\mathbb{R}^d$ to $\mathbb{R}$ based on the distribution the borders of cells. The definition of mapping function $\mathcal{M}$ is provided below. Suppose $x = (x_0, \ldots, x_{d-1})$ and $x \in C_i = \left[\theta_{j_0}^{(i)}, \theta_{j_0+1}^{(i)}\right) \times \ldots \times \left[\theta_{j_{d-1}}^{(d-1)}, x_{d-1}\right)$ then $\mathcal{M}(x) = i + \frac{\mu(H_i)}{\mu(C_i)}$, where $H_i = \left[\theta_{j_0}^{(i)}, \theta_{j_0+1}^{(i)}\right) \times \ldots \times \left[\theta_{j_{d-1}}^{(d-1)}, x_{d-1}\right)$. In the definition above, μ is the Lebesgue measure [2] on $\mathbb{R}^d$.

4.3 Shard Prediction Function SP

LISA defined this monotonic function SP for allocating shard id to every mapped value, which is consist of a series of piecewise function linear function. Compared to RMI [3] model, this function targeted to allocate the data layout in the disk pages rather than using regression model to get position, which would lead to impossible implementation with zero loss particularly when the keys are stored in disk pages instead of a dense array. SP is a staged model, and each stage contains multiple small regression models. For each small model, a monotonic regression model is constructed from the

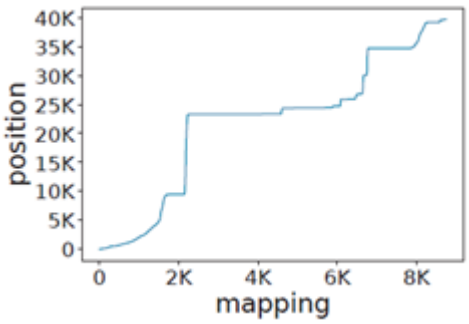

Figure 5: Example of keys' mapped values vs. indices

distribution of the number of mapped values. In the Figure 5, the relationship between the mapped values of keys and their indexes in the list is shown, which is difficult for neural network to get such characteristics. According to the distribution of the output of Shard Prediction Function, they would gather a series of points which are close in terms of distance into one shard. In this case, all the mapped value would be roughly sorted in the whole dataset.

4.4 Local Models for Shards

Based on the algorithm above, LISA could get the corresponding keys that allocated into every shard and the mapped values using Mapping Function $\mathcal{M}$ and Shard Prediction Function SP. But, once the shard overflows, it could not hold more keys. The local model $\mathcal{L}_i$ is used to solve this problem, specifically deciding how to partition them into more than one page. The local model $\mathcal{L}_i$ provides two different search functions $\mathcal{L}_i.lbound$ and $\mathcal{L}_i.ubound$ such that

$$\mathcal{L}_i.lbound\,(m) = j\,,\ where\ \mathcal{L}_i.PM\,[j-1] < m \leq \mathcal{L}_i.PM\,[j] \quad (3)$$

$$\mathcal{L}_i.ubound\,(m) = j\,,\ where\ \mathcal{L}_i.PM\,[j-1] < m \leq \mathcal{L}_i.PM\,[j] \quad (4)$$

PM is the sequence of sorted mapped values to partition the keys in every shard. These functions help us quickly find the location of a key for a given map value in our local model.

4.5 Range Query

LISA first gets the cells that intersect with query rectangle and then separate query rectangle into a series of smaller query rectangles, which intersects one and only one cell to ensure accuracy. In order to be more efficient, the consecutive small rectangles would be merged. Then mapping function are used to calculate mapped values of all query rectangle's vertices, after getting the results of shard prediction function, it is simple to get the corresponding search results combined with local model.

4.6 KNN Query

LISA uses a combination of lattice regression model [4] and the range query mentioned above to implement the KNN query, rather than using the traditional pruning strategies for KNN query in R-tree. In its implementation, KNN query could be converted to a series of range queries. The reason for choosing lattice regression model is that getting the distance bounds is still a regression model and especially, they did not have other features to be considered. Also, time consuming cost is another consideration. After lattice regression model processing, they could get the distance bound as $\delta_0 = \mathcal{LR}(q_{knn})$. Then they could query the keys in the corresponding range and get the corresponding keys. Depending on the number of the remaining keys, they would adjust δ_0 to produce another range query until fulfilling the K nearest keys.

4.7 Insertion

Based on the LISA structure, they could calculate the $i = \lfloor S\,P(M(k)) \rfloor$ to get the exact shard k to fall in. Depending on the existence of page P that contains k, LISA would execute different strategy to implement the insertion operation.

4.8 Deletion

Deletion operation is similar to the insertion operation. $M(k)$ and $SP(M(k))$ is needed. But the local model is also needed in the operation to check whether there is a page that contains k.

5 LEARNING MULTI-DIMENSIONAL INDEXES

This paper[7] proposes Flood, the first learned multi-dimensional in-memory index, which is read-optimized and can adapt itself to a particular dataset and workload by jointly optimizing the index structure and data storage layout using machine learning techniques.

5.1 Data Layout

The Flood index considers a general case of data with d-dimensions. This type of data does not have a natural sort order. In order to create an index, the first step Flood does is to impose an ordering over the data. To do so, it first ranks the d attributes using ML techniques which will be described later. Then, it overlays a (d-1)-dimensional grid on the data with the first (d-1) attributes, where the i-th dimension is divided into c_i equally spaced columns in its value range. With this division, every point in the dataset will be mapped to exactly one cell in the grid. The cell's coordinate can be given by:

$$cell\,(p) = (\left\lfloor \frac{p_1 - m_1}{r_1} c_1 \right\rfloor, \ldots, \left\lfloor \frac{p_{d-1} - m_{d-1}}{r_{d-1}} c_{d-1} \right\rfloor) \quad (5)$$

Where m_i is the minimum value of that dimension and r_i the range. In this grid layout, only $(d-1)$ dimensions are used and the last unused dimension will be referred to as the sort dimension and used to sort the points inside each cell. To conclude this layout, points are ordered using a depth-first traversal of the cells along the dimension ordering, whereas points in the same cell are ordered with the sort dimension.

5.2 Query Operation

The Flood learned index focuses on handling AND queries of given ranges. The process of performing a query in the Flood index is described as follows.

1. Projection: Find the cells in the grid that intersects with the query's hyper-rectangle using the above formula. Points satisfying the query can only exist in these cells.

2. Refinement: If the query involves the sort dimension of the data layout, we can further refine the range by filtering out points that do not satisfy the sort dimension range using binary search.

3. Scan: Finally, to get the exact results of the query, plainly check every point in the refined range to judge whether it satisfies the requirements.

From the query process described above, we can see that the ranking of the attributes and choice of sort dimension in data layout is essential to the efficiency of queries. Therefore, the Flood learned index uses ML techniques to provide an optimized layout based on the data distribution.

5.3 Building the Cost Model

To optimize the time of queries, we must first know how time is spent on different stages of the query. The Flood learned index models the query time of dataset D with data layout L as a sum of three parts given by:

$$Time\,(D, q, L) = w_p N_c + w_r N_c + w_s N_s \tag{6}$$

Where N_c is the number of cells and N_s is the number of scanned points, and w_p, w_r, w_s are averaged time to perform one of the operations in each stage and depend on the dataset and query.

Since the weights vary based on data, Flood trains a random forest regression model once to predict these weights on a given dataset. In order to train this model, Flood uses an arbitrary dataset and query workload, which can be synthetic. Flood generates random layouts by randomly selecting an ordering of the d dimensions, then randomly selecting the number of columns in the grid dimensions to achieve a random target number of total cells. Flood then runs the query workload on each layout, and measures the weights w and aforementioned statistics for each query. Each query for each random layout will produce a single training example. It is observed that 10 such training examples were sufficient to train a decent model for predicting the weights. In application, this random forest model is trained once on a given set of hardware to calibrate the weights.

5.4 Optimizing the Grid

After obtaining the time cost function using the aforementioned random forest regression, we can then apply ML methods to predict an optimal data layout for any given dataset and query distribution. This is done with the following steps:

1. Sample the dataset and query workload, then flatten the data sample and workload sample using RMIs trained on each dimension.

2. Try every selection of the d dimensions to be the sort dimension. Order the remaining $d - 1$ dimensions that form the grid by the average selectivity on that dimension across all queries in the workload. This gives us an ordering.

3. For each of these d possible orderings, run a gradient descent search algorithm to find the optimal number of columns for the $d - 1$ grid dimensions. The objective function is the query time given by the random forest regression model. For each call to the cost model, Flood computes the statistics $N = \{N_c, N_s\}$ and the input features of the weight models using the data sample instead of the full dataset D.

4. Select the layout with the lowest objective function cost amongst the d layouts.

The gradient descent performed in this step is efficient because the statistics are measured using a sample of D or predicted with the weights provided by the random forest model.

5.5 Learning from the Data

In addition to selecting the dimension ranking, Flood adopts two ways to optimize the layout according to the data, which are flattening for building cells with unequal range and piecewise linear model (PLM) for faster refinement step.

The flattening technique is to average the number of points in each cell so that query time can be more stable. This is implemented by using RMIs on each dimension to estimate its cumulative distribution function (CDF). This way, a point with value v in the kth dimension will be placed into column $[CDF(v)n]$.

To achieve faster refinement step than using binary search, Flood also builds a CDF model on the sort dimension to estimate the range before performing local search to calibrate. In order to achieve a controlled error bound, in the sort dimension Flood uses PLM where a sorted list of values V is partitioned into slices, each of which is modelled by a linear segment. The error bound in a slice is controlled by starting a new slice when the average error is about to exceed the required value.

6 COMPARISON FOR THREE METHODS

In comparing the three methods for learned indexes for spatial datasets, we can see that each method has its unique characteristics and trade-offs. RSMI and LISA both support updates, but RSMI requires periodic rebuilding, while LISA is more flexible in this regard. On the other hand, Flood does not support updates at all.

In terms of query support, LISA outperforms the other two methods by offering full support for point, window, and KNN queries. RSMI has limitations in the accuracy of window and KNN queries, whereas Flood does not support KNN queries altogether.

When it comes to machine learning methods and learning objectives, RSMI employs Deep Neural Networks to fit the Cumulative Distribution Function (CDF) to map Space-Filling Curve (SFC) values to ranks. LISA, on the other hand, uses piecewise linear models with the primary goal of predicting the rank from the output of the mapping function M. RSMI and LISA perform data rearrangement using the prediction results of the trained model. Flood differs from both RSMI and LISA by utilizing tree-based models to directly learn an optimized data layout, requiring samples of queries for its learning objective.

Overall, each of these methods has distinct strengths and weaknesses in terms of updates, query support, machine learning techniques, and learning objectives. Depending on the specific requirements of a spatial dataset application, one method may be more suitable than the others.

7 EVALUATION AND RESULTS

7.1 RSMI

Competitors. The baseline competitors included Z-order model (ZM), Grid File (Grid), K-D-B-tree (KDB), Rank space based R-tree (HRR), and Revised R*-tree (RR*). Datasets. Experiments were run

on two real data sets, Tiger and OSM, and three synthetic data sets, Uniform, Normal, and Skewed, with up to 128 million points.

Metrics. The performance metrics used were average response time and number of block accesses per query, with the latter serving as an indicator for external memory-based implementation performance. The recall was also calculated for kNN and window queries.

Results. The experiments measured the optimal RSMI Partition Threshold N and reported results on point, window, and kNN query processing, as well as update handling. The optimal value for N was found to be 10,000. RSMI outperformed other indices for all types of queries. It also provided consistently high recall over 88% although its range query algorithms are not exact. It had a longer construction time, which is a common characteristic of learned indices. Overall, the experiments provided a comprehensive analysis of the effectiveness and efficiency of RSMI in improving spatial data indexing and query processing.

7.2 LISA

Competitors. This paper compares LISA with Baseline (described before proposing LISA) and four existing methods: R-tree, R*-tree, KD-tree and ZM.

Datasets. Two real-world and ten synthetic datasets are used to evaluate indexes and they are deliberately readjusted and split into the dataset I, E and D.

The evaluation is conducted on three configurations: Init, AI, and AD, where Init involves building all five methods using the dataset I, AI involves inserting keys in E into the data, and AD involves deleting keys in D from the data.

Metrics. Four metrics are used to evaluate the performance of the methods:

1. Size: Indicates the disk storage space that a method consumes.
2. IO: The average number of pages to be loaded for a range/kNN query.
3. IO Ratio: The ratio of a method's IO cost to R-tree's IO cost.
4. Size Ratio: The ratio of a method's size to R-tree's size.

Results. The experiments conducted in the paper demonstrate that LISA outperforms other methods in terms of storage consumption and I/O cost for range queries, especially for large datasets and when there are no data updates. The advantage of LISA on the metric of IO may not be as clear if the number of inserted keys is significantly larger than the cardinality of the initial dataset. In such cases, re-generating the cells and rebuilding the mapping function M and the shard prediction function SP can help improve the performance. The average CPU time of LISA on KNN queries is much smaller than that of the alternatives. Overall, the results suggest that LISA has great potential in spatial data processing, especially for large databases with a large number of keys, where it can offer significant advantages in terms of storage consumption, I/O cost, and response time.

7.3 Flood

Competitors. This paper compares Flood with eight other approaches: Full Scan, Clustered Single-Dimensional Index, Grid Files, Z-Order Index, UB-tree, Hyperoctree, k-d tree and R*-Tree.

	sales	Tpc-h	osm	perfmon
records	30M	300M	105M	230M
queries	1000	700	1000	800
dimensions	6	7	6	6
Size (GB)	1.44	16.8	5.04	

Table 1: Datasets of Flood

Datasets. Three real-world and one synthetic datasets are used to evaluate indexes, summarized in Table x. Queries are either real workloads or synthesized for each dataset, and include a mix of range filters and equality filters.

Metrics. The main evaluation metric used is the total query time, which is the sum of the time spent on index lookups and scanning. Index creation time, as well as the number of disk seeks and disk blocks accessed during query processing are also reported. Other metrics include the space overhead of the indexes and the accuracy of the CDF models used in the per-cell models.

Results. Flood's performance was particularly strong on datasets with high dimensionality and high selectivity queries. For example, on a dataset of taxi trips, Flood achieved up to 5.5x faster query times than the best baseline index. On a dataset of OpenStreetMap points, Flood achieved up to 39x faster query times than the best baseline index.The paper also evaluated the robustness of their cost model and found that it did not need to be retrained for each dataset, as the resulting layouts and query times were similar across datasets. It also found that sampling the data and query workload during index creation could significantly reduce the learning time of the system without sacrificing performance. Finally, the paper evaluated the performance of the per-cell models used to accelerate the location of physical indexes along the sort dimension. It found that the piecewise-linear model (PLM) performed comparably to a learned B-tree and beat binary search by up to 4x on real and synthetic datasets. The PLM only required a single tuning parameter to strike a balance between accuracy and size.

8 POTENTIAL ISSUES

We will discuss some of the key future directions and open problems [1, 10] related to learned indexes for spatial data.

8.1 Efficiently Supporting Updates

Traditional index structures like B-trees and R-trees are designed to accommodate dynamic changes in the dataset, whereas learned indexes are mostly static, making it difficult to maintain and update the model as data changes. Developing efficient mechanisms for updating the learned index without retraining the entire model will be crucial for their practical adoption.

8.2 Support for Other Spatial Operations and Data Structures

While some spatial operations like range queries have already been explored with learned indexes, support for other essential operations like spatial join and closest pairs is still an open problem.

Besides, existing studies have focused on point data. For data structures with non-zero extent, such as polygons, more research is needed.

8.3 Choosing the Right ML Models

One of the critical aspects of learned indexes is selecting the appropriate ML model that can provide a good balance between efficiency, accuracy, and space consumption. It is worth exploring more types of models like SVM or other deep learning based methods to determine the best-suited one for different spatial data scenarios.

8.4 Concurrency Support

As learned indexes become part of database systems, they must support concurrent access from multiple transactions. This involves handling read and write conflicts, as well as ensuring isolation and consistency guarantees provided by the database system.

8.5 Benchmarking Learned Multidimensional Indexes

Finally, to assess the performance and effectiveness of learned indexes for spatial data, it is necessary to develop comprehensive benchmarks that can help in comparing different models and approaches. These benchmarks should encompass a wide range of data distributions, query types, and performance metrics to facilitate a fair comparison and identify the most promising techniques in the field.

9 CONCLUSION

In this report, we delved into learned indexes for spatial data, examining RSMI, LISA, and Flood as prominent approaches. By analyzing their unique contributions and evaluating their performance, we have highlighted their potential in addressing spatial data management challenges. We also pointed out that there are some open problems to be tackled. By building upon the insights gained from the analysis of RSMI, LISA, and Flood, we can continue to push the boundaries of efficient and adaptive spatial data management solutions, paving the way for a new generation of data-driven applications.

REFERENCES

[1] Abdullah Al-Mamun, Hao Wu, and Walid G. Aref. 2020. A Tutorial on Learned Multi-Dimensional Indexes. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems* (Seattle, WA, USA) *(SIGSPATIAL '20)*. Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/3397536.3426358

[2] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (sep 1975), 509–517. https://doi.org/10.1145/361002.361007

[3] Raphael Finkel and Jon Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.* 4 (03 1974), 1–9. https://doi.org/10.1007/BF00288933

[4] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '84)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[5] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2017. The Case for Learned Index Structures. *CoRR* abs/1712.01208 (2017). arXiv:1712.01208 http://arxiv.org/abs/1712.01208

[6] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020).

[7] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 985–1000. https://doi.org/10.1145/3318464.3380579

[8] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2341–2354. https://doi.org/10.14778/3407790.3407829

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

[10] Yao Tian, Tingyun Yan, Xi Zhao, Kai Huang, and Xiaofang Zhou. 2022. A Learned Index for Exact Similarity Search in Metric Spaces. arXiv:2204.10028 [cs.DB]

[11] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. 569–574. https://doi.org/10.1109/MDM.2019.00121