

Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:

Haozhan, Shi

Title:

Learned Hashmap for Spatial Queries

Date:

2020

Persistent Link:

<https://hdl.handle.net/11343/291586>

Terms and Conditions:

Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.



Learned Hashmap for Spatial Queries

by

Haozhan Shi

Student number: 989644

Supervisor: Jianzhong Qi

A thesis submitted in partial fulfillment for the
degree of Master of Science (Computer Science)

75-point Master Research Project

Subject Code: COMP90070

in the

Faculty of Science

School of Computing and Information Systems

THE UNIVERSITY OF MELBOURNE

2020/11/10

THE UNIVERSITY OF MELBOURNE

Abstract

Faculty of Science
School of Computing and Information Systems

Master of Science (Computer Science)

by [Haozhan Shi](#)

Student number: 989644

Supervisor: Jianzhong Qi

Spatial indexes such as R-Tree are widely used for managing spatial objects data efficiently, which is influenced by the popular one-dimensional range index B-Tree. Research has suggested that applying machine learning techniques such as linear regression or a neural network can improve the performance of traditional data structures. However, most studies are focused on tuning recursive learned models or learning a different ordering of data items. Many of them cannot guarantee query accuracy as in the traditional methods. This study investigates a different approach to the learned spatial index, namely Learned Spatial Hashmap (LSPH), which combines the learned model and hashmap. It only requires values from one of the data dimensions to build. Results from experiments on both synthetic and real-world datasets show that our approach significantly reduces the query processing time and maintains 100% accuracy, which is more efficient than traditional spatial indexes and more robust than recently proposed learned spatial indexes.

Declaration of Authorship

I certify that:

- This thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university;
- and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- the thesis is 18,000 words in length (excluding text in images, table, bibliographies and appendices).

Signed:

Date: 2020/11/10

Acknowledgements

I would like to acknowledge the help and constructive advice provided by my research supervisor Dr.Jianzhong Qi during the design and development of this research project.

I would also like to thank my parents for their help and assistance throughout my study.

Contents

Abstract	i
Declaration of Authorship	ii
Acknowledgements	iii
List of Figures	vi
List of Tables	vii
Abbreviations	viii
1 Introduction	1
1.1 Motivation	2
1.2 Research Questions	3
1.3 Approach and Outcomes	3
1.4 Thesis Structure	5
2 Related Work	6
2.1 Spatial Queries	7
2.2 Spatial Indexing Techniques	8
2.2.1 Transformation	8
2.2.1.1 Space-filling Curves	9
Z-ordering Curve	9
Hilbert Curve	10
2.2.2 Overlapping Regions	10
2.2.3 Clipping	11
2.2.4 Multiple Layers	11
2.3 Traditional Spatial Data Indexes	12
2.3.1 Grid File	12
2.3.2 Multidimensional Linear Hashing	13
2.3.3 K-D-Tree	14
2.3.4 BSP-Tree	15
2.3.5 Quadtree	16
2.3.6 R-Tree	16
2.4 Learned Indexes for One-dimensional data	18

2.4.1	The Recursive Model Index	18
2.4.2	ALEX	20
2.4.3	The PGM-index	22
2.4.4	RadixSpline	23
2.4.5	FITing-Tree	25
2.5	Learned Spatial Indexes	26
2.5.1	Learned ZM index	26
2.5.2	Recursive spatial model index	26
2.5.3	Flood	29
2.5.4	Tsunami	30
2.5.5	qd-tree	30
2.5.6	LISA	32
2.5.7	ML-Index	33
2.5.8	PolyFit	34
2.6	Research Gaps	34
2.7	Summary	36
3	Proposed Indexing Model	39
3.1	Design Goal	40
3.2	The Learned Spatial Hashmap	41
3.2.1	Overview	41
3.2.2	Single Dimensional Index Building	42
3.2.3	Learning a Hashmap	44
3.2.4	Model Choice	46
3.3	Basic Operations	46
3.4	Query Processing	47
3.4.1	Point Query	48
3.4.2	Range Query	49
3.4.3	k Nearest Neighbours Query	50
3.5	Minimum Bounding Box Handling	52
3.6	Summary	53
4	Experiments	55
4.1	Experimental Setup	55
4.1.1	Datasets	55
4.1.2	Competitors	56
4.1.3	Evaluation Metrics	57
4.2	Performance Results	57
4.2.1	Range Query Performance	59
4.2.2	KNN Query Performance	60
4.2.3	Load Factor and Query Performance	61
4.3	Summary	62
5	Conclusions and Future Work	64
5.1	Limitations	65
5.2	Future Work	66
	Bibliography	68

List of Figures

2.1	Z-ordering Curve and Hilbert Curve	9
2.2	Grid File	12
2.3	K-D-Tree	14
2.4	BSP-Tree	15
2.5	Quadtree	16
2.6	R-Tree	17
2.7	RMI	18
2.8	ALEX	21
2.9	PGM-index	22
2.10	RadixSpline	24
2.11	RSMI	27
2.12	Flood	29
2.13	LISA	32
3.1	The structure of the Learned Spatial Hashmap(LSPH). The search flow in LSPH takes one value from the selected dimension, then the learned hash function predicts the index to the estimated location in the hashmap.	41
3.2	Difference between building data with x-values and y-values	43
3.3	Lookup process on the dataset. Using the trained linear regression model to calculate the lookup hash value, then scanning the lookup region base on the lookup hash value.	46
3.4	<i>Range Query</i> in LSPH. Calculate the maximum and minimum hash value to get the search region on the hashmap.	49
3.5	<i>kNN Query</i> in LSPH. First to calculate the query hash value, then expand to neighbour buckets.	50
3.6	Store the top-right and bottom-right corners of a MBB in the hashmap	52
4.1	Distribution of post , g2 and osm datasets	56
4.2	Point query performance comparison on runtime memory, build time, total query time and average query time for all three spatial indexes	58
4.3	Range Query performance on dataset post . The window size is corresponding to the query selectivity, which is between 0.0-100.0%. The -1 in ZM represents the return results are out of search range.	60

List of Tables

2.1	Comparison of Data Indexes	36
4.1	Datasets	55
4.2	Point Query Accuracy	59
4.3	Average KNN Query Time	60
4.4	Load factor's effect on the query performance in LSPH	61

Abbreviations

LSPH	The L earned S Patial H ashmap
RMI	The R ecursive M odel I ndex
ZM	The Learned Z -order M odel
MBB	M inimum B ounding B ox
CDF	C umulative D istribution F unction

Chapter 1

Introduction

Large volumes of data have been collected and generated considerably over the recent years, among the increasing popularity of smartphones and Internet services. Multidimensional data such as geographic coordinates data and CAD data is considered as core infrastructure and component in the technology world [1–3]. Data queries in spatial data, such as range query and k nearest neighbour query has been widely used in map based products and location based services. Google maps use range queries and KNN to find points of interest. For example, finding restaurants in a region and finding restaurants near me. In VLSI CAD, spatial data index is also used to manage two-dimensional data objects such as polylines and polygons [4]. In some other areas, the spatial management system is also integrated with image processing and computer vision. For example, medical images and remote sensing images use spatial databases to store and location spatial objects from the images [5–8].

How to effectively manage spatial data have been studied significantly in the past few decades [9, 10]. Spatial index methods have been developed in many different approaches. Some have derived from a one-dimensional approach, such as linear hashing [11] and extendible hashing [12]. Some have derived from popular one-dimensional B-Tree structure [13], such as K-D-Tree [14], Quad-Tree [15], R-Tree [16], and so on. All spatial indexes support two-dimensional data points, and most indexes are also capable of handling polygons or line segments. Spatial queries are also different from regular database queries, range query and KNN query is especially important, and each index has a unique way to process these query tasks.

1.1 Motivation

A recent study on the learned index [17] demonstrates that a learned cumulative distribution function (CDF) from a data distribution can predict the position of the key, and invented the Recursive Model Index (RMI). Traditional data indexes follow certain policies to store data in the data structure, and they have to consider all data distribution. There are several advantages to a learned index. Firstly, the learned CDF could potentially reduce the search time complexity, especially in some cases where the CDF curve is perfectly fitted according to the dataset. Secondly, the memory size for a learned model is much smaller than the traditional data structure. For example, B-Tree stores data items in the nodes, the size of a data item must be taken into the memory size, whereas a learned B-Tree model is trying to find the page that contains the query item. The memory size of a learned model contains few parameters. Thirdly, a learned neural network can utilize the SIMD capabilities of CPU and GPU, and this can be a significant advantage on a large scale dataset [18]. All the advantages of the learned model open up a new field in databases, the power behind the learned function that describe the data distribution can be applied to many other data structures and algorithms, and potentially make the traditional data structures and algorithms more efficient.

Furthermore, the one-dimensional learned index can be expanded to handle two-dimensional data. ZM index [19] uses the Z-order curve to connect two-dimensional data points in space transformed into one-dimensional data, then it is easy to apply the learned index model upon the one-dimensional data. The Flood [20] uses a grid layout partitioning the space into different cells, then during the querying phase, the search window looks for intersecting cells, then refine and filter the query result from the cells. The RSMI [21] combines ideas from RMI, Z-order curve and R-Tree, and it uses Z-order numbering each partitioning block, then uses RMI to predict the corresponding block id. There are many more learned spatial indexes that integrate the traditional method with ideas from the learned index. All the learned spatial indexes have shown some performance gain or memory size reduction that benefits from the learned model. Therefore, introducing the learned model in the traditional spatial data index can further improve the efficiency of the spatial data management system.

1.2 Research Questions

There are two primary aims of this proposed research:

- Investigating the existing spatial indexes, the learned indexes and novel learned spatial indexes.
- To apply the learned model technique based on existing knowledge about data indexes.

The success achieved in the one-dimensional data index raises a question if we are reasoning about a similar task on multidimensional data: how can we improve the efficiency of a spatial data index through learned models? Specifically, can we improve the performance of the index by reducing the query lookup speed; or reducing the memory consumption; and compare to existing approaches to the learned spatial index, can we do better or worse than those methods?

1.3 Approach and Outcomes

The objectives of this research are to investigate existing data indexes and propose a learned index method that supports spatial queries. The main contribution of the study consist of three parts:

- We reviewed previous work on various data indexes, including traditional spatial indexes, one-dimensional learned indexes, and learned spatial indexes.
- We proposed a new spatial index method: Learned Spatial Hashmap (LSPH). The LSPH is a spatial index that is built on top of a hashmap with a learned hash function.
- We validated the LSPH with real-world and synthetic datasets, and performance is competitive against other spatial indexes.

Previous studies on the learned spatial indexes have focused on combining traditional spatial index methods with the learned CDF. Almost every learned spatial index has

claimed to have a faster search time and less memory size compared to the traditional methods. However, there are some disadvantages to those methods. First, most learned spatial indexes tended to focus on efficiency rather than accuracy, many of them predict the index position from a recursive model index, which the prediction result can not guarantee to locate the correct approximate position since the error can be very different from the maximum error bounds. Although the error can be reduced to an acceptable range, it still requires hyperparameter tuning for a single application. Second, many of the existing methods lack support for spatial queries. The ZM index does not support KNN queries [19]. Flood and Tsunami process a range scan for query points, but they also do not support KNN query [20, 22]. Third, learned indexes that use a multilayer neural network can have an expensive build time.

We proposed the Learned Spatial Hashmap (LSPH), which is a hashmap with a learned hash function for handling spatial queries. This method is straightforward, it uses values of one dimension from a two-dimensional data pair to train a monotonic CDF that describes the data distribution along one dimension in a set of spatial objects. The learned CDF is functioning as the hash function for the hashmap, and the predicted value from the learned CDF used as the hash value. Since the hash value from the CDF is going to be the same regardless of prediction error, it guarantees to locate the same spot and obtain the search result according to the query. We are also introduced to the single-dimensional index building, which is to train the CDF model with values that are from one dimension. The benefits of building indexes with the dimension with a larger variance are 1. Increasing the space utilization in the hashmap. 2. Query performance gain. 3. Solving skewness by choosing the dimension is less skewed. There are three types of spatial query supported by the LSPH: point query, range query and KNN query. In addition to handling spatial points, the LSPH is also supporting a minimum bounding box. Therefore, the LSPH manages spatial data efficiently and addresses some disadvantages of existing learned indexes.

Evaluation is measured from three aspects: memory consumption, build time, point query time, and accuracy of the result. LSPH has obtained better or even performance results compared to other methods under certain conditions. Overall, LSPH is balanced between query performance, memory consumption and building performance, and it guarantees to retrieve query results 100% accurately. Compared to other index methods, LSPH achieves up to $8\times$ query performance speedup over R-Tree and $40\times$ speedup over

a ZM index on a real-world dataset while maintaining similar memory consumption with R-Tree.

1.4 Thesis Structure

This thesis comprises 5 chapters in total, including this introduction. The rest of the chapters are organized as follows.

- Chapter 2 reviewed numerous traditional index methods targeting spatial objects, novel one-dimensional learned indexes, and learned multidimensional indexes.
- Chapter 3 introduces our proposed Learned Spatial Hashmap (LSPH) and discusses the index design, properties of the learned hashmap, and spatial query handling in detail.
- Chapter 4 provides an overview of experiments and compares the performance results of LSPH with the other two indexes on both real-world and synthetic datasets.
- Chapter 5 summarizes our study, and addresses some limitations and provides future research considerations.

Chapter 2

Related Work

A spatial database system is a data management system for organizing and querying multidimensional spatial objects. Different from one-dimensional data, spatial data objects are often more complex and more informative. One dimensional data usually represents a point object on a single axis. In a multidimensional space, an object can be represented as line segments, polygons or a cluster of points arbitrarily dispersed in the space. It is problematic to represent such a spatial object in a fixed-sized data element. Since there is more than one dimension, the relationship between data values is also more complex. In a one dimensional dataset, we can easily sort the data in a particular order (ascending or descending), and repeated values are obvious to find. In a multidimensional space context, spatial objects can overlap, contain or union with each other, and the order of each object is not so obvious.

Spatial objects are also dynamic [9]. Insert, delete and update spatial values into the database system is required to be robust over time, and a spatial data index should embrace the changes dynamically. Operations in the spatial database system are much broader than traditional databases [9]. The basic query operation can be retrieving points for given coordinates. The more complex ones are retrieving more than one point in the multidimensional space for a given range, or to check whether a point is contained within a spatial object. This can make a spatial index more difficult for handling different types of operations.

The scale of the spatial database system can be large. Presenting multiple types of spatial objects at the same location might take more space because a spatial data index

might use different data objects at this location. For example, a geographic map usually takes up a couple of gigabytes of memory.

Properties commonly shared between one-dimensional data index and spatial data index are space and time efficiency. Data indexes are attempting to reduce the operations that are required for accessing the memory. For example, B-Tree [13] is organizing data at a physical level to minimize the number of access to disk pages. The size of the data index should also be kept small to maximize storage utilization.

Overall, designing an efficient, functional and dynamic spatial index requires considering all the tasks above. In the following sections, we will review the definitions of spatial queries and analyze some popular techniques in spatial data management systems, and spatial indexes that apply these techniques.

2.1 Spatial Queries

We are assuming all the spatial queries are performing in two-dimensional spatial plane space.

Query 2.1.1. Point Match Query: given a point p with a coordinate (x, y) in the spatial extent $p_{x,y} \subseteq E^d$, find a exact point p' within the same spatial extent as the p :

$$PMQ(p) = \{p | p'.x = p.x \wedge p'.y = p.y\}.$$

The most important and frequently used query in spatial data is the point match query. The Point Match Query searches for a spatial point for given coordinates, and it is a fundamental query for more complex queries.

Query 2.1.2. Range Query: given a two-dimensional area of interest $R = [p_{x_{min}}, p_{y_{min}}] \times [p_{x_{max}}, p_{y_{max}}]$, where $[p_{x_{min}}, p_{y_{min}}]$ is the coordinate of the lower-left corner of the bounding box, and $[p_{x_{max}}, p_{y_{max}}]$ is the coordinate of the upper-right corner of the bounding box. Find all points p within or overlapping with the range R :

$$RQ(r) = \{p | (p_x, p_y) \cap R \neq \emptyset\}.$$

The Range Query retrieves multiple spatial objects that are within a window. The shape of a window is usually a rectangle, but it can be in other forms of shapes with arbitrarily unparallelled lines.

Query 2.1.3. Nearest-Neighbor Query: given a point p with spatial space $p \subseteq E^d$, find k points p' having a minimum distance from p :

$$NNQ(p) = \{p' | \forall p'' : dist(p', p) \leq dist(p', p'')\}.$$

The Nearest-Neighbor Query is to find k neighbour objects that are closest to the query point. The distance functions for NNQ to compare the closest points include the Euclidean and Manhattan distance.

2.2 Spatial Indexing Techniques

Many spatial indexes are developed based on one-dimensional methods. However, to fulfill requirements of query handling in spatial indexes, additional techniques such as transformation, overlapping regions, clipping and multiple layers are introduced in [23].

2.2.1 Transformation

The transformation technique constitutes spatial objects (polygons, line segments, and so on) into a format of point, it can allow one-dimensional data management methods directly applied to spatial objects. The space-filling curves are examples of transformation, the continuous curve transforms two-dimensional data into one-dimensional while preserving the locality. Other methods including minimizing the spatial data object into points, x and y coordinates of two diagonal corners can be used for endpoint transformation; or a centroid point of the object with two additional parameters that measure width and length. However, the disadvantage of transformation is that firstly the range queries can be more complicated than in the original space. Secondly, the distribution of data can also have a larger chance to be non-uniform due to useless points. Thirdly, two objects that are close to each other in the original space can be arbitrarily apart in the one-dimensional space.

2.2.1.1 Space-filling Curves

Space-filling curves can be used to represent a list of grid cells or, equivalently, a list of one-dimensional intervals that define the position of the grid cells concerned can be used to represent extended objects. It is especially useful for organizing multidimensional spatial data on one dimensional data access methods. How to order points on a grid layout and fill curves in between points is critical for an efficient data lookup. Popular space-filling techniques include the Z-ordering curve [24] and Hilbert curve [25]. Both of them tend to arrange data as close to each other as possible, as well as in terms of memory space.

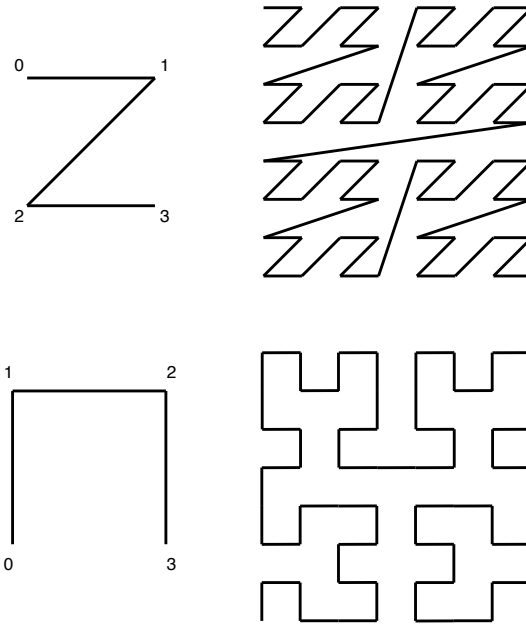


FIGURE 2.1: Z-ordering Curve and Hilbert Curve

Z-ordering Curve Z-ordering [24] is a space filling curve dynamically hashing points that are close to each other, which is based on the Peano curve. The process of obtaining a z-ordering curve on a space starting from splitting into two equal sized subspace by $(d - 1)$ -dimensional hyperplane. This process can be executed recursively until it meets the following criterias [9].

- The current subspace does not overlap the data object.
- The current subspace is fully enclosed in the data object.
- Some given level of accuracy has been reached.

Each subspace can be represented by a unique z-ordering hashed bit string. To construct z-ordering bit strings in a coordinate system, one uses a bit interleaving the two binary values from the coordinates. For example, a value from a coordinate system can be represented as $(1, 2)$, the binary string for the two values is 001 and 010 respectively. The first value in the bit string comes from the first bit of the first value 1, followed by the first bit of the second value 0. Therefore the first two bits are 01, and the completed resulting bit string is 001001. Z-ordered values created from the procedure are visually connected by a recursive z shaped curve. The resultant area collection is just an estimate of the initial entity. The accuracy and granularity (maximum number of bits) are required by the termination criterion. More Peano regions naturally will obtain more accuracy, but also sacrifice the size and complexity of estimating as a trade-off. The advantage of z-ordering is that spatial granularity shifts contribute directly to contextual improvements in the subsequent encoding. One drawback of z-hashing is that many useless data blocks will be created.

Hilbert Curve Another space-filling curve preserves locality well. Hilbert's order has been reported to be superior to the Z-order curve as it has a better locality-preserving behavior [23]. The Hilbert curve was first invented in 1891 by German mathematician David Hilbert [26]. The fundamental idea is using a continuous line connecting the closest element in a grid, the difference between the Hilbert curve and the Z-order curve is that Hilbert is a U-shape curve. The transition from one column or row to the next is also choosing the closest cell rather than the diagonal cell in the Z-ordering curve. The U-shaped curve will be passed to the next cell in the grid in a similar manner, extending the continuous curve until the entire grid is filled in. The mapping of two-dimensional data stretches to the one-dimensional data layout, preserving the locality very well. It means that for points (x_n, y_n) close to coordinates (x, y) , the output Hilbert value d_{x_n, y_n} is also going to be closed to $d_{x, y}$. The Hilbert curve has been applied to the R-tree, combined with overlapping regions technique to improve the search performance. The Hilbert R-tree will be discussed in the latter part of the review.

2.2.2 Overlapping Regions

Overlapping regions is a method that allows data buckets to access regions that have shared overlapped subspace. The linear transformation made point queries easier, but

it is a challenge for handling range queries and containment queries without a complex spatial structure. However, redundant data can be created in overlapping regions, as a result, the number of search paths will be increased. Each insertion of new data can increase the overlapping area, eventually, overlap becomes too large for index searching efficiently. Therefore performance can be an issue for overlapping regions.

2.2.3 Clipping

Clipping has to be mutually disjoint, which does not allow overlapping between subspace. Consequently, the search path will only run through one path to the corresponding nodes instead of searching for multiple paths in overlapping regions. The major problem with clipping is also related to the insertion of new data. For new data that spans outside of the existing bucket region, a subdivision is required for subspace partitioning. This leads to a problem where more than one bucket has to be created for the same entry. The redundancy [27] can increase in the average search time, and also increase in the frequency of bucket overflow. Another problem with clipping is that when inserting new data, it can cause enlargement of the region if the region did not cover it. In certain circumstances, it is impossible to construct an enlargement without having an overlap between bucket regions. A new region has to be created due to overlapping is not allowed. This can potentially produce a much more complicated structure and problems with handling bucket insertion and overflow. Finally, due to splitting into smaller regions, the number of splitting regions will be increased. Hence, the database size is more likely to be large.

2.2.4 Multiple Layers

Multiple layers [9] are similar to overlapping regions, since regions in different layers can be overlapped. A hierarchical structure is usually used to combine with the multiple layers. However, data regions within a layer are disjoint, and it does not support extension to accommodate data objects. To insert a new data object, we attempt to find the lowest layer in the hierarchy structure that does not require to split. If such a layer exists, the data object will be inserted into that layer. If not, a new hyperplane will be introduced in order to split the new data region, then insert the data object to the new data region. Object will be moved to a higher layer, if it intersects the hyperplane.

As we inserted more data into the structure, the higher layer will be collected on the higher layer, and the lower layers will have more segments of data objects. This can be inefficient for certain data distributions. Although, multiple layers still have advantage on selectivity of search path because of reduction of overlapping.

2.3 Traditional Spatial Data Indexes

Characteristics of multidimensional data is complex [9, 28]. Gaede and Robinson have summarized some requirements for spatial indexes: dynamic, secondary storage management, a broad range of supported operations, independence of input data, simplicity, scalability, time efficiency, space efficiency, concurrency and recovery, minimum impact. The following sections introduce some of the popular traditional spatial indexing methods, illustrate a wide range of techniques, as mentioned above, addressing the requirements for spatial index.

2.3.1 Grid File

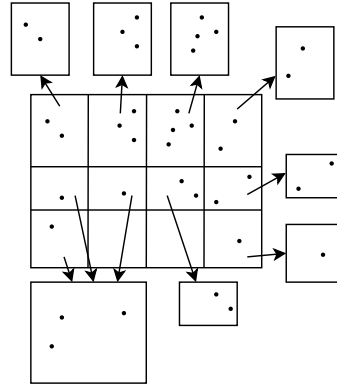


FIGURE 2.2: Grid File

The grid file [29] is a hashing-based spatial access method, which is a grid layout partition to store data objects. A grid file consists of a rectangular d -dimensional cell, each grid cell has a number of shapes and sizes. A bucket contains one cell and many its neighbouring cells, and each cell is bound to one bucket. The main structure of the grid file consists of d one-dimensional arrays called scales that are kept in the main memory.

To perform an exact point match query, the first step is to access the location of the cell in the scale. Then the located cell will redirect to the reference of the page that contains

all the potential points in that cell to perform an exact point match. The exact match query for the grid file is guaranteed to have 2 disk accesses. However for a range query, we must scan all cells that contain in the search region.

Inserting a point to the grid file, the exact point match is also required. The matching query is trying to locate the cell, the point can be inserted if there is enough space. Otherwise we need to test if there is a hyperplane in the scale usable for separating data sections. When there is such a hyperplane a new data page must be generated and the data distributed accordingly. When no other hyperplane occurs, it would produce a splitting hyperplane. A new data page will then be generated, and the newly inserted point will be spread among the new data with the old data page.

One of the major problems with grid files is dictionary split is expensive in high dimension, and even for uniformly distributed data it still has a super-linear growth of the directory. Some variations of the grid file tend to improve the structure and occupancy in the data buckets. EXCELL split the cell all in equal size, compared to the arbitrarily splitted space in the original grid file. The Two-Level Grid File added a level of grid file, limiting the splits without impacting too much of their surroundings to the subdirectory regions. The Twin Grid file introduced a second grid file parallel to the first one, which is a multilayer structure. The grid file has to create new data space by splitting thought hyperplanes when the number of points exceeds the limit. The twin grid file creates a new data space that requires redistribution of the data.

2.3.2 Multidimensional Linear Hashing

MOLHPE proposed by Kriegel and Seeger [30] a multidimensional linear hashing with partial expansions to maintain order. This arrangement is based on the concept of extending the buckets in part without necessarily increasing the file size. Page address splitting depends on the page size from 0 to N , and a pointer p keeps track of the page. Two important contributions in building access methods. Firstly, whether a document remains in the old page or shifts to the new page only relies on the data key. Secondly, the old and the new sections will be filled in uniformly. Pages are united into groups to achieve a more evenly distributed loading factor. A group of pages will be expanded by one page once a new page has been added to the file. The file page expansion using an

expansion pointer $ep = (ep_1, \dots, ep_d)$, and the address of the group is given by $G(ep)$. The expansion pointers will be updated along group partial expansion.

It performed well over uniformly distributed data, outperformed many other spatial data retrieval systems. It overcomes the superlinearly growth of dictionaries in some multidimensional hashing methods such as the grid file. However, it does not handle very well on the non-uniform data distribution. A year later the same writers suggested linear order-preserving (PLOP) hashing to improve the performance on the non-uniform distributed data. The space partitions in PLOP are similar to the grid file, hyperplanes are organized by the binary tree. The d dimensional binary tree is also kept in the main memory just like the scale in the grid file. All the entries are stored in the leaf node, each leaf node from the binary tree is linked to each other for speedup.

2.3.3 K-D-Tree

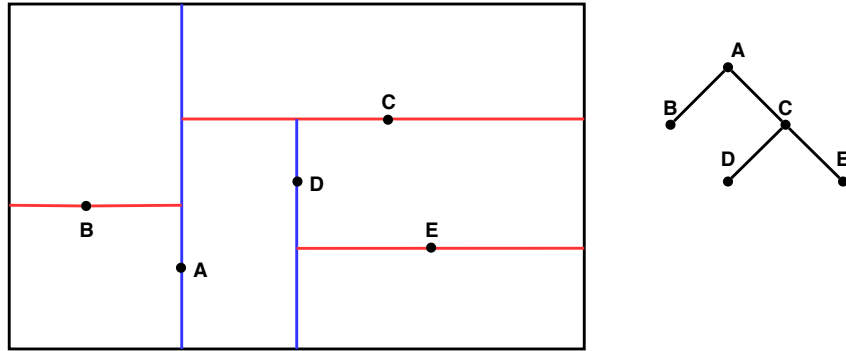


FIGURE 2.3: K-D-Tree

K-D-Tree [14] is a popular multidimensional data scheme that uses a hierarchical structure. It is a binary search tree, and each node represents a point in K dimensional space. The universe of K dimensional space is splitting by $K - 1$ dimensional hyperplanes recursively. Each data point is used as the coordinate of splitting hyperplanes, which depends on whether the node is X-align or Y-align. The X-align or Y-align decides which axis to compare for newly inserted data points. The intermediate nodes have one or two descendants for search. The searching procedure is straightforward, comparing the corresponding axis value from the tree level, then going to the right or left subtree, repeating the process until the goal data point is found.

One of the problems with the K-D-Tree is that the order of inserting points can have a large effect on the shape of the tree. Also, data points are distributed scattered over the

tree, which will be difficult to execute complex queries such as the nearest neighbours query. Keeping the tree balanced can also be hard since the structure of the tree is mostly decided by the order of insertion.

The adaptive K-D-Tree [14] improves on the splitting by choosing the point that between points are evenly distributed on both sides. The splitting point is no longer taken from data points, data points are only stored in the leaves. Intermediate nodes store the coordinates of splitting points and the dimension information. It improves the original structure trying to store point values evenly, but it is still sensitive to the order of points being inserted.

2.3.4 BSP-Tree

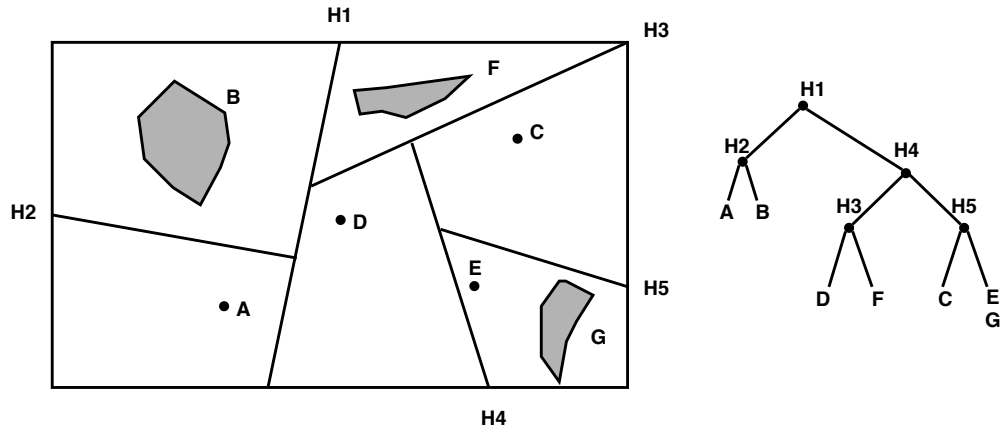


FIGURE 2.4: BSP-Tree

Comparable to K-D-Tree, the BSP-tree [31] is also a binary tree with a recursive subdivision subspace by splitting hyperplanes. Instead of vertical or horizontal lines in the K-D-Tree, the BSP-tree uses the arbitrary orientation of splitting hyperplanes. It allows the subdivided space to be more flexible and well suited for arbitrary shapes of spatial objects. Since data points are not used as the splitting hyperplane, each subspace can be divided independently. The partition hyperplane can be selected depending on the distribution of data objects. Inserting a point to the tree is also similar to the K-D-Tree. Starting from the root node, the search point has to compare with the value in the root node and determine whether to go to the right or left part of the tree. All data objects are stored in the leaf node, so the search will stop until reaching the leaf node of the tree. The BSP-Tree can handle non-uniformly distributed data well. However, arbitrary splitting hyperplanes also require more complex calculations and storage space.

2.3.5 Quadtree

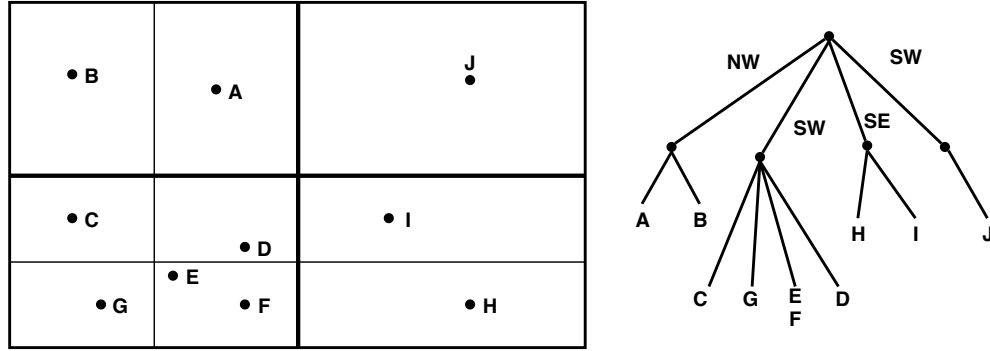


FIGURE 2.5: Quadtree

The Quadtree [15] is also considered to be closed to the K-D-Tree. The difference is the Quadtree is not a binary tree anymore. The intermediate node in the Quadtree will have 2^d descendants for d dimension. For example, $d=2$ will have four descendants, which each of the descendants is referred to as a rectangle region. These four descendants can be viewed as quadrants that represent four directions (NW, NE, SW, SE) prospectively. The recursive subdivision will be continued on the rest of the data. Consequently, the Quadtree will not be balanced since that more densely the region is deeper the tree will grow.

Searching and inserting in the Quadtree is similar to other tree structures. New data points will start from the root node, the node will decide one subtree from the four descendants. For point query, there always will be only one candidate subtree for further searching. For range queries, there might be one or more candidate subtrees. The process will continue recursively until finding the goal.

2.3.6 R-Tree

R-Tree[16] is a height-balanced hierarchical data structure that applies overlapping technique. Only the leaf nodes store the minimum bounding box (MBB) of data objects. For the interior nodes, each node has to store two information: children of the node and a MBB that contains all the nodes. The R-Tree contains the following properties:

- Each leaf node contains between m and M index records, except the root node.

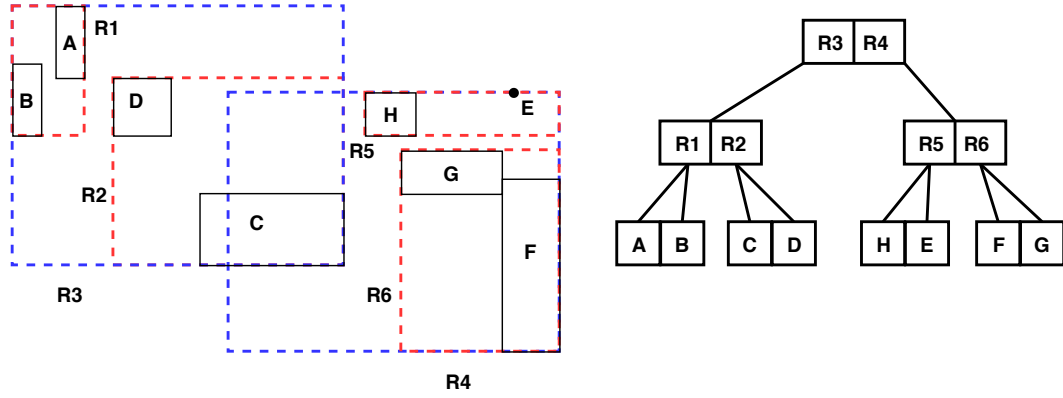


FIGURE 2.6: R-Tree

- For each entity in a leaf node, a minimum bounding box is used to contain the n -dimensional data object.
- The root node has at least two entries unless it is a leaf.
- All leaves are at the same level.

The increase of branching factor m will decrease the tree height and improve space utilization because most of the space will be used in the leaf nodes. Searching in R-Tree is similar to the B-Tree. At each node in the tree, they will be tested if within the search ranges. Since the bounding boxes in R-Tree can be overlapped, the search path can go through more than one subtree. Because R-Tree only stores MBBs, the search result is not an exact match result. For actual spatial context data, we need to search further within MBBs.

Insertion in R-Tree is also similar to B-Tree. The newly inserted object first recursively finds a path to the leaf node starting from the root node. If the leaf node is not full, insert the new object. Otherwise, the leaf node has to be split by a splitting heuristic. The old objects from the original node will be distributed among the original and new nodes. We can adjust the tree by propagating the newly added nodes up and change the height of the tree.

For the deletion of a node in the tree, the search query must be performed. If found in a leaf node, then delete it from the node. If there is not an underflow, the MBB will resize according to the updated nodes. If it causes an underflow, and the node occupation below m , then adjusting the tree is needed, the subtrees of the node will be dissolved and these nodes will propagate upwards. Old entries in the node will be inserted into a

new temporary node or another option is to merge with siblings node. In both options, the tree has to adjust the MBB size and propagate upwards.

To reduce the overlaps in insertion, the author proposes several different splitting algorithms in the original paper. A quadratic-cost algorithm picks two of the $M+1$ entries as the first elements of the two new groups. The criterion of choosing the pair is that they would take most areas if both were put in the same group, which means the region of a rectangle covering the two entries, minus the region of entries themselves, would be the largest. A linear-cost algorithm is similar to the quadratic-cost algorithm but uses different criteria to select the two entries. The two entries were selected based on the dimension of entries. For entries whose highest low side and lowest high side, record such separation. The separation will be normalized by the width of the entire set along the corresponding dimension. Then choose the pair with the greatest normalized separation along any dimension.

2.4 Learned Indexes for One-dimensional data

2.4.1 The Recursive Model Index

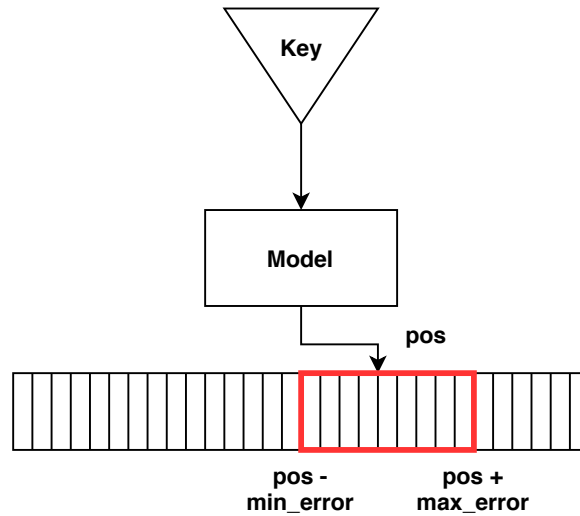


FIGURE 2.7: RMI

Kraska [17] proposed a method that uses a machine learning model to replace indexes. Kraska introduced the idea that data indexes can be treated as models, we can describe the data distribution by constructing the cumulative continuous functions. A B-Tree indexes a key into a hierarchical structure, retrieving the position of that key from

traversing the tree. It stores as many as children as possible to diminish the height of the tree, hence to reduce the number of disk access. The B-Tree is a self-balance tree, it only provides an approximate position with min and max error with stored keys. The optimal searching time complexity requires log time retrieval. To insert and delete a key, B-Tree needs to re-balance to provide the same search time guarantee. More importantly, strong error bounds are not needed since prediction results can be replaced by local precise search algorithms. Identical patterns obtained in a continuous function that used to describe a distribution of data can be used to build a data structure for indexing. As a result, range searching in the B-Tree index can be replaced by machine learning models such as linear regression, which can predict an approximate point to the next stage and error from the prediction does not affect the final search accuracy.

$$p = F(Key) * N$$

The CDF models data distribution and predicts the position of the key as shown above. P is the predicted position, F is the cumulative distribution function estimate likelihood for a range observed key and N is the total number of the key. The learned index is optimized by learning the CDF and minimizing the error from the linear regression model. The recursive-model indexes (RMI) is a series of machine learning models on a hierarchy structure. The RMI consists of multi-stages, where the model predicts a value from the input key at each stage. Due to range indexes that can point to a range of values, the predicted value does not need to be exact. The predicted value can be the final position result if at the final stage of the structure or a pointer to the next stage if it is not the final stage.

In the experiment, the recursive model can be built by a mixture of machine-learned index models and B-Tree. For a 2 stage model: the top stage is a 2 layer fully-connected neural network with 32 neurons per layer with the ReLU activation function, and the second stage can be thousands of simple linear regression models or B-Tree. The data distribution learned by the model can give a prediction of the pointer, the pointer is pointing to a specific range of values in the next stage. All models at final stages are trained by a specific range of values, and have much better accuracy for values within that specific range. In other words, we are reducing the searching range of values at each stage. However, like most neural net models, the prediction value can have uncertainty

with min-/max-error. In the case of the neural net model performing worse than a threshold, the neural net model can be replaced by a B-Tree to ensure the ‘last-mile’ accuracy.

Two search strategies mentioned in the paper, one is the model biased search and another is biased quaternary search. The model biased search is a simple binary search that it set the middle point of as the predicted value from the model. The biased quaternary search takes $pos - \sigma$, pos , $pos + \sigma$ as the initial three points, which can make the initial guess as accurate as possible. The min- and max-error sets upper and lower bounds that can guarantee all the points can be found. However, a wrong lower or upper bound can be generated for non-monotonic RMI models if the key does not exist.

Kraska also discussed some applications of the learned model on other data structures. The learned hashmap is trying to use a CDF to replace the hash function. The result shows the learned hashmap can reduce the number of conflicts by up to 77% on a certain dataset. Kraska suggests that for a uniform distributed dataset, the linear model is no better than a randomized hash function. For small keys, traditional hash functions with Cuckoo hashing provide the larger payload and reduce conflicts from distributed hashmaps.

2.4.2 ALEX

The ALEX [32] is an adaptive learned index improvement of the RMI. The major improvement is that ALEX uses different data structures to store data on the leaf nodes, whereas RMI uses only one single array. Gapped Array (GA) and Packed Memory Array (PMA) are the two node layouts that have been explored in the paper. Since all the data are stored in the leaf node, the search method in ALEX uses exponential search. The other feature is that ALEX adjusts the shape and height of the RMI dynamically depending on the updated workload. It also inserts records dynamically, which stores the key at where the model predicts.

Gapped Array is one of the node layouts that use the idea of exchanging extra storage for the speed. It creates a gap during insertion to maintain good exponential search, then utilize the gaps with adjacent keys. To insert a new key into a gapped array, we must find the insertion position. RMI is used to predict the insertion position, if this insertion

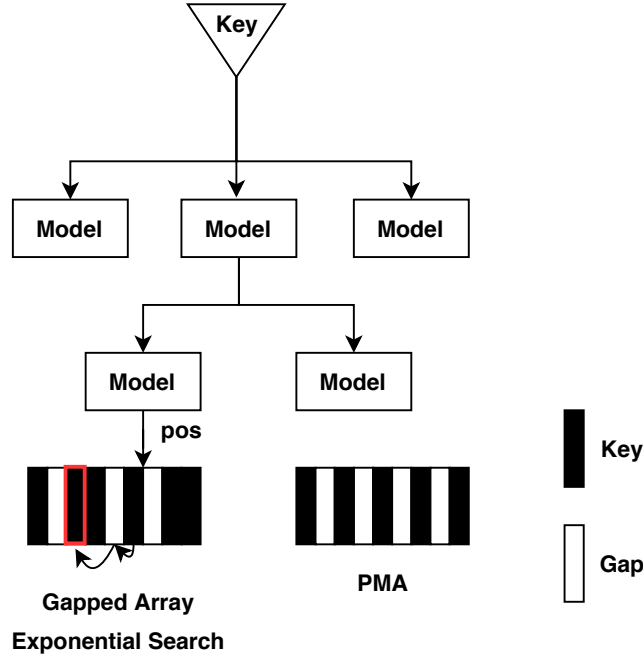


FIGURE 2.8: ALEX

position is maintaining the sorted order, and is a gap, then the new key is placed to the gap and finish. If the predicted insertion position is not right, we have to use an exponential search to find the correct insertion position. The correct insertion position must be a gap to insert the new key. If it is not a gap, we shift the elements at this position by one position in the direction of the closest gap. The insertion performance achieves $O(\log n)$. One procedure during insertion is if a new key pushes into the gapped array and exceeds the density bound d , then the gapped array needs to expand. A disadvantage of gapped arrays is when inserting a long contiguous region without any gaps, such *fully-packed region* can cause a dramatic increase in the insertion time.

Packed Memory Array is another node layout that has been shown to adopt and grow better to insert. The PMA is an array whose size is always a power of 2, and it divides itself into powers of 2 numbers of segments. The PMA builds an implicit binary tree, each segment is a leaf node and the root node represents the entire array. Each node will have a density bound that can determine the maximum number of elements to the position in the region of the array. The size of PMA will expand once the density in the array becomes too full. The rebalance local portions of the PMA allows data to be uniformly distributed.

Adaptive RMI Initialization in ALEX sets a predetermined maximum bound for the

number of keys in a leaf node so that ALEX can determine the RMI depth and the number of leaf models adaptively. The root node will be divided into a certain number of partitions, and each partition will have exactly the maximum number of keys. Node splitting on inserts is another optimization in ALEX. If the new key will push to the leaf node exceeds the maximum bound, then we can split the node. The corresponding leaf node will become the interior model, and a number of children nodes will be created from the model. This allows the ALEX to start with empty data, and the RMI will grow deeper along with more inserted nodes.

2.4.3 The PGM-index

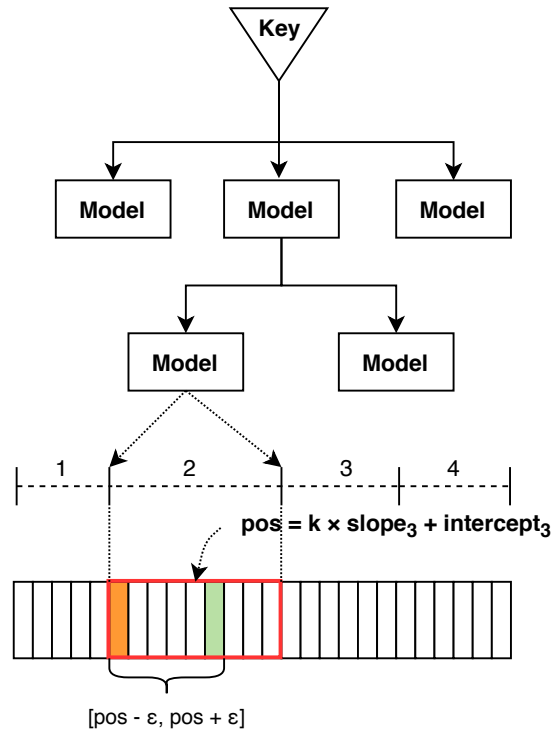


FIGURE 2.9: PGM-index

The Piecewise Linear Approximation model [33] is a pure learned data structure, rather than a hybrid learned model such as RMI. The PGM-index builds upon a sequence of segments, each one indexing a sequence of keys in constant space and constant query time to return the approximate position of the key in the array. Another property of the PGM-index is it is a recursive algorithm that constructs the index structure to fit the distribution of the keys.

The PGM-index is to focus on optimizing space-time trade-off. It has the following advantages in space-time complexity:

1. Since the PGM-index is built upon the minimum number of segments from bottom-up, which is more efficient than other learned indexes in terms of time and space.
2. The PGM-index uses these segments as a constant-space routing table, rather than storing large number keys depending on the disk-page size in other learned indexes.
3. The routing tables in the PGM-index take constant time to limit the search of a key in a node to a smaller subset of the indexed keys.

The author provides compression algorithms to reduce the number of distinct slopes to their optimal minimum number for the compressed PGM-index. The multicriteria PGM index exploits the space-time flexibility by optimizing in two different criteria: 1. Given a space constraint the output PGM that minimizes the query time. 2. Given a maximum query time the output PGM that minimizes the space. The number of ε -approximate segments controls trade-off between the space and time. The space occupancy decreases with increase of ε , whereas the query time $t(\varepsilon)$ increases with increase of ε .

The performance result shows that the PGM-index improves space occupancy by offering theoretical bounds, and it guarantees a 4X increase in the precision of the estimated position of the searched key compared to the RMI. The PGM-index is reported to provide better latency guarantees since it can dynamically adapt to the data distribution, and it can use the ε -approximation to optimize a better performance trade-off between space and time.

2.4.4 RadixSpline

RadixSpline [34] is another learned index that builds upon a single pass over the data. RadixSpline is built in a bottom-up approach in two steps. First, a linear spline is to model the cumulative distribution functionc (CDF) of the data, which guarantees certain error bounds. Second, a radix table will be built as an approximation index into the spline points.

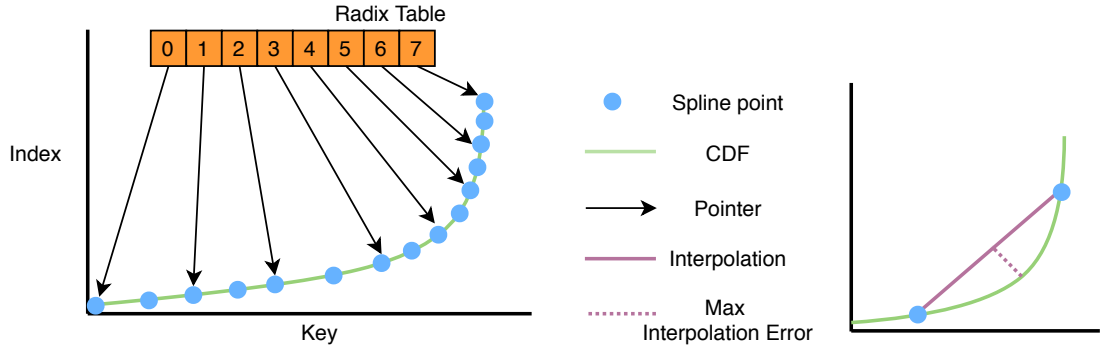


FIGURE 2.10: RadixSpline

To build a RadixSpline, we first construct an error-bounded spline model S on top of the underlying data. The spline model S always predicts the correct location of the data within a constant error of e . The parameters of the model S are a set of spline data points $Knots(S)$. For any look up key k , linearly interpolating between the two closest spline points in $Knots(S)$ will output an estimate within error e .

Then, the selected spline points are built to index in a radix table. The radix table can quickly find the two spline points surrounding the lookup key. It is a flat `uint32t` array that maps fixed-length key prefixes (“radix bits”) corresponding to the first spline point. The radix table uses r as a parameter, so array size will be 2^r . To increase the precision, the size of the array 2^r will grow exponentially with growth of r . The RadixSpline supports single pass building CDF, that means the sorted data points are passed in the spline construction and radix table allocation at the same iteration.

Lookup in the radix table takes following procedures: first to extract an r -bit radix prefix b from the key. Then, we can use the b radix prefix access to the radix table retrieving two spline points. A linear interpolation will be performed between two spline points to find an estimated position p of key. Finally, it uses a binary search to find the first occurrence of the key within the error bound $(p \pm e)$

The building performance of RadixSpline is significantly lower than the RMI. In lookup latency, RadixSpline performs similarly with RMI, which both are better than traditional indexes such as B-Tree. However, the lookup latency of learned indexes can be affected by the distribution of the dataset. The index size of RadixSpline is unstable for different dataset, the paper reports significantly high memory cost in one of the dataset.

2.4.5 FITing-Tree

FITing-Tree [35] is another learned index that uses a piecewise linear function for key index approximation. The idea of the FITing-Tree is to partition keys into fixed-sized segments within certain error bounds. The error is defined as associated with the maximum prediction error distance between the actual position and the predicted position. Two types of FITing-Tree have been proposed, which a clustered FITing-Tree manages sorted and indexed key in table pages, and a non-clustered FITing-Tree adds an “indirection layer” to store sorted pointers to the key.

Lookup and insert of the interior nodes in a clustered FITing-Tree are similar to B+tree, but once the operation reaches the leaf node, the FITing-Tree does some additional work. The FITing-Tree stores a segment’s slope, starting key, and a pointer to a segment at leaf level. Linear interpolation calculates the approximate position from a given slope.

The major difference between a non-cluster and a cluster FITing-Tree is the additional “indirection layer”. This layer indexes an array of pointers to the values that have been sorted within each segment, while the actual data remains unsorted in the table pages. All operations in the non-cluster FITing-Tree are performed on the indirection layer. To look up a data item, it must find the index of the pointer to the query data item in the indirection layer first, then the pointer returns the actual position in the table pages.

Besides point query, FITing-Tree also provides range queries. For a given start or end of a range in the tuple, the searching node performs a tree traverse to the leaf, where it stores the slope and starting key of the segment that can be used for index prediction. Since values in the clustered FITing-Tree are sorted, and the pointers are also sorted in the indirection layer in the non clustered FITing-Tree. The search can take place contiguous until reaching the end of the range.

Similar to other learned indexes, FITing-Tree requires local search at leaf nodes. The clustered method requires sorted data, while the non-clustered method doesn’t require data to be sorted at table pages but pointers in the indirection layer still need to be sorted. The reported result [35] showed that a non-clustered FITing-Tree is smaller than a non-clustered B+ tree with fixed-size pages because of fewer leaf and interior nodes that it has.

2.5 Learned Spatial Indexes

2.5.1 Learned ZM index

Inspired by the Learned Index, Wang [19] integrate the idea of using models to replace indexes and spatial technique to create an index model for spatial queries. One of the difficulties of the multidimensional access method contrasts with one-dimensional is to preserve the sequential order of spatial indexes. Learned spatial index can adopt a similar idea from one dimensional learned index models. The concept of the Z-order curve can help to convert multidimensional data into 1D space vectors. By interleaving two bit strings x and y from a coordinate point (x, y) , we can generate a unique number for this point on 1D vector space, which is referred to as Z-address. The multi-staged recursive model is similar to the Learned B-Tree model, using the predicted value as a pointer to access to models at the next stage. This can guarantee the lower bound of the error from the model prediction will not propagate to the next model.

The search strategy for range query processing is based on Model Biased Search (MBS) proposed in [17], which is different from the pointer approach in the B-Tree model. It is similar to binary search except that the first middle point is set to be the predicted value from the model. The MBS is to find the precise position of the data object after the RMI predicts the estimated position.

The Learned ZM index is reported to save above 90% memory compared to R-Tree since the structure itself only has a few parameters instead of storing data objects. The Point query performance of the ZM is 2.3X and 2.5X of the R-Tree on the two dataset RANDOM and POST respectively. For the range query, the performance is varying from selectivity from 0.0001 to 0.1. The ZM performed better than the R-Tree when the selectivity is low, but R-Tree has slightly fast query time when searching in a larger range of selectivity.

2.5.2 Recursive spatial model index

The recursive spatial model index (RSMI) [21] is another spatial data index that incorporates the Z-order curve and the RMI. Data points in the spatial space are first mapped to a $n \times n$ grid to describe the rank space. Then a Z-order curve maps every point on

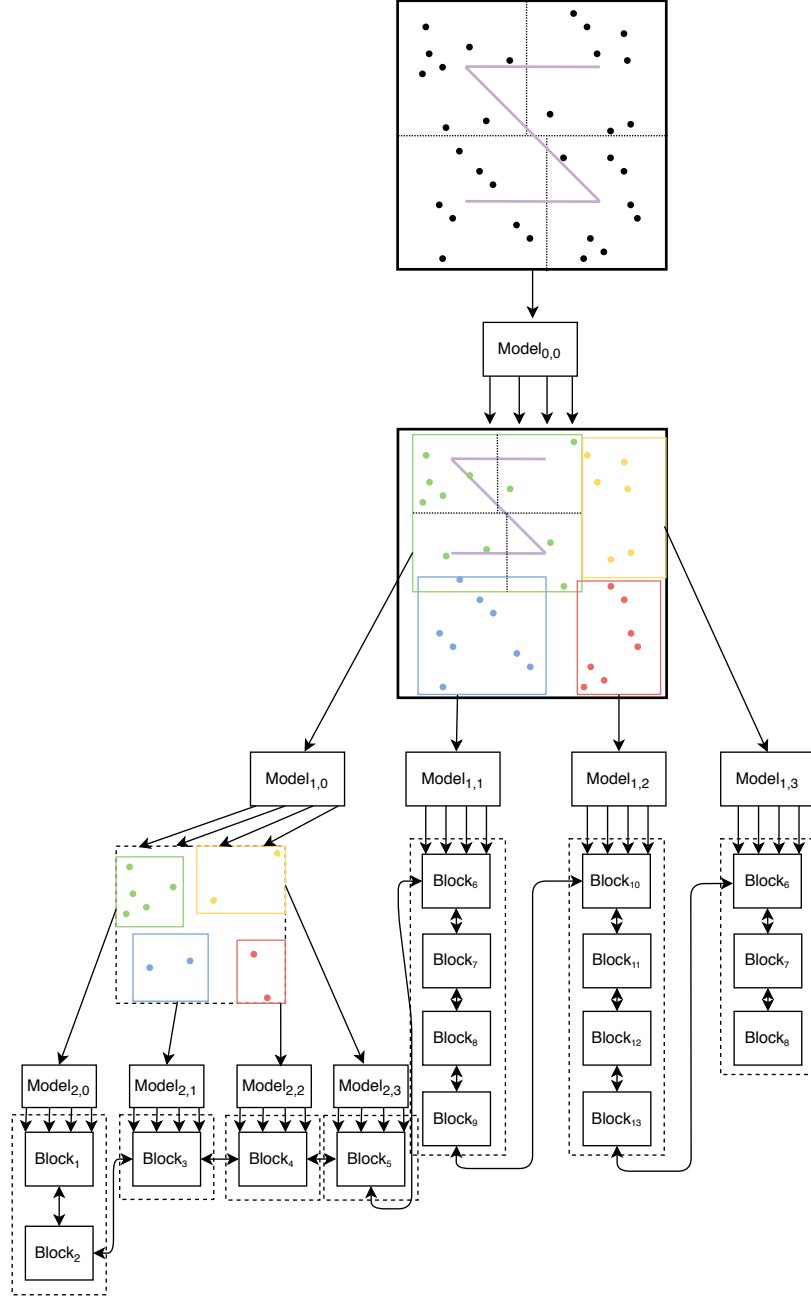


FIGURE 2.11: RSMI

the rank space with a curve value. All points are sorted in ascending order based on the curve value, then points can be packed into a block by $p.blk = \lfloor p.rank \cdot n/B \rfloor$, where B is the capacity of a block. Then, a regression model is trained to predict the $p.blk$ for given point coordinate.

For a larger scale data set, the RSMI partition all data points into a certain size of the grid. Each of the grid cells contains up to a certain number of data points. At the top level, the model is predicting a value that is not directly the exact matching result.

Then, we can regroup points with the same block curve value. The partition will be repeated to follow in the same manner, the learned model in each partition predicts which sub-partition the point belongs to. Once the partition is completed, data points along with block id from preceding and subsequent blocks are stored in each block. The pointer of block id makes it easier for access and searching in blocks at a different level.

To execute a point query in RSMI, the coordinates of the searching point is used as the input query that can be fed into the learned model. The searching process is done recursively, starting from the root model, the prediction value is pointing to the submodel at the next level. The searching procedure is repeated recursively until reaching the leaf model. The output prediction value from the leaf model is the block ID of the input point. At the final stage, we can examine the block and its neighbouring block within an error bound.

For the range query, we can locate the minimum and maximum curve value q_{min} and q_{max} from the input query. Then it retrieves all the candidate points between q_{min} and q_{max} . Finally, candidate points can be filtered out from the window range, and remaining points will be the result.

The K nearest neighbour query in RSMI starts with searching in a small region around the input query key. It will keep searching in the expanding region until k points are found. A window query will be performed when the search region is expanding. Measuring the distance from the point to the query point, and comparing to the existing points that have the minimum distance in the result. The initial search region size can be determined by the number of points and number of regions for the uniformly distributed data. Two skew parameters are introduced to adjust the search region based on the skewness of the data.

The result shows that the RSMI has a good performance on the point query, the number of block accesses is significantly lower than traditional data indexes and the ZM model. A disadvantage of the RSMI is that the training time cost is much higher compared to the traditional methods. The training time of RSMI is also higher than the ZM model since sorting is needed for each partition.

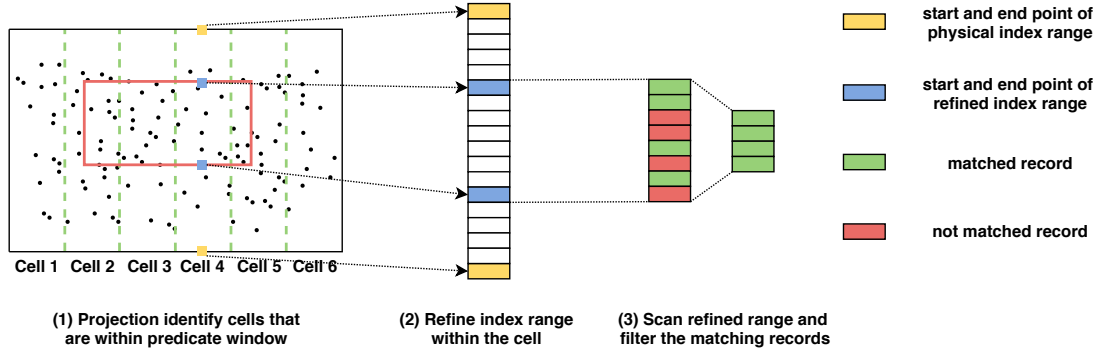


FIGURE 2.12: Flood

2.5.3 Flood

Flood [20] claims to be the first learned multi-dimensional in-memory index, which is based on the point access method grid file. The Flood model first preprocesses offline to choose an optimal layout for indexing, then an online component to handle executing queries.

The model breaks down into three parts: Projection, Refinement and Scan. Similar to the traditional grid file, projection first to match a query is to locate the cell containing the search point, which is to identify the range of position in storage. Compared to z-address, points in the columns that are along x and then sorted by their y-values, creating the serialization order. Then, refinement can narrow down the range of search. When query overflows the sort dimension, Flood can take advantage of the ordering of points within each cell to further refine the physical index ranges. The final step is to scan and find the matched query filter. The model time complexity depends on the weight for each step w_s , w_r and w_s and size of cells and scanning points.

Flood also uses a piecewise linear model (PLM) to train the CDF. By using PLM, the partition boundary between each column is not equally spaced. A flattened layout, as referred in the paper, is introduced to correlate with unevenly distributed data. They also mentioned that modelling based on a single-dimensional attribute was sufficient to reach an equal number of points in each partition.

Compared to other traditional multidimensional data structures, Flood shows significant performance improvement on query time. In measuring varying query selectivity, Flood also works well and outperforms other traditional methods.

2.5.4 Tsunami

A more recent study from the same authors of the Flood model proposed an improvement index Tsunami [22] based on the Flood model. Flood has some limitations when training on the non-uniform data distribution, each grid cell can have uneven distribution of data points. To address this problem, Tsunami proposes a data structure composed of the Grid Tree and Augmented Grid.

The Grid Tree divides space into partitions based on the values of split dimension. Each partition contains a number of data points, which are stored in the internal node of the Grid Tree. An optimization is applied to the Grid Tree to minimize the query skew, by setting the split value that reduces the skew the most.

The Augmented Grid can correlate the points within each sub-region. Several different strategies can be applied to create partitions. Notably, the strategy used in Flood is CDF on X independently from other dimensions. It can use a conditional $CDF(X|Y)$ that X dependent on another dimension Y . It can also use a functional mapping to some dimension other than X . They used adaptive gradient descent to optimize the augmented grid for the lowest average predicted query time and search space size.

The performance result shows that query time for Tsunami is $6\times$ faster than Flood and $11\times$ faster than the traditional index. The index size of the Tsunami is also much less than other indexes. The advantage of Tsunami is that it proposed strategies for data correlation to produce a more uniformly distributed data from a skewed dataset. There is still room for improvements, as mentioned in the paper, functional mapping can not handle outliers very well, the error bound can be enlarged; and the Augmented Grid can not handle higher dimensional data.

2.5.5 qd-tree

Qd-tree [36] is a high-dimensional data space partitioning binary tree, which is similar to the k-d tree. The qd-tree approaches the query task from a deep reinforcement learning-based framework. The structure of the qd-tree is built upon the binary tree. Each node in the tree contains data records that belong to the multidimensional data. The root node of the tree holds all the information for the entire dataset.

The strategy of querying entry in the tree is using a concept of routing data. Data is stored in the blocks on storage. Every record starts routing at the root of the tree, the same as most of the hierarchical structure, the record will recursively traverse down the tree. At each node, the tagged predicate p is evaluated on the record. Each predicate in the node does range comparison ($\{<, \leq, >, \geq\}$) and equality comparison ($=$). A record at the node does a series of range comparisons to decide which leaf (left or right) to go. Eventually, the record will be landed in the leaf node. The leaf node represents the physical location of the record on the disk. The block ID (BID) is stored along with the record on the leaf, indicating where the record belongs to.

The Greedy approach is beginning with a single root node that contains all the nodes. Then iterate through all the nodes to split the leaf node that is larger than $2b$ into two children nodes, the size of each of the children nodes is at least b . The Greedy approach has been reported [36] to work well to reach the local optimum and may be global suboptimality.

The Deep RL method is a black box learning process without assumption on the data distribution or query. The Deep RL agent Woodblock is built on top of the Markov Decision Process (MDP). It defines the state space to be any subspace of entire data space, and action space to be the set of splits. The Woodblock agent contains two parameterized networks: 1. *Policy Network*, which takes a state and emits a probability distribution over an action space. 2. *Value Network*, which estimates an expected cumulative reward for a given state. The learning process trained through a series of episodes:

1. Starts from a node off from the queue,
2. Evaluates the policy at current state,
3. Choice an action from the output distribution,
4. Applies this action on the current node to produce a new node.

A stopping condition is necessary for ending the episode training. As it is defined in the tree structure that the leaf node must contain at least b records, the agent will continue to make cuts on the current node if it contains more than b records. Overlapping region existing in the qd-tree. To handle such data redundancy in the leaf node, qd-tree

proposed a relaxed cutting condition that allows one of the children nodes to have a size smaller than b .

2.5.6 LISA

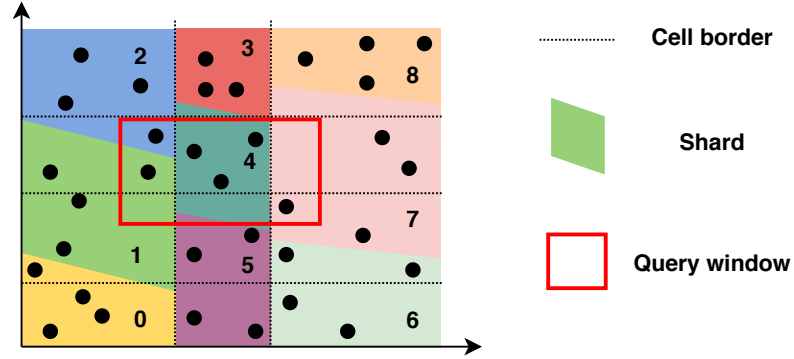


FIGURE 2.13: LISA

LISA proposed by Li [37], designated to generate data layout that allows using machine learning models to search in an arbitrary spatial dataset that is stored in the disk pages. Similar to many popular traditional approaches, LISA partitioning space into a series of grid cells, and numbering the cells along a sequence of axes. It builds a learned regression model SP to map the keys distribution into different shards, and assign an id for each shard. All shards preserve order concerning the corresponding interval in the mapped data. Keys in shards are stored in several disk pages, which means that one shard might contain keys from different pages. At the final stage, local models are required for page address lookup for the given query.

LISA also adopted a piecewise linear function as the shard prediction function. The piecewise linear function is a monotonic regression model, and it allows the mapped values to be sorted. Therefore, the operations for searching members of different partitions at the bottom level is efficient. The range query in LISA access to the keys that belong to the search region. Similar to traditional spatial indexes that applied **overlapping regions** technique, since the space partition might divide keys from different disk pages, it has to scan the overlapped pages in local models. For a KNN query, the given point and the number of return values k , LISA will estimate the distance bound base on the query. Then, the keys remaining within the bound are selected to perform repetitive range queries, until we get K nearest keys.

Compared to traditional such as R-Tree, KD-Tree, the advantage of LISA is that it benefits from the learned monotonic function. The learned monotonic function allows the mapping order of keys to be sorted, keys that to be appended to the structure are naturally sorted. Hence key lookup in neighbouring value and update handling in LISA is easy. Even though it still has overlapping regions in the local models, LISA still does less data updating than R-Tree.

2.5.7 ML-Index

The ML-Index [38] is another learned index for multidimensional data, and it supports point query, range query and KNN query. The ML-Index consists of two main components, the first part of the structure is to convert and scale the two-dimensional data into one dimension, and the second part is to use an RMI model as mentioned above by Kraska to learn the CDF of the data distribution.

The upper part of the algorithm is to scale a group of points that are similar to each other, and project into one-dimension. The first scaling method is to map distances of all points to a single reference point and use the distances as keys. It works for small datasets, but it is hard to scale to larger datasets. The second scaling method is called iDistance by Jagadish et al. [39], $key = i \times c + dist(O_i, d_l)$, where i is the index of the closest reference point O_i , and a constant c partitions the points into predefined ranges and stretches the range according to its value. The third method calculates the offset value as the scaling value $key = offset_i + dist(O_i, d_L)$, where the offset is calculated by sum of the radii of every partition that is based on their arbitrary reference points.

The second part of the index is to build a recursive learned index as described in Kraska's paper. Instead of building multiple stages models in the original paper, the ML-Index uses only two stages to reduce expected error by increasing the number of models in the second stage.

The point query procedure is the same as RMI, which uses scaled values to predict bounds containing the expected query result from $[pos - err, pos + err]$. The range query iterates through the reference points and calculates the closest and furthest point for each reference point from the query range. Then issue the model prediction to find

keys that are within the range. For the KNN query, the algorithm first predicts the position of the query key to locating the starting point of the search.

2.5.8 PolyFit

PolyFit [40] uses piecewise polynomial functions to process approximate range aggregate queries. Aggregate functions including COUNT, SUM, MIN and MAX. The key cumulative functions are extracted from those aggregate functions. Then the measured key cumulative function is used to construct the index.

The PolyFit index is built upon polynomial fitting for an interval of data. The polynomial function can achieve a better approximation of data distribution when compared to linear regression or piecewise linear regression. It is hard to fit the whole dataset with one polynomial function. Therefore, it also utilizes multiple polynomial functions to reduce the fitting error, which could get the approximate function more accurately. Memory consumption is also worth considering, one way is to use dynamic programming to minimize the number of polynomial functions while fitting the dataset well. Authors proposed Greedy Segmentation method (GS), it inserts one more key into the interval incrementally, then checks whether the interval can fulfill the bounded error constraint.

The PolyFit can be extended for two-dimensional queries. Similar to the one dimensional range aggregate queries, it estimates the key cumulative function with two keys for COUNT query. Then follow the same idea from GS to utilize the number of polynomial functions. However, the GS method takes $O(n^2)$ to obtain the minimum number of segmentations. They use a Quad-tree based method to get the segmentations. If a segmentation does not fulfill the error guarantee, the space partition can be continuously broken into four smaller partitions.

2.6 Research Gaps

Researchers have different perspectives on constructing access methods for multidimensional data, from a traditional hierarchical structure to a learned index.

The learned index is opening a new page for traditional data access methods by applying machine learning. The optimization of the machine-learned model seems to have a big

advantage over traditional hierarchical methods. The complexity of the model is scaled by the width h of the model and input length N : $O(hN)$. The linear model only using multiplication and addition operation: $w * x + b$, to calculate the prediction value. As a result, the Learned Index does not have searching phrases between stages. Consequently, it can achieve constant searching time complexity, while hierarchical structure B-Tree requires $O(\log n)$ searching time. On space complexity, the learned index also has an advantage over B-Tree. For inserting a new key, we might not need to re-train the entire model depending on the key. The new key could fall into a range at the next stage, the root stage model can predicate the same pointer to the next model that has the same range of keys.

However, there are some drawbacks to the learned index. For indexing, finding the min and max error of a position is more interesting to us. A typical machine learning model is to generalize the result, make it more suitable for real-world data. In the learned index, we wish to minimize the error as close to 0 as possible to achieve accurate results. It is different from a data science aspect problem, the fitting curve is preferred to overfit the dataset instead of generalizing the model.

The neural net model at the top stage does not produce the exact position of the key, requiring a 'last-mile' searching phase. The accuracy problem still exists in the 'last-mile' phase, which means the model still relies on hierarchical structure and traditional indexing method. The B-Tree is cache efficient. Machine learned models require all weights to compute a prediction, in which the multiplication and addition cost more memory.

Both Wang [19] and Nathan [20] represent multidimensional spatial data in a linear vector space. One of the difficulties of spatial data is the consistency of simultaneity ordered on different dimensions. For example, on a 2D spatial data, the growth of the Y-axis does not grow simultaneously along the X-axis.

The learned ZM index is trying to solve multidimensional problems in a one-dimensional way. The z-order curve or z-address is aiming to solve the consistency of simultaneously ordered spatial data on different dimensions. Interleaving bits of two-point value in a coordinate can convert the multidimensional data into one dimension. Applying the learned index model from Kafle [41], we could achieve a similar result as the learned B-Tree model.

Another problem with existing spatial indexes is that many of them lack support for spatial queries. The ZM index supports range queries but does not support KNN queries. Flood uses range scans for query processing, and it still does not support the KNN query. Qd-tree uses decision trees for routing data, which is also naturally difficult to support with range and KNN query.

Finally, the build time for the recursive model indexes can also be a problem. Generally, to train a simple linear regression model or neural network with few layers does not require a long training time. However, if a complex neural network is used in the root node and the subnodes, the build time for the entire recursive model index will be high. It might take up to hours to build a single index that can predict accurately, depending on the dataset.

TABLE 2.1: Comparison of Data Indexes

Index Type	Traditional Spatial Indexes						Learned Indexes					Learned Spatial Indexes							
	Grid File	MOLHPE	K-D-Tree	BSP-tree	Quadtree	R-Tree	RMI	ALEX	PGM-Index	RadixSpline	FTing-Tree	ZM	RSMI	Flood	Tsunami	qdt-tree	LISA	ML-Index	PolyFit
Model Name																			
Top-down Construction	●	●	●	●	●	-	●	●	-	-	-	●	●	●	●	-	●	●	●
Bottom-up Construction	-	-	-	-	-	●	-	-	●	●	●	-	-	-	-	●	-	-	-
Single-pass Training	●	●	●	●	●	●	●	●	●	●	●	●	-	-	-	-	-	-	-
Linear Structure	-	●	-	-	-	-	-	-	-	●	-	-	-	-	-	-	-	-	-
Hierarchical Structure	-	-	●	●	●	●	●	●	●	-	●	●	●	-	-	●	-	●	●
Grid Layout	●	-	-	-	-	-	-	-	-	-	-	-	-	●	●	-	●	-	●
CDF approximation	-	-	-	-	-	-	●	●	●	●	●	●	●	●	●	●	●	●	●
Piecewise Linear Model	-	-	-	-	-	-	-	-	●	-	●	-	-	●	-	-	●	-	-
Recursive Model	-	-	-	-	-	-	●	●	●	-	●	●	●	-	-	-	-	●	●
Reinforcement Learning	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	●	-	-	-
Require Sorted Data	-	-	-	-	-	-	●	●	●	●	●	●	●	-	-	●	●	●	●
Require Local Search	●	-	-	-	-	-	●	-	-	-	●	●	●	●	●	●	●	●	●
Transformation	-	●	-	-	-	-	-	-	-	-	-	●	●	-	-	-	-	-	-
Overlapping Regions	-	-	-	-	-	●	●	●	●	-	-	-	●	-	-	-	-	●	-
Clipping	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Multiple Layers	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Space Partitioning	●	-	-	-	-	-	-	-	-	-	-	-	-	●	●	-	●	-	●
Point Query	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Range Query	●	●	●	●	●	●	●	-	-	-	●	●	●	-	-	●	●	●	●
Nearest Neighbour Query	●	-	●	●	●	●	-	-	-	-	-	-	●	-	-	-	●	●	-

● = provides features; ● = partially provides features; - = does not provide features;

2.7 Summary

From this review, we have perceived the background knowledge and design flows of some popular traditional spatial indexes, the learned index for one-dimensional data

and current state-of-art learned spatial indexes. There are no perfect index methods, each method has its strengths and weaknesses.

To design a learned spatial index, we might consider some recipes from the current learned models and spatial indexes. The learned CDF curve can map an approximate shape of data distribution. Finding a fuzzy position of key only costs a complexity of prediction depending on the model. Transformation techniques in the spatial index provide a way to convert multidimensional spatial objects into one-dimensional data. Both ZM and RSMI applied Z-order space-filling curves in their models, so we can store spatial data in one-dimensional index fashion. Overlapping regions can handle complex operations for spatial queries, but overlapping areas can lead to searching on redundant paths in a hierarchical structure like R-Tree and even in the RMI.

The advantage[22] of the Flood is that the query performance can be boosted by automatically adjusting the number of partitions in each dimension. The more partitions in the data space, the fewer points are needed to be scanned. A drawback of Flood is that it is not optimized for skewed data. For a query that is non-uniform, it will scan a large range of points until reaching the result. Tsunami addresses this issue by correlating data to reduce the data skewness.

Most learned indexes are trying to optimize between space and time. The RMI looks for the position key regarding an upper bound and a lower bound. This bounded range can be large if the model under-fits the data. However, to train a well-fitted model, it requires more training time and more complex models. A complex model can be more expensive when inference during searching. The PGM-index offers a flexible turning strategy for either optimizing space for a query time constraint or optimizing query time for a space constraint. It is done by controlling the number of segments in the model. Radix spline's approach to this problem is to build a separate Radix table to look up a range of query points. The size of the Radix table can grow larger with the increase of query precision.

Integrating the learned model with traditional spatial indexes achieved good performance. The Flood follows the idea from the Grid File, it partitions the data layout into numbers of Grid cells, then narrows down the search region. The Flood learns from the data distribution and builds the model to determine the location of query points.

The Tsunami further improves the skewed data handling in Flood by using functional mapping and conditional CDFs.

Chapter 3

Proposed Indexing Model

Before we integrate machine learning models on conventional spatial data indexes, we should review the current state-of-art learned indexes and spatial indexes. The most important idea that comes from the learned indexes is to use learned models to predict the possible location of the key. In RMI, a query point is passed into a sequence of learned models to get the final range of location. ALEX improved on RMI using a *Gapped Array* or a *Packed Array* to store data points, instead of searching in a single array. The RMI is more memory efficient compared to ALEX since all the data points are allocated in a contiguous array, whereas ALEX allocates data points at a different location in memory. The PGM-index uses piecewise linear models to create segments of linear models. The number of segments controls the trade-off between time and space.

We must concede that the learned models can not predict the exact position, there always are errors that come with the predicted value. Reducing the error will rely on more complex computations, but inference cost will also be expensive. The question is how many trade-offs we can afford to build either a time-efficient spatial index or space-efficient spatial index.

Difficulties of spatial data is the consistency of simultaneity ordered on different dimensions. The ZM's approach is to combine the Z-order space-filling curve and the RMI model, but it does not guarantee to find data points accurately and it does not support k nearest neighbour query. The RSMI also adopted Z-order curves to apply to spatial space partitioning. The structure of the RSMI is also similar to the RMI and the R-Tree, because of the recursive model and space partitioning.

Current learned spatial indexes have a complex hierarchical structure. The hierarchical structure allows the index to handle complex spatial queries. The problem with adopting learned models to the hierarchical structure is the prediction value can have a large error range. The error propagation can be solved by introducing a sub-model to refine search regions. However, it can still cause problems when handling skewed data and error bounds are large. A large error bound can produce overhead on the number of access of sub-trees. As a result, it can degrade query performance.

3.1 Design Goal

Before investigating methods to build an efficient spatial index, the design goal of the learned spatial index meet the following requirements:

- The spatial index must retrieve query points **accurately**.
- The spatial index can handle **Point Match Query**, **Range Query** and the **Nearest Neighbour Query**.
- The data distribution should be mapped by a **cumulative distribution function** (CDF).

Firstly, retrieving the query point is the most important function in the spatial index. The Point Match Query is defined in 2.1.1, it retrieves a data point from the index for a given pair of coordinates. Therefore, accurate point retrieval is a basic and important function of a spatial index. Secondly, one of the major differences between a one-dimensional index and a spatial index is that a spatial index is able to handle spatial specific queries. The Point Match Query is the foundation query for supporting various spatial queries.

Efficiency is also an important criterion for measuring the performance of a spatial index. As mentioned above, it is hard to balance the trade-off between time and space. The ideal scenario is to achieve a fast query time with a low memory cost for a spatial index. However, in reality, we sometimes have to sacrifice one for another. The designed spatial index should exploit the space-time trade-off, and try to optimize from at least one aspect.

Overall, we are focusing on two aspects when designing a spatial index: query result accuracy, complete functionality of spatial query and performance improvement. The following sections describe how a *Learned Hashmap* can be useful in a spatial index and achieve these goals.

3.2 The Learned Spatial Hashmap

3.2.1 Overview

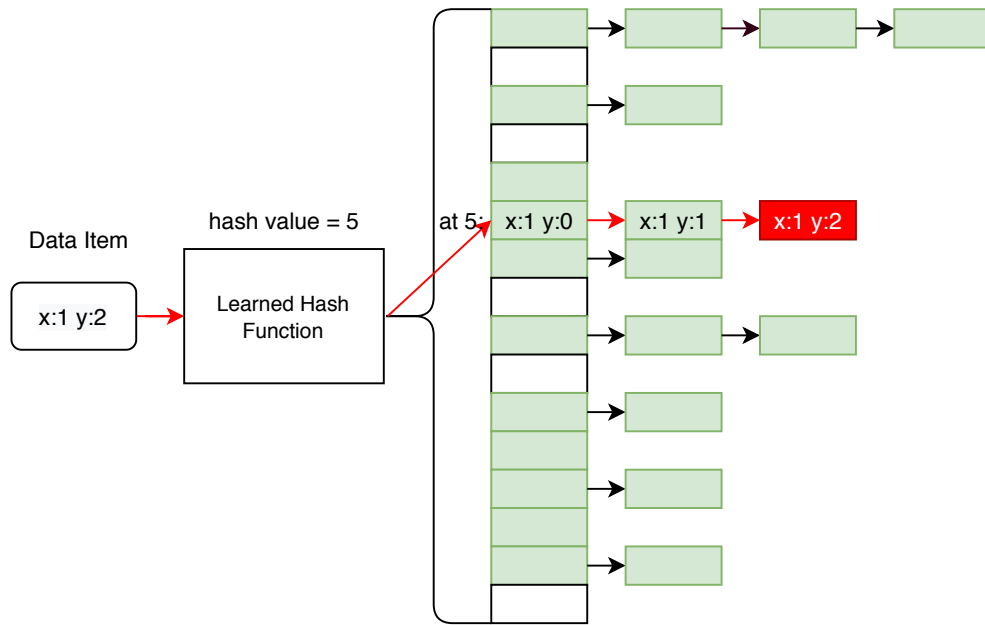


FIGURE 3.1: The structure of the Learned Spatial Hashmap(LSPH). The search flow in LSPH takes one value from the selected dimension, then the learned hash function predicts the index to the estimated location in the hashmap.

The Learned Spatial Hashmap(LSPH) is an efficient learned data index method for storing spatial data and handling spatial queries. The structure of the LSPH is straightforward, the main component of the index is a learned hashmap with chaining, as shown in Figure 3.1. The learned hashmap is a hashmap with a learned CDF as its hash function [17]. The CDF learned from a machine learning model and used it as a hash function in the hashmap. Linked lists are used in the LSPH for chaining. The design of the LSPH building process makes it suitable for storing spatial data points. The build process of the LSPH is shown as follows:

1. **Dimension Selection:** Choosing one dimension from the dataset for model training.
2. **Model Training:** A learned model trains on the selected dimension of data.
3. **Building of the LSPH:** Creating and Inserting to the Hashmap based on the prediction value.

The learned model in LSPH trained only on a single dimension of two-dimensional data points. We select a single dimension of value from a coordinate data pair: $\{n|\forall n \in E.p\}$, where E is every element in the dataset, and p is selected dimension. The dimension selection is based on the divergent of the values. A more divergent set of values can produce more divergent prediction values from the same model. Therefore, it will optimize the loading factor of the hashmap, thus improving the efficiency of the lookup speed.

Using simple linear regression to train on data along a selected dimension p , with corresponding ranks of the data points, we can estimate a range of candidate positions of the query data point on the p dimension, and trained model m .

Memory with a size of the model prediction range will be allocated in the hashmap: $\hat{y}_{max} - \hat{y}_{min}$. The building process of the hashmap is relatively simple, the trained model m infers the input values from the same selected dimension p . Then the trained model yields a prediction value that can be treated as the hash value for the hashmap. Finally, the data point can be inserted into the hashmap at the index of the hash value.

The LSPH is similar to a regular hashmap. It is simple but yet powerful. Some major differences distinguish itself from a regular hashmap, and suitable for storing and searching spatial objects. The LSPH uses only values from one dimension for the entire building procedure. Some properties about the learned hashmap guarantee the performance and accuracy of querying results, it will be discussed in the Section 3.2.3.

3.2.2 Single Dimensional Index Building

In the LSPH, the building process and the initial search range relies on one dimension of data. It is much easier to handle one-dimensional data than multidimensional data. The

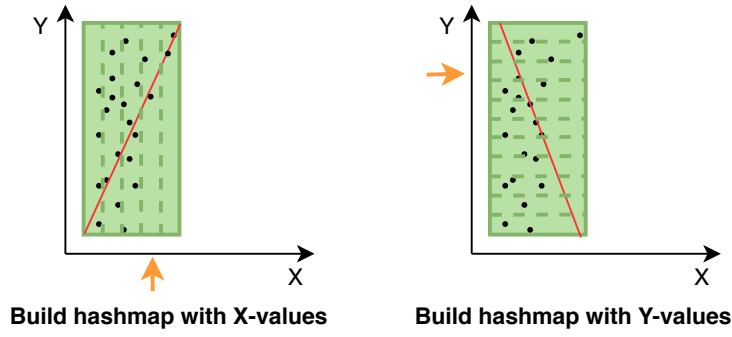


FIGURE 3.2: Difference between building data with x-values and y-values

selection of the dimension for training is determined by the variance. Values on which dimension with a larger variance will be selected for model training. Given a series of two dimensional data $S = (x_i, y_i)$, compare the variance between $Var(x) \forall x \subseteq S$ and $Var(y) \forall y \subseteq S$, and dimension p selected by the greater value between these two variances.

Transformation(space-filling curves, endpoint transformation and midpoint transformation) tends to be a popular choice for transferring multidimensional data into one-dimensional data in many traditional spatial indexes. In spatial data, it is hard to map two-dimensional data into one-dimensional data while preserving the spatial proximity [9]. The space-filling curves (SFC) integrate both x-value and y-value into one value, to preserve the locality of points in the space. The calculated SFC values are used to rank the data points. Two closely-ranked data points in sorted order are also located close to each other in the spatial space. However, it requires extra calculations, and more importantly, a learned model can not produce an accurate prediction for the SFC values.

There are some benefits for a spatial index to indexing from a single dimension of data. Training on a single dimension of values preserves the original information about the location, and it is already a one-dimensional data available. As a result, the LSPH has a fast query performance, since there is no pre-calculation for the transformation, values from original data can be used directly for inference. The building performance is also fast because of no extra calculation during the building process.

Training on only a single dimension might also avoid potential data skewness. In most spatial datasets, skewness is most likely to happen in only one dimension. It means if one dimension is skewed, we can check if another dimension is skewed, and the training

dimension p is chosen from the less skewed out of two. Although there are some special cases, which values in both dimensions are skewed (such as two clusters of data points). It only creates computation overhead during searching in the chaining, and will not affect the accuracy of query results.

The selection of p dimension can affect the space utility in the LSPH. A load factor in a hashmap is defined as: $Load\ Factor = \frac{n}{k}$, where n is *number of entries occupied in the hashmap* and k is *size of hashmap*. The performance of the hashmap is largely decided by the load factor [42]. For example, a hashmap with a load factor of 0.5 represents only half of the slots in the hashmap that are loaded with values. The hashmap in the LSPH is created with a size of prediction range, so in general, a high load factor can reduce chaining overhead and increase space utilization. For example, in Figure 3.2, building the learned hashmap with y-values is much more efficient than building from x-values, because it can create more slots in the hashmap, and reduce the number of nodes in the linked list. As a result, the performance can also be improved from the reduction of computational overhead in lookup nodes from linked list chainings.

3.2.3 Learning a Hashmap

The learned hashmap has been mentioned in Kraska's paper [41]. He suggested that the learned hash map is no better than traditional hashmaps such as Cuckoo hashing in terms of solving conflict and distributing key mapping. There are some properties of the learned hashmap not mentioned in Kraska's paper, which can be strengths for handling spatial query operations.

Hashmap is a two-dimensional data structure: The hashtable and linked list chaining are two-dimensional, which is naturally suitable for routing two-dimensional data in those two components with values that belong to different dimensions.

Small hash collision rate: According to Kraska, the learned hashmap can reduce hash conflict by up to 77%. The logic behind it is that a randomized hash function is a trend to generate a random hash value, whereas the learned CDF can produce a hash value that grows with an increase of the input value. Therefore, it is less likely to have hash collisions with learned hash functions.

Sorted head nodes: Since the prediction results monotonically grow with the input value, the order of head nodes in the learned hashmap is sorted. In a set of sorted data, for any values x' that are greater than a given input value x , the predicted value from the learned CDF function $y' = \text{CDF}(x')$ is also greater than the predicted value $y = \text{CDF}(x)$. Kristo [43] implemented a sorting algorithm based on this property of the learned hashmap.

Load factor depends on model and data distribution: The trained CDF maps all values from the dataset to the index value in the hashmap. Therefore, a good model choice and also the distribution of the data can affect the load factor. Whereas, the load factor in a hashmap that uses a random hash function is predetermined. This is critical to both memory space efficiency and query time performance. A high load factor can reduce the length of chaining to utilize the existing memory space in the hashmap, and also reduce the lookup overhead in the chaining.

Cost of hash function depends on the model: Query performance also depends on the complexity of the learned model. In general, we want to have a model that can map the approximate distribution of data. However, if the model is complicated, then the query performance at inference can be affected. Rather, a high load factor in the hashmap is more important than having a computationally more expansive model for the query performance.

Guarantee to find value(completeness): Most learned indexes are attempting to search the position of the value key directly, but the prediction value always comes with an error bound. The model in LSPH does not predict the position of the index directly but is used as a hash value to access the position in the hashmap and potential chaining at that location. Therefore, the LSPH can guarantee to find the position of the key or the chaining where the key belongs to with one shot.

Overall, these properties provide a foundation for the LSPH. The structure of the hashmap, monotonicity of head node and small collision rate are crucial for query accuracy, query performance and implementing spatial query algorithms. Compared to other learned indexes, the accuracy of a point query relies on the structure of the hashmap, rather than model prediction. Hence, the LSPH can guarantee the search result accuracy.

3.2.4 Model Choice

The query performance is largely dependent on the model and the distribution of data. Because the prediction value is used as a hash value instead of the actual position of the key in the hashmap, the model does not need to be well fitted with the dataset. Neural networks are better for fitting data well, but it also takes a long time for training, and more expensive at inference. A linear model can do a good job, because the computation is much simpler. With a trained weight w and bias b , linear function $y = wx + b$ can map an approximate distribution of data, and it can achieve a good performance at inference. Therefore, linear regression is chosen as the learned model for the LSPH, and the least square method is used for training the two parameters. As a result, the building process of LSPH is also fast.

3.3 Basic Operations

The basic operations including insert, delete and update in the LSPH are also almost identical to a regular hashmap. Rather than changing the way a hashmap is used, we changed the data item content and made it suitable for handling spatial objects. Points in spatial space usually contain the coordinates of its location, which is x, y . As mentioned above, a dimension p is chosen for model training, and it is also used for all operations of the LSPH.

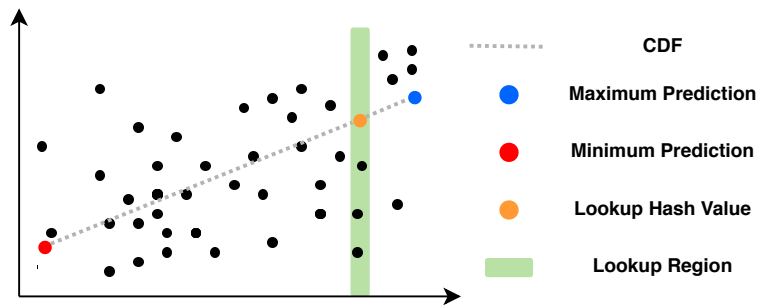


FIGURE 3.3: Lookup process on the dataset. Using the trained linear regression model to calculate the lookup hash value, then scanning the lookup region base on the lookup hash value.

Search: Insertion and deletion heavily rely on searching data items in a data index. Searching an item in the LSPH requires hashing and chaining. $x - value$ or $y - value$ is chosen for hashing, based on the chosen p . A hash value is generated by the prediction

value from the learned model: $h : V \rightarrow H$, where $H : \Leftrightarrow CDF$. The size of the LSPH is determined by the maximum and minimum prediction values: $m = \hat{y}_{max} - \hat{y}_{min}$, so when searching in the LSPH, the prediction value must subtract the minimum prediction value among the entire dataset to fit in the hashmap. The default rounding method to rounding from the float prediction value to integer is floor rounding. We found that rounding the closest integer can increase the variability of the resulting hash values, and raise the load factor.

Separate Chaining: When a slot is occupied in the hashmap, the hash collision happens. A linked list will be created for chaining to solve the hash collision, any items that collided at the index in the hashmap will be appended to the end of the linked list. The length of the chaining is also determined by the load factor of the hashmap. Since the size of the hashmap is corresponding to the range of all prediction values, the higher the load factor is, the shorter the chaining will be.

Update: To update an item, we need to determine whether the hash value of the data item is within the hashmap size of m . If the hash value is within the range, then we can insert the item as usual. If it is out of the range, the hashmap needs to be updated. The hash function model will be recalculated, and all existing items will be rehashed from the updated model. Since the linear model is used and the insertion process is relatively fast compared to neural network training, the rehash process is not considered expensive.

3.4 Query Processing

The structure of LSPH adopted from the learned hashmap. What makes it able to manage spatial data is spatial query algorithms. Three types of spatial queries that are currently supported by LSPH: *Point Query*, *Range Query*, and *k Nearest Neighbour query*. These three queries are focused on point object handling, but also developed a bounding box query algorithm that supports more complex spatial objects other than point objects.

The most important and frequently executed operation is lookup a key. Key lookup in LSPH requires a hashing and iterative searching in the separate chaining. The performance of the lookup query is bounded by the length of chaining since the cost of hashing

and indexing is always $O(1)$. The total cost of searching in the LSPH is $O(1 + \frac{n}{k})$, where $\frac{n}{k}$ is the load factor.

3.4.1 Point Query

Algorithm 1: Point Query

Input: Query point $q \ni x_q, y_q$

Output: Result query point q_{result} that stored in the structure

```

 $x \leftarrow x_q, y \leftarrow y_q;$ 
hashKey  $\leftarrow$  hashFunc(sort_by_x?x : y);
tempNode  $\leftarrow$  table[hashkey];
while tempNode is not empty do
    if tempNode.y = y then
        if tempNode.x = x then
            return tempNode;
        end
    else
        tempNode  $\leftarrow$  getNext();
    end
end
return null;
  
```

The procedure of spatial point lookup consists of two parts: 1. single dimensional value hashing 2. local search in chaining. The steps are defined in Algorithm 1.

As described in previous sections, LSPH training used values on only one dimension. Therefore, a boolean variable $sort_{by_x}$, which is deducted from the single-dimensional index building, is determining which dimension p will be used in point lookup. Next, the selected value of x will be hashed from a learned model. Since a linear model is mapping the approximate shape of the distribution of the dataset, the prediction/hash function is

$$\hat{y} = x * w + b + 0.5f - (x < 0) \quad (3.1)$$

, where $0.5f - (x < 0)$ is rounding float numbers to the closest integer value including negative numbers. Finally, a local iterative search is to find the actual position of

the target tuple of spatial points, by comparing the value from another dimension and validate the selected value on p dimension.

The performance of the *Point Query* is great for a uniformly distributed dataset since the querying process only requires one multiplication, one addition and local iteration with a size of $\frac{n}{k}$. For all keys that share the same hash value will be placed in the same chaining. Therefore, the LSPH will guarantee to find the point.

3.4.2 Range Query

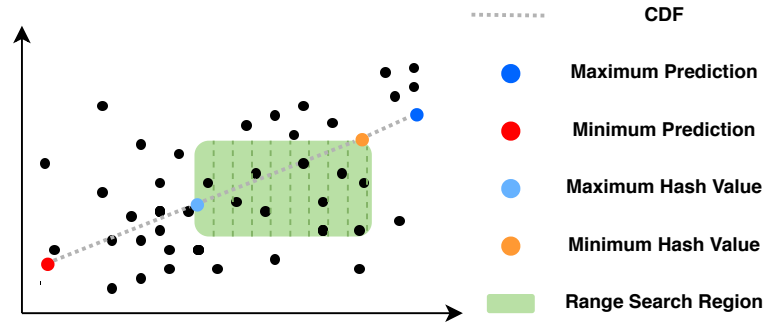


FIGURE 3.4: *Range Query* in LSPH. Calculate the maximum and minimum hash value to get the search region on the hashmap.

Range Query, in addition to *Point Query*, requires to specify the search range and scan through every data item if qualified. As illustrated in Figure 3.4, first to find the *hashKeyMin* and *hashKeyMax* for the minimum and maximum x value, or minimum and maximum y value, depending on the *sortBy_x*. Then we can scan through every chain within the search region in the hashmap, to check if a data item from a node is in the searching range.

One of the properties of the monotonic growing learned map is that the values in the head nodes are sorted. Therefore, the *hashKeyMin* and *hashKeyMax* is defining the bounds of the search region. Any index value on the hashmap that exceeds the *hashKeyMin* and *hashKeyMax* will be out of the target region. At each chaining bucket, it just needs to check if points are within the range and if true then append the point to the

result list. The time complexity for the range query depends on the size of the search region and the length of chaining.

Algorithm 2: Range Query

Input: Range of search window r

Output: Set of result points S

```

 $xMin, xMax, yMin, yMax \leftarrow r;$ 
 $hashKeyMin = hashFunc(sort\_by\_x?xMin : yMin);$ 
 $hashKeyMax = hashFunc(sort\_by\_x?xMax : yMax);$ 
 $S \leftarrow \emptyset;$ 
for  $hashkey$  in from  $hashKeyMin$  to  $hashKeyMax$  do
     $tempNode \leftarrow table[hashkey];$ 
    while  $tempNode$  is not empty do
        if  $tempNode.x$  in  $r$  then
            append  $tempNode$  to  $S$ ;
        end
         $tempNode \leftarrow getNext();$ 
    end
end
return  $S$ ;
  
```

3.4.3 k Nearest Neighbours Query

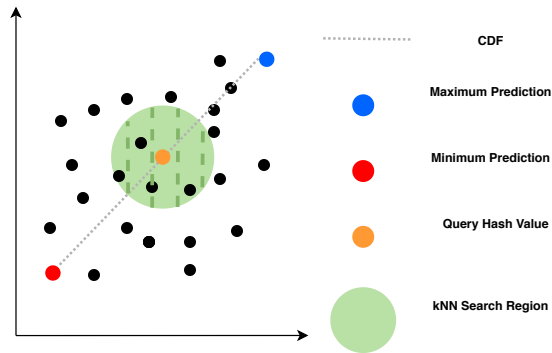


FIGURE 3.5: kNN Query in LSPH. First to calculate the query hash value, then expand to neighbour buckets.

The k Nearest Neighbours Query finds k number of point neighbours that have the closest distance to query point q . The algorithm starts with the same hashing process as in *Point Query* and *Range Query*. It first compares the distances with the current bucket

of chaining, and stores the candidate into a priority queue that prioritizes based on the closest distance. Once it has been explored at the current index, it can expand the exploration to neighbour indexes. Since the indexes are sorted based on the monotonic growth of head nodes, the neighbour indexes are containing the closest nodes to the query node. Hence, the next step of the searching process will use two pointers, which one is doing an incremental search, and another one is doing a decremental search along the hashmap. The expansion search process will stop until the minimum of distances that collected at the current index is greater than or equal to the maximum distances collected previously. Finally, k items pop out of the queue and return as the result of k nearest neighbours search

Algorithm 3: k Nearest Neighbours Query

Input: Query point q ; Number of nearest neighbours k

Output: Set of result points S

```

 $x \leftarrow x_q, y \leftarrow y_q;$ 
hashKey  $\leftarrow$  hashFunc(sort_by_x?x : y);
S  $\leftarrow$  Priority Queue;
def localSearch(hashKey):
    tempNode  $\leftarrow$  table[hashKey];
    while tempNode is not empty do
        if dist(tempNode, q)  $\leq$  minDist then
            | append tempNode to S
        end
        tempNode  $\leftarrow$  getNext();
    end
i  $\leftarrow$  hashKey + 1; j  $\leftarrow$  hashKey - 1;
previousMaxDist  $\leftarrow$  S.Max(); localMinDist  $\leftarrow$  0;
while localMinDist < previousMaxDist do
    previousMaxDist  $\leftarrow$  previous S.Max();
    localSearch(i); localSearch(j);
    localMinDist  $\leftarrow$  S.Min();
    i++; j--;
end
return S with first  $k$  elements;

```

The time complexity of Algorithm 3 is $O(1 + m \frac{n}{k})$, where m is the size of expansion. Unlike hierarchical structures such as B-Tree, the linear structure allows reducing redundant data scans caused by overlapping regions. Hence, LSPH can achieve more efficient kNN searching by fast hash computation and scanning regions reduction.

3.5 Minimum Bounding Box Handling

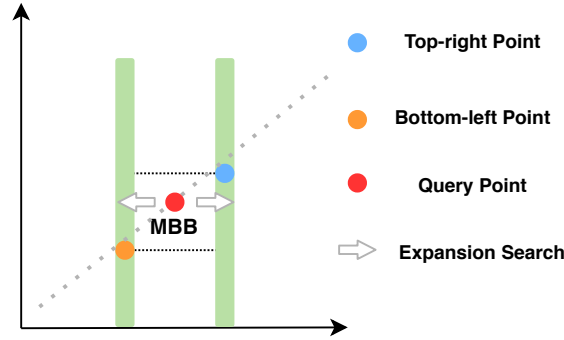


FIGURE 3.6: Store the top-right and bottom-right corners of a MBB in the hashmap

The current process discussed above is all based on the point object in spatial data. We also implemented a method that allows the spatial index to handle more complex spatial objects such as line segments and polygons. All two-dimensional objects can be bounded by their minimum bounding box (MBB), which is a rectangle shape box defined by the minimum and maximum width and height of the object. The representation of MBB simplifies the structure of spatial objects. The x and y coordinate of two diagonal corner pointers are taken to store MBB in the LSPH.

The fundamental operation of MBB supported by the LSPH is for a given query point q , the search result returns the MBB that the q belongs to. To check if a point is within a particular MBB, both of the corner points need to be compared with the query point. Therefore, we have to insert both of the corner points of the same MBB into the LSPH. Each data item contains the x and y coordinate of the corresponding corner, and a pointer to extra information of the data item that contains coordinates from both corners.

To retrieve the MBB from the query point, the procedure is similar to the k Nearest Neighbours Query:

1. Calculate the hash value from query point q .
2. Search at the current index, which is the hash value.
3. Iterative searching in the linked list at the current index.
 - (a) Retrieve coordinates of two corner pointers from data items in the linked list.
 - (b) Check if the query point is within the currently searching MBB (defined by the two corner coordinates).
 - (c) If it is within the MBB region, return that MBB.
4. If no MBB found at the current index, expand the index to both sides (add and subtract the pointer). Repeat the search process.

The performance of MBB searching is also similar to the performance of kNN query, except that MBB does not require a k number of return values. Since we insert both the lower-left corner and upper-right corner into the hashmap, the searching pointer can compare to either corner arbitrarily. A pointer stores coordinate values of both corners are pointing to the hashmap item, so no matter which corner that the current searching pointer is comparing with, it can access the full information of both corners.

There is one issue that can occur when two MBBs are overlapping, but the polygons themselves do not overlap, the result will only return the first found MBB. To find the finer search result over polygon spatial objects, the information of the shape of the polygon is needed to calculate if the search point is within the polygon.

3.6 Summary

Overall, the Learned Spatial Hashmap is a spatial index that applied model learning techniques. Although previous research has been carried out on learned models in spatial indexes, no single study has investigated applying learned hashmaps to handle spatial data. A large part of the study in this approach has devoted to the properties of learned hashmaps. The property of a two-dimensional data structure is a natural side effect of chaining in a hashmap, which provides convenience for handling two-dimensional spatial data. The monotonic CDF is also an important feature in LSPH structure, which allows

the inserted value to be sorted in the head node. Since the head nodes are sorted in the hashmap, consequently the support for range queries and KNN queries becomes easy.

The LSHP is also flexible to choose one of two axes to train the CDF. Building a learned index from a selected dimension can utilize hashmap slot space, and reduce the length of the linked list. This simple technique not only increases the load factor of the hashmap, but it reduces the lookup time in the linked list as well. Therefore, the performance of LSHP can be optimized by selecting the right axis.

In conclusion, LSHP is trying to balance the complexity of space and time. The search process finds a position of a key from a hashing function that only takes one multiplication and one addition and iterative search in the chaining linked list. The space complexity depends on the range of minimum and maximum prediction value, which is depending on the data distribution. Many learned indexes predict the position with an error bound. The benefit of this method is that it only needs to lookup keys in the disk page rather than store keys in the index structure. However, they do not guarantee the accuracy of results 100%. LSPH searches for the location of a query key based on the hash value, and a hash value is guaranteed to be the same from the same hash function. Therefore, as a tradeoff for memory, LSPH is guaranteed to have 100% query accuracy.

Chapter 4

Experiments

In this section, I will discuss how I evaluate the LSH with some real world data, and comparison with other spatial indexes. Finally, present the outcome that is measured from various aspects.

4.1 Experimental Setup

For evaluation, the LSPH is implemented in C++ and has been tested on a machine with 2.2GHz 6-core Intel Core i7 CPU and 16GB memory.

TABLE 4.1: Datasets

Name	Size	No. of data pairs	Data type
post	2.2M	123,593	Geographic Coordinates
g2	18K	2,048	Synthetic Clusters
osm	3.0G	114,932,854	Geographic Coordinates

4.1.1 Datasets

As shown in the Table 4.1 above, we have two real world datasets the `post` [44] and `osm` [45], and one synthetic dataset the `g2` [46].



FIGURE 4.1: Distribution of **post**, **g2** and **osm** datasets

- **post**: The **post** dataset contains 123,593 coordinates of post offices across North America. The distribution of this dataset is more evenly.
- **g2**: The **g2** is a synthetic cluster dataset that contains 2,048 points that are computed from Gaussian clusters with 2 dimensions and the standard deviation set to 10.
- **osm**: The dataset **osm** contains 114,932,854 coordinates in Australia and Oceania from OpenStreetMap. This dataset contains large amount of data points, and its distribution is highly skewed.

Figure 4.1 illustrates the general size and distribution of each dataset. The three datasets represent three different data distribution settings: the **post** dataset has relatively uniform data distribution, and with over 100,000 data points making it common scenarios for spatial data indexing; the **g2** is a synthetic two-dimensional dataset contains two clusters of data points, which is skewed on the both of axis, to test the skewness handling among the indexes; the **osm** is another real-world dataset with a sizable amount of data, and combine the heavily skewed data distribution, to test the performance under skewed high volume real-world scenario.

4.1.2 Competitors

We are comparing LSPH with one traditional spatial index **R-Tree** [16] and one learned spatial index **ZM** [19]

R-Tree: A templated C++ version implemented by Greg Douglas [47] is used for comparison. The partitioning method in this version uses the Quadratic-Cost splitting algorithm from the original R-Tree paper [16].

Learned ZM index: For the ZM model, the two-dimensional datasets convert to 64-bit one-dimensional data, using the Z-order curve transformation. We use an RMI implementation from the author of RadixSpline and SOSD [34, 48].

4.1.3 Evaluation Metrics

We measure the performance of all indexes from four aspects: memory consumption during runtime, time cost of index building, time cost of query and accuracy of the query result. The runtime memory reflects the space complexity of the index method, and the measurement is using the Linux command `time -v` to get the maximum resident set size of allocated memory size during runtime. Time measurements in the experiment were using the C++ `std::chrono` library to record the time during index building and query execution in nanoseconds.

In the experiment, we use sorted arrays to build all indexes. Each index takes two data values as a pair of keys for lookup query, and if it exists in the index, the returned results should match with the query keys. We execute the lookup query overall key pairs in all three datasets. Note that there are some duplicates in the dataset, and they have been filtered out before indexes building. Therefore, the input number of the query should match with the returned results if the index can locate the key without errors.

4.2 Performance Results

Overall performance: We first compare the point query performance of all indexes on three datasets. Figure 4.2 illustrates the overall performance of all indexes. The local search in the last step search in ZM used binary search, the total query time record the search range produced by a linear model plus the binary search execution time. R-Tree uses minimum bounding boxes to store all spatial objects. Therefore, for the point query in R-Tree, we set the minimum point and maximum point of a bounding

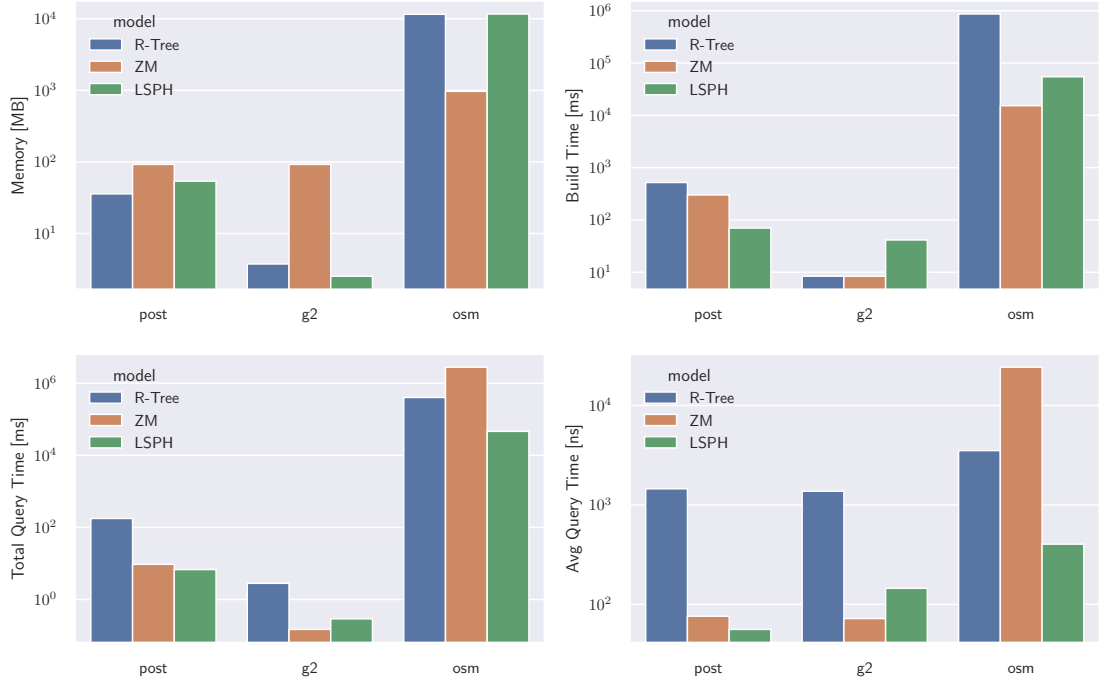


FIGURE 4.2: Point query performance comparison on runtime memory, build time, total query time and average query time for all three spatial indexes

box as the same pair of data values. As we see from the result, the LSPH has several advantages in memory and lookup query performance under certain conditions.

Runtime Memory: Comparing our method with R-Tree and ZM in terms of runtime memory consumption, as shown in the top-left diagram in Figure 4.2. LSPH consumes a similar amount of memory as R-Tree for all three sets of data. ZM costs more memory than LSPH and R-Tree in the `post` and `g2` dataset but costs less in the `osm` dataset. The ZM implementation requires running a rust code and converting it into C++ code, so it might consume more during conversion. The increase of numbers in the dataset causes a consistent growth in LSPH and R-Tree. For ZM, the memory cost is maintained at a certain level without concerning too much on the dataset size.

Build Time: The build time of LSPH on a uniformly distributed dataset is outperforming the other two indexes. However, it cost more on skewed dataset `g2` and `osm`. Since we are using a linear model for the ZM, the build time is much faster than as if we were training a neural network. The reason for the slower build time of LSPH on the skewed dataset is that skewed data creates more chaining in the hashmap, which increases the node access during insertion.

Query Time: Lookup query time showed a lookup performance advantage of LSPH. Note the log time scale in the total query time on the bottom of Figure 4.2, we also add an average query time for comparison. LSPH performed the best out of three indexes in the post and osm, and second place in the g2. The lookup speed in LSPH performs best in uniformly distributed datasets, it even can reach exact $O(1)$ if in a perfectly distributed dataset. The skewed data creates linked list overflow that causes some performance drop, but the average lookup speed is still kept at a slower growth compared to the ZM. Although the lookup speed is much slower on an extreme cluster case where values on both axes are skewed, the LSPH still does a better job than ZM in terms of accuracy, which will be discussed in the next section.

TABLE 4.2: Point Query Accuracy

	R-Tree	ZM	LSPH
post	100%	99.99%	100%
g2	100%	99.71%	100%
osm	100%	96.84%	100%

Accuracy: Table 4.2 shows the accuracy of point query results from three indexes. R-Tree and LSPH can find every single query point from all three datasets. Some query points are not matching the query points in the ZM, due to the prediction positions are bounded by errors. If the errors have large ranges, for example, in the skewed datasets g2 and osm, some predictions will exceed the maximum error ranges.

4.2.1 Range Query Performance

LSPH outperformed the other two indexes in point query speed. However, the range query can be considered as a weakness of the LSPH. The result is shown in Figure 4.3. The range query performance was tested on the **post** dataset, since the **post** is more uniformly distributed that helps various window sizes can retrieve good amounts of data. The position of the range query window is located at the centre of the dataset, the selectivity of window size grows from 0 - 100

The performance of range queries of LSPH is worse than the R-Tree and the ZM. R-Tree is a hierarchical structure that its nodes contain the containment relationship between

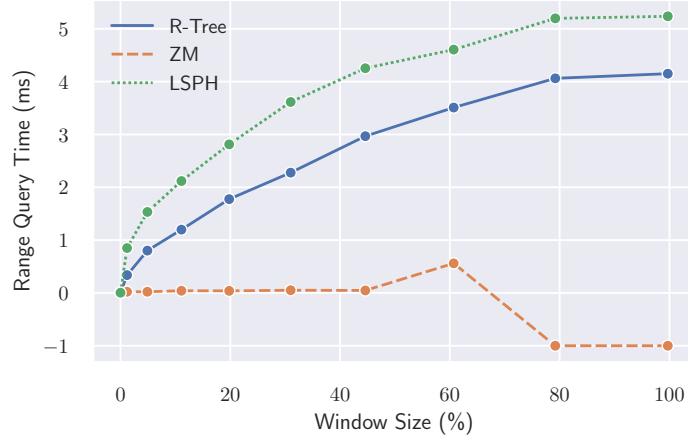


FIGURE 4.3: Range Query performance on dataset *post*. The window size is corresponding to the query selectivity, which is between 0.0-100.0%. The -1 in ZM represents the return results are out of search range.

the minimum bounding box and spatial object. In range queries, the results are more likely to be found in the upper part of the tree, then check its child nodes. Therefore, R-Tree performed better at range query tasks.

The overall performance of the ZM index is fast. However, in the last two window sizes 80% and 100%, the ZM can not retrieve points from the corresponding range, due to the predicted position being out of the maximum position. Hence, the ZM can not guarantee the accuracy of the search result.

Our approach LSPH for the range query takes the longest time among the three. The range query algorithm in LSPH requires iterative search between a minimum hash value and a maximum hash value. However, it has to check every single point in the hashmap chaining. This means that the search is not only the head nodes in the hashmap but also every data item that is stored in the linked list. Therefore, it creates a lot of computational overhead.

4.2.2 KNN Query Performance

TABLE 4.3: Average KNN Query Time

	post	g2	osm
average query time	16.407ms	0.008ms	434.478ms

We generate 1000 queries from randomly selected points from each dataset to test KNN query performance in LSPH. Figure 4.2.2 shows the average total query time result from the 1000 queries. Since our KNN algorithm scans through the entire dataset and returns the top k value from a priority queue, the speed is approximately the same for all k values. This is not the most efficient algorithm to find k closest neighbours. However, it only takes 16ms to scan the entire `post` dataset, 0.008ms for the `g2`, and 434.478ms for the `osm`. This performance of KNN query time is similar to the time performance to scan the entire dataset. Due to the scope of the study is to focus on the two-dimensional data objects, the KNN query performance cannot provide a comprehensive view of the capability of LSPH. Further improvements need to be made to the KNN algorithm to complete this task more efficiently.

4.2.3 Load Factor and Query Performance

TABLE 4.4: Load factor’s effect on the query performance in LSPH

	<code>post</code>	<code>g2</code>	<code>osm</code>
load factor ¹	0.638%	0.059%	0.159%
average query time	52ns	155ns	403ns

¹ load factor = $\frac{n}{k}$, where n is the total number of entries, and k is the size of hashmap;

The load factor in the LSPH is dependent on data distribution and modelling. Comparing the load factors from all three datasets and their average query time is shown in the Figure 4.2.3. Overall, our statistics are consistent with the theoretical hypothesis that higher load factor in the hashmap will expect a better query performance. The `post` dataset has the best average query time with the highest load factor. The load factor in the `g2` is much lower than the load factor in the `osm`, but the average query time in the `g2` is still higher than the `osm`. The inconsistency of the result is because of differences between quantity of datasets. The `osm` contains significantly more data points than the `g2`, which takes more time in linked list search for LSPH. The load factor in `g2` is $10\times$ smaller than the load factor in `post`, and the average query time is $3\times$ slower than `post`.

The load factor in `osm` is $4\times$ smaller than the load factor in `post`, and the average query time is $7.75\times$ slower than `post`.

Therefore, a higher load factor in the learned hashmap can achieve a better query performance. Besides the data distribution, models can also affect the load factor. Linear regression is great for mapping the approximate distribution of data and maintaining a very low computational cost. However, linear regression is not able to illustrate more complex curve shapes. It is worth considering piecewise monotonic regression in the future.

4.3 Summary

In general, the performance of LSPH is well-balanced between query time and memory assumption. Especially, the point query time outperformed the traditional index R-Tree by up to $8\times$, and over the ZM index by up to $40\times$. LSPH shows that its advantage in single point finding, in which the simplicity of its structure is aiming to find the two-dimensional data point straight from hash function and hashmap chaining scans. It is much more efficient than certain complex data routing algorithm designs in hierarchical structures. The runtime memory of LSPH is similar to traditional R-Tree, but storage memory is not as efficient as the ZM index. However, LSPH has shown more robust performance in accuracy when compared to the ZM index. The build time of LSPH is also efficient in certain datasets. Since we used a linear model in the ZM index, the build time of the ZM index is better than LSPH in two of the three datasets. If a neural network is used in the experiments, it could increase the training time of the ZM index.

LSPH demonstrated that it combines performance advantages from traditional data indexes and novel learned indexes. It is more efficient than R-Tree, and more robust than the ZM index. However, LSPH is not perfect. One of the drawbacks is that LSPH does not handle range queries and KNN queries efficiently. The weak range query performance in LSPH might be due to the linear expansion scan in a flat structure, whereas the R-Tree can take advantage of hierarchical relationships to execute the query more efficiently. The KNN algorithm is also not efficient cause the scanning process takes the length of the entire dataset. There also is a limitation for LSPH handling skewed data distributions. Selecting single-dimensional values in LSPH can avoid certain cases

of skewed data in two-dimensional space, but it performs poorly in cluster shape data distribution that both axes are skewed. Even the performance of LSPH on the skewed data is declined, but it will not be affected by the large error bounds in terms of accuracy, which can still guarantee to find query results 100%.

Chapter 5

Conclusions and Future Work

The primary objective of this study is to develop an understanding of learned spatial data indexes and propose a new approach of a spatial index with the learned model technique. Firstly, we have examined some previous works on the spatial data management techniques and spatial indexes based on that. We have also reviewed and compared some of the state-of-art learned one-dimensional indexes and learned spatial indexes. The conclusions obtained from the comparison suggest that a learned index model maps a data distribution that could potentially reduce the time complexity of query time, and as a benefit from directly locating the position of keys from models, the storage memory size of indexes is also lower than traditional data indexes. Notably, there are still some limitations in existing learning indexes. The RMI does not guarantee to find the query result 100% accurately without hyperparameter tuning for a particular dataset. Lack of spatial query support in ZM and Flood also limit the functionalities of real-world application as a spatial data index. Neural network training in RMI, ZM and reinforcement learning in the qd-tree takes a long time to build, which can be a problem in real-world applications.

Secondly, we proposed a new index method for handling spatial data and queries, the Learned Spatial Hashmap (LSPH). It integrates ideas that come from data science and machine learning to combine a learned CDF with a traditional hashmap. As a result of the learned CDF, the two-dimensional point query performance in LSPH achieved up to $8\times$ speedup compared to R-Tree and $40\times$ speedup compared to ZM index. The runtime memory cost of LSPH is similar to the R-Tree, which is overall balanced between

space and time complexity. More importantly, LSPH addresses some disadvantages from existing learned model indexes. LSPH supports range queries, KNN queries, and can handle minimum bounding boxes. It is also worthwhile mentioning that LSPH guarantees query results to be 100% accurate.

5.1 Limitations

Our proposed spatial index method has a potential influence on studies in learned data indexes and real-world database applications. Folmer et al. [49] proposed a Learning Spatial Buckets method for handling spatial data, which uses a similar idea of single dimension index building from our approach. However, there are not many studies that investigate the use of learned hashmap or other kinds of hashmap in the spatial index. Therefore, LSPH contributes a new idea to studies in spatial data management systems.

LSPH provides the same result retrieving accuracy as the traditional spatial indexes and supports various spatial queries. LSPH has also been examined to outperform one of the popular traditional spatial data indexes and one of the state-of-art learned spatial indexes in real-world datasets.

It must also be mentioned that LSHP is still facing many challenges:

Skewness handling: Notably, we have obtained a poor performance result from LSPH on the g2 dataset. The single-dimensional index building has limitations when handling data skewness on both axes. Therefore, we would like to investigate the methods that can improve performance on skewed data.

Memory consumption: LSHP might consume more memory in certain cases, because the size of the hashmap is determined by the minimum prediction value and the maximum prediction value, and even if there are very few data items between this prediction range, the hashmap will still allocate the same amount of slots.

Range query and KNN query performance: Different from most spatial indexes, LSPH is a two-dimensional linear structure, rather than a hierarchical structure. The faster range query performance of the hierarchical structure indexes results from the containment relationship between parent nodes and child nodes. The KNN algorithm is also not very efficient by scanning the entire dataset. Future work is going to focus

on building relationships between entries by experimenting with different structures and propose a more efficient KNN algorithm.

Model Choice: Linear regression works well as the learned CDF in the hashmap. However, a major limitation of linear regression is it can not fit data distribution very well. Even the hashmap does not require a perfectly fitted CDF, but a more fitted model will result in better performance in terms of memory utilization and lookup speed. It is worth considering a piecewise linear function or interpolated polynomial model [50].

5.2 Future Work

Some of the issues listed from the limitations above are related to each other. Skewed data requires a more fitted model, and as a result of a more fitted model, the memory consumption can be reduced due to increase of hashmap load factor. We can investigate this problem from two aspects: model selection and data transformation.

It is worth considering a piecewise linear function or interpolated polynomial model [50]. Piecewise linear models have achieved good performance results from experiments in learned indexes such as PGM-index, FITing-index and Flood [20, 33, 35]. The piecewise linear model constructs a set of points and builds the segments of models according to the data range between two breakpoints. The resulting model learns the distribution of data better and differentiates the prediction values. The interpolated polynomial model approximates the regression function from small training data. Such interpolated models generalize data better than linear regression, and it also requires much less time to train and less memory space compared to neural networks.

Tsunami [22] addresses data skewness problems from statistical tools: functional mapping and conditional CDFs to correlate skewed data. Functional mapping creates monotonically correlations between x values and y values. If the x value increases, the y value also increases in one direction. The transformed y values obtain a semantically linear relationship with the x values, so a linear regression function is enough for making predictions. Conditional CDFs maps x values dependent on the y values, in uniform $CDF(X|Y)$. Both methods are intended to construct a data correlation between two dimensions.

Piecewise linear function and interpolated polynomial model or data correlation can both improve LSPH to handle skewed data better, and predict the accurate desired label values more effectively.

In summary, LSPH contributes a new idea to the study of the learned multidimensional index, and improves the robustness and efficiency of spatial data management. Future work is going to focus on improving the skewed data handling in LSPH and exploring more to utilize the efficiency of LSPH.

Bibliography

- [1] O. Günther and A. Buchmann, “Research issues in spatial databases,” *ACM Sigmod Record*, vol. 19, no. 4, pp. 61–68, 1990.
- [2] G. M. Morton, “A computer oriented geodetic data base and a new technique in file sequencing,”
- [3] H.-P. Kriegel, S. Brecheisen, P. Kröger, M. Pfeifle, and M. Schubert, “Using sets of feature vectors for similarity search on voxelized cad objects,” in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 587–598, 2003.
- [4] D.-R. Liu, S. Shekhar, and M. Coyle, “An evaluation of access methods for spatial networks,” in *Proceedings of the Second ACM Workshop on Advances in Geographic Information Systems*, pp. 93–78, 1994.
- [5] B. Borah and D. K. Bhattacharyya, “An improved sampling-based dbscan for large spatial databases,” in *International conference on intelligent sensing and information processing, 2004. proceedings of*, pp. 92–96, IEEE, 2004.
- [6] J. Adhikary, J. Han, and K. Koperski, “Knowledge discovery in spatial databases-progress and challenges,” *School of Computing Science, Simon Fraser University*, 1996.
- [7] D. Mantel and U. Lipeck, “Matching cartographic objects in spatial databases,” *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences-ISPRS Archives 35 Part B4 (2004)*, vol. 35, pp. 172–176, 2004.
- [8] H. D. Tagare, C. C. Jaffe, and J. Duncan, “Medical image databases: A content-based retrieval approach,” *Journal of the American Medical Informatics Association*, vol. 4, no. 3, pp. 184–198, 1997.

- [9] V. Gaede and O. Günther, “Multidimensional access methods,” *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 170–231, 1998.
- [10] B. C. Ooi, *Efficient query processing in geographic information systems*, vol. 471. Springer Science & Business Media, 1990.
- [11] P.-Å. Larson, “Linear hashing with partial expansions,” in *VLDB*, vol. 6, pp. 224–232, 1980.
- [12] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, “Extendible hashing—a fast access method for dynamic files,” *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 3, pp. 315–344, 1979.
- [13] R. Bayer and E. McCreight, “Organization and maintenance of large ordered indexes,” in *Software pioneers*, pp. 245–262, Springer, 2002.
- [14] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- [15] H. Samet, “The quadtree and related hierarchical data structures,” *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.
- [16] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pp. 47–57, 1984.
- [17] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, “The case for learned index structures,” in *Proceedings of the 2018 International Conference on Management of Data*, pp. 489–504, 2018.
- [18] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan kaufmann, 2016.
- [19] H. Wang, X. Fu, J. Xu, and H. Lu, “Learned index for spatial queries,” in *2019 20th IEEE International Conference on Mobile Data Management (MDM)*, pp. 569–574, IEEE, 2019.
- [20] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, “Learning multi-dimensional indexes,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 985–1000, 2020.

- [21] J. Qi, G. Liu, C. S. Jensen, and L. Kulik, “Effectively learning spatial indices,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2341–2354, 2020.
- [22] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, “Tsunami: A learned multi-dimensional index for correlated data and skewed workloads,” *arXiv preprint arXiv:2006.13282*, 2020.
- [23] C. Faloutsos and Y. Rong, “Dot: A spatial access method using fractals,” in *Proceedings of the Seventh International Conference on Data Engineering*, pp. 152–159, IEEE Computer Society, 1991.
- [24] J. A. Orenstein and T. H. Merrett, “A class of data structures for associative searching,” in *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pp. 181–190, 1984.
- [25] I. Kamel and C. Faloutsos, “Hilbert R-tree - An Improved R-tree using Fractals.,” *VLDB*, 1994.
- [26] D. Hilbert, “Über die stetige abbildung einer linie auf ein flächenstück,” in *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes*, pp. 1–2, Springer, 1935.
- [27] J. A. Orenstein, “Redundancy in spatial databases,” *ACM SIGMOD Record*, vol. 18, no. 2, pp. 295–305, 1989.
- [28] J. T. Robinson, “The kdb-tree: a search structure for large multidimensional dynamic indexes,” in *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pp. 10–18, 1981.
- [29] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, “The grid file: An adaptable, symmetric multikey file structure,” *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 1, pp. 38–71, 1984.
- [30] H.-P. Kriegel and B. Seeger, “Multidimensional order preserving linear hashing with partial expansions,” in *International Conference on Database Theory*, pp. 203–220, Springer, 1986.
- [31] H. Fuchs, Z. M. Kedem, and B. F. Naylor, “On visible surface generation by a priori tree structures,” in *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pp. 124–133, 1980.

- [32] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, *et al.*, “Alex: an updatable adaptive learned index,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 969–984, 2020.
- [33] P. Ferragina and G. Vinciguerra, “The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds,” *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, 2020.
- [34] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, “RadixSpline: A Single-Pass Learned Index,” *arXiv.org*, Apr. 2020.
- [35] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, “Fiting-tree: A data-aware index structure,” in *Proceedings of the 2019 International Conference on Management of Data*, pp. 1189–1206, 2019.
- [36] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 193–208, 2020.
- [37] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, “Lisa: A learned index structure for spatial data,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 2119–2133, 2020.
- [38] A. Davitkova, E. Milchevski, and S. Michel, “The ml-index: A multidimensional, learned index for point, range, and nearest-neighbor queries,” in *EDBT*, pp. 407–410, 2020.
- [39] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, “idistance: An adaptive b+-tree based indexing method for nearest neighbor search,” *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 364–397, 2005.
- [40] Z. Li, T. N. Chan, M. L. Yiu, and C. S. Jensen, “Polyfit: Polynomial-based indexing approach for fast approximate range aggregate queries,” *arXiv preprint arXiv:2003.08031*, 2020.

- [41] K. Kafle and C. Kanan, “Visual question answering: Datasets, algorithms, and future challenges,” *Computer Vision and Image Understanding*, vol. 163, pp. 3–20, Oct. 2017.
- [42] “java.util class hashmap,” Nov 2015.
- [43] A. Kristo, K. Vaidya, U. Çetintemel, S. Misra, and T. Kraska, “The case for a learned sorting algorithm,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 1001–1016, 2020.
- [44] Y. Theodoridis, “The r-tree-portal,” 2003.
- [45] OpenStreetMap contributors, “Australia and Oceania retrieved from <https://download.geofabrik.de/>.” <https://www.openstreetmap.org>, 2017.
- [46] P. F. R. Mariescu-Istodor and C. Zhong, “Xnn graph,” *Joint Int. Workshop on Structural, Syntactic, and Statistical Pattern Recognition*, vol. LNCS 10029, pp. 207–217, 2016.
- [47] G. Douglas, “Templated c++ version of r-tree,” 2004.
- [48] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, “Sosd: A benchmark for learned indexes,” *NeurIPS Workshop on Machine Learning for Systems*, 2019.
- [49] M. Folmer, R. Neumann, and T. Karunanithi, “A Learned Bucket Index Supporting Spatial Queries,” *projekter.aau.dk*.
- [50] N. F. Setiawan, B. I. Rubinstein, and R. Borovica-Gajic, “Function interpolation for learned index structures,” in *Australasian Database Conference*, pp. 68–80, Springer, 2020.