UNIVERSITY OF CALIFORNIA
RIVERSIDE

Query Optimization for Big Spatial Databases using Theoretical Analysis and Machine Learning

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Tin Khac Vu

June 2021

Dissertation Committee:

Dr. Ahmed Eldawy, Chairperson
Dr. Vassilis J. Tsotras
Dr. Vagelis Hristidis
Dr. Amr Magdy

The Dissertation of Tin Khac Vu is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my advisor, Dr. Ahmed Eldawy, without whose support, I would not have been able to complete this journey. Thank you for your continued guidance, support and encouragement from the beginning to the end of my PhD study. I especially would like to thank my dissertation committee members: Dr. Vassilis J. Tsotras, Dr. Vagelis Hristidis and Dr. Amr Magdy, for reviewing my dissertation. I also would like to thank to my collaborators: Dr. Alberto Belussi and Dr. Sara Migliorini from University of Verona, Italy, for your support in multiple research projects. Finally, thank you to my colleagues at the Big Data Lab: Saheli Ghosh, Samriddhi Singla, Akil Sevim and Puloma Katiyar, for all the discussions we had inside and outside the lab.

The text of this dissertation, in part, is a reprint of the material as it appears in ACM Transactions on Spatial Algorithms and Systems (TSAS) 2020, Frontiers in Big Data 2020, SpatialGems 2019, DeepSpatial 2020, ACM SIGSPATIAL 2018, 2019 and 2020. The co-author Ahmed Eldawy listed in that publication directed and supervised the research which forms the basis for this dissertation.

This research was partially supported by National Science Foundation under grants IIS-1838222 and CNS-1924694. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of the National Science Foundation.

To my parents and my twin brothers for all the support.

ABSTRACT OF THE DISSERTATION

Query Optimization for Big Spatial Databases using Theoretical Analysis and Machine Learning

by

Tin Khac Vu

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2021
Dr. Ahmed Eldawy, Chairperson

Spatial data is being produced at increasing rates from various sources such as mobile applications and satellite data. For example, there is an average of 500 million tweets sent every day from users at different spatial locations. NASA EOSDIS adds about 6.4 TB of data to its archives every day. These data sources urged the research community and industry to develop new systems for big spatial data. Regardless of their architecture, one of the fundamental requirements of query optimization in these systems is to spatially partition the data efficiently across machines. Existing spatial databases rely on traditional index search structures such as R-tree, STR, Kd-tree, Quad-tree, etc. These approaches are not always suitable with the demands of current big data applications. My dissertation proposes new partitioning techniques based on theoretical analysis. First, this work introduces a balanced spatial partitioning, termed R*-Grove, which provides load balanced partitions with high spatial quality. Second, this dissertation proposes an incremental spatial partitioning framework for distributed file systems that allows high ingestion rates and efficient spatial analytical queries.

The proposed systems above are built based on theoretical analysis of spatial query performance. In recent years, there are many works that employ the power of machine learning techniques to address classical problems in big data systems. Motivated by the success of these approaches, my dissertation also proposes some machine learning based systems to solve several query optimization problems in spatial databases such as spatial partitioning, selectivity estimation, and spatial join cost estimation problem. The experimental results show that machine learning is a promising approach to efficiently solve query optimization problems in big spatial databases.

Contents

List of Figures **xii**

List of Tables **xv**

1 Introduction **1**

2 Balanced spatial partitioning for big data **6**
2.1 Introduction . . . 6
2.2 Related Work . . . 10
2.3 Background . . . 13
2.3.1 R*-tree . . . 13
2.3.2 Sample-based Partitioning Workflow . . . 13
2.3.3 Quality Metrics . . . 15
2.4 R*-Grove Partitioning . . . 17
2.4.1 R*-tree-based Partitioning . . . 18
2.4.2 Load Balancing for Partitions with Equal-size records . . . 20
2.4.3 Load Balancing for Datasets with Variable-size Records . . . 26
2.4.4 Implementation Considerations . . . 32
2.5 Case Studies . . . 34
2.5.1 Indexing . . . 34
2.5.2 Range Query . . . 35
2.5.3 Spatial Join . . . 35
2.6 Experiments . . . 36
2.6.1 Experimental Setup . . . 37
2.6.2 Effectiveness of the proposed improvements in R*-Grove . . . 38
2.6.3 Results Overview . . . 40
2.6.4 Partition quality . . . 42
2.6.5 Spatial query performance . . . 47
2.6.6 Performance on larger datasets and multi-dimensional data . . . 49
2.7 Conclusion . . . 53

3 Using deep learning for big spatial data partitioning 54
3.1 Introduction . . . 54
3.2 Problem definition . . . 61
3.3 Training Phase . . . 65
3.3.1 Training set generation . . . 66
3.3.2 Dataset Summarization . . . 70
3.3.3 Evaluation of Quality Metrics . . . 80
3.3.4 Model Training . . . 87
3.4 Application Phase . . . 90
3.4.1 Overview . . . 90
3.4.2 Application of the model in a real system . . . 91
3.5 Experiments . . . 95
3.5.1 Experimental Setup . . . 96
3.5.2 Algorithm Selection . . . 100
3.5.3 Model Selection . . . 101
3.5.4 Model Accuracy . . . 103
3.5.5 The effect of histogram size . . . 109
3.5.6 Stability of Quality Metrics . . . 111
3.5.7 Performance of the Summarization Phase . . . 112
3.5.8 Training Data with Skewed Shapes . . . 114
3.6 Related work . . . 116
3.6.1 Spatial Partitioning . . . 116
3.6.2 Data Summarization . . . 117
3.6.3 Deep Learning . . . 118
3.7 Conclusion . . . 118

4 Incremental partitioning for efficient spatial analytics 120
4.1 Introduction . . . 120
4.2 A generic incremental partitioning framework . . . 126
4.2.1 Partition layout . . . 128
4.2.2 Data flushing phase . . . 128
4.2.3 Partition selection . . . 129
4.2.4 Partition reorganization . . . 129
4.2.5 Multiversion Control / Garbage Collection . . . 130
4.3 Partition optimization problem . . . 131
4.3.1 Preliminaries and problem definition . . . 131
4.3.2 The NP-Hardness of the problem . . . 133
4.4 Cost-benefit analysis of the partition selection process . . . 135
4.4.1 Range query cost model in HDFS . . . 137
4.4.2 Reorganization benefit . . . 139
4.5 Proposed Incremental Partitioning Algorithms . . . 144
4.5.1 R*-Tree-inspired partitioning (R*P) . . . 145
4.5.2 LSM-tree-inspired partitioning (LSMP) . . . 146
4.5.3 Cost-based partitioning (CBP) . . . 148
4.6 Experiments . . . 150

4.6.1 Cost model and benefit model validation . . . 152
4.6.2 Performance of proposed partitioning algorithms . . . 153
4.6.3 Comparison with existing spatial data systems . . . 156
4.7 Related Work . . . 158
4.8 Conclusion . . . 160

5 Selectivity estimation with predicted error and response time 162
5.1 Introduction . . . 162
5.2 Related Work . . . 164
5.3 Selectivity Estimation with Predicted Error and Response Time . . . 166
5.3.1 Problem definition . . . 166
5.3.2 Prediction with mixed data sources . . . 167
5.3.3 DeepSampling architecture . . . 169
5.4 Preliminary results . . . 171
5.4.1 Experimental setup . . . 171
5.4.2 Accuracy Prediction . . . 173
5.4.3 Sampling Ratio Estimation . . . 174
5.4.4 Effect of histogram resolution . . . 174
5.5 Summary and future work . . . 175

6 A learned query optimizer for spatial join 177
6.1 Introduction . . . 177
6.2 Related Work . . . 181
6.3 Overview of SJML . . . 183
6.4 Model training and testing . . . 185
6.4.1 Training Data Generation . . . 185
6.4.2 Feature Extraction . . . 188
6.4.3 Training Set Preparation . . . 196
6.4.4 Definition of the models . . . 197
6.5 Experiments . . . 200
6.5.1 Experimental Setup . . . 200
6.5.2 Baseline methods . . . 201
6.5.3 Feature Selection . . . 202
6.5.4 Training Set Generation . . . 205
6.5.5 Spatial join cost estimation model . . . 207
6.6 Conclusion . . . 208

7 Conclusions 210

Bibliography 211

A Spatial data generator 227
A.1 Introduction . . . 227
A.2 Data Generators . . . 229
A.2.1 Uniform . . . 232

A.2.2 Diagonal . . . 232
A.2.3 Gaussian . . . 233
A.2.4 Sierpinski triangle . . . 234
A.2.5 Bit distribution . . . 235
A.2.6 Parcel distribution . . . 236
A.3 Post Transformations . . . 238
A.3.1 Affine Transformation . . . 238
A.3.2 Compound Datasets . . . 239
A.3.3 Identifying Datasets . . . 239
A.4 Spider: Web-based Spatial Data Generator . . . 240

List of Figures

2.1 Comparison between STR and R*-Grove 8
2.2 The sampling-based partitioning process 14
2.3 Load balancing for datasets with variable-size records 29
2.4 Auxiliary search structure for R*-Grove 32
2.5 Partition quality with variable-record-size dataset in R*-Grove and its two variants, R*-tree-black-box and R*-tree-gray-box 39
2.6 The advantages of R*-Grove when compared to existing partitioning techniques 40
2.7 Indexing performance and partition quality of R*-Grove and other partitioning techniques in `OSM-Nodes` datasets with similar-size records. 41
2.8 Indexing performance and partition quality of R*-Grove and other partitioning techniques in `OSM-Objects` dataset with variable-size records. 43
2.9 Indexing performance and partition quality of R*-Grove in `OSM-Objects` datasets with different sampling ratios. 45
2.10 Indexing performance and partition quality of R*-Grove in `OSM-Objects` datasets with different minimum splitting ratios. 46
2.11 Spatial join performance in R*-Grove and STR partitioning 47
2.12 The scalability of R*-Grove partitioning in Spark and Hadoop 49
2.13 Indexing performance and partition quality of R*-Grove and other partitioning techniques on `synthetic` multi-dimensional dataset. 50
2.14 Indexing performance and partition quality of R*-Grove and other partitioning techniques on multi-dimensional `NYC-Taxi` dataset. 52

3.1 The workflow of the proposed solution 57
3.2 Partitions produced by applying different techniques (1st row: regular grid, 2nd row: QuadTree-based grid, 3rd row: RTree-based grid) for both synthetic and real datasets. 66
3.3 Example of distributions contained in the training set. 67
3.4 Example of rectangles contained in the training set. (a) regular rectangles of different sizes, (b) oblong rectangles of different sizes. 70

3.5 Example of Box-counting plot for: (a-c) a synthetic dataset with the distribution of a Sierpinski's triangle, (d-e) a synthetic dataset containing a diagonal line with buffer and (g-i) a real-world dataset representing the primary roads of Australia. 75
3.6 Example of Moran's index computation on the Diagonal Line dataset (a). In (b) the considered histogram is shown. In (c) the cells of the histogram are labelled with their count (# geometries they intersect) and two cells are highlighted together with their adjacent cells. Finally in (d) the contribution of the cells to the computation of the numerator (N) and the denominator (D) of the Moran's index is shown. 77
3.7 The relation of partition quality and query performance 84
3.8 A sample fully connected model that we adopt in the chapter. In this work, we vary the number of hidden (blue) layers and the number of hidden units per layer 87
3.9 Flow chart of the optimization task. 91
3.10 Algorithm comparison between CNN and FC model 100
3.11 Tuning model parameters for the **histogram-based** summarization technique 101
3.12 Tuning model parameters for the **fractal-based** summarization technique . . 102
3.13 Model accuracy when train and test on synthetic and real datasets 104
3.14 Model accuracy when varying the ratio of train data / test data 106
3.15 Confusion matrices for different index techniques and considering different quality measures. 108
3.16 The effect of histogram size 109
3.17 Stability of quality metrics 110
3.18 Summarization performance 113
3.19 Experiments on training data with skewed shapes 115

4.1 Our proposed system in the context of other systems 123
4.2 Overview of incremental partitioning of big spatial data 127
4.3 Different relationships of a range query with a partition. 136
4.4 Different scenarios when reorganizing a partition or a group of partitions to reduce the estimated cost and improve the quality 137
4.5 Group benefit: (1) Cluster the overlapping partitions (2) Compute total benefit of disjoint groups. 143
4.6 Data flushing in different partitions 146
4.7 Partition optimization in different partitioning techniques 147
4.8 Cost model and benefit model validation 152
4.9 Performance of proposed implementations with insert-only workload 154
4.10 Performance of proposed implementations with insert-delete workload 154
4.11 Performance comparison of CBP with existing techniques: AsterixDB, GeoMesa, Beast 156
4.12 Different approaches for accessing big spatial data 159

5.1 Trade-off between accuracy and efficiency in AQP 163
5.2 DeepSampling addresses critical problems on existing AQP systems 166

5.3 How sampling ratio (σ) relates to accuracy (α) . . . 168
5.4 DeepSampling architecture . . . 169
5.5 Accuracy of DeepSampling and linear regression . . . 172
5.6 Effect of histogram resolution . . . 175

6.1 Related work in join optimization . . . 181
6.2 Overview of SJML . . . 183
6.3 Join selectivity and execution time in different cardinality scales . . . 188
6.4 The confusion matrix of the algorithm selection model . . . 203
6.5 Effect of training distributions in M1 and M2 model . . . 206
6.6 The impact of join result size and number of MBR tests to spatial join cost . 207

A.1 Overview of the spatial generator . . . 229
A.2 Examples of the first three distributions . . . 232
A.3 Examples of the last three distributions . . . 235
A.4 Example of compound dataset obtained by combining two different Gaussian distributions and one diagonal distribution. . . . 239
A.5 Spider Web: Online service for spatial data generation . . . 240

List of Tables

2.1 Datasets for experiments 37

3.1 Execution of the DJ in SpatialHadoop with different kinds of indexes (i.e., Gr = regular grid, Qt = Quadtree, Rt = R-tree) and different distributions of the datasets (i.e., Uni = uniform distribution, Skw = skewed distribution). # tasks is the total number of map tasks, AVG time is the average time for a map task, and %RSD is the relative standard deviation for the running time of map tasks. 55
3.2 Symbols 71
3.3 Computation of the Moran's indexes M_0, M_1 and percentage of empty cells for the datasets presented in Fig.3.2: (a) a synthetic dataset with the distribution of a Sierpinski's triangle, (b) a synthetic dataset containing a diagonal line with buffer, (c) a synthetic dataset containing a double cluster and (d) a real-world dataset representing the primary roads of USA. 79
3.4 Correlation between quality metrics and query performance 83
3.5 Execution times of the spatial join operations in seconds. All possible combinations of partitioning techniques applied to datasets D_{PRoads} and D_{Builds} are considered. 95
3.6 Experiments and performance metrics 96
3.7 Training set generation: datasets of different distributions generated for the training phase. 98
3.8 Real datasets used for testing. 99
3.9 The independence of the best index in terms of total area and dataset size . . 112

4.1 Table of notations 131

5.1 Parameters for the selectivity estimation (SE) query 171

6.1 Different possibilities of feature selection. 197
6.2 Accuracy of the theoretical model in predicting the join selectivity and the best two algorithms in ranking. 203
6.3 Effect of feature selection to join selectivity estimation models 203
6.4 Effect of feature selection to MBR selectivity estimation models 204

6.5 Effect of feature selection to algorithm selection models 204
6.6 Regression score of the linear model between join result, number of MBR tests and join time 208

A.1 Identifiers for the six sample datasets shown in this paper. For simplicity, all of them use the identity transformation (affine) matrix 239

Chapter 1

Introduction

Spatial data is being produced at increasing rates from various sources such as mobile applications[102], satellite imagery [81], social networks [135], and VGI [94], IoT sensors, autonomous vehicles. For example, there is an average of 500 million tweets sent every day from users at different spatial locations [179]. NASA EOSDIS adds about 6.4 TB of data to its archives every day [84]. These data sources urged the research community and industry to develop new systems for big spatial data. In these systems, the query optimizer plays an important role, which aims to help users to work efficiently with a huge amount of data. Although most of existing distributed systems were designed to handle the large datasets, they are not optimized yet for spatial data processing. Motivated by this gap, the research projects in this dissertation mainly focus on the query optimization problems of spatial data processing on distributed systems.

Regardless of their architecture, all spatial data systems need an essential preliminary step that partitions the data across machines before the execution can be parallelized. This

is also known as *global indexing* [74]. A common method that was first introduced in SpatialHadoop [71], is the sample-based STR partitioner. This method picks a small sample of the input to determine its distribution, packs this sample using the STR packing algorithm [125], and then uses the boundaries of the leaf nodes to partition the entire data. Despite their wide use, the existing partitioning techniques all suffer from one or more of the following three limitations. First, some partitioning techniques (STR, Kd-tree) prioritize load balance over spatial quality which results in suboptimal partitions. Second, they could produce partitions that do not fill the HDFS blocks in which they are stored. Third, when records are highly variant in size, all existing techniques end up with extremely unbalanced partitions. Chapter 2 proposes a novel spatial partitioning technique for big data, termed R*-Grove, which completely addresses all of three aforementioned limitations.

Although R*-Grove is the most advanced partitioning technique for big spatial data, it is not a silver bullet that is suitable for all kind of spatial datasets. Unfortunately, there is no single partitioning technique that all the systems agree on. Rather, most of these systems provide a wide range of spatial partitioning techniques and it is up to the user to choose an appropriate one. Past studies showed that the spatial partitioning approach is critical to the performance of many spatial analytic operations such as indexing [62], computational geometry [68], visualization [82], spatial joins [72], kNN joins [134], and others. Choosing an appropriate spatial partitioning technique is a very challenging and complicated problem. Chapter 3 introduces an auto-select partitioning system based on deep learning, which is able to assist spatial data systems to efficiently determine the best partitioning scheme for their datasets.

The proposed works in previous chapters could speed up the performance of existing spatial data systems. However, all of them are only designed to work with *static* datasets. In many applications, data is not only large in volume, but it is also continuously growing and changing. When organizing spatial data, there are two main approaches, depending on the query processing needs of the system. If the focus is on highly selective queries (e.g. point look-up, top-k) data is indexed; in the first approach (termed *record-level*) every record finds its exact position in the index structure (e.g., R-tree [97], quad-tree [165]) so that the highly selective query will access very few records using the index structure. While such queries are fast, there is an overhead in maintaining the index. On the other hand, if the focus is on analytical queries (e.g. aggregates, spatial joins [160], kNN joins [134], polygon union [69], convex hull, Voronoi diagram [126], DBSCAN [101], etc.), it is better to partition the data at a coarser granularity. Here, the exact record position is not important; rather records are organized in partitions (e.g., hash or range partitioning). After a record's partition is identified, its position within the partition is not important. This is because an analytical query will read all records in each partition that it accesses. As a result, the second approach (termed *block-level*) incurs less overhead in creating and maintaining the partitions, compared to the index maintenance of the record-level approach. Record-level approaches using the LSM-Tree include Apache HBase[100], Accumulo[8], AsterixDB [16], MD-HBase [147], Parallel Secondo [130], BBoxDB [144], and GeoMesa [106], among others. However, there is no block-level approach that can support incremental spatial datasets with efficient analytical queries. Chapter 4 introduces a generic framework for incremental spatial partitioning, which completely fulfills that need of spatial data systems.

The next step after spatial data partitioning is to execute spatial queries. The main method that data scientists use to process large-scale datasets is through *interactive exploratory queries*; i.e., an ad-hoc query that should be answered in a fraction of a second. Existing studies show that a response time of more than a few seconds to these queries would negatively impact the productivity of the users [128]. Unfortunately, existing big-spatial data systems [66, 202, 198, 24, 177], require way more than that to run even the simplest queries, hence, they cannot answer interactive exploratory queries. The most viable solution to the interactive exploration problem is approximate query processing (AQP) which uses a small data synopsis, e.g., a sample, to provide an approximate answer within a fraction of a second. Existing solutions either provide answers without any performance guarantee or make unrealistic assumptions such as uniform distribution or independence between dimensions [172, 49, 171, 170, 109, 5, 148, 11, 127, 157, 107]. This problem is particularly challenging due to the intertwined relationship between the sample size, query parameters, algorithm logic, data distribution, and result accuracy. Chapter 5 discusses about a model that supports selectivity estimation with predicted error and response time.

One of the most important and challenging operations in all these systems is *spatial join*, which combines multiple datasets together based on their spatial features. It is a resource demanding operation in spatial databases and becomes even more challenging with big data [75, 161, 39]. Since big spatial data systems run in a distributed environment, the best algorithm has to balance the computation across machines, reduce disk access from the distributed file system, and minimize network overhead. At the same time, the skewed distribution of the inputs and the hardware specification of the machines have to be taken

into account. The complexity of the problem encourages the researchers to develop many spatial join algorithms for big data [108, 63, 201]. However, this creates a complex query optimization problem to choose the best spatial join algorithm given the input datasets and hardware resources. Chapter 6 proposes the first learned query optimizer for distributed spatial join on big data, which supports spatial data systems to make wise decision in spatial join execution.

Chapter 2

Balanced spatial partitioning for big data

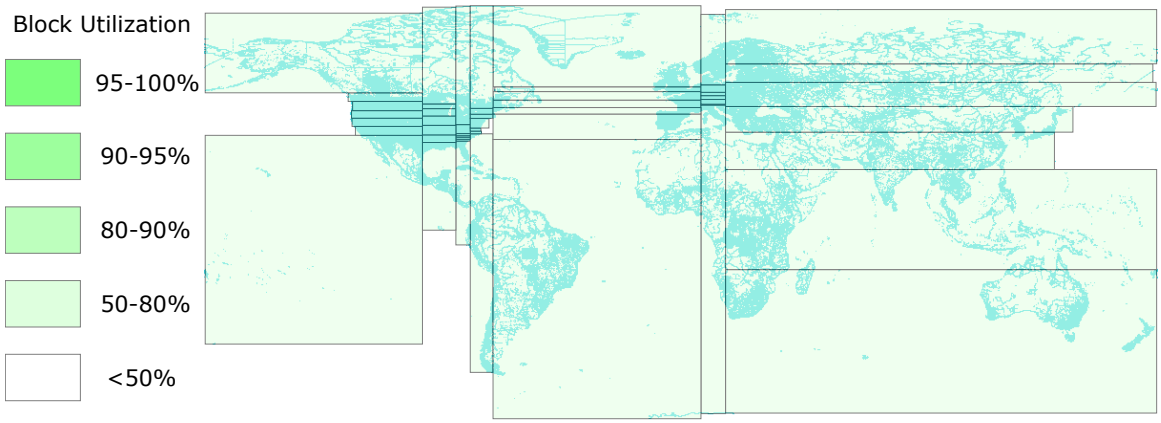
2.1 Introduction

The recent few years witnessed a rapid growth of big spatial data collected by different applications such as satellite imagery [81], social networks [135], smart phones [102], and VGI [94]. Traditional Spatial DBMS technology could not scale up to these petabytes of data which led to the birth of many big spatial data management systems such as SpatialHadoop [71], GeoSpark [203], Simba [198], LocationSpark [177], and Sphinx [67], to name a few.

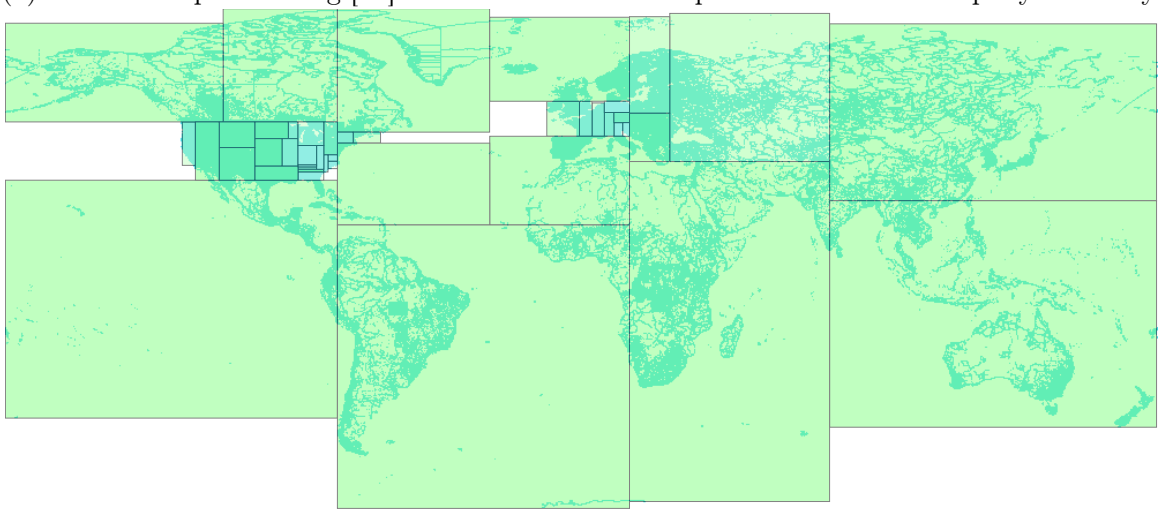
Regardless of their architecture, all these systems need an essential preliminary step that partitions the data across machines before the execution can be parallelized. This is also known as *global indexing* [74]. A common method that was first introduced in SpatialHadoop

[71], is the sample-based STR partitioner. This method picks a small sample of the input to determine its distribution, packs this sample using the STR packing algorithm [125], and then uses the boundaries of the leaf nodes to partition the entire data. Figure 2.1(a) shows an example of an STR-based partitioning where each data partition is depicted by a rectangle. The method was later generalized by replacing the STR bulk loading algorithm with other spatial indexes such as Quad-tree [165], Kd-Tree, and Hilbert R-trees [110, 65]. That STR-based partitioning was very attractive due to its simplicity and good load balancing which is very important for distributed applications. Its simplicity urged many other researchers to adopt it in their systems such as GeoSpark [203] and Simba [198] for in-memory distributed processing; Sphinx [67] for SQL big spatial data processing; HadoopViz [82, 92] for scalable visualization of big spatial data; and in distributed spatial join [160].

Despite their wide use, the existing partitioning techniques all suffer from one or more of the following three limitations. First, some partitioning techniques (STR, Kd-tree) prioritize load balance over spatial quality which results in suboptimal partitions. This is apparent in Figure 2.1(a) where the thin and wide partitions result in low overall quality for the partitions since square-like partitions are preferred for most spatial queries. Square-like partitions are preferred in indexing because they indicate that the index is not biased towards one dimension. Also, since most queries are shaped like a square or a circle, square-like partitions would minimize the overlap with the queries [26]. Second, they could produce partitions that do not fill the HDFS blocks in which they are stored. Big data systems are optimized to process full blocks, i.e., 128 MB, to offset the fixed overhead in processing each block. However, the index structures used in existing partitioning techniques, e.g.,



(a) STR-based partitioning [71]. All the thin and wide partitions reduce the query efficiency.



(b) The proposed R*-Grove method with square-like and balanced partitions.

Figure 2.1: Comparison between STR and R*-Grove

R-trees, Kd-tree, Quad-tree, produce nodes with number of records in the range $[m, M]$, where $m \leq M/2$. In practice, m can be as low as $0.2M$ [26, 29]. While those underutilized index nodes were desirable for disk indexing as they can accommodate future inserts, they result in underutilized blocks as depicted in Figure 2.1(a) where all blocks are less than 80% full. Moreover, this design might also produces poor load balance among partitions due to the wide range of partition sizes. Third, all existing partitioning techniques rely on a sample and try to balance the number of records per partition. This resembles traditional indexes

where the index contains record IDs. However, in big spatial data partitioning, the entire record is written in each partition, not just its ID. When records are highly variant in size, all existing techniques end up with extremely unbalanced partitions.

This chapter proposes a novel spatial partitioning technique for big data, termed R*-Grove, which completely addresses all of three aforementioned limitations. First, it produces high quality partitions by utilizing the R*-tree optimization techniques [26] which aim at minimizing the total area, overlap area, and margins. The key idea of the R*-Grove partitioning technique is to start with one partition that contains all sample points and then use the node split algorithm of the R*-tree to split it into smaller partitions. This results in compact square-like partitions as shown in Figure 2.1(b). Second, in order to ensure that we produce full blocks and balanced partitions, R*-Grove introduces a new constraint that puts a lower bound on the ratio between the smallest and the largest block, e.g., 95%. This property is theoretically proven and practically validated by our experiments. Third, when the input records have variable sizes, R*-Grove combines a data size histogram with the sample points to assign a weight for each sample point. These weights are utilized to guarantee that the size of each partition falls in a user-defined range.

Given the wide adoption of the previous STR-based partitioner, we believe the proposed R*-Grove will be widely used in big spatial data systems. This impacts a wide range of spatial analytics and processing algorithms including indexing [65, 182], range queries [71, 203], kNN queries [71], visualization [82, 92], spatial join [108], and computational geometry [68, 126]. All the work proposed in this chapter is publicly available as open source and supports both Apache Spark and Apache Hadoop. We run an extensive experimental

evaluation with up-to 500 GB and 7 billion record datasets and up-to nine dimensions. The experiments show that R*-Grove consistently outperforms existing STR-based, Z-curve-based, Hilbert-Curve-based, and Kd-tree-based techniques in both partitions quality and query efficiency.

The rest of this chapter is organized as follow. Section 2.2 describes the related works. Section 2.3 gives a background about big spatial data partitioning. Section 2.4 describes the proposed R*-Grove technique. Section 2.5 describes the advantages of R*-Grove in popular case studies of big spatial data systems. Section 5.4 gives a comprehensive experimental evaluation of the proposed work. Finally, Section 5.5 concludes the paper.

2.2 Related Work

This section discusses the related work in big spatial data partitioning. In general, distributed indexes for big spatial data are constructed in two levels, one global index that partitions the data across machines, and several local indexes that organize records in each partition. Previous work [71, 133, 74] showed that the global index provides far much improvement than local indexes. Therefore, in this chapter we focus on global indexing and it can be easily combined with any of the existing local indexes. The work in global indexing can be broadly categorized into three approaches, namely, sampling-based methods, space-filling-curve (SFC)-based methods, and quad-tree-based methods.

The *sampling-based* method picks a small sample from the input data to infer its distribution. The sample is loaded into an in-memory index structure while adjusting the data page capacity, e.g., leaf node capacity, such that the number of data pages is roughly

equal to the desired number of partitions. The order of sample objects does not affect the partition quality, since the sample is uniformly taken from the entire input dataset. Furthermore, most of algorithms sort the data as part of the partitioning process so the original order is completely lost. Some R-tree bulk-loading algorithms (STR [125] or OMT [123]) can also be used to speed up the tree construction time. Then, the minimum bounding rectangles (MBRs) of the data pages are used to partition the entire dataset. This method was originally proposed for spatial join and denoted the seeded-tree [129]. It was then used for big spatial indexing in many systems including SpatialHadoop [71, 65], Scala-GiST [133], GeoSpark [203], Sphinx [67], Simba [198], and many other systems. This technique can be used with existing R-tree indexes but it suffers from two limitations, load imbalance and low quality of spatial partitions. Additionally, when there is a big variance in record sizes, the load imbalance is further amplified due to the use of the sample. We will further discuss these limitations in Section 2.4.

The *SFC-based* method builds a spatial index on top of an existing one-dimensional index by applying any space-filling curve, e.g., Z-curve or Hilbert curve. MD-HBase [147] builds Kd-tree-like and Quad-tree-like indexes on top of HBase by applying the Z-curve on the input data and customizing the region split method in HBase to respect the structure of both indexes. GeoMesa [91] uses geo-hashing which is also based on the Z-curve to build spatio-temporal indexes on top of Accumulo. Unlike MD-HBase which only supports point data, GeoMesa can support rectangular of polygonal geometries by replicating a record to all overlapping buckets in the geohash. While this method can ensure a near-perfect load balance, it produces an even bigger spatial overlap between partitions as compared to the

sampling-based approach described above. This drawback leads to the inefficient performance of spatial queries.

The *quad-tree-based* method relies heavily on the Quad-tree structure to build efficient and scalable Quad-tree index in Hadoop [196]. It starts by splitting the input data into equi-sized chunks and building a partial Quad-tree for each split. Then, it combines the leaf nodes of the partial trees based on the Quad-tree structure to merge them into the final tree. While highly efficient, this method cannot generalize to other spatial indexes and is tightly tied to the Quad-tree structure. In addition, this Quad-tree-based partitioning tends to produce much more than the desired number of partitions which also leads to load imbalance.

Although there are several partitioning techniques for large-scale spatial data as mentioned above, sampling-based method is the most ubiquitous option, which is integrated in most of existing spatial data systems. Sampling-based methods are preferred as they are simple to implement and provide very good results. In this chapter, we follow the sampling-based approach, and propose a method which utilizes R*-tree's advantages that were never used before for big spatial data partitioning. The proposed R*-Grove index has three advantages over the existing work. First, it inherits and improves the R*-tree index structure to produce high-quality partitions that are tailored to big spatial data. Second, the improved algorithm produces balanced partitions by employing a user-defined parameter, termed *balance factor*, α, e.g., 95%. In addition, it can produce spatially disjoint partitions which are necessary for some spatial analysis algorithms. Third, R*-Grove can couple a sample with a data size histogram to guarantee the desired load balance even when the input

record sizes are highly variant. While R*-Grove is not the only framework for big spatial partitioning, it is the first one that is tailored for large-scale spatial datasets while existing techniques reuse traditional index structures, such as R-tree, STR, or Quad-tree, as black boxes.

2.3 Background

2.3.1 R*-tree

The R*-tree [26] belongs to the R-tree family [98] and it improves the insertion algorithm to provide high quality index. In R-tree, the number of children in each nodes has to be in the range $[m, M]$. By design, m can be at most $\lfloor M/2 \rfloor$ to ensure that splitting a node of size $M + 1$ is feasible. In this chapter, we utilize and enhance two main functions of the R*-tree index, namely, ChooseSubtree and SplitNode which are both used in the insertion process. For the ChooseSubtree method, given the MBR of a record and a tree node, it chooses the best subtree to assign this record to. The SplitNode method takes an overflow node with $M + 1$ records and splits it into two nodes.

2.3.2 Sample-based Partitioning Workflow

This section gives a background on the sampling-based partitioning technique [71, 65, 182], just partitioning hereafter, that this chapter relies on. Figure 2.2 shows the workflow for the partitioning algorithm which consists of three phases, namely, sampling, boundary computation, and partitioning. The sampling phase (Phase 1) draws a random sample of the input records and converts each one to a point. Notice that sample points

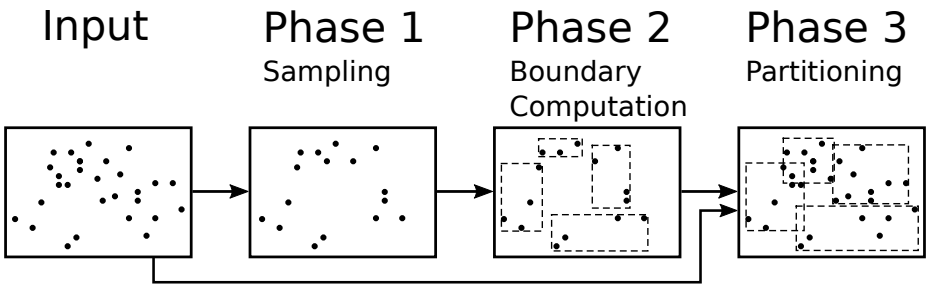


Figure 2.2: The sampling-based partitioning process

are picked from the entire file at no particular order so the order of points does not affect the next steps. The boundary computation phase (Phase 2) runs on a single machine and processes the sample to produce partition boundaries as a set of rectangles. Given a sample S, the input size D, and the desired partition size B, this phase adjusts the capacity of each partition to contain $M = \lceil |S| \cdot B/D \rceil$ sample points which is expected to produce final partitions with the size of one block each. The final partitioning phase (Phase 3) scans the entire input in parallel and assigns each record to these partitions based on the MBR of the record and the partition boundaries. If each record is assigned to exactly one partition, the partitions will be spatially overlapping with no data replication. If each record is assigned to all overlapping partitions, the partitions will be spatially disjoint but some records can be replicated and duplicate handling will be needed in the query processing [58]. Some algorithms can only work if the partitions are spatially disjoint such as visualization [82] and some computational geometry functions [126].

The proposed R*-Grove method expands Phase 1 by optionally building a histogram of storage size that assists in the partitioning algorithm at Phase 2. In Phase 2, it adapts R*-tree-based algorithms to produce the partition boundaries with desired level of load

balance. In Phase 3, we propose a new data structure that improves the performance of that phase and allows us to produce spatially disjoint partitions if needed.

2.3.3 Quality Metrics

this chapter uses the quality metrics defined in [66]. Below, we redefine these metrics while accounting for the case of partitions that span multiple HDFS blocks. A single partition π_i is defined by two parameters, minimum bounding box mbb_i and size in bytes $size_i$. Given the HDFS block size B, e.g., 128 MB, we define the number of blocks for a partition π_i as $b_i = \lceil size_i/B \rceil$. Given a dataset that is partitioned into a set of l partitions, $\mathcal{P} = \{\pi_i\}$, we define the five quality metrics as follows.

Definition 1 (Total Volume - Q_1)**.** *The total volume is the sum of the volume of all partitions where the volume of a partition is the product of its side lengths.*

$$Q_1(\mathcal{P}) = \sum_{\pi_i \in \mathcal{P}} b_i \cdot volume(mbb_i)$$

We multiply by the number of blocks b_i because big spatial data systems process each block separately. Lowering the total volume is preferred to minimize the overlap with a query. Given the popularity of the two-dimensional case, this is usually used under the term total area.

Definition 2 (Total Volume Overlap - Q_2)**.** *This quality metric measures the sum of the overlap between pairs of partitions.*

$$Q_2(\mathcal{P}) = \sum_{\pi_i,\pi_j \in \mathcal{P}, i \neq j} b_i \cdot b_j \cdot volume(mbb_i \cap mbb_j) + \sum_{\pi_i \in \mathcal{P}} \frac{b_i(b_i - 1)}{2} \cdot volume(mbb_i)$$

where $mbb_i \cap mbb_j$ is the intersection region between the two boxes. The first term calculates the overlaps between pairs of partitions and the second term accounts for self-overlap *which treats a partition with multiple blocks as overlapping partitions. Lowering the volume overlap is preferred to keep the partitions apart.*

Definition 3 (Total Margin - Q_3)**.** *The margin of a block is the sum of its side lengths. The total margin is the sum of all margins as given below.*

$$Q_3(\mathcal{P}) = \sum_{\pi_i \in \mathcal{P}} b_i \cdot margin(mbb_i)$$

Similar to Q_1, multiplying by the number of blocks b_i treats each block as a separate partition. Lowering the total margin is preferred to produce square-like partitions.

Definition 4 (Block Utilization - Q_4)**.** *Block utilization measures how* full *the HDFS blocks are.*

$$Q_4(\mathcal{P}) = \frac{\sum_{\pi_i \in \mathcal{P}} size_i}{B \cdot \sum_{\pi_i \in \mathcal{P}} b_i}$$

The numerator $\sum size_i$ represents the total size of all partitions and denominator $B \sum b_i$ is the maximum amount of data that can be stored in all blocks used by these partitions. In big data applications, each block is processed in a separate task which has a setup time of a few

seconds. Having full or near-full blocks minimize the overhead of the setup. The maximum value of block utilization is 1.0*, or* 100%*.*

Definition 5 (Standard Deviation of Sizes)**.**

$$Q_5(\mathcal{P}) = \sqrt{\frac{\sum_{\pi_i \in \mathcal{P}} (size_i - \overline{size})^2}{l}}$$

Where $\overline{size} = \sum size_i/l$ *is the average partition size. Lowering this value is preferred to balance the load across partitions.*

2.4 R*-Grove Partitioning

This section describes the details of the proposed R*-Grove partitioning algorithm. R*-Grove employs three techniques that overcome the limitations of existing works. The first technique adapts the R*-tree index structure for spatial partitioning by utilizing the ChooseSubTree and SplitNode functions in the sample-based approach described in Section 2.3. This technique ensures a high spatial quality of partitions. The second technique addresses the problem of load balancing by introducing a new constraint that guarantees a user-defined ratio between smallest and largest partitions. The third technique combines the sample points with its storage histogram to balance the sizes of the partitions rather than the number of records. This combination allows R*-Grove to precisely produce partitions with a desired block utilization, which cannot be achieved by any other partitioning techniques.

2.4.1 R*-tree-based Partitioning

This part describes how R*-Grove utilizes the R*-tree index structure to produce high quality partitions. It utilizes the SplitNode and ChooseSubtree functions from the R*-tree algorithm in Phases 2 and 3 as described shortly. A naïve method [187] is to use the R*-tree as a blackbox in Phase 2 in Figure 2.2 and insert all the sample points into an R*-tree. Then it emits the MBRs of the leaf nodes as the output partition boundaries. However, this technique was shown to be inefficient as it processes the sample points one-by-one and does not integrate the R*-tree index well in the partitioning algorithm. Therefore, we propose an efficient approach that runs much faster and produces higher quality partitions. It extends Phases 2 and 3 as follows.

Phase 2 computes partition boundaries by only using the SplitNode algorithm from the R*-tree index which splits a node with $M + 1$ records into two nodes with the size of each one in the range $[m, M]$. This algorithm starts by choosing the split axis, e.g., x or y, that minimizes the total margin. Then, all the points are sorted along the chosen axis and the split point is chosen as depicted in Algorithm 1. The ChooseSplitPoint algorithm simply considers all the split points and chooses the one that minimizes some cost function which is typically the total area of the two resulting partitions.

We set $M = \lceil |S| \cdot B/|D| \rceil$ as explained in Section 2.3 and $m = 0.3M$ as recommended in the R*-tree paper. In particular, this phase starts by creating a single big tree node that has all the sample points S. Then, it recursively calls the SplitNode algorithm as long as the resulting node has more than M elements. This top-down approach has a key advantage over building the tree record-by-record as it allows the algorithm to look at

Algorithm 1 A simplified version of the traditional R*-tree splitting mechanism.
Inputs: P is the all sample records; m is the minimum size of a node.
Output: the optimal splitting position.

```
1: function ChooseSplitPoint(P,m)
2:     chosenK = -1; minCost = ∞
3:     for k in [m, |P| − m] do
4:         P_1 = P[1..k]                          ▷ P_1 is the first k records of P
5:         P_2 = P[k + 1..|P|]                    ▷ P_2 is all the remaining records P − P_1
6:         Calculate the cost of the partitions P_1 and P_2
7:         if the cost is smaller than minCost then
8:             Set chosenK = k and update minCost
9:     return chosenK
```

all the records at the beginning and optimize for all of them. Furthermore, it avoids the ForcedReinsert algorithm which is known to slow down the R*-tree insertion process. Notice that this is different than the bulk loading algorithms as it does not produce a full tree. Rather, it just produces a set of boundaries that are produced as partitions. Phase 3 treats all the MBRs as leaf nodes in an R-tree and uses the ChooseLeaf method from the R*-tree to assign an input record to a partition.

Run-time analysis: The SplitNode algorithm can be modeled as a recursive algorithm where each iteration sorts all the points and runs the linear-time splitting algorithm to produce two smaller partitions. The run-time can be expressed as $T(n) = T(k) + T(n - k) + O(n \log n)$, where k is the size of one group resulting from the partitioning, n is the number of records in the input partition. In particular, $T(k)$ and $T(n - k)$ are the running times to partition two partitions from splitting process. The term $O(n \log n)$ is the running time for the splitting part which requires sorting all the points. This recurrence relation has a worst case of $n^2 \log n$ if k is always $n - 1$. In order to guarantee a run-time of $O(n \log^2 n)$, we define a parameter $\rho \in [0, 0.5]$ which defines the minimum splitting ratio k/n. Setting

Algorithm 2 R*-tree-based split while ensuring valid partitions
Inputs: P is the all sample records; $[m, M]$ is the target range of sizes for final partitions.
Output: the optimal splitting position.

```
1: function ChooseValidSplitPoint(P,m)
2:     for k in [m, |P| − m] do
3:         if either k or |P| − k is invalid then                ▷ Lemma 1
4:             Skip this iteration and continue
5:         Similar to Lines 4-8 in Algorithm 1
6:     return chosenK
```

this parameter to any non-zero fraction guarantees an $O(n \log^2 n)$ run-time. However, the restriction of k/n also limits the range of possible value of k. For example, if $n = 100$ and $\rho = 0.3$, k must be a number in the range $[30, 70]$. As this parameter gets closer to 0.5, the two partitions become closer in size and the run-time decreases but the quality of the index might also deteriorate due to the limited search space imposed by this parameter. To incorporate this parameter in the node-splitting algorithm, we call the ChooseSplitPoint function with the parameters (P,max$\{m, \rho \cdot |P|\}$), where $|P|$ is the number of points in the list P.

2.4.2 Load Balancing for Partitions with Equal-size records

In this section, we focus on balancing the number of records in partitions assuming equal-size records. We further extend this in the next section to support variable-size records. The method in Section 2.4.1 does a good job in producing high-quality partitions similar to what the R*-tree provides. However, it does not address the second limitation, that is, balancing the sizes of the partitions. Recall that the R-tree index family requires the leaf nodes to have sizes in the range $[m, M]$, where $m \leq M/2$. With the R*-tree algorithm explained earlier, some partitions might be 30% full which reduces block utilization and load

balance. We would like to be able to set m to a larger value, say, $m = 0.95M$. Unfortunately, if we do so, the SplitNode algorithm would just fail because it will face a situation where there is no valid partitioning.

To illustrate the limitation of the SplitNode mechanism, consider the following simple example. Let us assume we choose $m = 9$ and $M = 10$ while the list P contains 28 points. If we call the SplitNode algorithm on the 28 points, it might produce two partitions with 14 records each. Since both contain more than $M = 10$ points, the splitting method will be called again on each of them which will produce an incorrect answer since there is no way to split 14 records into two groups while each of them contain between 9 and 10 records. A correct splitting algorithm would produce three partitions with sizes 9, 9, and 10. Therefore, we need to introduce a new constraint to the splitting algorithm so that it always produces partitions with sizes in the range $[m, M]$.

The Final Finding: The SplitNode algorithm can be minimally modified to guarantee final leaf partitions in the range $[m, M]$ by satisfying the following validity constraint:

$$\lceil S_i/M \rceil \leq \lfloor S_i/m \rfloor, i \in \{1, 2\}$$

, where S_1 and S_2 are the sizes of the two resulting partitions of the split. Algorithm 2 depicts the main changes to the algorithm that introduces a new constraint test in Line 3 that skips over invalid partitioning. The rest of this section provides the theoretical proof that this simple constraint guarantees the algorithm termination with leaf partitions in the range $[m, M]$. We start with the following definition.

Definition 6. ***Valid Partition Size:*** *An integer number S is said to be a valid partition size with respect to a range $[m, M]$ if there exists a set of integers $X = \{x_1, \cdots, x_n\}$ such that $\sum x_i = S$ and $x_i \in [m, M]$ $\forall$ $i \in [1, n]$. In words, if we have S records, there is at least one way to split them such that each split has between m and M records.*

For example, if $m = 9$ and $M = 10$, the sizes 14, 31, and 62, are all invalid while the sizes 9, 27, and 63, are valid. Therefore, to produce balanced partitions, the SplitNode algorithm should keep the invariant that the partition sizes are always valid according to the above definition. Going back to the earlier example, if $S = 28$, the answer $S_1 = S_2 = 14$ will be rejected because $S_1 = 14$ is invalid. Rather, the result of the first call to the *SplitNode* algorithm will result in two partitions with sizes $\{10, 18\}$ or $\{9, 19\}$. The following lemma shows how to test a size for validity in constant time.

Lemma 1. ***Validity Test:*** *An integer S is a valid partition size w.r.t a range $[m, M]$ iff $L \leq U$ in which L (lower bound) and U (upper bound) are computed as:*

$$L = \lceil S/M \rceil$$

$$U = \lfloor S/m \rfloor$$

Proof. First, if S is valid then, by definition, there is a partitioning of S into n partitions such that each partition is in the range $[m, M]$. It is easy to show that $L \leq U$ and we omit this part for brevity. The second part is to show that if the inequality $L \leq U$ holds, then there is at least one valid partitioning. Based on the definition of L and U, we have:

$$U = \lfloor S/m \rfloor \Rightarrow U \leq S/m \Rightarrow S \geq m \cdot U \Rightarrow S \geq m \cdot L \Rightarrow S - m \cdot L \geq 0 \tag{2.1}$$

$$L = \lceil S/M \rceil \Rightarrow L \geq S/M \Rightarrow S \leq M \cdot L \Rightarrow S - m \cdot L \leq (M - m) \cdot L \tag{2.2}$$

Based on Inequalities 2.1 and 2.2, we can make a valid partitioning as follows:

1. Start with L empty partitions. Assign m records to each partition. The remaining number of records is $S - m \cdot L \geq 0$. This is satisfied due to Inequality 2.1.

2. Since each partition now has m records, it can receive up-to $M - m$ additional records in order to keep its validity. Overall, L partitions of size m can accommodate up-to $(M - m) \cdot L$ records to keep a valid partitioning. But the remaining number of records $S - m \cdot L$ is not larger than the upper limit of what the partitions can accommodate, $(M - m) \cdot L$ as shown in Inequality 2.2. Therefore, this condition is satisfied as well.

In conclusion, it follows that if the condition $L \leq U$ holds, we can always find a valid partitioning scheme for S records which completes the proof. □

If we apply this test for the example above, we find that 28 is valid because $L = \lceil 28/10 \rceil = 3 \leq U = \lfloor 28/9 \rfloor = 3$ while 62 is invalid because $L = \lceil 62/10 \rceil = 7 > U = \lfloor 62/9 \rfloor = 6$. This approach works fine as long as the initial sample size S is valid but how do we guarantee the validity of S? We show that this is easily guaranteed if the size S is above some threshold S^* as shown in the following lemma.

Lemma 2. *Given a range $[m, M]$, any partition of size $S \geq S^*$ is valid where S^* is defined by the following formula:*

$$S^* = \left\lceil \frac{m}{M-m} \right\rceil \cdot m \tag{2.3}$$

Proof. Following Definition 6, we will prove that for any partition size $S \geq S^*$, there exists a way to split it into k groups such that the size of each group is in the range $[m, M]$.

First, let $i = \left\lceil \frac{m}{M-m} \right\rceil$, we have:

$$S \geq S^* = \left\lceil \frac{m}{M-m} \right\rceil \cdot m = i \cdot m \tag{2.4}$$

$$\Rightarrow S = i \cdot m + X, X \geq 0. \textit{ Let } X = a \cdot m + b \qquad , a \geq 0, 0 \leq b < m \tag{2.5}$$

$$\Rightarrow S = i \cdot m + (a \cdot m + b) = (i + a) \cdot m + b \qquad , a \geq 0, 0 \leq b < m \tag{2.6}$$

Second, since $b < m$, we have:

$$\frac{b}{M-m} < \frac{m}{M-m} \leq i \Rightarrow \frac{b}{i} < M - m \tag{2.7}$$

From Equation 2.6 and 2.7, we can make a valid partitioning for a partition size S as follows:

1. Start with $i + a$ empty partitions. Assign m records to each partition. The remaining number of records is b. This step is based on Equation 2.6.

2. Equation 2.7 means that we can split b records over i groups such that each group receives at most $M - m$ records. Since we already have $i + a$ groups each of size m, adding $M - m$ to i groups out of them will increase their sizes to M which still keeps them in the valid range $[m, M]$. The remaining groups will still have m records making them valid too.

This completes the proof of Lemma 2. □

Based on Lemma 2, a question is raised as how large the size of sample points S should be to ensure that a good block utilization is achievable. As we mentioned from beginning, R*-Grove allows us to configure a parameter $\alpha = m/M$, that called *balance factor*, is computed as the ratio between minimum and maximum number of records of a leaf node in the tree. α should be close to 1 to guarantee a good block utilization. Let's assume that $0 < r \leq 1$ is the sampling ratio and p is the storage size of a single point. The maximum number of records M is computed in the Section 2.4.1 as:

$$M = \left\lceil \frac{|S| \cdot B}{D} \right\rceil \Rightarrow M = \left\lceil \frac{|S| \cdot p}{D} \cdot \frac{B}{p} \right\rceil \Rightarrow M = \left\lceil \frac{r \cdot B}{p} \right\rceil \tag{2.8}$$

From Equation 2.8, we can rewrite Lemma 2 as:

$$|S| \geq S^* = \left\lceil \frac{m}{M - m} \right\rceil \cdot m \Rightarrow |S| \geq \left\lceil \frac{\alpha}{1 - \alpha} \right\rceil \cdot \alpha \cdot \left\lceil \frac{r \cdot B}{p} \right\rceil \Rightarrow |S| \cdot p \geq \left\lceil \frac{\alpha}{1 - \alpha} \right\rceil \cdot \alpha \cdot \lceil r \cdot B \rceil \tag{2.9}$$

Therefore, assume that we want to configure the *balance factor* as $\alpha = 0.95$, sample ratio $r = 1\%$ and block size $B = 128$ MB, then the term $|S| \cdot p$ in Equation 2.9 would be

Algorithm 3 Choose the split point with weights
Inputs: P is the all sample records; w is an array of weights of corresponding records in P; $[m, M]$ is the target range of sizes for final partitions.
Output: the optimal splitting position.

```
1: function ChooseWeightedSplitPoint(P,w,m)
2:     W = ∑_{1≤i≤|P|} w_i
3:     for k in [m, |P| − m] do
4:         W_1 = ∑_{1≤i≤k} w_i
5:         if either W_1 or W − W_1 is invalid then                ▷ Lemma 1
6:             Skip this iteration and continue
7:         Similar to Lines 4-8 in Algorithm 1
8:     return chosenK
```

computed as 23 MB. In other words, if the storage size of sample points $|S| \cdot p \geq 23$ MB, it will be guaranteed to produce a valid partitioning. This is a reasonable size that can be stored in main memory and processed in a single machine.

2.4.3 Load Balancing for Datasets with Variable-size Records

The above two approaches can be combined to produce high-quality and balanced partitions in terms of number of records. However, the partitioning technique needs to write the actual records in each partition and often these records are of variable sizes. For example, the sizes of records in the `OSM-Objects` dataset [15] range from 12 bytes to 10 MB per record. Therefore, balancing the number of records can result in a huge variance in the partition sizes in terms of number of bytes.

To overcome this limitation, we combine the sample points with a *storage size histogram* of the input as follows. The storage size histogram is used to assign a weight to each sample point that represents the total size of all records in its vicinity. To find these weights, Phase 1 computes, in addition to the sample, a storage size histogram of the input.

This histogram is created by overlaying a uniform grid on the input space and computing the total size of all records that lie in each grid cell [49, 172]. This histogram is computed on the full dataset not the sample, therefore, it catches the actual size of the input. After that, we count the number of *sample* points in each grid cell. Finally, we divide the total weight of each cell among all sample points in this cell. For example, if a cell has a weight of 1,000 bytes and contains five sample points, the weight of each point in this cell becomes 200 bytes.

In Phase 2, the SplitNode function is further improved to balance the *total weight* of the points in each partition rather than the number of points. This also requires modifying the value of M to be $M = \lceil \sum w_i / N \rceil$, where w_i is the assigned weight to the sample point p_i, and N is the desired number of partitions. Algorithm 3 shows how the algorithm is modified to take the weights into account. Line 4 calculates the weight of each partitioning point which is used to test the validity of this split point as shown in Line 5.

Unfortunately, if we apply this change, the algorithm is no longer guaranteed to produce balanced partitions. The reason is that the proof of Lemma 1 is no longer valid. That proof assumed that the partition sizes are defined in terms of number of records which makes all possible partition sizes part of the search space in the **for-loop** in Line 2 of Algorithm 2. However, when the size of each partition is the sum of the weights, the possible sizes are limited to the weights of the points. For example, let us assume a partition with five points all of the same weight $w_i = 200$ while $m = 450$ and $M = 550$. The condition in Definition 6 suggests that the total weight $1,000$ is valid because $L = \lceil 1000/550 \rceil = 2 \leq U = \lfloor 1000/450 \rfloor = 2$. However, given the weights $w_i = 200$ for $i \in [1, 5]$, there is no valid partitioning, i.e., there is no way to make two partitions each with a total weight in the range $[450, 550]$.

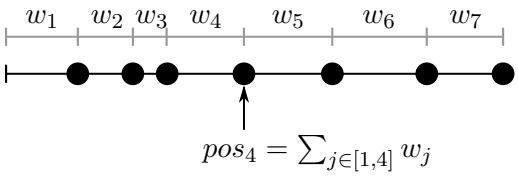
To overcome this issue, this part further improves the SplitNode algorithm so that it still guarantees a valid partitioning even for the case described above. The key idea is to make minimal changes to the weights to ensure that the algorithm will terminate with a valid partitioning; we call this process *weight correction*. For example, the case described earlier will be resolved by changing the weights of two points from 200 and 200 to 100 and 300. This will result in the valid partitioning $\{200, 200, 100\}$ and $\{300, 200\}$ which is valid. Keep in mind that these weights are approximate anyway as they are based on the sample and histogram so these minimal changes would not hugely affect the overall quality, yet, they ensure that the algorithm will terminate correctly. The following part describes how these weight changes are applied while ensuring a valid answer.

First of all, we assume that the points are already sorted along the chosen axis as explained in Section 2.4.1. Further, we assume that Algorithm 3 failed by not finding any valid partitions, i.e., return -1. Now, we make the following definitions to use them in the weight update function.

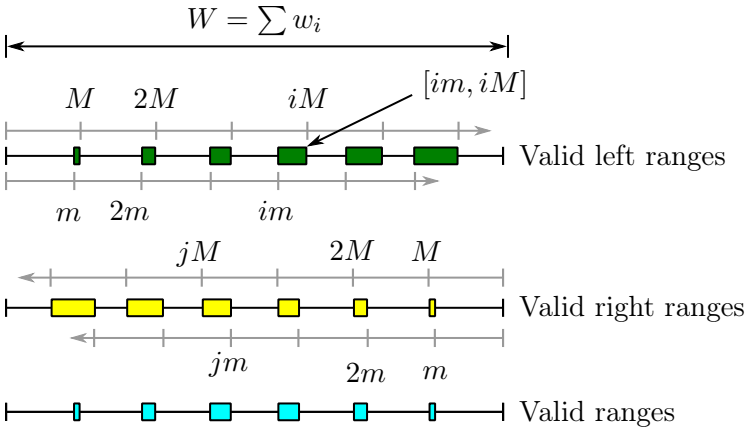
Definition 7. ***Point position:*** *Let p_i be point #i in the sort order and its weight is w_i. We define the position of the point i as $pos_i = \sum_{j \leq i} w_j$.*

Based on this definition, we can place all the points on a linear scale based on their position as shown in Figure 2.3(a).

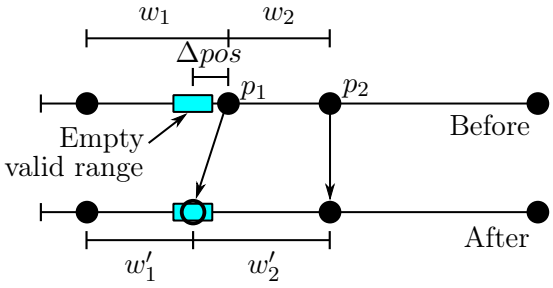
Definition 8. ***Valid left range:*** *A range of positions $VL = [vl_s, vl_e]$ is a valid left range if for all positions $vl \in VL$ the value vl is valid w.r.t. $[m, M]$. All the valid left ranges can be written in the form $[im, iM]$ where i is a natural number and they might overlap for large values of i. (See Figure 2.3(b).)*



(a) Positions of points



(b) Valid ranges



(c) Weight correction

Figure 2.3: Load balancing for datasets with variable-size records

Definition 9. ***Valid right range:*** *A range of positions $VR = [vr_s, vr_e]$ is a valid right range if for all positions $vr \in VR$ the value $W - vr$ is valid w.r.t. $[m, M]$. Similar to valid left ranges, all valid right ranges can be written in the form $[W - jM, W - jm]$, where $W = \sum w_i$. (See Figure 2.3(b).)*

Definition 10. ***Valid range:*** *A range of positions $V = [v_s, v_e]$ is valid if for all positions $v \in V$, v belongs to at least one valid left range and at least one valid right range. In other words, the valid ranges are the intersection of the valid left ranges and valid right ranges.*

Figure 2.3(b) illustrates the valid left, valid right, and valid ranges. If we split a partition around a point with a position in a valid left range, the first partition will be valid. Similarly for valid right positions the second partition (on the right) will be valid. Therefore, we would like to split a partition around a point in one of the valid ranges (intersection of left and right).

Lemma 3. ***Empty valid ranges:*** *If Algorithm 3 fails by returning -1, then none of the point positions in P falls in a valid range.*

Proof. By contradiction, let a point p_i has a position pos_i that falls in a valid range. In this case, the partitions $P_1 = \{p_k : k \leq i\}$ and $P_2 = \{p_l : l > i\}$ are both valid partitions because the total weight of P_1 is equal to the position pos_i which is valid because pos_i falls in a valid left range. Similarly, the total weight of P_2 is valid because pos_i falls in a valid right range. In this case, Algorithm 3 should have found this partitioning as a valid partitioning because it tests all the points which is a contradiction. □

A corollary to Lemma 3 is that when Algorithm 3 fails by returning -1, then all valid ranges are empty.

As a result, we would like to slightly modify the weights of some points in the sample points in order to enforce some points to fall in valid ranges. We call this the *weight correction* process. This process is described in the following lemma:

Lemma 4. ***Weight correction:*** *Given any empty valid range $[v_s, v_e]$, we can modify the weight of only two points such that the position of one of them will fall in the range.*

Proof. Figure 2.3(c) illustrates the proof of this lemma. Given an empty valid range, we modify the two points with positions that follow the empty valid range, p_1 and p_2, where $pos_1 < pos_2$. We would like to move the point p_1 to the new position $pos'_1 = (v_s + v_e)/2$ which is in the middle of the empty valid range. To do that, we reduce the weight w_1 by $\Delta pos = pos_1 - pos'_1$. The updated weight $w'_1 = w_1 - \Delta pos$. To keep the position of p_2 and all the following points intact, we have to also increase the weight of p_2 by Δpos; that is, $w'_2 = w_2 + \Delta pos$. □

We do the weight correction process for *all* empty valid ranges to make them non-empty and then we repeat Algorithm 3 to choose the best one among them.

The only remaining part is how to enumerate all the valid ranges. The idea is to simply find a valid left range, an overlapping valid right range, and compute their intersection, all in constant time. Given a natural number i, the valid left range is in the form $[im, iM]$. Assume that this range overlaps a valid right range in the form $[W - jM, W - jm]$. Since they overlap, the following two inequalities should hold:

$$W - jm \geq im \Rightarrow j < \frac{W - im}{m}$$

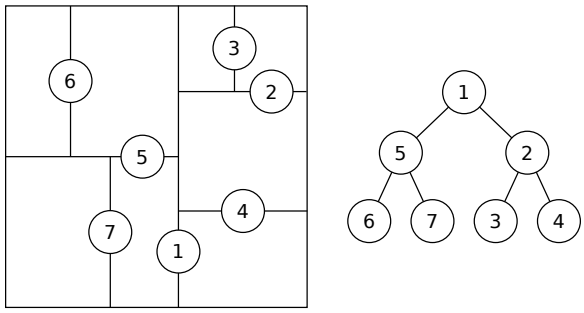


Figure 2.4: Auxiliary search structure for R*-Grove

$$W - jM \leq iM \Rightarrow j > \frac{W - iM}{M}$$

Therefore, the lower bound of j is $j_1 = \lceil \frac{W - i \cdot M}{M} \rceil$ and the upper bound of j is $j_2 = \lfloor \frac{W - i \cdot m}{m} \rfloor$. If $j_1 \leq j_2$, then there is a solution to these inequalities which we use to generate the bounds of the valid range $[v_s, v_e]$. Notice that if there is more than one valid solution to j, all of them should be considered to generate all the valid ranges but we omit this special case for brevity.

2.4.4 Implementation Considerations

Optimization of Phase 3: The ChooseSubTree operation in R*-tree chooses the node that results in the least overlap increase with its siblings [26]. A straight-forward implementation of this method is $O(n^2)$ as it needs to compute the overlap between each candidate partition and all other partitions. In the R*-tree index, this cost is limited due to the limited size of each node. However, this step can be too slow as the number of partitions in R*-Grove can be extremely large. To speed up this step, we use a K-d-tree-like auxiliary search structure as shown in Figure 2.4. This index structure is generated during Phase 2 as the partition boundaries are computed. Each time the NodeSplit operation

completes, the search structure is updated by adding a corresponding split in the direction of the chosen split axis. This auxiliary search structure is stored in memory and replicated to all nodes. It will be used in Phase 3, when we physically store the input records to the partitions. Given a spatial record, it will be assigned to the corresponding partition using a search algorithm which is similar to the K-d-tree's point search algorithm [41]. Based on this similarity, we can estimate the running time to choose a partition to be $O(log(n))$. Notice that this optimization is not applicable in traditional R*-trees as the partitions might be overlapping while in R*-Grove we utilize the fact that we only partition points which guarantees disjoint partitions.

Since the partition MBRs in Phase 2 are computed from sample objects, there will be objects which do not fall in any partition in Phase 3. R*-Grove addresses this problem in two ways. First, if no disjoint partitions are desired, it chooses a single partition based on the ChooseLeaf method in original R*-tree. In short, an object will be assigned to the partition in which the enlarged area or margin is minimal. Second, if disjoint partitions are desired, R*-Grove uses the auxiliary data structure, which covers the entire space, to assign this record to all overlapping partitions.

Disjoint indexes: Another advantage of using the auxiliary search structure described above, is that it allows for building a disjoint index. This search structure naturally provides disjoint partitions. To ensure that the partitions cover the entire input space, we assume that input region is infinite, that is, starts from $-\infty$ and ends at $+\infty$ in all dimensions. Then, Phase 3 replicates each record to all overlapping partitions by directly searching in this k-d-tree-like structure with range search algorithm, which has the $O(\sqrt{n})$ running time [122].

This advantage was not possible with the black-box R*-tree implementation as it is not guaranteed to provide disjoint partitions.

2.5 Case Studies

This section describes three case studies where the R*-Grove partitioning technique can improve big spatial data processing. We consider three fundamental operations, namely, indexing, range query, and spatial join.

2.5.1 Indexing

Spatial data indexing is an essential component in most big spatial data management systems. The state-of-the-art global indexing techniques rely on reusing existing index structures with a sample which are shown to be inefficient in terms of quality and load balancing [66, 202, 182].

R*-Grove partitioning can be used for the *global indexing* step which partitions records across machines. In big spatial data indexing, the global index is the most crucial step as it ensures load balancing and efficient pruning when the index is used. If only the number of records needs to be balanced or if the records are roughly equi-sized, then the techniques described in Sections 2.4.1 and 2.4.2 can be used. If the records are of a variable size and the total sizes of partitions need to be balanced, then the histogram-based step in Section 2.4.3 can be added to ensure a higher load balance. Notice that the index would hugely benefit from the balanced partition size as it reduces the total number of blocks in

the output file which improves the performance of all Spark and MapReduce queries that create one task per file block.

2.5.2 Range Query

Range query is a popular spatial query, which is also the building block of many other complex spatial queries. Previous studies found a strong correlation between the performance of range queries and the performance of other queries such as spatial join [66, 104]. Therefore, the performance of range query could be considered as a good reflection about the quality of a partitioning technique. A good partitioning technique allows the query processor to make two optimization techniques. First, it can prune the partitions that are completely outside the query range. Second, it can directly write to the output the partitions that are completely contained in the query range without further processing [67]. For very small ranges, most partitioning techniques will behave similarly as it is most likely that the small query overlaps one partition and no partitions are completely contained [65]. However, as the query range increases, the differences between the partitioning techniques become apparent. Since most range queries are expected to be square-like, the R*-Grove partitioning is expected to perform very well as it minimizes the total margin which produces square-like partitions. Furthermore, the balanced load across partitions minimizes the straggler effect where one partition takes significantly longer time than all other partitions.

2.5.3 Spatial Join

Spatial join is another important spatial query that benefits from the improved R*-Grove partitioning technique. In spatial join, two big datasets need to be combined

to find all the overlapping pairs of records. To support spatial join on partitioned big spatial data, each dataset is partitioned independently. Then, a spatial join runs between the partition boundaries to find all pairs of overlapping partitions. Finally, these pairs of partitions are processed in parallel. An existing approach [208] preserves spatial locality to reduce the processing jobs. However, it still relies on traditional index like R-Tree, which also inherited its limitations. The R*-Grove partitioning has two advantages for the spatial join operation. First, it is expected to reduce the number of partitions by increasing the load balance which reduces the total number of pairs. Second, it produces square-like partitions which is expected to overlap with fewer partitions of the other dataset as compared to the very thin and wide partitions that the STR or other partitioning techniques produce. These advantages allows R*-Grove to significantly outperform other partitioning techniques in spatial join query performance. We will validate these advantages in the Section 2.6.5.

2.6 Experiments

In this section, we carry out an extensive experimental study to validate the advantages of R*-Grove over widely used partitioning techniques, such as bulk loading STR, Kd-tree, Z-Curve and Hilbert curve. We will show how R*-Grove addresses the current limitations of those techniques, leads to a better performance in big spatial data processing. In addition, we also show other capabilities of R*-Grove in the context of big spatial data, for example, how it works with large or multi-dimensional datasets. The experimental results in this section provide an evidence to the spatial community to start using R*-Grove if they would like to improve the system performance of their spatial applications.

Table 2.1: Datasets for experiments

Dataset	Type	Dimensions	Size	# records
(1) OSM-Nodes	Point	2	500GB	7.4 billions
(2) OSM-Roads	Line segments	2	20GB	59 millions
(3) OSM-Parks	Polygon	2	7.2GB	10 millions
(4) OSM-Objects	Polygon	2	96GB	264 millions
(5) NYC-Taxi	Point	4,5,7	46GB	173 millions
(6) Diagonal points	Point	3,4,5,9	100GB	80 millions

2.6.1 Experimental Setup

Datasets: Table 2.1 summarizes the datasets will be used in our experiments. We use both real world and synthetic datasets for our experiments: (1) Semi-synthetic OpenStreetMap (`OSM-Nodes`) dataset with 7.4 billion points and a total size of 500 GB. This is a semi-synthetic dataset which represents all the points in the world. The points in this dataset are generated within a pre-specified distance from original points from `OSM-Nodes` dataset; (2) `OSM-Roads` with size 20 GB and (3) `OSM Parks` with size 7.2 GB, which contain line segments and polygons for spatial join experiments. (4) `OSM-Objects` dataset with size 92GB, which contains many variable-size records. (5) `NYC-Taxi` dataset with size 41.7GB with up-to seven dimensions. All of those datasets are available online on UCR-STAR [180] - our public repository for spatial data; (6) Synthetic multi-dimensional `diagonal` points, with the number of dimensions are $3, 4, 5$, and 9. This synthetic dataset is generated using our open source Spatial Data Generator [191]. Dataset (5) and (6) allow us to show the advantages of R*-Grove in multi-dimensional datasets.

Parameters and performance metrics: In the following experiments, we partition the mentioned datasets with different datasets size $|D|$ in different techniques then we measure: (1) partition quality metrics, namely, total partition area, total partition margin,

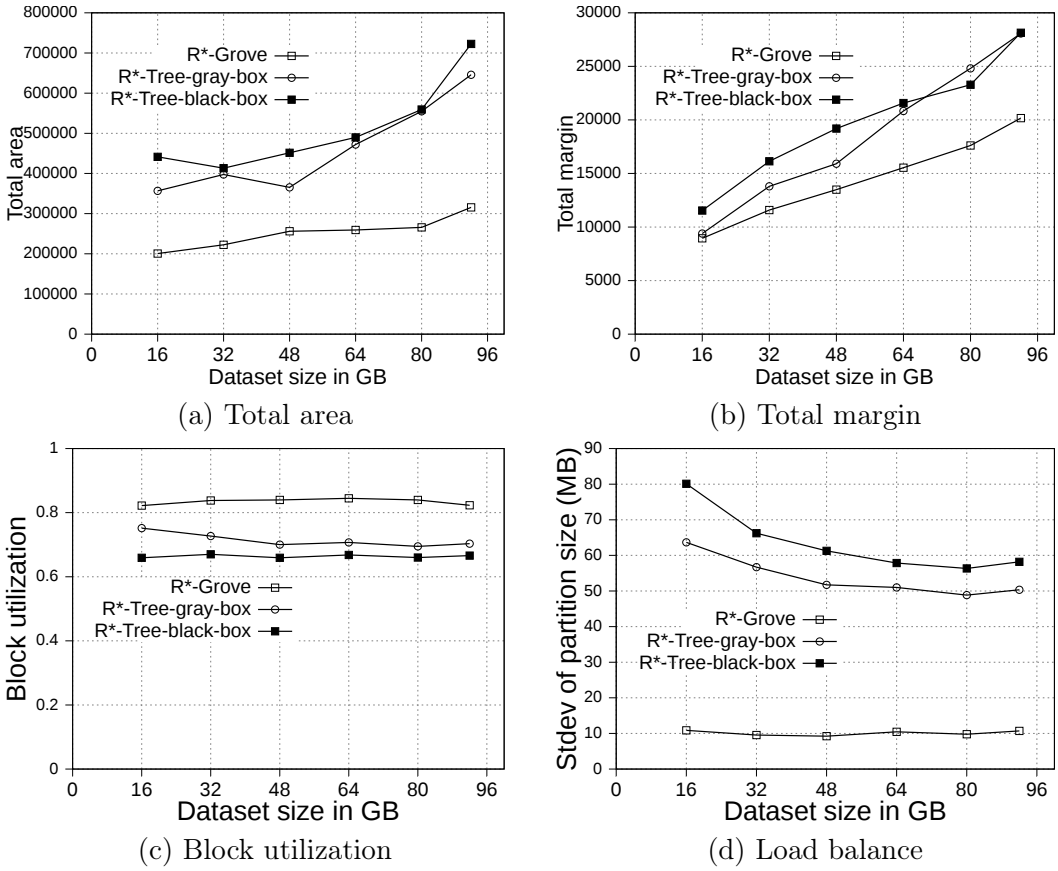
total partition overlap, block utilization(maximum is 1.0, i.e. 100%), standard deviation of partition size in MB (load balance). Notice that unit is not relevant for area, margin and overlap metric; (2) total partitioning time (in seconds), (3) for range queries, we measure the number of processed blocks and query running time, (4) for spatial join, we measure the number of processed blocks and total running time. We fix the balance factor $\alpha = 0.95$ and HDFS block size at 128 MB.

Machine specs: All the experiments are executed on a cluster of one head node and 12 worker nodes, each having 12 cores, 64 GB of RAM, and a 10 TB HDD. They run CentOS 7 and Oracle Java 1.8.0_131. The cluster is equipped with Apache Spark 2.3.0 and Apache Hadoop 2.9.0. The proposed indexes are available for running in both Spark and Hadoop. Unless otherwise mentioned, we use Spark by default. The source code is available at `https://bitbucket.org/tvu032/beast-tv/src/rsgrove/`. The implementation for R*-Grove (*RSGrovePartitioner*) is located at *indexing* package.

Baseline techniques: We compare R*-Grove to K-d Tree, STR, Z-curve and Hilbert curve (denoted H-Curve thereafter) which are widely used in existing big spatial data systems [74]. Z-Curve is adopted in some systems under the name Geohash which behaves in the same way.

2.6.2 Effectiveness of the proposed improvements in R*-Grove

In this experiment, we compare the three following variants of R*-Grove: (1) *R*-tree-black-box* is the application of the method in Section 2.4.1. Simply, it uses the basic R*-tree algorithm to compute high-quality partition but it does not guarantee a high block



(a) Total area (b) Total margin

(c) Block utilization (d) Load balance

Figure 2.5: Partition quality with variable-record-size dataset in R*-Grove and its two variants, R*-tree-black-box and R*-tree-gray-box

utilization or load balance. (2) *R*-tree-gray-box* applies the improvements in Sections 2.4.1 and 2.4.2. In addition to the high-quality partition, this method can also guarantee a high block utilization in terms of number of records per partition but it does not perform well if records have highly-variable sizes since it does *not* include the size adjustment technique in Section 2.4.3. (3) *R*-Grove* applies all the three improvements at Sections 2.4.1, 2.4.2 and 2.4.3. It has the advantage of producing high-quality partitions and can also guarantee a high block utilization in terms of storage size even when the record sizes are highly variable.

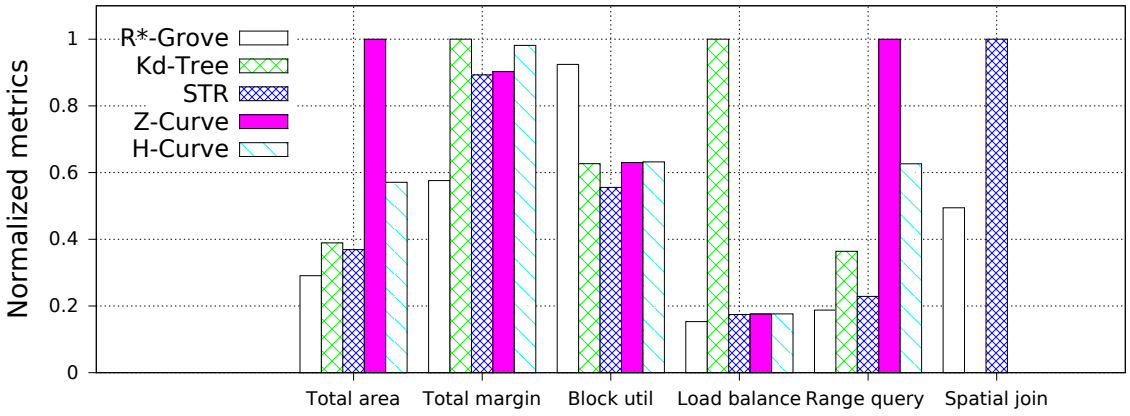
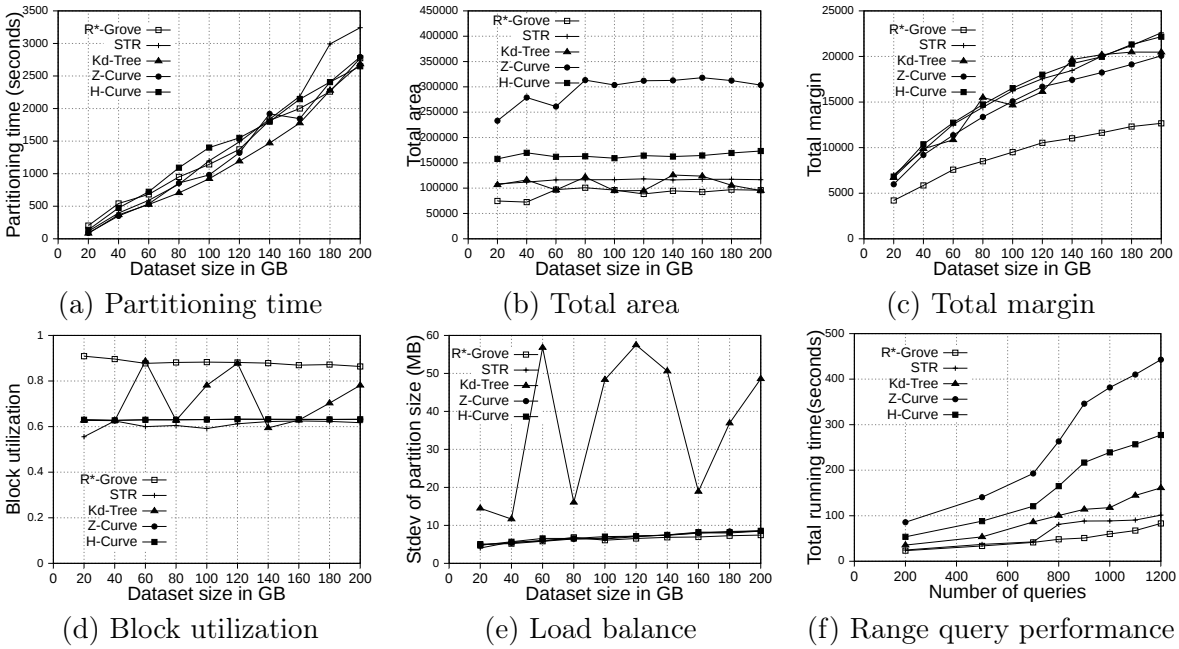


Figure 2.6: The advantages of R*-Grove when compared to existing partitioning techniques

In Figure 2.5, we partition the `OSM-Objects` dataset, which contains variable-size records to validate our proposed improvements. Overall, R*-Grove outperforms R*-tree-black-box and R*-tree-gray-box in all of spatial quality metrics. Especially, R*-Grove provides excellent load balance between partitions as shown in Figure 2.5(d), which is the standard deviation of partition size in `OSM-Objects` dataset. Given the HDFS block size is 128MB, R*-Grove has the standard deviation of partition size $5 - 6$ times smaller than R*-Tree-gray-box and R*-Tree-black-box. Since then, the following experiments will evaluate the performance of R*-Grove with existing widely-used spatial partitioning techniques.

2.6.3 Results Overview

Figure 2.6 shows an overview of the advantages of R*-Grove over other partitioning techniques for indexing, range query, and spatial join. In this experiment, we compare to four popular baseline techniques, namely, STR, Kd-Tree, Z-Curve and H-Curve. We use `OSM-Nodes` dataset [180] for this experiment. The numbers on the $y - axis$ are normalized to the largest number for a better representation except for block utilization which is reported



(a) Partitioning time (b) Total area (c) Total margin

(d) Block utilization (e) Load balance (f) Range query performance

Figure 2.7: Indexing performance and partition quality of R*-Grove and other partitioning techniques in `OSM-Nodes` datasets with similar-size records.

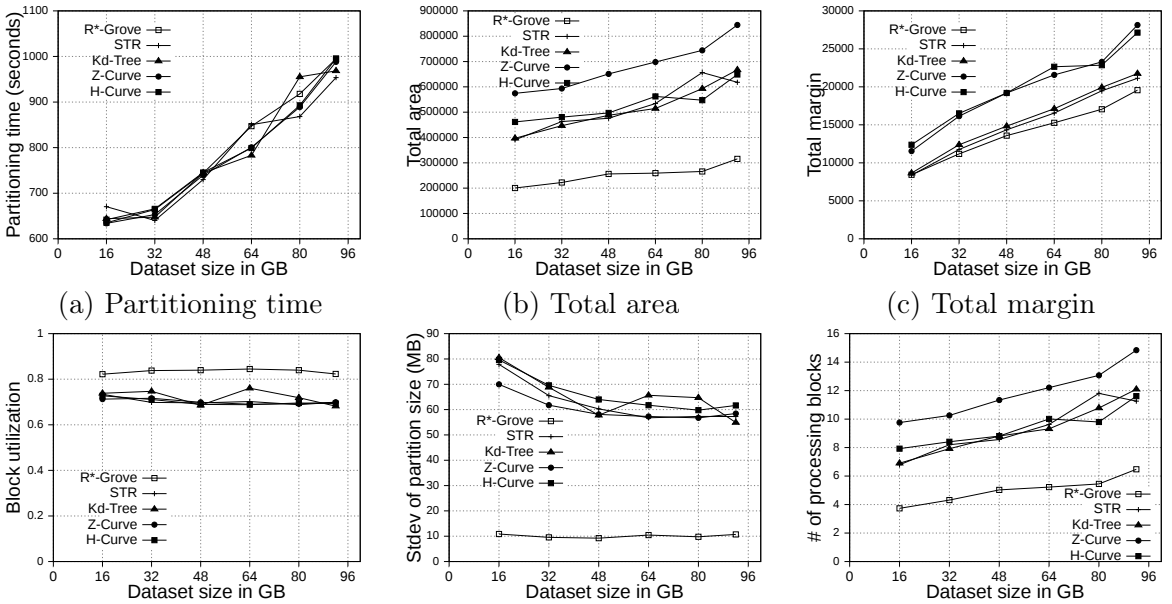
as-is. Except for block utilization, the lower the value in the chart the better it is. The first two groups, total area and total margin, show that index quality of R*-Grove is clearly better than other baselines in both measures. For block utilization, on average, a partition in R*-Grove occupy around 90%, while other techniques could only utilize $60 - 70\%$ storage capacity of an HDFS block. R*-Grove also has a better load balance when compared to other techniques. The last two groups indicate that R*-Grove significantly outperforms other partitioning techniques in terms of range query and spatial join query performance. We will go into further details in the rest of this section.

2.6.4 Partition quality

This section shows the advantages of R*-Grove for indexing big spatial data when compared to other partitioning techniques. We use `OSM-Nodes` and `OSM-Objects` dataset with size up to 200 and 92 GB, respectively. We compare five techniques, namely, R*-Grove, STR, Kd-Tree, Z-Curve and H-Curve. We implemented those techniques on Spark with sampling-based partitioning mechanism. Figures 2.7(a) and 2.8(a) show that there is no significant difference of indexing performance between different techniques. This result is expected since the main difference between them is in Phase 2 which runs on a single machine on a sample of a small size (and a histogram in case of R*-Grove). Typically, Phase 2 takes only a few seconds to finish. These results suggest that the proposed R*-Grove algorithm requires the same computational resources as the baseline techniques. Meanwhile, it provides a better query performance by providing a higher partition quality as detailed next.

Total area and total margin

Figures 2.7(b) and 2.8(b) show the total area of indexed datasets when we vary the `OSM-Nodes` and `OSM-Objects` dataset size from 20GB to 200GB and 16GB to 92GB, respectively. R*-Grove is the winner since it minimizes the total area of all partitions. While H-Curve performs generally better than Z-Curve, they are both doing bad since they do not take partition area into account in their optimization criteria. Specially, Figure 2.8(b) strongly validates the advantages of R*-Grove in non-point datasets. Figures 2.7(c) and 2.8(c) report the total margin for the same experiment. R*-Grove is the clear winner because it inherits the splitting mechanism of R*-Tree, which is the only one among all those that tries



(a) Partitioning time (b) Total area (c) Total margin

(d) Block utilization (e) Load balance (f) Average range query cost

Figure 2.8: Indexing performance and partition quality of R*-Grove and other partitioning techniques in `OSM-Objects` dataset with variable-size records.

to produce square-like partitions. As the input size increases, more partitions are generated which causes the total margin to increase.

Block utilization

Figures 2.7(d) and 2.8(d) show the block utilization as the input size increases. R*-Grove outperforms other partitioning techniques due to the proposed improvements in Section 2.4.2 and 2.4.3 specifically improve block utilization. Using R*-Grove, each partition almost occupies a full block in HDFS which increases the overall block utilization. Z-Curve and H-Curve perform similarly since they produce equi-sized partition by creating split points along the curve. The high variability of the Kd-tree is due to the way it partitions the space at each iteration. Since it always partitions the space along the median, it only works perfectly if the number of partitions is a power of two; otherwise, it could be very

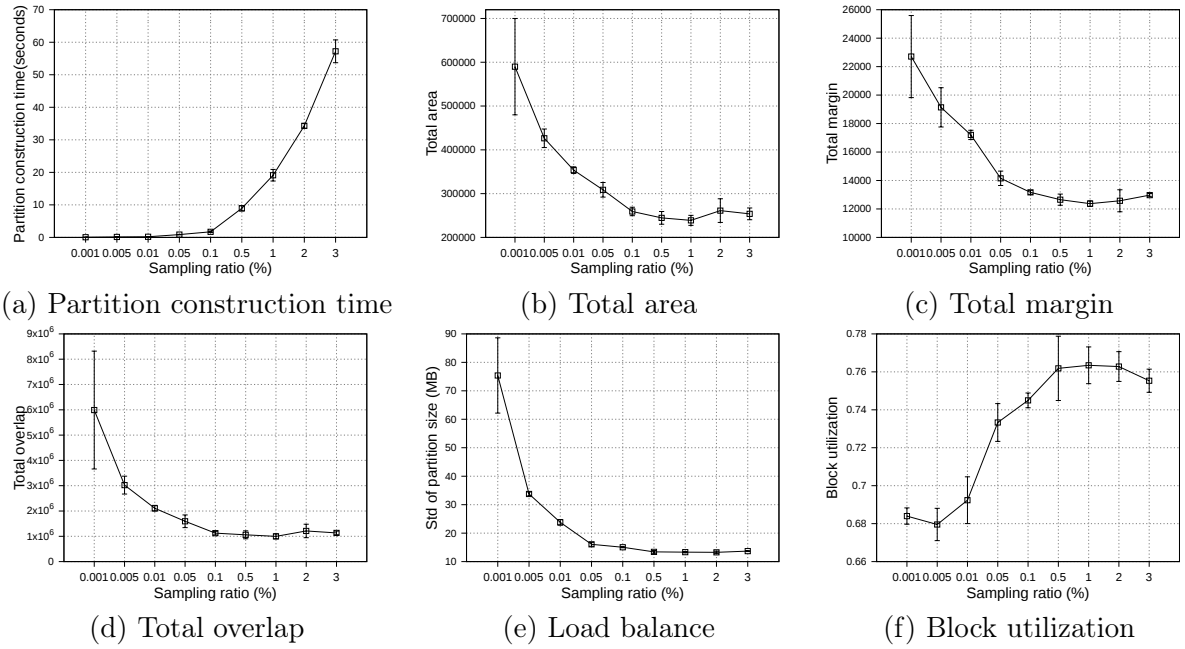
inefficient. This occasionally results in partitions of high block utilization but they could be highly variable in size.

Load balance

Figures 2.7(e) and 2.8(e) show the standard deviation of partition size in MB for the `OSM-Nodes` and `OSM-Objects` datasets, respectively. Note that the HDFS block size is set to 128 MB. A smaller standard deviation indicates a better load balance. In Figure 2.7(e), the dataset `OSM-Nodes` contains records of almost the same size so R*-Grove performs only slightly better than Z-Curve, H-Curve and STR even though these three techniques try to primarily balance the partition sizes. In Figure 2.8(e), the `OSM-Objects` dataset contains highly variable record sizes. In this case, R*-Grove is way better than all other techniques as it is the only one that employs the storage histogram to balance variable size records. In particular, we observe that the standard deviation of partition size on STR, Kd-Tree, Z-Curve and H-Curve is about $50 - 60\%$ of the HDFS block size. Meanwhile, R*-Grove maintains a value around 10 MB, which is only 8% of the block size.

Effect of sampling ratio

Since the proposed R*-Grove follows the *sampling-based* partitioning mechanism, a valid question is how the sampling ratio affects partition quality and performance? In this experiment, we execute several partitioning operations using R*-Grove in `OSM-Objects` datasets. All the partitioning parameters are kept fixed, except the sampling ratio, which is varying from 0.001% to 3%. For each sampling ratio value, we execute the partition

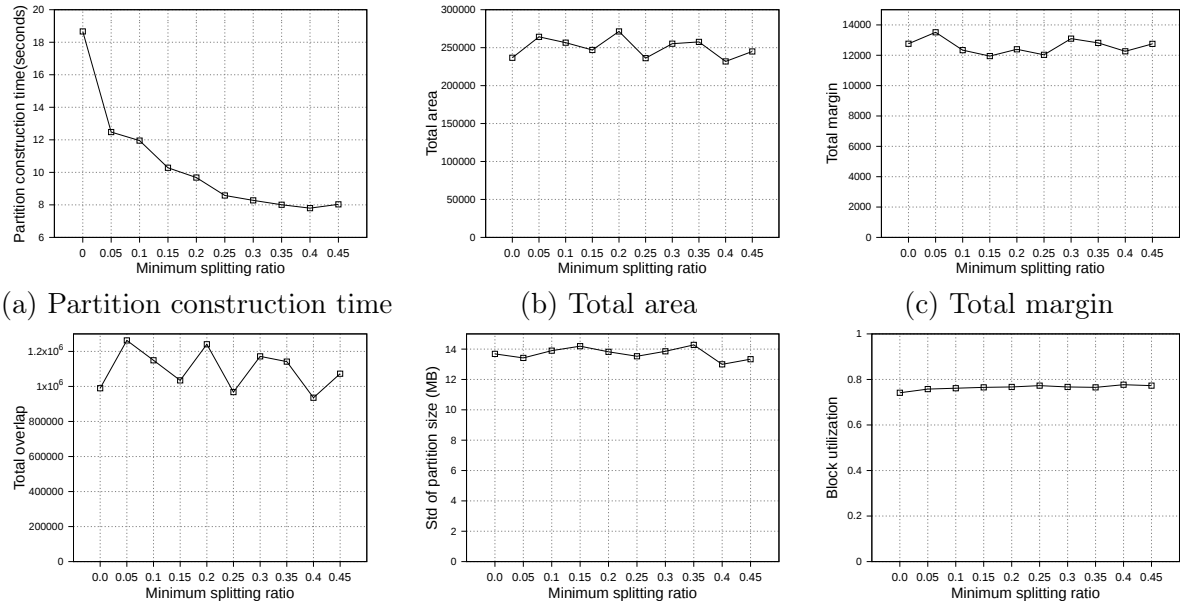


(a) Partition construction time (b) Total area (c) Total margin

(d) Total overlap (e) Load balance (f) Block utilization

Figure 2.9: Indexing performance and partition quality of R*-Grove in `OSM-Objects` datasets with different sampling ratios.

operation three times, then compute the average and standard deviation of quality measures and partition construction time. Partition construction is the process that compute partition MBRs from the sample. Figure 2.9 uses the average values to plot the line and the standard deviation for the error bars. First, Figure 2.9(a) shows that the higher sampling ratio requires higher time for the partition construction process. This is expected due to the number of sample records which the partitioner has to use to compute partition MBRs. Figures 2.9(b,c,d,e) show the downward trend of total area, total margin, total overlap and standard deviation of partition size. Figure 2.9(f) shows the upward trend of block utilization when the sampling ratio increases. In addition, the standard deviation of small sampling ratios is much higher than that for high sampling ratios. These results indicates that the higher sampling ratios promise better partition quality. However, an important observation



(a) Partition construction time (b) Total area (c) Total margin

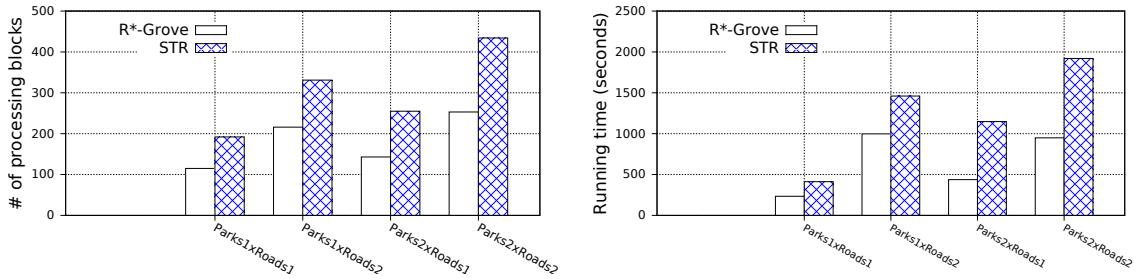
(d) Total overlap (e) Load balance (f) Block utilization

Figure 2.10: Indexing performance and partition quality of R*-Grove in `OSM-Objects` datasets with different minimum splitting ratios.

is that the partition quality measures start stabilizing for sampling ratios larger than 1%. This behavior was also validated in a previous work [66]. In short, this work shows that a sample ratio of 1% dataset is enough to achieve virtually a same partition quality as sample ratio 100%. In the following experiments, we choose 1% as the default sampling ratio for all partitioning techniques.

Effect of minimum split ratio

In Section 2.4.1, we introduced parameter ρ, namely minimum splitting ratio, to speed up the running time of SplitNode algorithm used in Phase 2, boundary computation. In this experiment, we verify how the minimum splitting ratio impacts the partition quality and performance. We also use `OSM-Objects` dataset with R*-Grove partitioning as the



(a) Number of processing blocks (b) Running time in seconds

Figure 2.11: Spatial join performance in R*-Grove and STR partitioning

previous experiment in Section 2.6.4. We vary ρ from 0 to 0.45. Figure 2.10 shows the overview of the experimental results. First, Figure 2.10(a) shows that the running time of Phase 2, boundary computation, decreases as ρ increases which is expected due to the balanced splitting in the recursive algorithm which causes it to terminate earlier. According to the run-time analysis in Section 2.4.1, a larger value of ρ reduces the depth of the recursive formula which results in a lower running time. However, this minimum splitting ratio also shrinks the search space for optimal partitioning scheme. Fortunately, the number of records in the 1% sample is usually large enough such that the boundary computation algorithm could still find a good partitioning scheme even for high value of ρ. In the following experiments, we choose $\rho = 0.4$ as the default value for R*-Grove partitioning.

2.6.5 Spatial query performance

Range query

Figure 2.7(f) shows the performance of range query on the `OSM-Nodes` dataset with size 200GB. For partitioned `OSM-Nodes` dataset, we run a number of range queries (from 200 to $1,200$) all with the same range query size which is 0.01% of the area covered by the

entire input. All the queries are sent in one batch to run in parallel to put the cluster at full utilization. It is clear that R*-Grove outperforms all other techniques, especially when we run a large number of queries. This is the result of the high-quality and load-balanced partitions which minimize the number of blocks needed to process for each query. Figure 2.8(f) shows the average cost of a range query on the `OSM-Objects` dataset in terms of number of blocks that need to be processed, the lower the better. This value is also computed for a range query with size 0.01% of space area. This result further confirms that R*-Grove provide a better query performance for variable-size records datasets.

Spatial join

In this experiment, we split `OSM-Parks` and `OSM-Roads` datsets to get multiple datasets as follows: `Parks1`, `Park2` with sizes 3.6 and 7.2 GB; `Roads1` and `Roads2` with sizes 10 and 20 GB, respectively. This allows us to study the effect of the input size on the spatial join query while keeping the input data characteristics the same, i.e., distribution and geometry size. We compare to STR since it is the best competitor of R*-Grove in previous experiments. Figure 2.11 shows the performance of the spatial join query. In general, R*-Grove significantly outperforms STR in all query instances.

Figure 2.11(a) shows the number of accessed blocks for each spatial join query over the datasets which are partitioned by R*-Grove and STR. We can notice that R*-Grove needs to access 40%-60% fewer blocks than STR for two reasons. First, the better load balance in R*-Grove reduces the overall number of blocks in each dataset. Second, the higher partition quality in R*-Grove results in fewer overlapping partitions between the two datasets. The number of accessed blocks is an indicator to estimate the actual performance of spatial

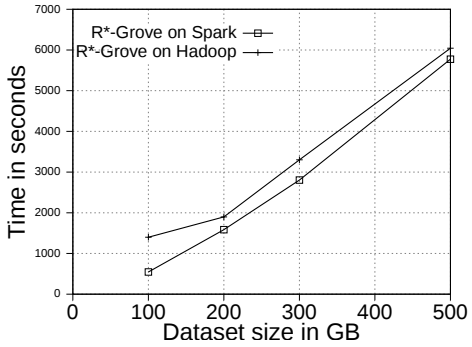


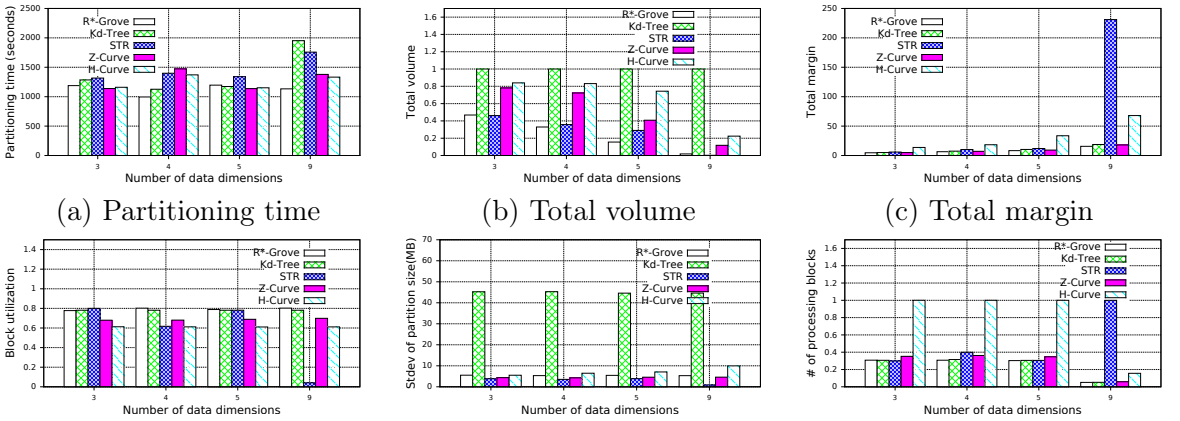
Figure 2.12: The scalability of R*-Grove partitioning in Spark and Hadoop

join queries. Indeed, this is further verified in Figure 2.11(b), which shows actual running time for those queries. As we described, STR does not produce high quality partitions, thus the compound effect will even make it worst for spatial join query, which always relates to multiple STR partitioned datasets. On the other hand, R*-Grove addresses the limitations of STR so it can significantly improve the performance of spatial join query.

2.6.6 Performance on larger datasets and multi-dimensional data

Scalability

Figure 2.12 shows the indexing time for two-dimensional `OSM-Nodes` dataset with sizes $100, 200, 300$ and 500GB. We executed the same indexing jobs in both Spark and Hadoop to see how the processing model affects the indexing performance. We observed that Spark outperforms Hadoop in terms of total indexing time. This experiment also demonstrates that R*-Grove is ready to work with large volume datasets on both Hadoop and Spark. We also observe that the gap between Hadoop and Spark decreases as the input size increases as Spark starts to spill more data to disk.



(a) Partitioning time (b) Total volume (c) Total margin

(d) Block utilization (e) Load balance (f) Range query performance

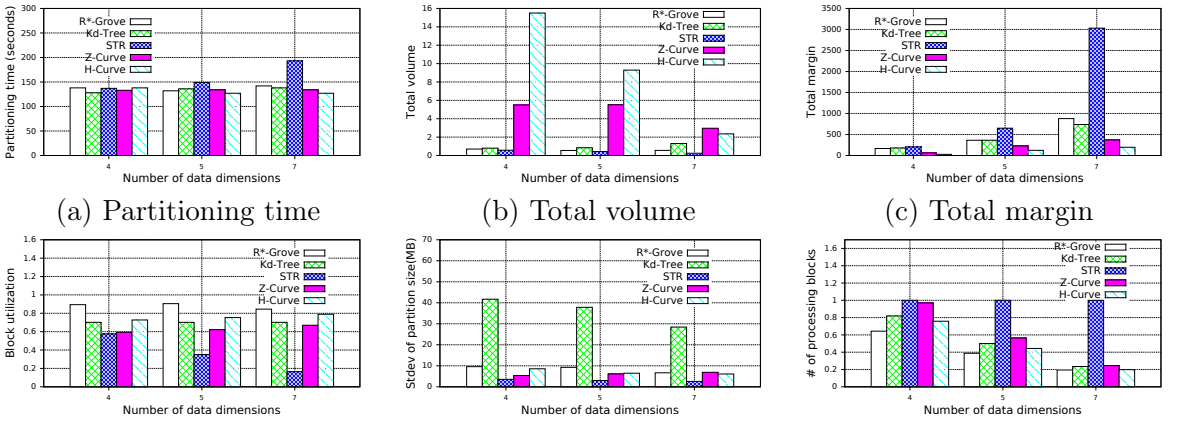
Figure 2.13: Indexing performance and partition quality of R*-Grove and other partitioning techniques on `synthetic` multi-dimensional dataset.

Multi-dimensional datasets

In this experiment, we study the quality of R*-Grove on multi-dimensional datasets. Inspired by [28], we use four `synthetic` datasets with number of dimensions 3, 4, 5, and 9. We measure the running time and the quality of the five partitioning techniques: R*-Grove, STR, Kd-Tree, Z-Curve and H-Curve. Figure 2.13(a) shows that R*-Grove is mostly the fastest technique to index the input dataset due to the best load balance among partitions. Figure 2.13(b) shows that R*-Grove significantly reduces total area of partitions. Figure 2.13(c) shows the total margin of all the techniques. While the total margin varies with the number of dimensions since they are different datasets, the techniques maintain the same order in terms of quality from best to worst, i.e., R*-Grove, Z-Curve, Kd-tree, STR and H-Curve, except the last group, where H-Curve is better than STR. This experiment indicates that R*-Grove could maintain its characteristics for multi-dimensional datasets. Figure 2.13(d) and 2.13(e) report the block utilization and standard deviation of partition size,

respectively. R*-Grove is the best technique that keeps both measures good. Figure 2.13(f) depicts the normalized range query performance of different techniques, which affirms the advantages of R*-Grove. Notice that this is the only experiment where Z-Curve performs better than H-Curve. The reason is that the generated points are generated close to a diagonal line in the d-dimension. Since the Z-Curve just interleaves the bits of all dimensions, it will result in sorting these points along the diagonal line which results in a good partitioning. However, the way H-Curve rotates the space with each level will cause it to jump across the diagonal.

Additionally, STR becomes very bad as the number dimensions increases. This can be explained by the way STR computes the number of partitions given a sample data points. The existing STR implementation always creates a tree with a fixed node degree n and d levels where d is the number of dimensions. This configuration results in n^d leaf nodes or partitions. It computes the node degree n as the smallest integer that satisfies $n^d \geq P$ where P is the number of desired partitions. For example, for an input dataset of 100 GB, $d = 9$ dimensions, and a block size of $B = 128$ MB, the number of desired partitions $P = 100 \cdot 1024/128 = 800$ partitions and $n = 3$. This results in a total of $3^9 = 19683$ partitions. Obviously, as d increases, the gap between the ideal number of partitions P and the actual number of partitions n^d increases which results in a very poor block utilization as shown in this experiment. Finally, Figure 2.13(f) shows the average cost of a range query in terms of number of processed blocks, which indicates that R*-Grove is the winner when we want to speed up spatial query processing.



(a) Partitioning time (b) Total volume (c) Total margin

(d) Block utilization (e) Load balance (f) Range query performance

Figure 2.14: Indexing performance and partition quality of R*-Grove and other partitioning techniques on multi-dimensional `NYC-Taxi` dataset.

To further support our findings, we also execute similar experiment on `NYC-Taxi` dataset, which contains up to seven dimensions as follows: $pickup\_latitude$, $pickup\_longitude$, $dropoff\_latitude$, $dropoff\_longitude$, $pickup\_datetime$, $trip\_time\_in\_secs$, $trip\_distance$. These attribute values are normalized in order to avoid the dominance of some columns. We decide to partition this dataset using multiple attributes which is picked in the aforementioned order with size $4, 5$ and 7. Figure 2.14 shows that R*-Grove balances all the different quality metrics. Specially, Figure 2.14(f) indicates that R*-Grove is the winner when compared to other techniques in terms of spatial query performance. We also notice that H-Curve performs better than Z-Curve with this real dataset. We conclude that R*-Grove is a better option for indexing multi-dimensional spatial data since it outperforms or got an equivalent performance with other indexes in all metrics.

2.7 Conclusion

This chapter proposes R*-Grove, a novel partitioning technique which can be widely used in many big spatial data processing systems. We highlighted three limitations in existing partitioning techniques such as STR, Kd-Tree, Z-Curve and Hilbert Curve. These limitations are the low quality of the partitions, the imbalance among partitions, and the failure to handle variable-size records. We showed that R*-Grove overcomes these three limitations to produce high quality partitions. We showed three case studies in which R*-Grove can be used to facilitate big spatial indexing, range query, and spatial join. An extensive experimental evaluation was carried out on big spatial datasets and showed that R*-Grove is scalable and speeds up all the operations in the case studies. We believe that R*-Grove promises to be a good replacement to existing big spatial data partitioning techniques in many systems. In the future, we will further study the proposed technique for in-memory and streaming applications to see how it behaves under these architectures.

Chapter 3

Using deep learning for big spatial data partitioning

3.1 Introduction

In recent years, there has been a notable increase in the amount of spatial data produced by IoT sensors, social networks, and autonomous vehicles, among others. This led to many research efforts for developing *big spatial data* frameworks that are able to absorb and process these huge amounts of data such as SpatialHadoop [72], Simba [199], GeoSpark [203], and others [74, 176, 167]. Regardless of their internal architecture, all these systems have a common and necessary first step, that is, *spatial data partitioning*. These systems *scale out* by partitioning the data across machines and then processing these partitions in parallel. However, there is no single partitioning technique that all the systems agree on. Rather, most of these systems provide a wide range of spatial partitioning techniques and it is up to the

user to choose an appropriate one. Past studies showed that the spatial partitioning approach is critical to the performance of many spatial analytic operations such as indexing [62], computational geometry [68], visualization [82], spatial joins [72], kNN joins [134], and others.

Choosing an appropriate spatial partitioning technique is a very challenging and complicated problem for two reasons. First, the efficiency of these partitioning techniques rely on the characteristics and distribution of the dataset, e.g., uniform Vs skewed data, points Vs rectangles, or clustered Vs scattered data. Second, the requirements of the analytic operations play a huge role in choosing a partitioning technique, e.g., maximize load balancing, minimize partition overlap, or prefer square-like partitions. Recent studies provided both theoretical [38, 40] and experimental [62] evaluations of several partitioning techniques for big spatial data and highlighted the complexity of choosing one technique over the others. As new partitioning techniques are developed [187], the problem becomes even more complex.

Table 3.1: Execution of the DJ in SpatialHadoop with different kinds of indexes (i.e., Gr = regular grid, Qt = Quadtree, Rt = R-tree) and different distributions of the datasets (i.e., Uni = uniform distribution, Skw = skewed distribution). # tasks is the total number of map tasks, AVG time is the average time for a map task, and %RSD is the relative standard deviation for the running time of map tasks.

Dataset distribution	**Dataset index**	**Tot. time (mills)**	**Map tasks**		
			# tasks	AVG time (millis)	%RSD time
Uni/Uni	**Gr/Gr**	**145,307**	37	15,833	4%
Uni/Uni	Gr/Qt	150,458	51	18,902	9%
Uni/Uni	Gr/Rt	147,646	54	16,231	7%
Uni/Skw	Gr/Gr	125,327	33	22,710	90%
Uni/Skw	Gr/Qt	96,001	52	11,209	50%
Uni/Skw	**Gr/Rt**	**40,205**	21	18,087	28%

To illustrate the complexity of the problem, Table 3.1 shows the result of the execution in SpatialHadoop of the Distributed Join (DJ) [64, 39] applied to two synthetic datasets, where the first one is uniformly distributed (i.e., "*Uni*") and partitioned using a regular grid (i.e., "*Gr*"), while the second one varies from a uniform (i.e., "*Uni*") to a skewed (i.e., "*Skw*") distribution and has been partitioned using different techniques, namely regular grid (i.e., "*Gr*"), Quad-tree (i.e., "*Qt*") and R-tree (i.e., "*Rt*"). Interestingly, when both datasets are uniformly distributed, the response time of the DJ is the best with the uniform grid partitioning with Rt and Qt coming as close second and third. On the other hand, when a skewed distributed dataset (*Skw*) is considered, then the differences are significant and in this particular case are in favor of the R-tree-based partitioning technique. This is due mainly to the fact that when the distribution is skewed the partitioning of the geometries based on a regular grid does not produce balanced splits, while the Quad-tree and the R-tree-based partitioning techniques perform better and produce more balanced splits. This is evident from columns 4, 5 and 6 of Table 3.1, which report the characteristics of the map tasks in the different cases. In particular, column 4 contains the number of instantiated map tasks (which depends on the pair of intersecting partitions from both datasets), column 5 reports the average time taken by a map task, and column 6 shows the relative standard deviation of the execution time of the map tasks w.r.t to their mean signifying the load balance. It is clear that balancing the cost of the single map tasks is crucial for the total cost of the MapReduce job, in particular when the implemented operation is performed primarily in the map phase.

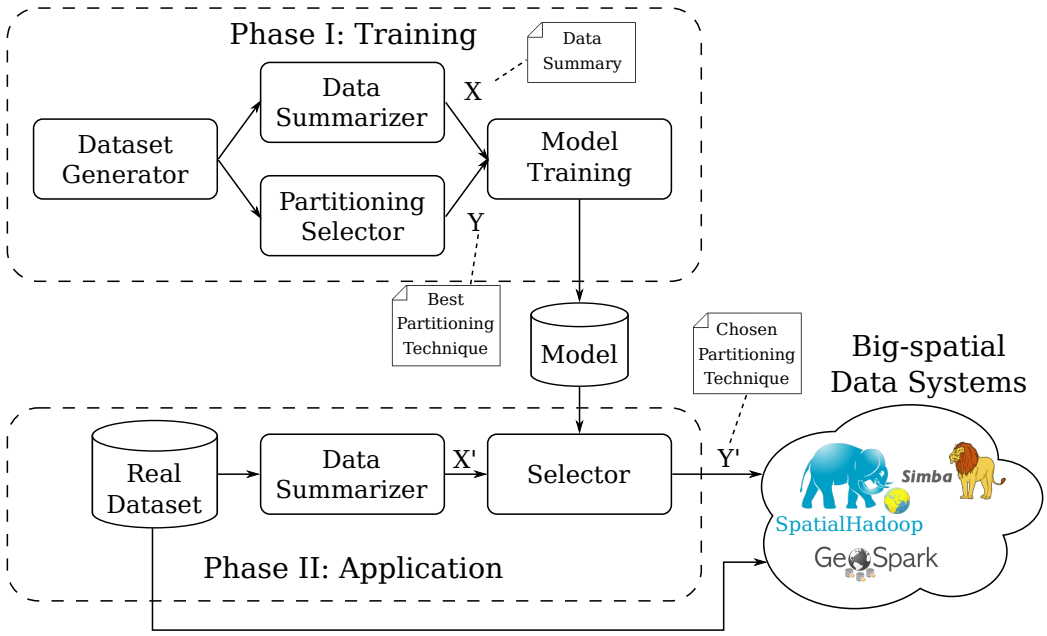


Figure 3.1: The workflow of the proposed solution

The aim of this chapter is to define a new mechanism for choosing the most appropriate partitioning technique for a given dataset. There are three design goals for the proposed work: (1) ability to make a decision based on parameters that can be computed quickly, (2) support arbitrarily many partitioning techniques, and (3) provide different choice criteria based on the requirements of the analytic operation and the user preferences.

To achieve the three goals mentioned above, we propose the framework illustrated in Figure 3.1. The framework works in two main phases, namely, *training* and *application* phases. In the training phase, we build the *partition selection model* that is able to choose an appropriate index for any given dataset. This phase is executed as an offline phase and it consists of the following four components.

- **Dataset Generator:** This component generates many diverse synthetic datasets that are used to train the model. This step is important for deep learning which needs a very large training set that catches as many input features as possible.

- **Data Summarizer:** This component takes every input dataset and computes a set of descriptors that summarizes the dataset and catches its details. This step transforms the variable-size input dataset to a fixed-size feature vector X that the deep learning algorithm can process. This chapter considers two summarization techniques, *fractal-based* techniques, which utilize skewness measures that are developed by experts, and a simple *histogram* that represents a detailed density map.

- **Partitioning Selector:** This component assesses the quality of all supported spatial partitioning techniques to choose the best one. It evaluates the performance of all the available partitioning techniques using a set of standard quality metrics and generates a label Y that contains the best partitioning technique for each quality metric. The deep learning can use the pair (X, Y) for training the model.

- **Model Training:** This last step takes the feature vector X and the label vector Y and uses deep learning to build a model that can estimate the performance vector Y given the features X.

The second phase, *application phase*, uses the model produced by Phase I and applies it on a new (real) dataset, provided by the user, and chooses the most appropriate partitioning technique for it. This phase first computes the feature vector X' exactly as in Phase I but on the real dataset. Then applies the model M on the vector X' to produce an estimated performance vector Y' that encodes the most appropriate partitioning technique.

The chosen technique, taking into account also the user requirements (i.e., which operation she/he needs to apply), can then be passed to any big-spatial data system, e.g., SpatialHadoop or GeoSpark, for actual data partitioning and analysis.

In this chapter, we build a prototype for the proposed system using six different partitioning techniques, five quality metrics, and two data summarization techniques. The first summarization technique uses a few well-crafted skewness measures for spatial data including box counting [38, 40] and Moran's Index [142]. The second summarization technique uses a simple histogram for the entire dataset which represents a details density map but could be harder to use by the machine learning component due to their big size. One of the goals of this chapter is to study which summarization technique works better for this problem. In other words, can the deep learning technique extract its own skewness measures from the histogram that outperforms the ones developed by the experts? We test the proposed framework using both synthetic and real big datasets to show the effectiveness of the proposed framework. The initial experiments show up-to 90% accuracy with synthetic data and 80% with real data.

In summary, the contributions of the chapter are listed and presented hereby.

1. **Training set generation:** Deep learning model require a sufficiently large and representative training set. In the considered context where the problem to address is to choose the most suitable partitioning technique for a given spatial dataset of unknown distribution, no training set is available (unlike the image classification problem where huge repositories are freely available on Internet). So the first contribution of this chapter is to propose an approach for generating a training set addressing this kind

of problems, and this includes a set of algorithms for producing the training set in a reasonable amount of time. In particular, the application that generates the training set has been implemented in Spark.

2. **Feature extraction:** Once a training set is generated, we need to decide the features that should be extracted from the dataset to use as input to the machine learning model. There is an agreement that the distribution of the dataset is the key feature for choosing the best partitioning technique but the question is: which descriptors should be chosen? Which statistical descriptor is the best one for supporting the choice of a correct partitioning technique? To answer this question, this chapter proposes two techniques. The first technique extracts an ensemble of carefully selected skewness measures that have been shown to catch several important features of spatial data including box-counting [38, 40] and Moran's Index. This techniques resembles classical image processing techniques that extracts manually designed image features. The second technique uses the dataset histogram as one big feature and let the modern deep learning method extracts its own features from the histogram. We show in this chapter that this method is easier to implement since it avoids hand-picking the skewness measures and, thanks to deep learning, can provide a very high accuracy.

3. **Experimental evaluation:** Finally, as third contributions we configure, train and test a Neural Network proving that the proposed idea is feasible. In the experiments we use a considerable amount of synthetic datasets with different distributions and some real huge datasets. The results support our intuition that the histograms can be a good choice for addressing the optimization issue regarding data partitioning.

The rest of this chapter is organized as follows. Section 3.2 formalizes the problem. Section 3.3 describes the training phase which builds the partition selection model. Section 3.4 describes the application phase applying the model to real datasets provided by the user. Section 5.4 provides an extensive experimental evaluation of the proposed system using real datasets. Section 3.6 describes the related work. Finally, Section 5.5 concludes the chapter.

3.2 Problem definition

The problem that this chapter addresses is, given a spatial dataset, how to choose the best partitioning technique that will provide the best performance. Considering the case study shown in Table 3.1, it is evident that this is an important yet challenging problem given the complexity of big spatial datasets. In addition, the objectives of the spatial partitioning vary by the spatial operation that will be applied and the requirements of the system that applies this operation. For example, in selection and join operations, it could be desired to minimize the total area or total margin of the partitions [26, 29, 62]. On the other hand, for computational geometry operations [68], minimizing or eliminating the overlap between partitions could be more beneficial. For scanning and aggregate operations, load balance (i.e., minimize the variance) could be of a high advantage to minimize the straggler effect. This section aims at clearly defining the problem which includes how to identify the *best* partitioning technique.

Definition 11 (Feature (f))**.** *A spatial feature f represents a record that contains a geometry g and a set of non-spatial attributes $A = \{a_i\}$. The minimum bounding rectangle $f.MBR$ is the smallest orthogonal rectangle that encloses the geometry g. The size $f.s$ is the total size*

of the feature representation, i.e. geometry plus attributes, in bytes. In this chapter, we do not process the actual geometry or attributes, rather, we only consider the MBR and size. A feature is also referred to as a record *following the database terminology.*

Definition 12 (Partition (P))**.** *A spatial partition $P = \{f_1, \ldots, f_m\}$ is a set of spatial features that are stored in the same file block(s). The MBR, size, total number of blocks and average cardinality of blocks of the partition are defined as:*

$$P.MBR = MBR\left(\bigcup_{f \in P} f.g\right) = MBR\left(\bigcup_{f \in P} f.mbr\right)$$

$$P.s = \sum_{f \in P} f.s$$

$$P.blocks = \lceil P.s/B \rceil$$

$$P.card = \frac{\mid P \mid}{P.blocks}$$

where B is the block size of the file system which has a default value of 128 MB in HDFS.

Any partitioning technique aims at producing partitions having at most one block, however in practice the application of a technique to a real dataset D might produce also partitions containing more than one block, due to the particular distribution of the features of D in the reference space.

Definition 13 (Partitioning Technique (PT))**.** *A partitioning technique ($PT : D \rightarrow \mathcal{P}$) is a function that can be applied to a dataset $D = \{f_i\}$ to produce a set of partitions $\mathcal{P} = \{P_k\}$ such that each feature f_i is assigned to at least one partition, i.e., $\bigcup_{P_k \in \mathcal{P}} P_k = D$.*

Definition 14 (Quality Metric (QM)). *The quality metric ($QM : \mathcal{P} \rightarrow \mathbb{R}$) is a function that is applied to a set $\mathcal{P}$ of partitions to quantify its quality as real number $qm \in \mathbb{R}$, e.g., the total area of partitions or standard deviation of the partition sizes or a combination of them.*

Notice that the quality metric to be chosen might depend on the user requirements, i.e., the requested operation. Moreover, the quality metrics used in this chapter are better when having lower values, e.g., total area or total margin. However, there exist other quality metrics for which the higher the value the better, e.g., disk utilization. The approach proposed in this chapter can handle both types of quality metrics.

Next, we define the main problem that we address in this chapter.

Definition 15 (Partitioning Selection Problem (PSP)). *Given a spatial dataset D, a set of partitioning techniques $PT = \{PT_1, \ldots, PT_n\}$, and a quality metric QM, choose the best partitioning technique PT_i that will minimize/maximize the quality metric QM when applied to the dataset D.*

A naïve solution to the PSP problem is to apply all partitioning techniques to the big dataset and then compute the quality metric for all the resulting partitions and choose the best one. However, since the big spatial data frameworks deal with peta bytes of data, it is not feasible or effective to apply all possible partitioning techniques.

This chapter proposes a solution to this problem through a framework that uses deep learning to predict the best partitioning technique based on a history of how all partitioning techniques behave with datasets that are similar to the input dataset D. At a very high-level, the proposed framework works in two phases, training and application. The training phase looks at a huge number of reference datasets and their quality when partitioned with all

the available partitioning techniques. Then, it builds a small model M that captures this complicated relationship. The application phase takes a new dataset D and applies that model on D to choose a partitioning technique that is expected to be the best. This entails the following *challenging problems* that we address in this chapter.

- **Dataset generation:** How to generate large and diverse reference datasets that can be used for training? These datasets should capture as many aspects of the partitioning techniques as possible. They should also simulate real datasets so that the generated model can be used with real data. We address this problem by surveying a large number of synthetic data distributions used in literature and choosing a set of representative distributions that are close to real datasets. Then, we generate a large number of datasets for each distribution by varying its parameters. Finally, we combine the generated datasets to generate more compound distributions that cannot be represented by a single distribution. This process has been done with the support of our open-source spatial data generator [192].

- **Dataset similarity:** One of the biggest problems is how to measure the similarity between different datasets including real datasets that are only available in the second phase. We evaluate and contrast two directions. The first direction uses some skewness measures defined by the experts such as box counting [38] and Moran's index [142]. The second direction uses a simple uniform histogram that is easier to compute but of a much larger size. The second option is particularly intriguing to use with deep learning as the histogram looks like an image which deep learning is particularly good at.

- **Performance evaluation:** Given a dataset D, a set of partitioning techniques PT, and a set of quality metrics QM, how to measure all the quality metrics for all the partitioning techniques on the dataset D to be able to identify the best one for training purpose? We address this problem by proposing a distributed Spark-based algorithm that is able to generate the partitions $\mathcal{P}$ for all partitioning techniques as one distributed job without really having to partition the actual features of D. This technique allows us to generate a large number of reference datasets in a short time to improve the accuracy of the model during the training phase.

- **Model training:** Given the reference datasets and their corresponding quality measures, how to build a model that captures this complicated relationship? To address this problem, we use deep learning to build such a model and explain in this chapter how we choose the parameters for this model and do the model training.

3.3 Training Phase

The training phase is responsible of building the machine learning model M that can choose the best partitioning technique for a dataset D. This phase works in four steps. (1) Generate a set of reference datasets to use as training set. (2) Summarize each training dataset into a fixed-size vector that is used for training. (3) Compute all quality metrics for each dataset and label each dataset with the best partitioning technique for each quality metric. (4) Apply deep learning to learn the relationship between the data summary and the best technique. Details of the four steps are provided below.

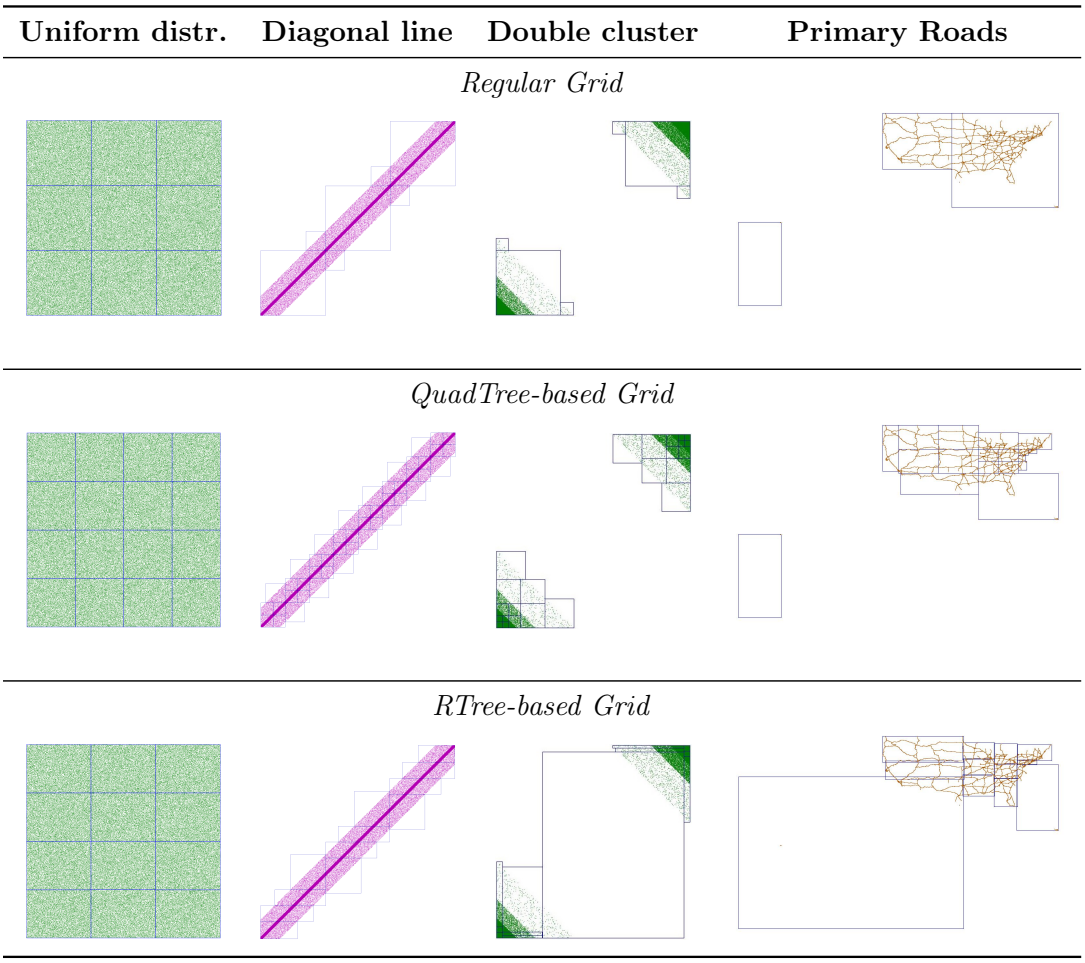


Figure 3.2: Partitions produced by applying different techniques (1st row: regular grid, 2nd row: QuadTree-based grid, 3rd row: RTree-based grid) for both synthetic and real datasets.

3.3.1 Training set generation

This section describes the distributions of the synthetic datasets that we use for model training. Different distributions of geometries in the reference space produce different behavior of the partitioning techniques, which provide very different subdivisions of the features in the resulting partitions. Fig. 3.2 illustrates an example with four datasets: a uniformly distributed set of rectangles (*Uniform distr.*), a set of rectangles distributed around

the diagonal of the reference space (*Diagonal line*), a set of rectangles distributed around the lower left and upper right corners of the reference space (*Double cluster*) and a real dataset containing the primary roads of the USA (*Primary roads*). Three partitioning techniques have been applied: regular grid, QuadTree and RTree to all the datasets. The resulting partitions are shown by drawing the boundary of their MBRs on top of the datasets plots. Notice that the MBRs produced by different techniques are very different from each other. Thus, the dataset distribution is a vital characteristic for deciding the correct partitioning technique. In order to build an effective training set, it is crucial to generate datasets with different distribution, in particular with different kind of skewed distributions.

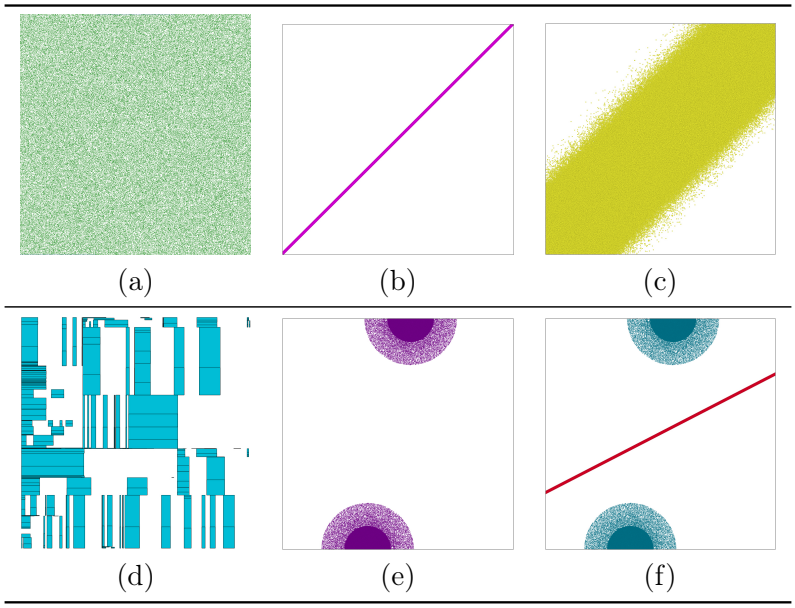


Figure 3.3: Example of distributions contained in the training set.

For all datasets, two common parameters are set, the reference space (a bounding rectangle of the input space), and the total size. In addition, each distribution can have some

additional parameters that control the dataset generation. In particular, we consider the following dataset distributions exemplified in Fig. 3.3:

- Uniform distribution: the dataset geometries are uniformly distributed inside the reference space (Fig. 3.3.a). A parameter s is adjusted to represent the maximum side length of each rectangle. This distribution models real datasets that are uniformly distributed, e.g., houses in suburbs.

- Linear distribution: the dataset geometries are all located very close to a line, namely they are uniformly distributed inside a small buffer around it (Fig. 3.3.b). The training set considers as reference line both the main diagonal of the reference space, and about 100 possible rotations of it. This distribution can be customized by setting the maximum side length of a rectangle (s) and the size of the buffer (b). This distribution can represent data that are centered around a line, e.g., shops along a highway or houses along a river.

- Diagonal distribution: the dataset geometries are located around a line with a normal distribution. More specifically, the concentration of the geometries decreases as the distance from the main line increases (Fig. 3.3.c). In generating the various datasets, the percentage of geometries concentrated around the line and the dimension of overall buffer are changed. Moreover, beside to the main diagonal, we consider as reference line also about 100 possible rotations of it. This distribution can model data around a linear region such as river banks.

- Parcel distribution: this dataset is generated by recursively splitting the reference space by horizontal and vertical lines. After that, each resulting rectangle is randomized by

slightly changing its size (Fig. 3.3.d). The parameter r represents the randomization factor as a percentage of the rectangle size. Parcel distribution can model some real datasets such as farm lands and green areas that cover a large region with slight or no overlap.

- Cluster distribution: the dataset geometries are located around two main kernels. In particular, the majority of geometries are placed inside a smaller buffer around one of the two kernels, while the remaining ones are inside of a bigger buffer (Fig. 3.3.e). In order to produce the various datasets the percentage of closer geometries and the dimension of the two buffers are changed, as well as their position. The parameters for this distribution consist of the locations and sizes of the two centers. Cluster distributions can represent urban areas that are centered around big cities.

- Combinations of two of the previous distributions: several combinations of the above distributions have been produced. Fig. 3.3.f shows an example of combination between a cluster and linear distribution. These combinations allow for producing more complicated datasets that cannot be represented with a single distribution.

In generating the synthetic datasets, also the length of the rectangle sides have been changed in order to obtain datasets with small and big rectangles. A separate group of datasets have been generated for representing the MBRs produced by linear networks or similar real data where oblong rectangles are very frequent. Some snapshots of diagonal datasets extracted from the generated data are shown in Fig. 3.4. This method is also applied to the other distributions to vary the shapes of the rectangles.

The experiments section provides the details of the parameters and sizes of the synthetic datasets that we use in our experimental evaluation.

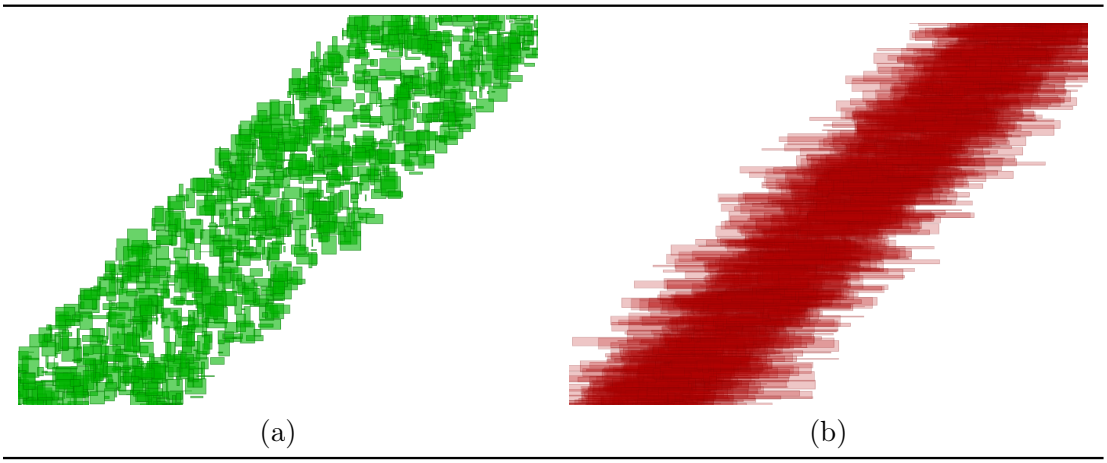


Figure 3.4: Example of rectangles contained in the training set. (a) regular rectangles of different sizes, (b) oblong rectangles of different sizes.

3.3.2 Dataset Summarization

This part describes how we summarize the big and variable-size datasets into a fixed-size vector that catches their characteristics and can be used as an input to the deep learning model. We consider two types of summarization techniques, fractal-based and histogram-based techniques. The fractal-based technique is inspired by sophisticated skewness measures developed by research experts in literature, e.g., box-counting [37] and Moran's-Index [142]. Since inspired by experts, these skewness measures are supposed to make an effective summary of the input dataset. On the other hand, the *histogram* technique is basically a uniform histogram which is much bigger in terms of representation size but might be able to catch more details about the dataset. The research question that we address in this chapter is: Can the machine use deep learning to come up with its own skewness

measures based on the histogram that outperforms the fractal-based techniques developed by experts?

Considering the case study shown in Table 3.1, it is clear that an easy and efficient way for evaluating the skewness of a spatial dataset is crucial for choosing the right partitioning technique. The parameters that we have chosen for describing the dataset distribution are presented below; they represent one of the main contributions of this work and are called *distribution descriptors* in the rest of the chapter.

Two distinct approaches have been considered: the first one, called *histogram-based*, computes a histogram by superimposing a fixed grid onto the dataset in order to describe extensively its distribution: each cell of the grid stores the number of geometries intersecting it; the second one, called *fractal-based*, computes some synthetic parameters deriving from the application of the fractal dimension concept and the Moran's index for capturing in a synthetic way the dataset distribution. Other statistics can be exploited to produce different descriptors, among them we can list the Ripley's K and L functions and other spectral analysis, but we choose the fractal-based ones since firstly they have already proved to be effective for the partitioning decision problem and secondly we need a representative technique for comparing the feature-extraction based approach with the usage of histograms, that is instead an approach based on row data, thus more deep learning oriented.

Table 3.2 summarises the symbols used in the formal presentation of the descriptors.

Table 3.2: Symbols

Symbol	**Meaning**
D	D represents a spatial dataset containing geometries.

G	G represents the grid used for computing an histogram on D.
n	n is # of cells on one side of the grid G. G has $n \times n$ cells.
l	l is the length of one side of the grid G.
r	r is the width of a cell belonging to G.
$hs_D^r(i)$	it is # of features of D intersecting the i-th cell of G with side length r (Def. 16).
q	An integer that represents the exponent of the Box-counting function.
$BC_D^q(r)$	it represents the computation of the box-counting function with exponent q on dataset D with a grid with cell of width r (Def. 17).
α	it is the constant of proportionality used in Eq. 3.3.
E_q	it is the exponent of the power law (see Eq.3.3), it represents the fractal dimension of the dataset.
$x_k(i)$	it represents the variable of interest in the computation of the Moran's index.
$\overline{x_k}$	it is the average of the variable of interest computed on all cells of the histogram used in the Moran's index computation.
N	$N = n \times n$ is the total number of cell of the histogram used in the Moran's index computation.
$w_{i,j}$	it represents the weight that is assigned to the pair of cells (i, j) in the computation of the Moran's index. Notice that each cell is identified by a single index i (or j).
EMP_D	it is # of empty cells in the histogram of a spatial dataset D.

Histogram-based Summarization Regarding the histogram-based approach, given a spatial dataset D containing geometries, we compute the histogram by choosing a regular grid G and computing for each cell of G the number of geometries of D that it intersects.

Definition 16 (Histogram)**.** *Given a dataset D, containing a set of geometric features, and a grid $G(n \times n)$ with cell size $r = l/n$ (l being the length of the grid side) and covering the reference space of D (i.e., the MBR of D), the histogram hs^r_D is defined as follows:*

$$hs^r_D(i) = count(\textit{features of } D \textit{ with an MBR intersecting the } i\textit{-th cell}) \tag{3.1}$$

In the *histogram-based* approach given a dataset D the ordered list of values representing the counts in the histogram cells is used for describing its distribution $(hs^r_D(1), \ldots, hs^r_D(n \times n))$. The histogram is computed efficiently using either Spark (used in this chapter) or Hadoop as shown in [49, 169]. The choice of the parameter r can have an impact on the effectiveness of the histograms in representing the dataset distribution. In Section 3.5.5 we illustrate the results of some specific experiments devoted to the analysis of this issue.

Fractal-based Summarization. Since the list of values in the histogram representation can be quite long, an alternative approach is to use the concept of fractal dimension to describe the dataset distribution by mean of a single number. This approach is usually applied to theoretically infinite set of points and has been extended to finite set of geometries, as proposed in [38, 40]. Using this idea, given a dataset D a family of histograms are computed and from each histogram a single number is obtained by summing up all the values contained in its cells. This sum is called *Box-counting* and the trend of this function, by varying the

size r of the grid cells, provides information about the dataset distribution, in particular this is straightforward when the dataset presents the self-similarity property (like, any fractal does), which occurs quite often on real datasets. More than one Box-counting function can be defined by considering different values for the exponent q, producing different fractal dimensions (E_0, E_2, ...) as theoretically defined in fractal theory.

Definition 17. *Given a dataset D, containing a set of features, the Box-counting plot is the plot of $BC_D^q(r)$ versus r in logarithmic scale, where:*

$$BC_D^q(r) = \sum_i (hs_D^r(i))^q \qquad \text{with } q \neq 1 \tag{3.2}$$

Now, we can consider such plot and exploit the following observation of [37]: for real datasets the box-counting plot reveals a trend of the box-counting function that, in a large interval of scale values r, behaves as a power law:

$$BC_D^q(r) = \alpha \cdot r^{E_q} \tag{3.3}$$

where α is a constant of proportionality and E_q is a fixed exponent that characterizes the power law.

The Box-counting plot is vital for the computation of the exponent E_q for a given dataset D, since this exponent becomes the slope of the straight line that approximates $BC_D^q(r)$ in a range of scales (r_1, r_2), thus it can be computed by a linear regression procedure. In our case we choose to consider the exponents E_0, E_2 and E_3. Fig. 3.5 shows the

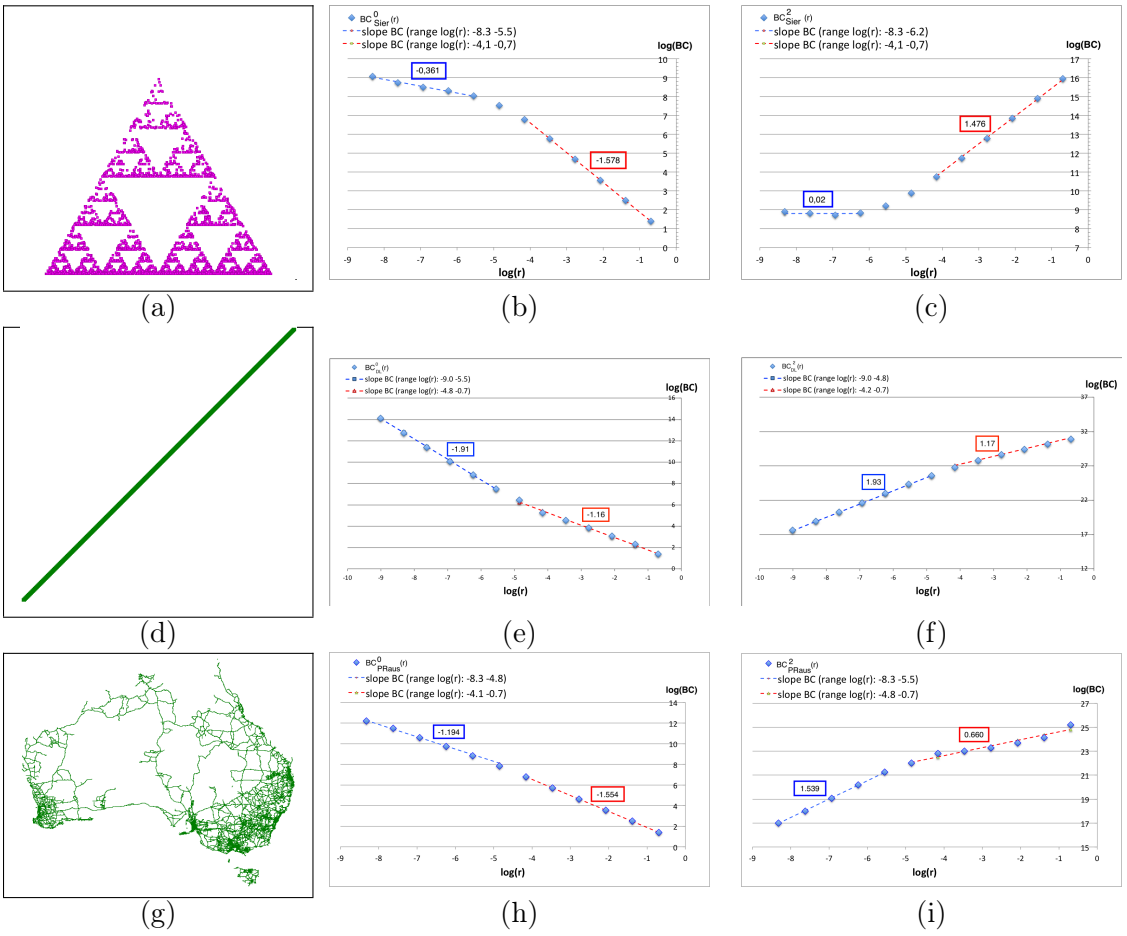


Figure 3.5: Example of Box-counting plot for: (a-c) a synthetic dataset with the distribution of a Sierpinski's triangle, (d-e) a synthetic dataset containing a diagonal line with buffer and (g-i) a real-world dataset representing the primary roads of Australia.

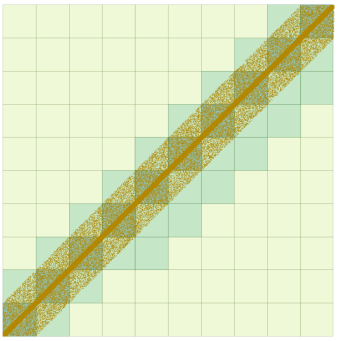
computation of E_0 and E_2 for some synthetic and real datasets. The first dataset contains small polygons with the distribution of the Sierpinski's triangle, which is a well-known fractal whose dimension is theoretically fixed to the value $log(3)/log(2) \approx 1.585$. The computed value of E_0 and E_2 in this case are very closed to the expected value 1.585. The first part of the plots, both for E_0 and E_2 has a different slope, this is due to the fact that the considered dataset is finite and thus when the cells of the grid becomes small enough, they will contain only one geometry each and as a consequence the value of $BC_D^q(r)$ tends to be

constant. Also the second dataset can be described as a fractal with dimension 1, since its distribution follows a straight line representing the diagonal of the reference space. Also here the computed slopes are very closed to the expected value. Finally, a real dataset has been considered, representing the primary roads of Australia. In this case we can notice that the dataset behaves indeed like a fractal, since we can measure slopes in the Box-counting plot. Notice that the values of E_0 and E_2 vary according to the considered intervals of values for r (representing the length of the cell side) and for higher values of r they are considerably less than two. This means that the dataset is not uniformly distributed in the reference space.

Again, a MapReduce implementation of this procedure allows the efficient computation of these descriptors as described in [38, 40].

Moran's Index Another well-known index that we have adopted for characterizing the dataset distribution is the Moran's index, which is a measure of spatial autocorrelation first presented in [142]. This index is able to detect the grade of autocorrelation regarding a variable of interest x that assumes different values in the cells of a grid, representing the domain of x. In our case the histograms computed for the previous descriptors E_* are used and the variable of interest x is represented by the count stored in each cell of the grid in the considered histogram, thus the reference space where the geometries are embedded, represents the domain of x. As shown in the following definition, the Moran's index analyses each cell of the histogram and evaluates how the value stored in the cell is correlated to the values stored in the adjacent cells.

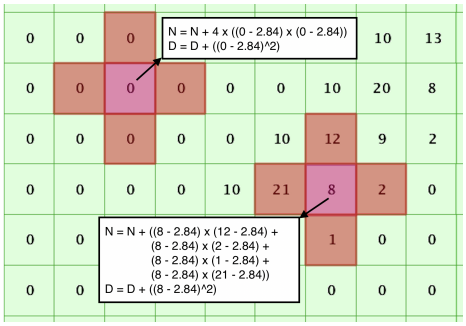
(a) Dataset *Diagonal Line*



(b) One of the computed histograms

0	0	0	0	0	0	0	0	9	18
0	0	0	0	0	0	0	10	13	9
0	0	0	0	0	0	10	20	8	2
0	0	0	0	0	10	12	9	2	0
0	0	0	0	10	21	8	2	0	0
0	0	0	3	13	8	1	0	0	0
0	0	4	13	7	1	0	0	0	0
0	4	13	5	1	0	0	0	0	0
4	13	4	0	0	0	0	0	0	0
13	4	0	0	0	0	0	0	0	0

(c) Chosen cells for showing the computation of MI_1 (numbers are reduced for sake of readability)



(d) Contributions of the chosen cells to the numerator N and denominator D of MI_1 (average = 2.84)

Figure 3.6: Example of Moran's index computation on the Diagonal Line dataset (a). In (b) the considered histogram is shown. In (c) the cells of the histogram are labelled with their count (# geometries they intersect) and two cells are highlighted together with their adjacent cells. Finally in (d) the contribution of the cells to the computation of the numerator (N) and the denominator (D) of the Moran's index is shown.

Definition 18 (Moran's index)**.** *Given a spatial dataset D together with its histogram $(hs^r_D(1), \ldots, hs^r_D(n \times n))$, the Moran's index have been computed considering the variable of interest $x_k = (hs^r_D(i))^k$, with the exponent $k \in \{0, 1\}$. The reasoning behind this choice can be explained as follows: with $k = 0$ the presence (1) or absence (0) of geometries inside a cell is considered, conversely with $k = 1$ the variation of concentration of geometries inside the cells are evaluated.*

$$MI_k = \frac{N}{W} \frac{\sum_i \sum_j w_{i,j}(x_k(i) - \overline{x}_k)(x_k(j) - \overline{x}_k)}{\sum_i (x_k(i) - \overline{x}_k)^2}$$

where:

- $w_{i,j}$ *is a matrix of spatial weights with zeroes on the diagonal, given a row i it contains ones only for the cells that are adjacent to the i-th cell and zeroes everywhere else.*
- $N = n \times n$ *(i.e., the histogram size)*
- $W = \sum_i \sum_j w_{i,j}$ *(i.e., the sum of all spatial weights)*
- $x_k(i) = (hs^r_D(i))^k$ *is the variable of interest in the considered case*
- $\overline{x}_k$ *is the average of the variable $x_k(i)$.*

In general, the typical values of the Moran's index belongs to the range -1,+1. Values near -1 indicates negative spatial autocorrelation (dispersion), while values near +1 means positive spatial autocorrelation (concentration), finally values around zero represent a random arrangement.

In Fig. 3.6 an example of computation of the Moran's index is shown. We consider the dataset representing a collection of small polygons distributed along the diagonal of the space with a portion of the data that are spread within a given distance (buffer) from the diagonal (an example of this dataset is shown also in Fig. 3.2, second column *Diagonal line*). Notice that, the cell on the left provides a positive contribution to the index calculation, since it detects similar values of the variable of interest in its neighbors, while the cell on the right, on the contrary, produces a negative contribution to the index, since very different values of the variable of interest are stored in its neighboring cells.

The MapReduce procedure for computing the descriptors E_0, E_2 and E_3 has been extended to compute also the values of the Moran's indexes: MI_0 and MI_1. In the experiments, in order to emphasis the spatial autocorrelation, we introduce also a discretization in five classes of the variable of interest x_1.

Empty Cells We also consider an additional descriptor that simply counts the percentage of empty cells (cells that are not intersected by any geometry) we call it EMP_D when computed on a spatial dataset D.

Table 3.3: Computation of the Moran's indexes M_0, M_1 and percentage of empty cells for the datasets presented in Fig.3.2: (a) a synthetic dataset with the distribution of a Sierpinski's triangle, (b) a synthetic dataset containing a diagonal line with buffer, (c) a synthetic dataset containing a double cluster and (d) a real-world dataset representing the primary roads of USA.

Dataset	M_0	M_1	EMP
Uniform distribution	0.719	0.011	12.6%
Diagonal line with buffer	0.917	0.739	81.6%
Double cluster	0.847	0.875	87.9%
Primary roads of USA	0.655	0.552	96.7%

In Tab. 3.3 the value of M_0, M_1 and EMP for some datasets are shown. Notice that M_0 is often close to 1, since it tends to be influenced by the spatial autocorrelation produced by the fact that empty cells are closed to other empty cells, or by the fact that not empty cells are closed to other not empty cells. Thus, a similar value is obtained both for the uniform distribution and the real dataset representing the primary roads of USA. In this situations M_1 can distinguish the two cases more effectively, but definitively the EMP value separates them clearly. The other two datasets are not well separated by these values, but they are if we consider the other descriptors E_0, E_2 and E_3.

3.3.3 Evaluation of Quality Metrics

In this section we briefly describe the quality metrics that characterize the partitioning techniques that we consider in this chapter and we show their effect on skewed distributed datasets. We also describe an efficient way to compute all quality metrics for all partitioning techniques in one Spark job.

Why do we need quality metrics? All big spatial data frameworks that run on multiple machines have to partition the data across machines before being processed. This applies to disk-based systems such as Hadoop, memory-based systems such as Spark, streaming systems such as Storm, key-value stores such as HBase, and big data managements systems such as AsterixDB [86]. Unfortunately, there is no agreement in the community of a single spatial partitioning technique that is universally recommended. The common partitioning techniques are based on grid, R-tree, Quad-tree, and space filling curve. Furthermore, there are many variations under each of these techniques. One of the reasons for having so many spatial

partitioning techniques is that the requirements of the systems vary by their architecture and the type of spatial analytics they perform.

In order to be able to quantify the *goodness* of the different partitioning techniques, several *quality metrics* have been developed. Each quality metric measures one aspect of the spatial partitioning techniques. Depending on the user requirements, one or more of these quality metrics might be chosen to minimize or maximize. The problem is that the quality of the resulting partitions depends on both the dataset distribution and the spatial partitioning technique.

Quality Metrics In order to measure the quality of the partitioning techniques when applied to a certain dataset D, we define four quality metrics that have been previously shown to improve the query performance of range query, kNN, and spatial join [62]. These quality metrics are total area (Q_1), total margin (Q_2), total area overlap (Q_3), standard deviation of partition cardinality (Q_4) and average range query cost (ARQ), all defined below.

Definition 19 (Total area - Q_1)**.** *Given a set of partitions $\mathcal{P} = \{P_i\}$, this quality measure is obtained by computing the sum of the areas of all partitions P_i:*

$$Q_1(\mathcal{P}) = \sum_{P_i \in \mathcal{P}} area(P_i.MBR) \cdot P_i.blocks$$

The multiplication by number of blocks $P_i.blocks$ allows the quality metric to take into account the processing mechanism of big spatial data frameworks. Simply, a partition

with multiple blocks is treated by those query processing engines as multiple partitions each with one block. This multiplication ensures that it is counted as multiple partitions.

Definition 20 (Total margin - Q_2)**.** *Given a set of partitions $\mathcal{P} = \{P_i\}$, this quality measure is obtained by computing the sum of the length of the semiperimeter of all partitions P_i:*

$$Q_2(\mathcal{P}) = \sum_{P_i \in \mathcal{P}} semiperimeter(P_i.MBR) \cdot P_i.blocks$$

where $semiperimeter(MBR)$=$MBR.width + MBR.height$.

Similar to Q_1, the multiplication by number of blocks ensures that a partition with multiple blocks is treated as multiple partitions with one block.

Definition 21 (Total overlaps - Q_3)**.** *Given a set of partitions $\mathcal{P} = \{P_i\}$, this quality measure is obtained by computing the sum of the area of the overlapping regions produced by intersecting each partition P_i with all other partitions P_j $(i \neq j)$:*

$$Q_3(\mathcal{P}) = \sum_{P_i, P_j \in \mathcal{P} \wedge i \neq j} area(P_i.MBR \cap P_j.MBR) \cdot P_i.blocks \cdot P_j.blocks + \sum_{P_i \in \mathcal{P}} area(P_i.MBR) \cdot \frac{P_i.blocks \cdot (P_i.blocks - 1)}{2}$$

The first term in the equation above calculates the total area of overlap between every pair of different partitions. The multiplication by $P_i.blocks \cdot P_j.blocks$ ensures that each partition is treated as separate partitions each with one block. The second term calculates the overlap that results when we have one partition with more than one block. In this case, if we treat each block as a separate partition, and all blocks will be completely overlapping with all

Table 3.4: Correlation between quality metrics and query performance

Partitioning technique	Total area	Total margin	Total overlap	Load balance
All techniques	0.9853496033	0.4225531469	0.9775651893	0.1635463961
R*-Grove	0.9639500086	0.989214543	0.9639500086	0.9094351766
STR	0.9456511908	0.970685173	0.8974233437	0.9588232937
Z-Curve	0.9927921874	0.9741114139	0.9949534266	0.9516883534

others. Notice that, if P_i has only one block its contribution to the second term is equal to zero.

Definition 22 (Standard deviation of partition cardinality (Q_4))**.** *Given a set of partitions $\mathcal{P} = \{P_i\}$, this quality measure is obtained by computing the average of the deviation from the average of the cardinality of each partition:*

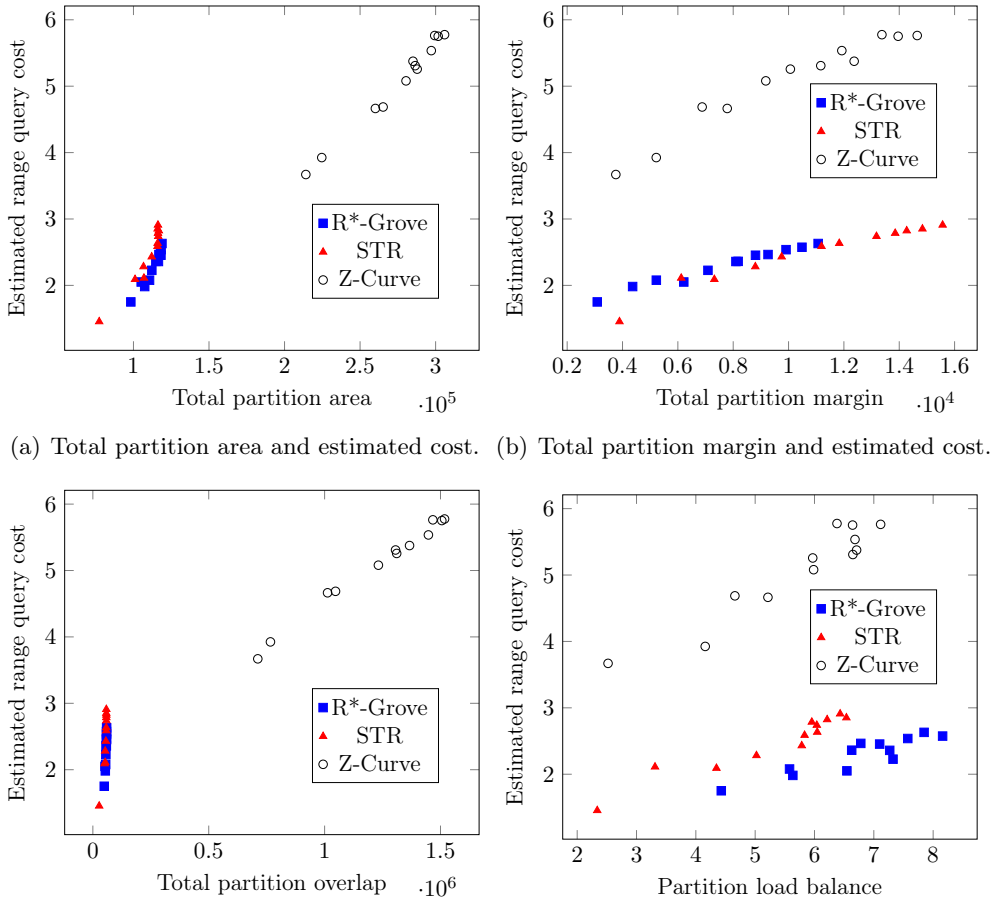
$$Q_4(\mathcal{P}) = \sqrt{\frac{\sum_{P_i \in \mathcal{P}} (P_i.card - \mathcal{P}.\overline{card})^2}{\mid \mathcal{P} \mid}}$$

where $\mathcal{P}.\overline{card}$ represents the average cardinality of the blocks of the partitions belonging to $\mathcal{P}$.

Definition 23 (Average range query cost (ARQ))**.** *Given a set of partitions $\mathcal{P} = \{P_i\}$, this quality measure is obtained by computing the sum of the average number of blocks which would be scan, if we execute a square query of size SxS at a random position on the dataset's space that is partitioned by $\mathcal{P}$.*

$$ARQ(\mathcal{P}) = \sum_{P_i \in \mathcal{P}} \frac{(P_i.width + S) \cdot (P_i.height + S)}{\mathcal{P}.MBR} \cdot P_i.blocks$$

Similar to previous ones, the multiplication by number of blocks is necessary to consider the actual query processing cost as each block is treated as a separate partition.



(a) Total partition area and estimated cost. (b) Total partition margin and estimated cost.

(c) Total partition overlap and estimated cost. (d) Partition load balance and estimated cost.

Figure 3.7: The relation of partition quality and query performance

A previous work [62] proved that the quality metrics Q1-Q4 are good indicators to evaluate the efficiency of a partitioning technique for a specific dataset. In other words, given two partitioning techniques, the one which achieves better quality metrics would provide a better query performance as well. In order to validate this statement, we carry an experiment which partitions the OSM-Nodes [93] datasets of different sizes by R*-Grove [187], STR and Z-Curve as shown in Figure 3.7. Figure 3.7(a) and 3.7(c) clearly shows that there is a linear relationship between total partition area and overlap with average range

query cost. In Figure 3.7(b) and 3.7(d), there are gaps of cost between different techniques, because the cost for a single technique is affected by the combination of all quality metrics. However, there is still upward trends for each partitioning technique. This observation was also mentioned in [62]. In more detail, Table 3.4 verifies this observation by showing the correlation values between quality metrics and query performance in different partitioning techniques. These high values validate our claim that we could use partition quality metrics to evaluate performance of a partitioning scheme.

The following part describes how these quality metrics are computed efficiently using Spark.

Quality Metrics Computation This part describes how we compute a set of quality metrics for a given dataset while considering many partitioning techniques. In our discussion, we borrow some terminology from SpatialHadoop [72] but the approach can generalize to other systems including Spark-based systems. This step is critical as it needs to be done for each training dataset that we consider. A naïve approach is to simply partition the dataset using all possible partitioners and then evaluate their quality using all quality metrics. However, this would be too slow and would limit the performance of the training phase. Rather, we consider a more efficient technique that can accurately calculate the quality metrics without having to actually partition the data. Below, we first describe the notion of a *master file* and then explain how we use it to compute the quality metrics efficiently.

Master files: SpatialHadoop manages the metadata of a partitioned dataset by a small text file, called master file. The master file of a partitioned dataset contains a list of metadata

for all partitions of that dataset. The metadata of a partition includes partition ID, total number of records, partition size, and partition MBR. In order to execute a query, for example range query, the query executor would take a look at the master file for early pruning the partitions which certainly do not contribute to the answer. According to previous work, the data partitioning, as encoded in the master file, is the main driving factor for query performance [72].

The key observation is that we can produce many master files for all partitioning techniques in one Spark/MapReduce job without having to actually partition the data. In particular, the quality metrics that we consider in this chapter are total area ($Q1$), total margin ($Q2$), total overlaps ($Q3$), or average range query cost (ARQ) of all partitions, as we mentioned in Section 3.3.3. All of those metrics could be computed from the master file of the partitioned dataset as they only require the MBR and total size of each partition. Furthermore, other researchers can easily extend these quality metrics based on the demands of the desired analytic operation, e.g., standard deviation of partition size, and disk utilization.

Efficient Computation of Master Files: In order to create a training data point, we have to find the best partitioning technique among several options (e.g., kd-tree, R*-Tree, STR, Grid, and Z-Curve) in terms of a specific quality metric, for example total area of all partitions. Instead of physically partitioning the data using these techniques, we observe that all information encoded in the master files, i.e., MBR and total size, are associative and commutative aggregate functions that can be computed in a local/global manner without the need to group all records of one partition in one machine. In other words, instead of partitioning the data into partitions and then computing those aggregate functions, we can

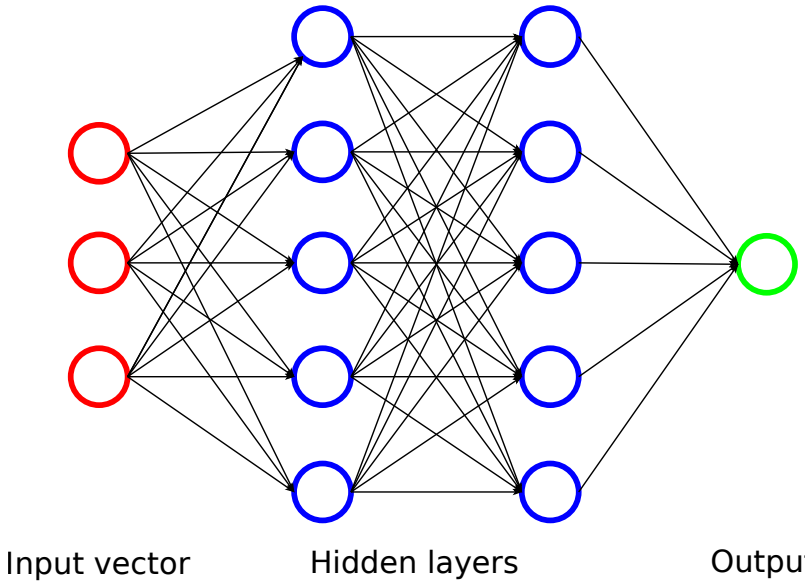


Figure 3.8: A sample fully connected model that we adopt in the chapter. In this work, we vary the number of hidden (blue) layers and the number of hidden units per layer

directly compute these aggregate functions. Simply, each machine computes local values for *all partitions* and then they are grouped by partition ID to be further aggregated into final values. Furthermore, we can compute these aggregate values for *all* partitioning techniques in one job by extending the grouping key to be ⟨partition*er* ID, partition ID⟩.

Once all the master files are computed, we can then compute the quality metrics (i.e., $Q1$, $Q2$, $Q3$, and ARQ) as described above on a local machine since the size of the master files is sufficiently small.

3.3.4 Model Training

In this section, we describe how we conduct a deep learning model which is able to predict the best partitioning technique for a spatial dataset in terms of a specific quality metric. The first challenge that we have to address is to choose a suitable deep learning

algorithm for this problem. As we mentioned, the partitioning selector problem is analogous to image classification problem. Thus, we could consider several novel classification models such as Convolutional Neural Network(CNN) or a fully connected neural network (FC). If the number of data points is large enough, e.g. millions of data points, CNN would mostly outperform a fully connected model. However, this might not be applicable for our system, where the number of data points is only in thousands. Based on a previous work[159], we carried an experiment for algorithm selection at Section 3.5.2. Finally, we chose a fully connected neural network to train and test our model. Fig. 3.8 shows the architecture of a fully connected neural network that we use for spatial partitioning selection model. The input vector (X) consists of a summary of the input dataset, either the histogram or the fractal-based descriptors. In particular, the vector X is composed of:

Histogram-based vector : in this case the vector contains exactly all the counts collected in the cells of the histogram computed on dataset D:

$$X = \langle hs^r_D(1), \ldots hs^r_D(n) \rangle$$

This means that the size of the input layer is always equal to the size of the histogram, i.e., number of bins in the histogram.

Fractal-based vector: : in this case the vector contains the descriptors computed on dataset D:

$$X = \langle \mid D \mid, E_0(D), E'_0(D), E_2(D), E'_2(D), E_3(D), E'_3(D),$$

$$M_{0_{min}}(D), M_{0_{avg}}(D), M_{0_{max}}(D), M_{2_{min}}(D), M_{2_{avg}}(D), M_{2_{max}}(D),$$

$$M_{3_{min}}(D), M_{3_{avg}}(D), M_{3_{max}}(D), EMP_D\rangle$$

where: (i) $E_0(D)$ is the exponent E_0 computed for a dataset D, we have two value E_q and E'_q for each exponent, since in many cases the behaviour of the dataset follows trends similar to those shown in Fig. 3.5; (ii) $M_q(D)$ is the Moran's index computed for the variable of interest $(hs^r_D(i))^q$, here we use three values, since we consider a family of histograms and thus we produce several values for $M_q(D)$, thus we take the minimum, maximum and average value for representing the behaviour of the dataset D.

The hidden layers are fully connected and we vary their sizes in the experiments section to tune the system. The size and number of hidden layers can be tuned differently according to which summarization technique we use. The output vector is a single categorical value, which is the best partitioning technique between Kd-tree, R*-Grove, STR, Z-Curve, Grid and RR*-Tree. The output value is encoded as a number in range $[0-5]$, which is the order of the corresponding partitioning technique. We choose to build a separate model for each quality metric as each one of them might need to catch different aspects of the input vector.

The activation function for hidden units is $ReLU$ function, except for the last layer, where we use $softmax$ function. Since the output value is categorical, we use $categorical\_crossentropy$ as the loss function.

As recommended in deep learning, we separate the input dataset into three parts, *training*, *validation*, and *testing*. The training set is used to train the model and adjust the

weights on all the connections in the neural network. The validation set is used during the training phase to evaluate the current model and avoid over fitting. The validation set is *never* fed through the input layer so it is always a *new* dataset to the model. Finally, the test dataset is used for final evaluation as shown in the experiments section.

To give the network enough time to stabilize without over fitting, we periodically measure the accuracy of both the training and validation sets. When the accuracy of the validation set stops improving or starts to drop, it is a signal of overfitting. Therefore, we terminate the training phase and retrieve the last good model right before the accuracy dropped.

3.4 Application Phase

3.4.1 Overview

In this phase, the system takes a dataset D that was not inspected earlier by the framework and a quality metric (QM). The goal is to predict which partition technique (PT) among the ones considered by the framework will produce the best behaviour in the chosen quality metric QM. The main challenge of this step is that is has to be much faster than applying all partitioning techniques and choosing the best. This phase works in two steps.

The first step summarizes the data to produce a fixed-size vector (X') that describes the input data distribution as described in Section 3.3.2.

The second step feeds the vector (X') computed in the first step into the machine learning model (M) that corresponds to the quality metric QM. The output of the model is a label (Y') that simply names one of the partitioning techniques (PT) that is estimated by

the model to produce the best quality metric (QM). The selected partitioning technique is then passed to any big spatial data system, e.g., SpatialHadoop or GeoSpark, to actually partition the data. In other words, our framework does not actually partition the data, it just chooses a partitioning technique to apply and it is up to the user to choose how to apply it.

More specifically in the following subsection the application of the system to the spatial join operation is illustrated by considering a test case with real datasets.

3.4.2 Application of the model in a real system

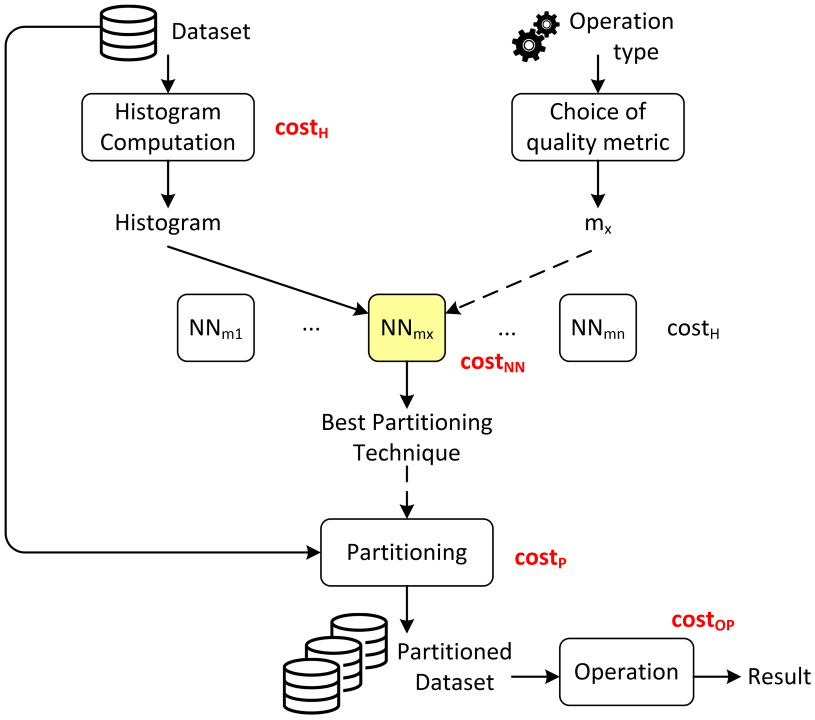


Figure 3.9: Flow chart of the optimization task.

In order to show how the proposed model can be applied in a real system we show in Figure 3.9 the flow chart describing the necessary steps to perform a given operations OP,

for example a spatial join, on a pair of datasets, D_1 and D_2, with unknown distributions. In the figure the optimization task is composed of the following steps:

1. **Histogram computation**: For each dataset D_i the corresponding histogram H_i is computed, representing the input vector X'; the cost of this operation is denoted as $COST_H(D_i)$.

2. **Quality metric choice**: Given the operation to execute OP the corresponding quality metric QM_x is chosen; the cost of this operation is trivially close to zero.

3. **Partitioning technique choice**: Given QM_x, the corresponding model NN_x is activated passing as input the vector X', obtaining the suggested partitioning technique PT_i, one for each input dataset. The cost of this operation is again close to zero, thanks to the trained machine learning model NN_x.

4. **Partitioning**: Each chosen technique PT_i is applied to the corresponding dataset D_i, producing a partitioned dataset PD_i.

5. **Operation computation**: the operation OP is executed on the partitioned datasets PD_i; the cost of this execution is denoted as $COST_{OP}(PD_1, PD_2)$.

The application of the proposed approach is convenient since the following conditions are very often satisfied in particular for the spatial join operation ($\bowtie$); in this case the quality metric is total margin (QM_{TM}):

- **Basic condition**: the cost for generating the histogram (i.e., the input vector X') for a given dataset must be significantly less than the cost of partitioning the same

dataset:

$$COST_H(D_i) << COST_P(D_i)$$

This operation is preformed by a parallel task implemented in Spark and as shown in Figure 3.18 of the next section, this cost is an order of magnitude less than the cost for partitioning a dataset.

- **Specific condition for $\bowtie$**: the average cost of the execution of OP on the partitioned datasets must be less than the cost of executing it on the original datasets:

$$COST_{\bowtie}(PD_1, PD_2) < COST_{\bowtie}(D_1, D_2)$$

- **Optimization condition for $\bowtie$**: the average cost of the optimization phase must be less than the gain produced by the optimization:

$$COST_H(D_1)+COST_H(D_2)+COST_P(D_1)+COST_P(D_2) < COST_{\bowtie}(D_1, D_2)-COST_{\bowtie}(PD_1, PD_2)$$

In order to test the application phase and verify the satisfaction of the second and the third condition, we performed some experiments in a specific case using real datasets. In particular, we consider two real datasets D_{PRoads} and D_{Builds} containing the primary roads and buildings of the USA, respectively. Each dataset has been partitioned by applying the six considered techniques. Then, the spatial join between all possible combinations of partitioned datasets has been performed (in total 36 joins). In Table 3.5 the execution time in seconds of each combination is shown.

Finally, considering the pair (RR*-Tree, RR*-Tree) chosen by the proposed machine learning model NN_{TM} (i.e., the neural network for the chosen quality metric: total margin) we can observe that: (i) NN_{TM} is able to detect the pair that is in the top positions in the ranking of spatial join time execution; (ii) the gain with respect to the join performed on the original datasets is about 99.4% (iii) the gain with respect to the average performance of all pairs is about 31.9% and finally the gain with respect to the worst pair is about 61.4%. The gain obtained by applying the suggested partitioning techniques is:

$$COST_{\bowtie}(D_1, D_2) - COST_{\bowtie}(PD_1, PD_2) = 28,859\, sec$$

and the cost of optimization is:

$$COST_H(D_1) + COST_H(D_2) + COST_P(D_1) + COST_P(D_2) = 1,238\, sec$$

This results allow us to confirm that the above mentioned conditions are all satisfied. A wider analysis of applicability considering other operations is out of the scope of this chapter, also because previous works [62] about spatial partitioning techniques already confirm the effectiveness of their use.

Table 3.5: Execution times of the spatial join operations in seconds. All possible combinations of partitioning techniques applied to datasets D_{PRoads} and D_{Builds} are considered.

Combinations of part. tech.	D_{Builds}					
D_{PRoads}	**Grid**	**Kd-Tree**	**RR*-Tree**	**R*-Grove**	**STR**	**Z-Curve**
Grid	413.2	303.4	245.0	259.6	293.1	279.2
Kd-Tree	318.2	172.3	142.8	148.3	134.9	217.0
RR*-Tree	325.2	208.3	159.4	160.0	148.6	232.9
R*-Grove	357.8	212.3	170.7	171.9	166.9	217.0
STR	343.8	200.2	312.2	16.95	145.2	292.7
Z-Curve	361.0	251.5	220.2	217.2	229.2	235.3

A tutorial showing the steps for applying the proposed system is available on GitHub [1].

3.5 Experiments

This section provides the details of our extensive experimental evaluation. The goal of this experimental evaluation is to measure how accurate the proposed approach is in choosing the best partitioning technique. The experiments will also compare the two summarization techniques to verify which one is more effective for this problem. In the rest of this section, Section 5.4.1 provides the experimental setup. Section 3.5.3 describes how we tune the deep learning model. Then, we evaluate the accuracy of the proposed model for both synthetic and real data in Section 3.5.4. Section 3.5.5 illustrates the effect of histogram size to the model accuracy, model complexity and training cost. After that, Section 3.5.6 shows the effect of the dataset size on quality metrics and justify the medium sizes of the synthetic datasets that were used for training. Section 3.5.7 will focus on evaluating the performance of the system in terms of running time considering both the creation of the

[1] https://github.com/tinvukhac/deep-spatial-partitioning

Table 3.6: Experiments and performance metrics

Experiment	Parameters	Metrics
Model tuning	# of hidden layers, units	Accuracy
Model accuracy	dataset distribution	Accuracy
Histogram effect	histogram size	Accuracy, model complexity
Stability of quality metrics	dataset size, HDFS block size	Quality metrics
Summarization performance	dataset size	Running time

training set on one side and the computation of the Histograms-based and Fractal-based summarizations on the other side. Notice that, in the fractal-based summarization we also include the Moran's index. Section 3.5.8 considers the effect of including in the training set also collection of data with oblong rectangles.

3.5.1 Experimental Setup

Table 3.6 shows the list of experiments that we are carrying out. (1) First, to tune the parameters of the deep learning models, we vary the number of hidden layers and the number of units per layer to find a suitable fully connected architecture for each summarization technique and for each quality metric. (2) Second, we conduct several experiments to see how the model learns to predict the best partitioning technique from training data for both synthetic and real data. (3) Third, we vary histogram size to see how it affects the model accuracy and complexity. In particular, for each histogram size, we feed the data to several models with different number of hidden layers/units to see which configuration is suitable for a specific histogram size. This experiment explains how we choose histogram size and model architecture for the second experiment. (4) Fourth, to justify the parameters that we use for synthetic data generation, we show the stability of the quality metrics as the dataset size varies. This allows us to generate many medium-size synthetic datasets to save time

instead of generating a few large datasets. (5) Fifth, we measure the running time of the summarization process to show that the proposed solution can be applied in practice since the required effort is significantly less than the cost for partitioning the dataset with all six available techniques. (6) Sixth, we measure the effects on model accuracy of the addition to the training set of new synthetic datasets containing oblong rectangles.

We run our experiments on a cluster of one head node and 12 worker nodes, each having 12 cores, 64 GB of RAM, and a 10 TB HDD. They run CentOS 7 and Oracle Java 1.8.0_131. The cluster is equipped with Apache Spark 2.3.0 and Apache Hadoop 2.9.0. We implement our deep learning model on Keras [96] with TensorFlow 1.12.0 as the backend.

Datasets: In Tab. 3.7 the characteristics of the generated synthetic datasets are presented [192]. Notice that for training the model, the generated datasets do not have to be very big. They just have to be diverse enough to represent various characteristics of real data. In Section 3.5.6 below, we justify this decision by showing the independence of the relative quality of partitioning techniques with dataset size. For each distribution, we generate 100 different datasets with different seeds. The collection contains 1,600 datasets with about 210 millions of geometries in total.

Tab. 3.8 shows the real datasets that we use for testing the model. All datasets are publicly available through the SpatialHadoop website [72]. We picked three datasets, buildings, lakes, and roads. To have a decent number of datasets with different distributions, we split each dataset into five parts that roughly enclose North America, South America, Europe, Africa, and Asia+Australia. The size of each part is shown in the table.

Table 3.7: Training set generation: datasets of different distributions generated for the training phase.

Distribution	**Num. of Datasets**	**Size**	**Num. of Features**
Uniform	100	512 Mb	7,000,000
Linear	100	512 Mb	7,000,000
Linear rotated	100	420 Mb	6,000,000
Diagonal	100	1.1 Gb	15,000,000
Diagonal rotated	100	1.1 Gb	15,000,000
Parcel	100	512 Mb	7,000,000
Cluster	100	1.0 Gb	5,000,000
Linear/Linear rot.	100	1.0 Gb	11,000,000
Linear/Uniform	100	1.0 Gb	14,000,000
Linear rot./Uniform	100	1.0 Gb	11,000,000
Diagonal/Diagonal rot.	100	2.2 Gb	30,000,000
Diagonal/Uniform	100	1.6 Gb	22,000,000
Diagonal rot./Uniform	100	1.6 Gb	22,000,000
Parcel/Uniform	100	1.0 Gb	14,000,000
Parcel/Linear rot.	100	1.0 Gb	13,000,000
Cluster/Linear rot.	100	1.5 Gb	11,000,000

Notice that the aim of this experiment session is to verify the quality of the model, i.e. to test the performance of the neural network that predicts the right technique to choose in order to obtain the best partition with respect to a given quality measure. We are not testing the impact of the choice on the final operation that is applied by the user on the partitioned datasets.

Training sets: We produced the data points of the training sets from the generated synthetic datasets, listed in Table 3.7. For deep learning, a data point is a pair (X, Y), where: X represents the summarization of one dataset (using one of the two proposed summarization techniques), and Y represents the corresponding best partition. Since we have in total five quality metrics and two summarization techniques, Histograms-based and Fractal-based, for

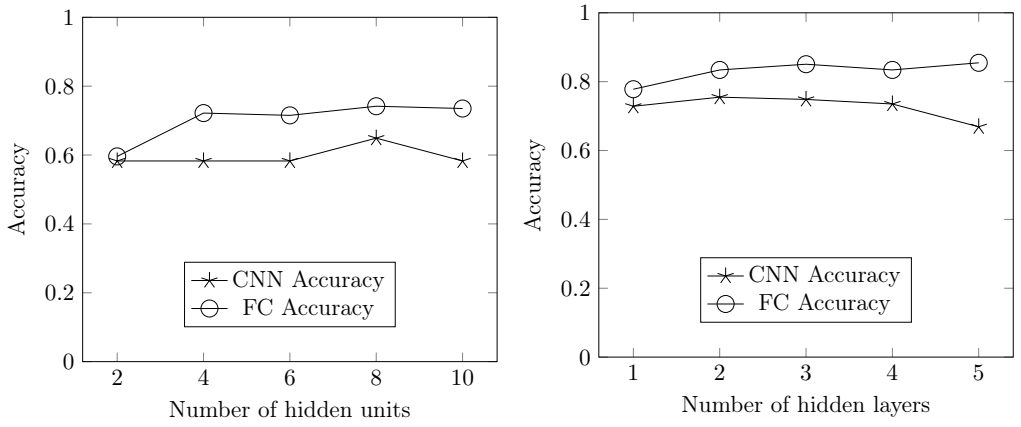
Table 3.8: Real datasets used for testing.

Dataset	**Num. of Features**
Buildings-1	1,393,451
Buildings-2	8,708,373
Buildings-3	91,657,814
Buildings-4	7,925,531
Buildings-5	5,111,326
Lakes-1	828,221
Lakes-2	4,246,874
Lakes-3	2,072,660
Lakes-4	619,689
Lakes-5	652,583
Roads-1	3,672,499
Roads-2	19,729,459
Roads-3	33,078,006
Roads-4	6,725,578
Roads-5	9,137,471

each dataset we produce 10 data points, one for each training set dedicated to one model: $(X_H, Y_i)_{Q_i}$, $1 \leq i \leq 5$ for the Histograms-based summarization, and $(X_F, Y_i)_{Q_i}$, $1 \leq i \leq 5$ for the Fractal-based one.

In total, in each training set we have 1,600 data points generated from the synthetic datasets. Unless otherwise mentioned, we use 80% of generated data points as training data and the other 20% as the testing data. Out of the 80% training set, 20% of it is used as a validation set, i.e., 16% of the overall data.

Accuracy Metrics: We use a Boolean accuracy metric. That is, we compare the label generated by the model with the true label that was selected by computing the actual quality metrics and choosing the best. If they match, the accuracy is 1.0, otherwise it is 0.0. Then, we take the average over all the test set. Notice that since we have six possible labels that



(a) Accuracy and # of hidden units on CNN and FC model

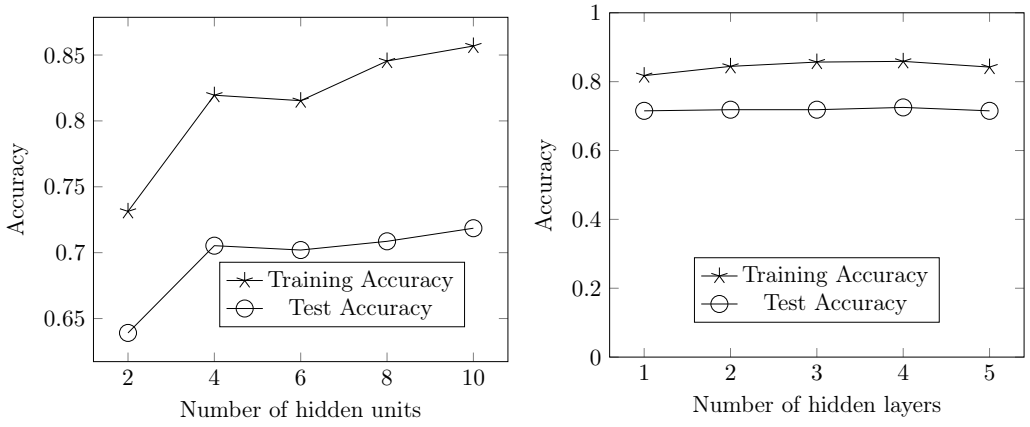
(b) Accuracy and # of hidden layers on CNN and FC model

Figure 3.10: Algorithm comparison between CNN and FC model

correspond to the six partitioning techniques, a completely random baseline would have an accuracy of $1/6 \approx 17\%$.

3.5.2 Algorithm Selection

In this experiment, we compare two approaches in the context of spatial partitioning selection problem: convolutional neural network (CNN) model and a fully connected (FC) model. The choice of these two candidate models is inspired by the observation that our problem is analogous to an image classification problem. We are trying to set up the same parameters, e.g. number of hidden layers and number of hidden units in each layer for both model. After that, we train these models and evaluate their accuracy. Figure 3.10 shows that FC models outperforms CNN models in terms of model's accuracy. Furthermore, CNN also requires more training time before its convergence point for a same given dataset. The reason is that CNN typically requires a training and testing set with very large number of data



(a) Accuracy and # of hidden units with histogram input

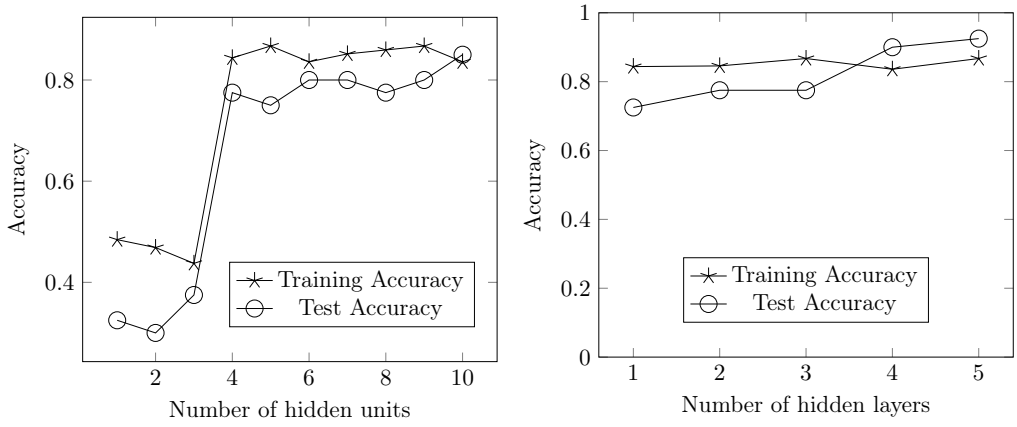
(b) Accuracy and # of hidden layers with histogram input

Figure 3.11: Tuning model parameters for the **histogram-based** summarization technique

points. However, the training set we created from histograms is limited in size, which might be more suitable to a simple architecture like FC models. Finally, we chose to use FC model for our following experiments. In our published repository, we provide the implementation fo both CNN and FC model. Therefore, users could choose any model which is suitable for their own datasets.

3.5.3 Model Selection

This experiment shows our effort to find the suitable model architecture for our training datasets. As we mentioned in Section 3.3.4, there are two options to generate training dataset with data points (X, Y). First, X could be the flatten vector of the histogram matrix, which is chosen as $50x50$ in this experiment (see Section 3.5.5 for more details about this choice). Second, X can be considered as the ordered skewness values which are computed by fractal-based summarization methods (fractal dimensions and Moran's indexes). Y is the



(a) Accuracy and # of hidden units with skewness input

(b) Accuracy and # of hidden layers with skewness input

Figure 3.12: Tuning model parameters for the **fractal-based** summarization technique

single number that reflects the order (base 0) of the best partitioning options among Kd-tree, R*-Grove, STR, Z-Curve, Grid, and RR*-tree.

In this experiment, we use total partition area as reference quality metric to evaluate partitioning techniques. The best technique should have the smallest total area. The different kinds of feature vector might require different configurations of the learning model. Moreover, we use a fully connected neural network and vary both the number of hidden layers and the number of units per layer to find the suitable model for each kind of input vector.

Figure 3.11(a) shows the accuracy of a fully connected model with 3 hidden layers for the training dataset with histogram vector as the input vector. We vary the total number of units in each layer to see how the accuracy changes. When the number of hidden units is small, e.g. 2, the model is not able to capture the complex information from training data. As the number of hidden units increases, the accuracy for both training and testing process are stabilized. Thus, we choose 10 as the number of hidden units for each layer.

In the next experiment shown in Figure 3.11(b), we fix the number of units per layer as 10 then vary the number of hidden layers to see how it affects the model accuracy. We observe that the accuracy is stable when the number of layers changes with the best value at 3 hidden layers. Based on these two experiments, for the model with an input vector composed of the flatten representation of the histogram matrix with size $50x50$, we choose the fully connected model with 3 hidden layers and 10 hidden units per layer.

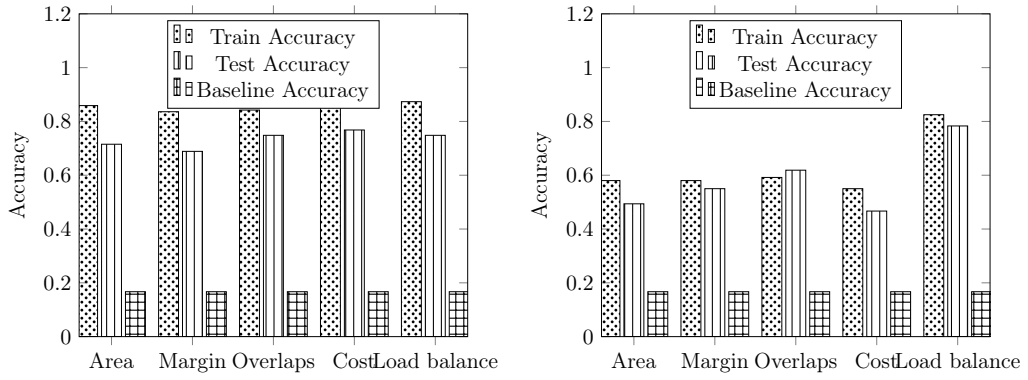
We repeat the same procedure with the other summarization technique, i.e., the fractal-based one, as shown in Figures 3.12(a) and 3.12(b). In this case, we choose an architecture with 3 hidden layer and 5 hidden units in each layers.

In these experiments, we can also observe that the model can reach up-to 90% and 80% accuracy when applied on synthetic and real test data, respectively, which shows the applicability of the proposed approach to the problem of spatial partitioning.

3.5.4 Model Accuracy

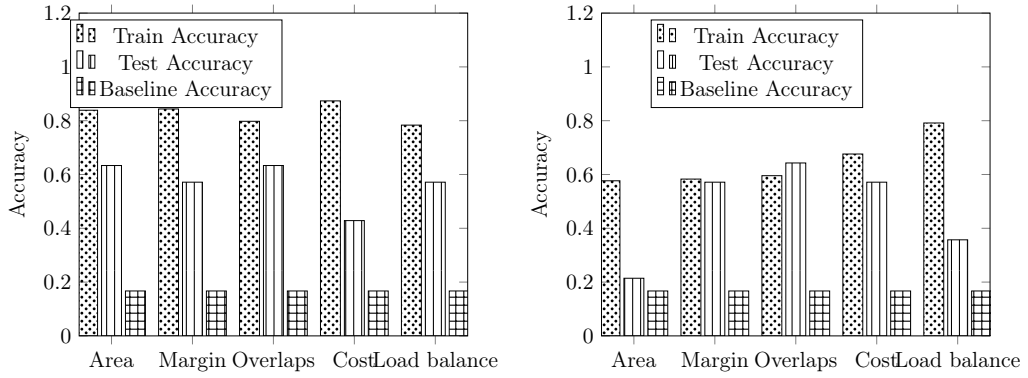
This section shows the accuracy of our model to predict the best partitioning technique for datasets with different distributions including synthetic and real datasets. We only use the synthetic datasets for training and we use both synthetic and real data for testing, reporting their accuracy separately.

In such experiments, we measure the accuracy of our predictive models considering two configurations: (i) in the first one the input vector is the ordered list of skewness values, which are computed by fractal-based summarization methods, while (ii) in the second one the flatten vector representing the histogram of the dataset is the input.



(a) Histogram summarization with test on **synthetic** data

(b) Skewness summarization with test on **synthetic** data



(c) Histogram summarization with test on **real** data

(d) Skewness summarization with test on **real** data

Figure 3.13: Model accuracy when train and test on synthetic and real datasets

The quality metrics include: total area (Q_1), total margin (Q_2), total overlaps (Q_3), the standard deviation of the partition size (Q_4) and the average range query cost (ARQ) of partitioned datasets. We evaluate such metrics in six different partitioning techniques: Kd-tree, R*-Grove, STR, Z-Curve, Grid, RR*-tree. Generally, if we randomly choose a technique between those options, the probability that we can choose the best one is 17%. Since there are no similar work that exists in literature, we choose this number as the baseline

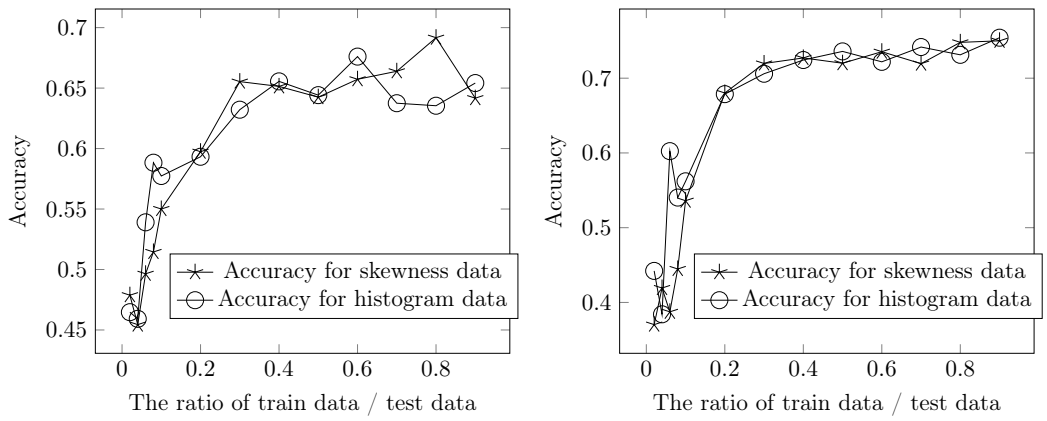
accuracy to compare with our proposed method which is a common practice in machine learning evaluation.

Figure 3.13(a) and 3.13(b) show the accuracy of training and testing process when we train and test our model with data points coming from synthetic datasets. As we can observe, in both configurations, the models can predict the best partitioning technique for different quality metrics with an accuracy of up to 78%, which is significantly better than the baseline method.

Figure 3.13(c) and 3.13(d) show the accuracy of training and testing process when we train our model on data points coming from synthetic datasets, and *test it on data points from real datasets*. Although the test accuracy is not as high as the synthetic datasets, it still gives us a good accuracy with up-to 64%. Keep in mind that the model was trained on synthetic data only and the real datasets used in testing are observed by the model for the first time.

Comparison of the two summarization techniques: One interesting observation in this experiment is that the histogram-based summarization outperforms the fractal-based skewness measures developed by the experts for *synthetic data*. However, when it comes to real data, the results for both summarizations are very similar (only in some cases the experts' measures outperform the simple histogram).

This indicates that the deep learning model can learn and produce an accurate model for the datasets it sees during the training phase and can outperform existing methods. However, the skewness measures, as developed by experts, are good at extracting meaningful measures of skewness and allow to find hints of similarity between two datasets that, from a



(a) Predict the technique with best total area (b) Predict the technique with best total margin

Figure 3.14: Model accuracy when varying the ratio of train data / test data

simple comparison of their plots, might seem very different. In the histogram configuration the models learn to detect similarity between datasets mainly considering a visualization-based comparison working at the granularity of the histogram. This implies that the model that learns from histograms need a training set containing a higher variety of distributions and datasets with a higher similarity to the real ones in order to increase its accuracy.

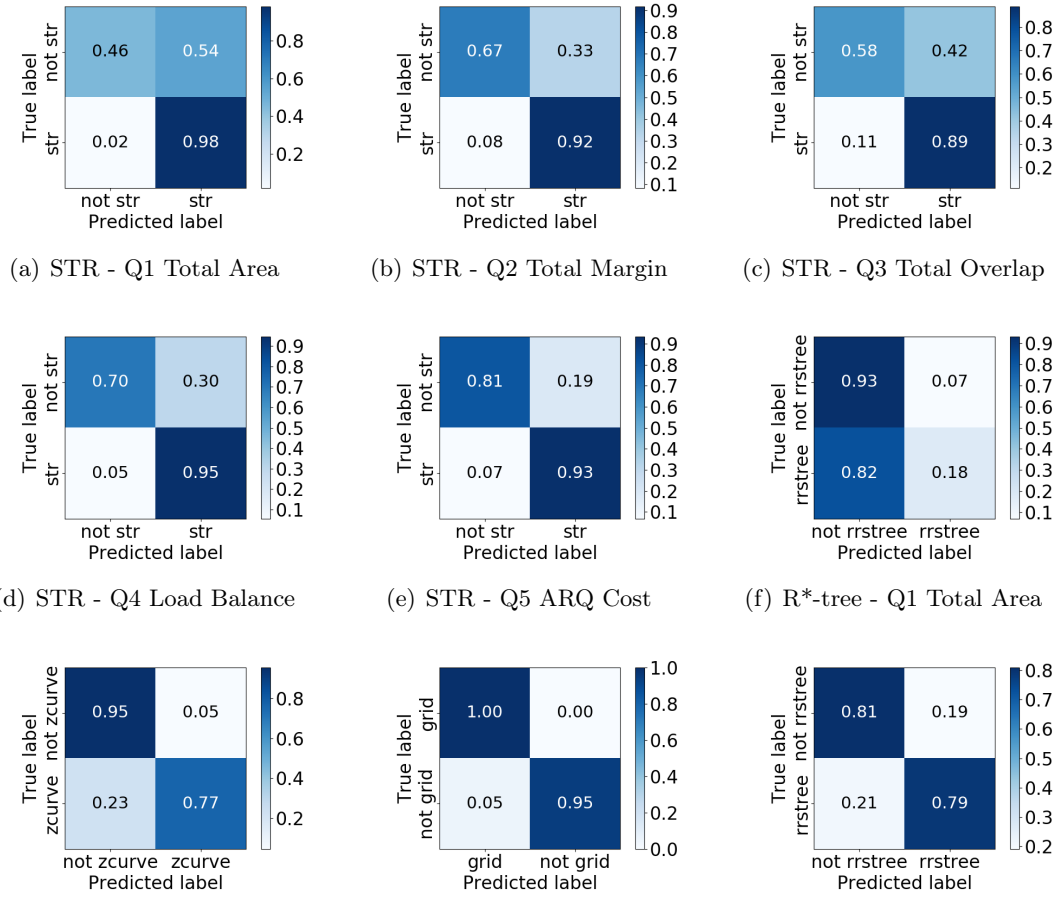
We expect that if we have a bigger training set with more diverse synthetic datasets, the deep learning approach with histogram can produce better results. We plan to verify this conjecture in future works by adding more distributions and more datasets to the training set.

Figure 3.14 shows the accuracy of the predictive models with skewness and histogram input when we vary the ratio between number of train and test data points. As expected, the accuracy increases and then stabilizes as the ratio of the training set increases. This verifies that the predictive models are able to capture the characteristics of the input datasets and that they get more accurate with more training points.

Another accuracy metric that is usually used in multi-labeled deep learning models is the *confusion matrix*. As shown in Figure 3.15, this matrix shows for each pair $(label, metric)$, a square divided in four parts containing the percentage of: (i) true positive cases obtained in the test (lower right sub-square), (ii) true negative cases (upper left sub-square), (iii) false positive (lower left sub-square) and (iv) false negative (upper right sub-square).

Figure 3.15.(a-e) shows the five confusion matrices regarding the *STR* partitioning technique, one for each each quality metric. Notice that for this technique, which is the one having more samples in the training set, the true positive percentage is always over 89%. However, for the other techniques we do not reach the same optimal results. For instance, considering the *RR*-tree* no test cases are available for such technique with the load balance metric, and in the other metrics the results are varying: good for the range query cost metric (Fig. 3.15.j), but not as good for the total area metric (Fig. 3.15.f). The same is true for other techniques, for instance the *Grid* technique show very good results with the load balance metric (Fig. 3.15.i), but for the others no test cases are available.

The main reason for the above described results is that many techniques (but *STR*) are underrepresented in the training data which is a known problem that causes machine learning models to be incapable of learning their characteristics. In our problem, it was not easy to balance the training data as we cannot directly specify which partitioning technique is the best for a specific dataset, rather, we generate different synthetic data and run them through all the partitioning techniques and the best is selected based on their behavior.



(a) STR - Q1 Total Area (b) STR - Q2 Total Margin (c) STR - Q3 Total Overlap

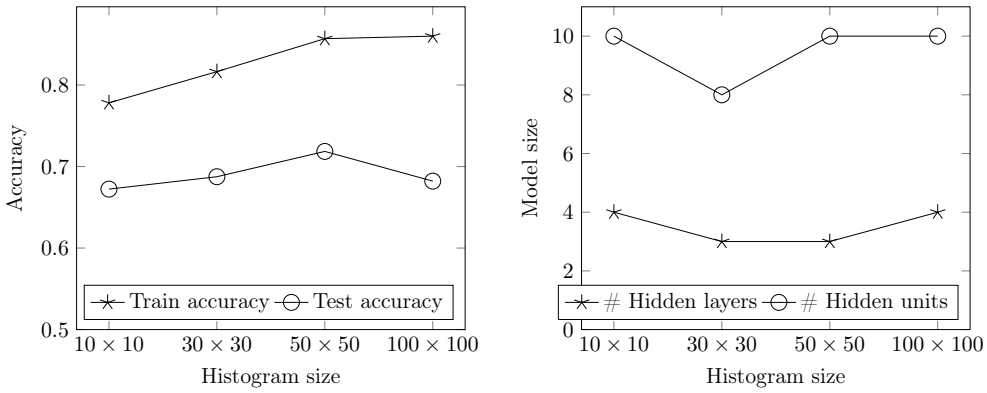
(d) STR - Q4 Load Balance (e) STR - Q5 ARQ Cost (f) R*-tree - Q1 Total Area

(g) ZCurve - Q2 Total Margin (h) Grid - Q4 Load Balance (i) R*-tree - Q5 ARQ Cost

Figure 3.15: Confusion matrices for different index techniques and considering different quality measures.

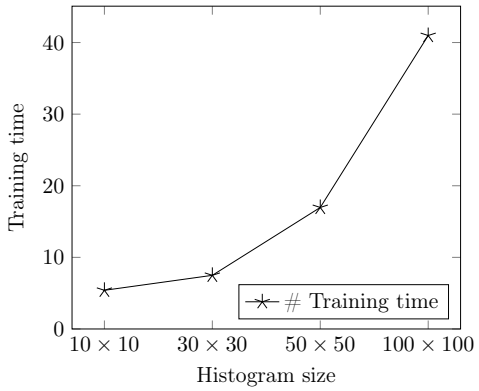
3.5.5 The effect of histogram size

In this section, we study the effect of the histogram size. Since the histogram size controls the size of the input, the optimal model parameters, i.e., number and size of hidden layers. Therefore, for each histogram, we repeat the model tuning experiments described in Section 3.5.3 and we report here the results of the optimal model.



(a) Accuracy

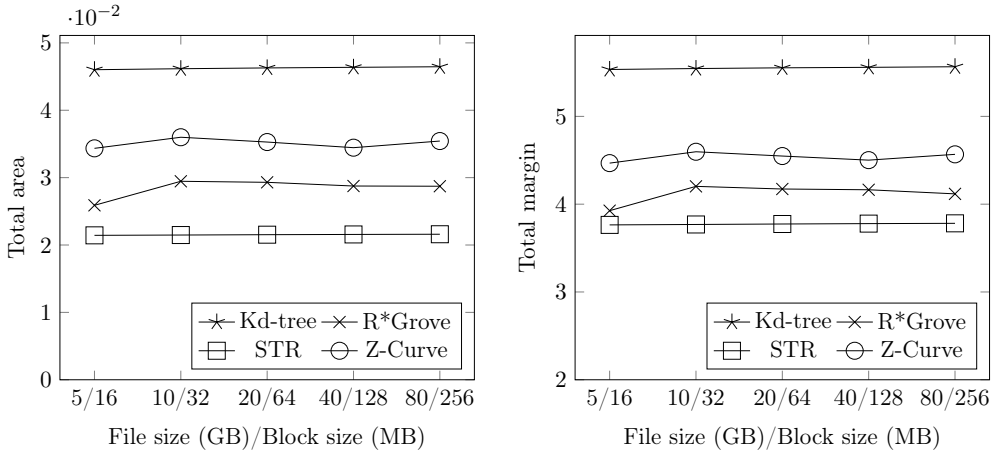
(b) Best model parameters



(c) Training time in seconds

Figure 3.16: The effect of histogram size

Figure 3.16 reports the accuracy of the best model found as the histogram sizes from 10×10 to 100×100. This experiments shows the trade-off between the model complexity



(a) Total area

(b) Total margin

Figure 3.17: Stability of quality metrics

and accuracy. One one hand, when the histogram size is small, the model also tends to be small but can be trained accurately. On the other hand, when the histogram is large, the model becomes more complex, but it cannot be trained accurately given the amount of training data that we have. The histogram size of 50×50 tends to strike a balance between these two.

Figure 3.16(b) further confirms this observation by showing the optimal model parameters that we found for each histogram size, i.e., number of hidden layers and size of the hidden layers. For the largest histogram size, the neural network model becomes more complex with more layers and more neurons per layer. We expect that if there are more training data, a larger histogram size could be more suitable. Additionally, we also show in Figure 3.16(c) that a 100×100 histogram requires a significantly 2.5 times longer to stabilize as compared to the 50×50 histogram which is also attributed to the complexity of the model.

3.5.6 Stability of Quality Metrics

In this experimental evaluation, we used moderate-size synthetic data with about 1.0 GB each. Although the real datasets can be arbitrarily large, we chose to keep the synthetic datasets small to be able to generate many datasets in a short time. We experimentally show in this part that this is still a valid approach by showing that the relative performance of the synthetic datasets is the same regardless of the size. Which means that the deep learning model will see no difference between the small and big datasets in terms of which partitioning technique is better as long as the distribution is fixed.

In the experiment shown in Figure 3.17 we fix the distribution type to the diagonal dataset and we vary the generated dataset size from 5 to 80 GB. As we increase the dataset size, we also increase the block size to ensure that the number of blocks is roughly the same for a fair comparison. For example, when we increase the size from 5 GB to 10 GB, we also increase the block size from 16 MB to 32 MB. We evaluate the performance of four partitioning techniques (Kd-tree, R*-Grove, STR, and Z-Curve) in two quality metrics (Q1 - total area and Q2 - total margin). The main observation from Figure 3.17 is that the quality measures of partitioning technique do not change as long as the ratio of dataset size / block size remains constant. Therefore, the best partitioning technique (STR) is also consistent over different dataset sizes as well. Given this result, instead of spending hours to compute the best partitioning technique of a dataset with size 128GB with normal HDFS block size (128 MB), we could execute the same operation for a dataset size of 1GB with HDFS block size 1MB and get the same result. This observation allow us to significantly reduce the time

Table 3.9: The independence of the best index in terms of total area and dataset size

Dataset size(GB)	Kd-tree	R*-Grove	STR	Z-Curve	Best Index
20	0.057	0.030	0.026	0.032	STR
50	0.039	0.028	0.029	0.034	R*-Grove
100	0.037	0.030	0.020	0.041	STR
200	0.036	0.032	0.032	0.050	STR

to generate our training data points. In practice, we generate the training data points from datasets in Table 3.7 with HDFS block size is 4MB.

In order to show that there is no dependency also on the HDFS block size and therefore on the number of blocks that the technique produces, we perform an additional experiment where we consider a collection of datasets with diagonal distribution from 20GB to 200GB. For each dataset the master files of four partitioning techniques are generated and quality measure $Q1$ (i.e., total area) is computed. The number of blocks generated by the different techniques is changing, since the dataset size changes while the HDFS block size is fixed to 128MB. Results are shown in Table 3.9. Notice that again the best technique is almost the same one (STR): the only case in which it is not corresponds to a size of 50 GB. However, in this case STR is very close to the best technique (R*-Grove) in terms of quality with a difference of only 4%. This confirms that we can train the model considering medium-size synthetic datasets without sacrificing the accuracy of the model.

3.5.7 Performance of the Summarization Phase

This section discusses the performance of the proposed approach for generating the summarization of each dataset, which is particularly important, since this computation has

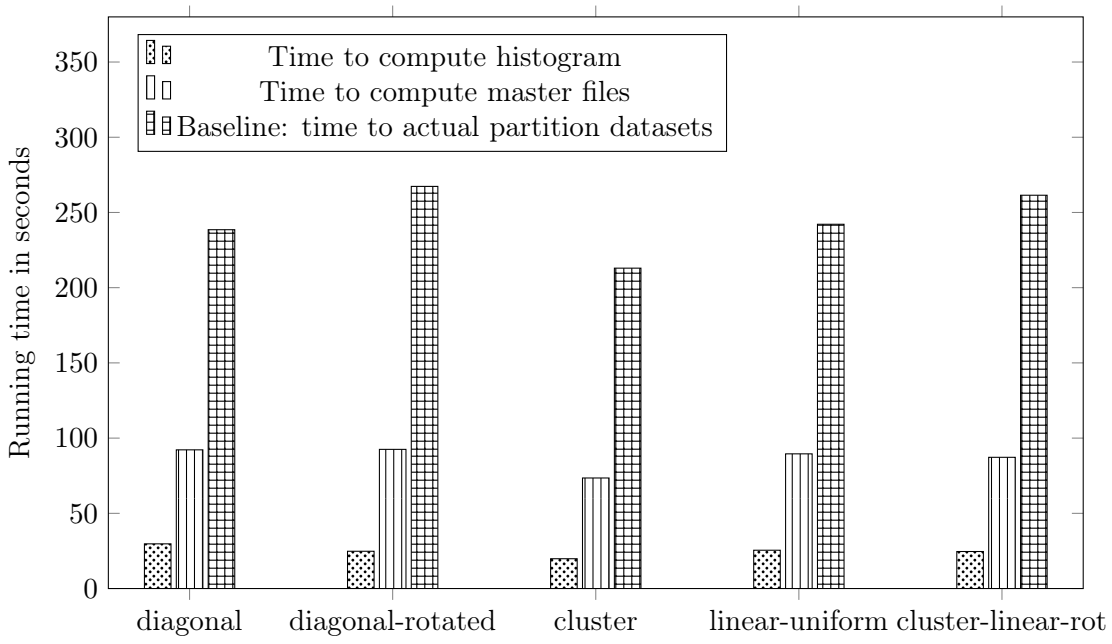


Figure 3.18: Summarization performance

an impact on both the generation of the training set and the application phase. Notice that the first is only applied once, while the second is at work when the solution is operative.

We focus on the algorithm that computes the master files, and hence the quality metrics, on one side, and the procedure that generates the histogram of a dataset, on the other side. The first one is used only for generating the training set, the second one is used also in the application phase.

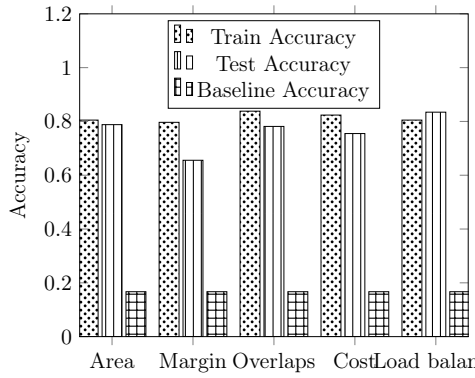
For the latter we compute the histograms using Spark as further explained in [49], while to compute the master files for the six partitioning techniques we use our optimized algorithm, which is mentioned in Section 3.3.3 to correctly compute the six collections of master files in one job without physically partitioning the data.

In both cases we consider as baseline approach the algorithm that physically partitions the data using the six partitioning techniques and then collects the master files from the outputs and determines the best technique by considering five quality metrics.

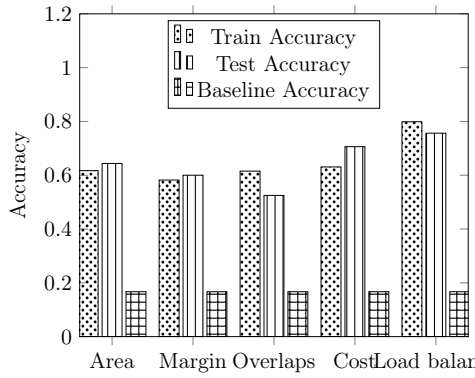
Figure 3.18 shows the efficiency of computing the histogram and the master files. It is clear that our method of generating the master files is much faster than the baseline method; this allow us to save a lot of time when producing the training set for synthetic datasets. Notice that we only compute master files for training purpose, where we compute the label for a dataset by determining the best partitioning option based on master files. In the application phase, we only need to compute the histogram or skewness features of the given dataset to predict the best partitioning technique. Figure 3.18 also indicates that the time to compute histogram of a dataset is very small when compared to the time to compute master files. This promises that if we have a good enough trained model, we can quickly predict the best partitioning option instead of actual compute the master files for all techniques to determine the best one.

3.5.8 Training Data with Skewed Shapes

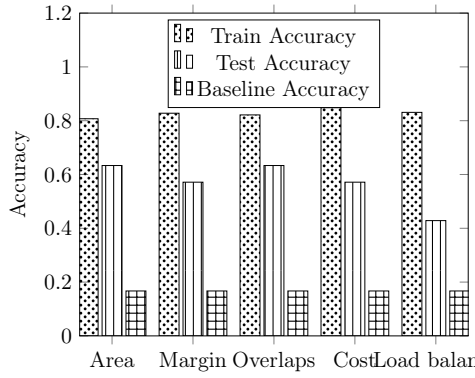
In this section, we study the effect of using non-point training data. More precisely, in the previous experiments the generated synthetic datasets contain rectangles that, considering the reference space, are relatively small, since they have to represents real objects like buildings or road segments compared to the extension of a state or continent. The goal is to explore whether a non-point dataset, i.e., dataset containing big and oblong rectangles, would enrich the model by extending the dataset characteristics. Figure 3.4 illustrates an example how



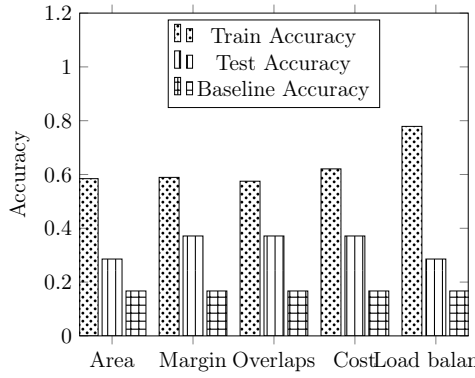
(a) Histogram summarization with test on **synthetic** data



(b) Skewness summarization with test on **synthetic** data



(c) Histogram summarization with test on **real** data



(d) Skewness summarization with test on **real** data

Figure 3.19: Experiments on training data with skewed shapes

the new datasets look like. We generate a total of 60 new datasets that follow the six distributions illustrated in Figure 3.3.

Figure 3.19 shows the results of the model when the input dataset contains a mix of points and rectangular datasets. Comparing these results with the results in Figure 3.13, we have two observations. First, the accuracy improves when adding the oblong rectangle datasets to the training sets which affirms that non-point data enriches the training set by adding new characteristics. This is especially true when testing on real data (Figure 3.19(c)

and 3.19(d) as compared to Figure 3.13(c) and 3.13(d)). Second, the gap between the histogram-based summarization and the fractal-based skewness measure summarization is reduced after using the non-point dataset. This shows a promise in deep learning being more efficient than the hand-crafted skewness measures provided that we can generate training datasets with diverse distributions and characteristics.

3.6 Related work

In this section, we review the related work in literature in three categories: spatial partitioning, data summarization, and deep learning.

3.6.1 Spatial Partitioning

Spatial partitioning is an essential operation in all big spatial data frameworks [74]. Regardless of the underlying architecture, e.g., disk-based or memory-based, data partitioning is essential to scale out to multiple machines. SpatialHadoop [72] proposed the idea of sampling-based partitioning in which a sample is used to estimate the data distribution and then a partitioning is applied to the big dataset in parallel. This idea was generalized to seven partitioning techniques including Grid-based, R-tree-based, Quad-tree-based, and space-filling-curve-based techniques [62]. Other systems follow a similar approach such as Scala-GiST [131], SATO [183], GeoSpark [203], and Simba [199]. AQWA [19] uses an adaptive histogram rather than a sample to summarize the data and query workload for data partitioning. In [134], a Voronoi-diagram-based partitioning technique is proposed to solve the kNN-join operation. R*-Grove [187] is another spatial partitioning technique

that extends the R-tree family for big spatial data systems. Other research work reuses some of these partitioning techniques to address other spatial analytic operations such as computational geometry operations [68] and visualization [82].

This chapter does not propose a new partitioning technique; rather, it proposes a framework that can suggest one of these partitioning techniques based on the input data distribution and analytic operation requirements.

3.6.2 Data Summarization

Many statistical techniques are used in data processing systems in order to provide a summarized description of a dataset, for instance through a sample, a histogram or a distribution model. These descriptors, often called sketches, are used to speed up the query processing by providing approximated answers based on them [55, 169, 172, 171]. One of their main uses in spatial big data analysis can be the estimation of selectivity for a join operation. The two sketching techniques that are relevant to this chapter can be classified into two main categories: sampling-based methods and histogram-based methods. Sampling-based methods are the basis of most existing spatial partitioning technique available in big data systems, like SATO [183], SpatialHadoop [72, 62], ScalaGiST [131], and Simba [199]. A histogram-based technique was employed by AQWA [19] to provide an adaptive partitioning technique for big spatial data based on query workload. The histogram is used to summarize the query workload which is then used to adaptively partition the data. In general, histogram-based methods are shown to be superior for accurate spatial selectivity estimation [4, 156], and some attempts have been made in order to use them to answer range queries in constant time [103, 49].

This chapter uses histogram-based techniques to summarize the data into a fixed-size vector. Unlike AQWA [19], which used Euler histogram, this chapter also uses skewness measures based on these histograms including Moran's Index and box counting [38, 40].

3.6.3 Deep Learning

With the rise of deep learning, more research work aim at utilizing it improving decisions and recommendations such as visualization recommendation [105], query optimization [185]. One of the notable works is the learned index structures [119] which replaces the complex index structures for datasets with certain characteristics with a small neural network model and an auxiliary data structure. Similarly, there has been some work on learning locality sensitive hashing (LSH) to build approximate nearest neighbor (ANN) indexes [194]. In this work, we do not aim to replace existing methods but to alleviate the choice between existing ones using deep learning.

3.7 Conclusion

This chapter explores the use of deep learning techniques to choose an appropriate spatial partitioning method. It formally defines partitioning techniques, quality metrics, and the partitioning selection problem which aims at choosing the partitioning technique that will maximize a given quality metric for a dataset. The proposed framework runs in two phases, training and application. The training phase builds a deep learning model by generating synthetic datasets of diverse distributions. It uses these synthetic datasets to train a model by choosing the best partitioning technique for each one. To allow the deep learning

model to work with a variable size dataset, we choose and contrast two summarization techniques termed fractal-based and histogram-based techniques. The application phase uses this model to choose the best spatial partitioning technique. We build a prototype of this framework that uses six partitioning techniques and four different quality metrics. The experimental results show up-to 87% accuracy of the proposed model in recommending the best partitioning technique. We also found that the histogram-based summarization is more efficient for synthetic data while the fractal-based techniques are more efficient with real data. This suggests that we can increase the size and diversity of the training data to achieve a higher accuracy with histogram-based technique. In summary, the results show that deep learning can be used to catch the spatial data distribution in an efficient and concise way.

Chapter 4

Incremental partitioning for efficient spatial analytics

4.1 Introduction

Spatial data is being produced at increasing rates from various sources such as mobile applications and satellite data. For example, there is an average of 500 million tweets sent every day [179] from users at different spatial locations. NASA EOSDIS adds about 6.4 TB of data to its archives every day [84]. In all these applications, data is not only large in volume, but it is also continuously growing and changing. These characteristics urged the research community and industry to develop new systems for big spatial data [74, 12, 198, 204].

When organizing spatial data, there are two main approaches, depending on the query processing needs of the system. If the focus is on highly selective queries (e.g. point look-up, top-k) data is indexed; in the first approach (termed *record-level*) every record finds

its exact position in the index structure (e.g., R-tree [97], quad-tree [165]) so that the highly selective query will access very few records using the index structure. While such queries are fast, there is an overhead in maintaining the index. On the other hand, if the focus is on analytical queries (e.g. aggregates, spatial joins [160], kNN joins [134], polygon union [69], convex hull, Voronoi diagram [126], DBSCAN [101], etc.), it is better to partition the data at a coarser granularity. Here, the exact record position is not important; rather records are organized in partitions (e.g., hash or range partitioning). After a record's partition is identified, its position within the partition is not important. This is because an analytical query will read all records in each partition that it accesses. As a result, the second approach (termed *block-level*) incurs less overhead in creating and maintaining the partitions, compared to the index maintenance of the record-level approach.

In order to support high ingestion or deletion rates while providing indexed access to files (*record-level* approach) systems use the Log-Structured Merge-tree (LSM tree) data organization [150, 48, 17]. In LSM-Tree, new records are inserted sequentially in (fast) main memory to create a component file (also called memtable). After a component file gets full, it is written sequentially to pages. Eventually component files in the pages are merged together and records find their exact position in the index. Record-level approaches using the LSM-Tree include Apache HBase[100], Accumulo[8], AsterixDB [16], MD-HBase [147], Parallel Secondo [130], BBoxDB [144], and GeoMesa [106], among others. In these systems, a new record is sent to one of the participating nodes and is then indexed by a spatial index residing in that node. A highly selective query will run in parallel on each node accessing

only relevant records through each node's spatial index. While LSM allows such systems to have high update rates, the amortized maintenance cost per record remains high.

On the other hand, distributed query processing engines that focus on analytical queries, such as Spark and Hadoop, follow the *block-level* approach. Typically the block size is 128 MB, which is much larger than a disk page (4-8KB) of the *record-level* storage engines. Example systems that use this approach include Slalom [149], for general purpose data analytics, and systems that are tailored towards spatial data analytics (i.e. using spatial partitions) such as SpatialHadoop [73], Simba [198], Hadoop-GIS[12] and GeoSpark [204], among others [74]. Unfortunately, due to the sequential write limitation of DFS (files are written sequentially to avoid expensive random writes), like HDFS [168], Amazon S3 [20] and Microsoft Azure [23] Blob storage, there is no mechanism that current block-level systems can use to maintain their spatial partitions incrementally.

The problem is exacerbated by the presence of *updates*. For example, a Twitter analytics system must periodically delete bot tweets, which are discovered by bot classifiers that run periodically on the data. This requires efficiently handling batch updates, which are currently not adequately handled by existing block-level platforms.

In this chapter, we study how a system can combine both efficient spatial analytic queries and high ingestion rates, as shown in Figure 4.1. Single-machine Spatial DBMS systems, e.g., PostGIS, provide record-level indexing for relatively low ingestion rate that a single machine can provide. BDMS systems and key-value stores, e.g., AsterixDB and GeoMesa, are able to support very high ingestion rates for record-level indexes by using distributed LSM-Tree indexes. On the other hand, block-level partitioning on big spatial data

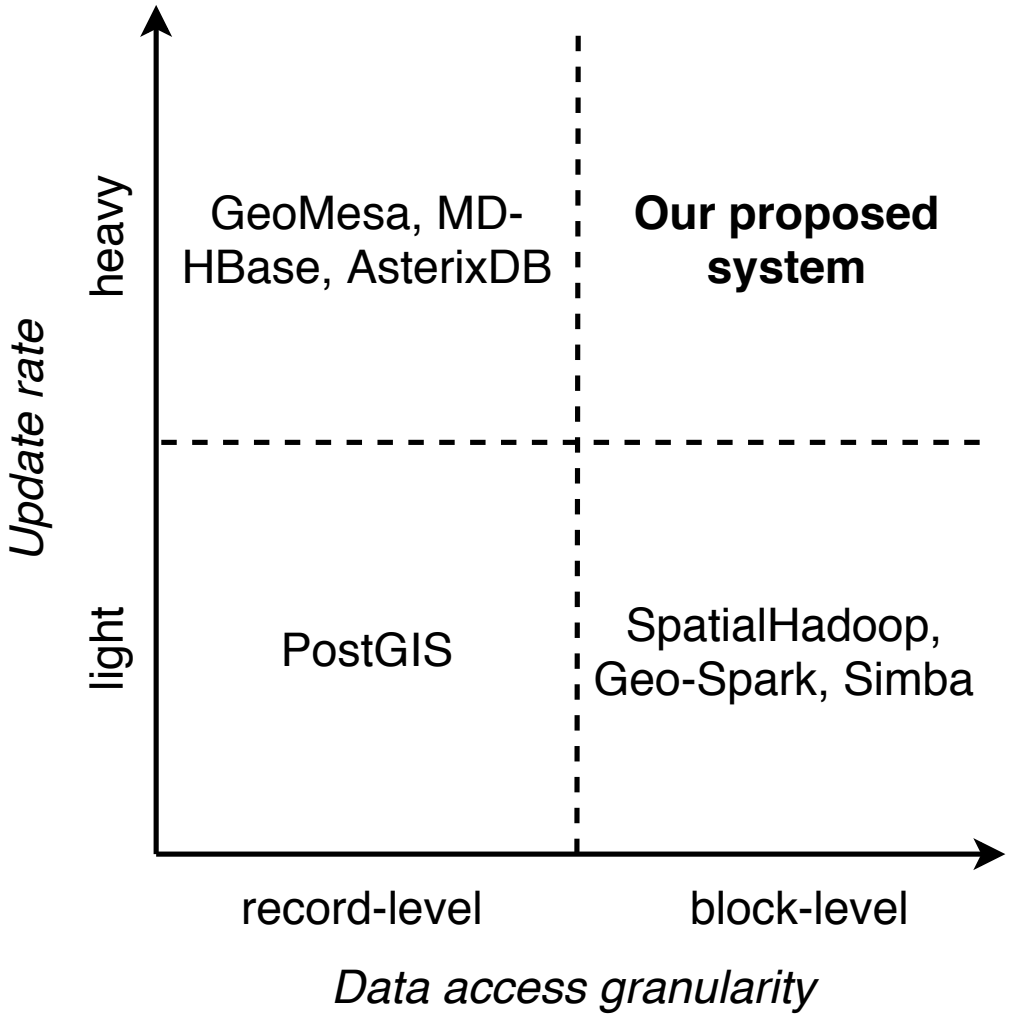


Figure 4.1: Our proposed system in the context of other systems

is supported by systems like SpatialHadoop and Simba but they can only support applications with low ingestion rates where the data can be occasionally repartitioned. Furthermore, block-level systems naturally do not support delete operation. The proposed framework in this chapter supports high insertion and deletion rates while incrementally partitioning the data in blocks. *Hence, our proposed work can be viewed as the LSM-equivalent for block-level spatial data, where the goal is to avoid the high overhead of the record-level LSM merges.*

This chapter proposes a new framework that can efficiently support incremental big spatial datasets with batch ingestion, addressing the limitations of the state-of-the-art. We focus on ingestion/deletion since for the applications we consider new data is continuously added, deleted, or updated. A *first key property* of the proposed framework is to facilitate partitioning of data based on their spatial attributes, so as to perform fewer accesses during query time. The partitions are created and stored directly in DFS which allows MapReduce and RDD programs to run directly on the partitions. A *second key property* is to facilitate a pay-as-you-go partition maintenance mechanism that can be optimized based on the objectives of the application.

To achieve these properties, the proposed framework has two phases, namely, (i) *data flushing*, where new or deleted data is periodically pushed to secondary storage, and (ii) *partition optimization*, where the newly added/deleted data and the old partitions are jointly maintained given cost budget constraints.

Initially, we formalize the partition optimization problem and prove its NP-Hardness, which implies that building optimal partitions is non-trivial. Therefore, we break the problem into two subproblems, namely: *partition selection* and *partition reorganization*, which can be solved separately. The former gives us a set of potential partitions to reorganize and the latter physically reorganizes the records in the selected partitions. To solve the *partition selection* problem, we propose three incremental partitioning techniques, termed, *R*-Tree-Inspired Partitioning(R*P)*, *LSM-Tree-Inspired Partitioning(LSMP)*, and *Cost Based Partitioning(CBP)*. R*P is motivated from the node insert and node split algorithm in R*-tree. In the other place, LSMP utilizes the idea of LSM-Tree merge policy to maintain its

partitions. Finally, CBP takes a range query cost model into account in its optimization process so that it can minimize the estimated query cost, given a limited reorganization budget.

The key challenges in the partition selection problem are *(a)* how to estimate the benefit of selecting a specific set of partitions to reorganize, and *(b)* how to efficiently navigate the combinatorial search space of partition subsets, given a reorganization budget. For that, we introduce a novel block-based cost model for spatial partitioning that estimates the number of accessed disk blocks for a range query, and propose an algorithm to select a set of partitions to optimize this cost. The experimental results indicate that this cost model allows us to create a high performance incremental partitioning scheme in terms of both partitioning time and query throughput, as compared to our two straightforward solutions and to state-of-the-art big spatial data systems.

In summary, the main contributions of this chapter are:

- We introduce a comprehensive framework for incremental spatial partitioning of large-scale spatial datasets.
- We formalize the partition optimization problem for incremental spatial partitioning systems and prove its NP-Hardness.
- Further, we propose three different implementations of the partitioning frameworks, including an approximate algorithm to solve partition optimization problem.
- We perform a comprehensive experimental evaluation on incremental big spatial systems to measure their performance and partitioning quality.

The rest of this chapter is organized as follows: Section 4.2 gives an overview of the proposed framework. Section 4.3 formalizes the partition optimization problem and proves its NP-Hardness. Section 4.4 proposes a new cost model for distributed spatial indexes. Section 4.5 shows the different incremental spatial partitioning techniques which implemented the proposed framework. Section 5.4 shows experimental results to validate our proposed work. Section 4.7 reviews related work while Section 5.5 concludes the chapter.

4.2 A generic incremental partitioning framework

Figure 4.2 gives an overview of the proposed framework for incremental partitioning of big spatial data. This is a generic framework in the sense that its steps can be implemented differently to produce various types of incremental partitioning schemes. This chapter provides three different implementations that follow this framework. The framework consists of two phases, *data flushing* and *partition optimization*. In the *data flushing* phase, a set of new records or deletion markers is ingested into an existing, initially empty, partition structure. Given the limitation of the distributed file systems (which prevents random file updates), the flushing phase is only allowed to append to existing files or create new files. This results in an intermediate (transient) state of the partition, which can be used in query processing but might not be optimized.

In the second *partition optimization* phase, the intermediate partitions are optimized by reorganizing some or all of them. This phase first identifies a subset of partitions, then it reorganizes their contents into a new set of partitions. In Section 4.3, we prove that the *partition optimization* problem is NP-hard. As it is impractical to find an optimal solution,

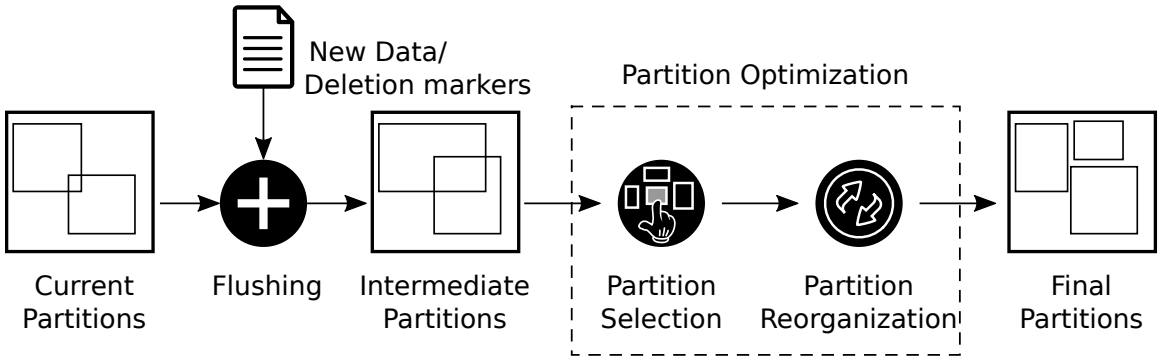


Figure 4.2: Overview of incremental partitioning of big spatial data

we break the problem into two sub-problems, *partition selection* and *partition reorganization*, solved separately as discussed below.

The *partition selection* step identifies a subset of *bad* partitions to be reorganized. The second step, *partition reorganization*, processes the selected partitions by reorganizing their contents into a set of *new* partitions; old partitions are then deleted. Since HDFS does not allow random updates, the modified files have to be completely rewritten. In summary, the partition optimization phase transforms the partition from an intermediate unoptimized state into a final optimized state. In both the data flushing and partition optimization phases, the index is kept under multiversion concurrent control which enables existing data analytics jobs to continue running on older versions while new jobs access the new version. A periodical garbage collection process cleans up non-used files.

We proceed by first describing the layout of the partition on the distributed file system; then we provide more details about the three main steps of our framework (data flushing, partition selection, and partition reorganization).

4.2.1 Partition layout

This chapter focuses on block-level partitioning which partitions the data such that each partition fits in one HDFS block [168] of a default size 128 MB. Local indexing can be employed to determine the internal format of each 128 MB block but this is outside the scope of this chapter. Each partition is stored on disk as a separate file. Additionally, a *master* file stores the metadata of the partitions which consists of a partition ID, the minimum bounding rectangle (MBR) of the partition, number of records, total size of records marked for deletion, and total size of non-deleted records. A spatial query starts by examining the master file to decide which partitions to process, e.g., the partitions that overlap the area of interest. Then, the selected partitions are processed in parallel using MapReduce [57] or RDD [206]. If multiple master files exist, the most recent one is used to adhere with multiversion concurrency control schemes.

4.2.2 Data flushing phase

As shown in Figure 4.2, the data flushing phase takes a batch of new data or deletion markers and ingests it to the partitioning. Typically, systems that deal with big data, buffer these updates in memory and trigger the flushing phase when the in-memory component reaches a pre-specified threshold, e.g., 4GB. Since this phase is triggered while the system is *hot* and accepting updates to the data, it prioritizes the insertion time over the quality of the partition. This allows the system to continue accepting new records at the highest rate with minimal partition maintenance overhead. In this proposed framework, we limit the flushing phase to appending to existing files and writing new files. This limitation is driven by the

DFS limitation and also helps in reducing the amount of disk IO, which is equal to the batch size. The *master* file is not changed until the flushing phase is complete which makes all the changes hidden to the query processing. Once the flushing process completes, all the changes become visible by writing a new version of the master file that reflects the updates.

This chapter considers two flushing techniques, namely, *append*, which appends records to existing partitions; and *LSM flush*, which creates a new set of partitions (that logically form a new LSM component). Both are described in detail in Section 4.5.

4.2.3 Partition selection

The partition selection step identifies *target* partitions that need to be deleted and reorganized. We design this step to run under a system constraint which limits the amount of disk IO in this process (read + write). The goal is to choose a small subset of *bad* partitions, e.g., overlapping partitions, that are lowering the quality of the partitioning and reorganize them to boost its quality. For efficiency, this step operates only on the partition metadata, e.g., MBR and size, from the latest master file. We will show that this step has a significant impact on the quality of the partition and the overall reorganization time.

4.2.4 Partition reorganization

Given a list of target partitions from the previous step, the partition reorganization step completes the partition optimization phase by physically rewriting those partitions into highly-optimized partitions, with the removal of records which are marked for deletion. This step is generic and can use any existing static partition construction algorithm for big spatial data.

4.2.5 Multiversion Control / Garbage Collection

Since this proposed scheme is designed for long-running analytic jobs, all the updates (flushing and optimization) are done through a multiversion concurrency control (MVCC) scheme [197]. In this scheme, old files are not automatically deleted, but existing files can be appended and new files are created. While an update is going on, the master file is kept intact. As a result, none of the updates are visible to the queries because the master file is not updated. Once the update process is completed successfully, a new version of the master file is created to reflect the updates. Existing jobs can continue to run because they used an older valid version of the master file. New queries will only access the new files listed in the newest version of the master file.

At the beginning of each job, a light-weight *garbage collection* (GC) step deletes non-used master files and corresponding data files. The GC step starts by finding the earliest active job, i.e., the job with the earliest start time (t_s). Then, it marks all master files that were created before t_s for deletion, except the most recent master file. For each master file marked for deletion, a simple `diff` operation with the subsequent master file identifies which partitions are no longer accessible and deletes them. Then, the master file itself is deleted. This step takes only a fraction of a second and does not require stopping or pausing any existing jobs. The system can continue accepting and running jobs without the need of a locking mechanism.

Table 4.1: Table of notations

Notation	Description
$r(mbr, size, is\_deleted)$	a record.
b	default block size, e.g., 128 MB.
$D = \{r_1, \ldots, r_n\}$	a dataset D is a set of records.
$MBR(D)$	the MBR of a set of records D.
$psize(D)$	the physical size of D.
$csize(D)$	the condensed size of D.
$pblocks(D)$	the number of physical blocks of D.
$cblocks(D)$	the number of condensed blocks of D.
$P = \{p_1, \ldots, p_m\}$	A global partitioning state
$T(P, P')$	The disk IO cost to transform P to P'
$C(P)$	Function to compute query cost on P

4.3 Partition optimization problem

4.3.1 Preliminaries and problem definition

To formulate the problem, we first present the following definitions; notations are summarized in Table 4.1:

- $r(mbr, size, is\_deleted)$: a record r is represented by its minimum bounding rectangle (MBR) and size in bytes. In addition, $is\_deleted$ is a Boolean tombstone flag with value $\{0, 1\}$ to indicate whether this is a deleted record.
- b: default block size in the file system, e.g., 128 MB.
- $D = \{r_1, \ldots, r_n\}$: a dataset D is a set of records.
- $MBR(D) = \bigcup_{i=1}^{n} mbr(r_i)$: the MBR of a set of records is the minimum MBR that contains the MBRs of all its records.

- $psize(D) = \sum_{i=1}^{n} size(r_i)$: the physical size of a set of records is the sum of all record sizes. This indicates the disk space needed to store these records.

- $csize(D) = \sum_{i=1}^{n} (1 - 2 \cdot is\_deleted_i) \cdot size(r_i)$: the condensed size of a set of records corresponds to the size this set would occupy if deleted records are removed. It is computed by subtracting the size of deleted (tombstone) records.

- $pblocks(D) = \left\lceil \frac{psize(D)}{b} \right\rceil$: is the number of physical blocks for a set of records.

- $cblocks(D) = \left\lceil \frac{csize(D)}{b} \right\rceil$: is the number of physical blocks D will occupy if condensed..

- $P = \{p_1, \ldots, p_m\}$: A global partitioning state, hereafter will be simply called a *partitioning*, of a dataset is a set of partitions p_i, where each partition is a subset of the input D. Partitions satisfy the following three constraints, (1) $p_i \subseteq D$, (2) $p_i \cap p_j = \emptyset, \forall i \neq j$, and (3) $\bigcup_i p_i = D$. Similar to the dataset D, we can also compute MBR, *psize*, *csize*, *pblocks* and *cblocks* of every $p_i \in P$. In this work, we assume the records inside a partition are not indexed (i.e., there is no local index), but this decision is orthogonal to the reorganization problem that we study. Previous work[73] showed that local indexes have little impact on the overall performance of analytical queries on big spatial data.

- $T(P, P')$: Given two partitions, P and P', the cost to transform P to P' is defined as the number of blocks to read from P and the cost of writing new condensed partitions in P'.

$$T(P, P') = \sum_{p_i \in P \setminus P'} pblocks(p_i) + \sum_{p_i \in P \setminus P'} cblocks(p_i)$$

- $C(P, s)$ is a function that measures the average cost of running a range query with size $s \times s$ on P. This metric, formally defined in Section 4.4.1, reflects how good the partitioning is for spatial query processing.

Partition Optimization Problem *Given a dataset D with partitioning P, a fixed number of accessed blocks B and a query size $s \times s$, find a new partitioning P' which can be transformed from P where $T(P, P') \leq B$ and $C(P', s)$ is minimized.*

Input:

- Dataset $D = \{r_1, \ldots, r_n\}$
- Current index $P = \{p_1, \ldots, p_m\}$
- Budget B: maximum blocks we can read in the reorganization process.
- A quality function Q

Output: Another index $P' = \{p'_1, \ldots, p'_{m'}\}$ such that $Q(P')$ is maximized subject to $Cost(P, P') \leq B$

4.3.2 The NP-Hardness of the problem

We proceed to prove that a simplified version of the partition optimization problem is NP-Hard (and hence the partition optimization problem is also NP-Hard). In particular, we show that if we have a quality function that is defined independently for each partition, the problem can be reduced from the well-known 0-1 Knapsack problem.

(0-1 Knapsack) Given a set of n items numbered from 1 to n, each with weight w_i and value v_i, and a maximum weight capacity W, find a vector $X = \{x_1, \ldots, x_n\}$ that:

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0, 1\}$$

Reduction Algorithm: We transform the 0-1 Knapsack problem to the following partition optimization problem:

- Define D as a set of n records where the size of each record r_i is w_i.
- The current partitioning P contains n partitions, where each partition contains only one record, $p_i = \{r_i\}$.
- The budget B is equal to the weight capacity W, and the block size $b = 1$.
- The cost function for one partition $c(p, s)$ is:

$$c(p, s) = \begin{cases} 0 & if\ |p| = 1 \\ \sum_{r_i \in P} v_i - \sum_{r_i \in p} v_i & otherwise \end{cases}$$

  Finally, we define $C(P, s) = \sum_{p_i \in P} c(p_i, s)$. In particular, the cost for a partitioning state P is the total of cost for each partition in P.

To complete the proof, we need to show that (1) the optimal answer can be mapped between the two problems, and (2) both the problem reduction and the answer mapping require a polynomial time.

(1) Assume that the optimal solution for the 0-1 Knapsack problem is $X = \{x_1, \ldots, x_n\}$. The set of items with $x_i = 1$ correspond to a subset of records $R = \{r_i | x_i = 1\}$. In this case, there is an optimal answer P' where each item $x_i = 0$ maps to a partition with one record $p_i = \{r_i\}$, and all the items with $x_i = 1$ are combined in one partition R. On the other hand, given an optimal answer P' to the partition optimization problem, we can map it to an optimal answer to the 0-1 Knapsack as follows: Each record $r_i \in p'_j$ where $|p'_j| = 1$ will be mapped to $x_i = 0$, otherwise, if $|p'_j| \geq 2$, r_i is mapped to $x_i = 1$. The key idea is that all partitions selected to be modified by the partition optimization problem as an optimal solution will have a total cost equal to the weights of their records and the answer will have a quality equal to the total value of those selected records.

(2) Both the reduction algorithm and the answer mapping above require $O(n)$ time complexity, i.e., the reduction process is a polynomial-time algorithm.□

4.4 Cost-benefit analysis of the partition selection process

Given that the partition optimization problem is NP-hard, we break it into two smaller sub-problems: *partition selection*, which selects a subset of partitions to be reorganized, and *partition reorganization*, which reorganizes the records in the selected partitions. Previous work on static indexes for big spatial data [73, 66, 188, 198, 133] can be used to solve the second problem while there has been little attention to the first one. This section focuses on the first problem, provides a theoretical analysis, and develops a cost model for it.

The key idea is to create an accurate cost model for range queries and use it as a proxy for the quality of the partitions (Section 4.4.1). We use range queries as they are the

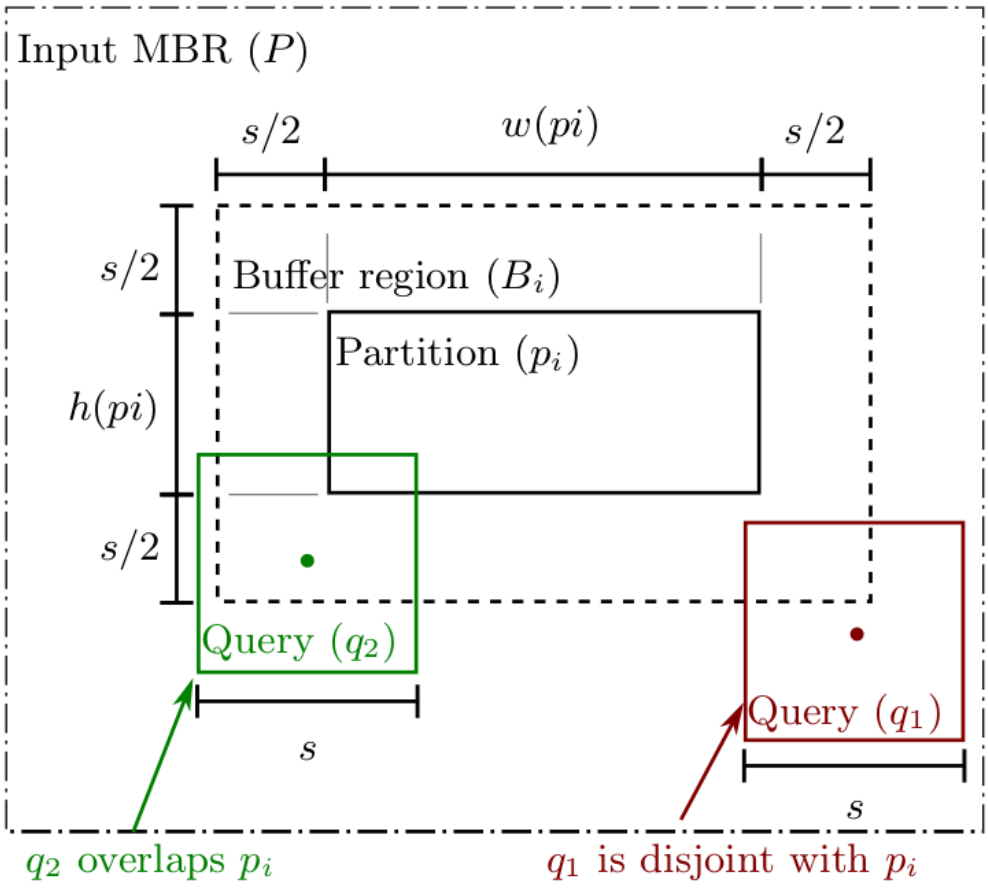
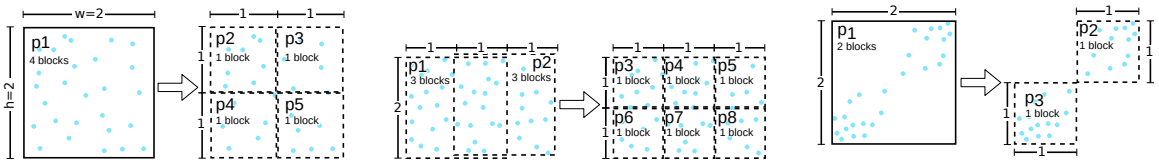


Figure 4.3: Different relationships of a range query with a partition.

most fundamental operation in spatial data analytics [66, 104]; range query is commonly used as the building block for other spatial operations such as joins or aggregations, as we discuss below. Then, we define a *benefit function* that uses the cost model to estimate the improvement in the partition quality for any subset of selected partitions (Section 4.4.2). The next section will show how to use this cost model to solve the partition selection problem.



(a) Split a partition with uniform data

(b) Split overlapping partitions

(c) Uncover dead area by splitting skewed data

Figure 4.4: Different scenarios when reorganizing a partition or a group of partitions to reduce the estimated cost and improve the quality

4.4.1 Range query cost model in HDFS

We build on previous research which showed that the cost of a range query is a good proxy for the quality of a spatial partitioning method (i.e., a partitioning state P) even for analytical queries such as spatial join [66, 104]. Given a fixed range query of size $s \times s$, our cost model estimates the total number of blocks and total size of data that it will process.

Traditional range query cost models [50, 35, 9, 178] focused on estimating the number of disk pages required to answer the query or the query size. This made sense for traditional DBMS algorithms which access data from a regular disk with a relatively small disk page, e.g., 8 KB. The assumption was that a disk page is the smallest access unit to a disk which makes the cost uniform on all disk pages regardless of how many actual records are in each page.

When transitioning to the distributed file system, the previous assumptions no longer hold. In HDFS, data is stored in blocks which can vastly vary in size from a few megabytes up-to 128 MB. Therefore, the cost of accessing each block is no longer uniform. In addition, we have an opportunity to build a more accurate estimate as compared to traditional models. In traditional DBMS, the cost estimation is part of the query optimization which should take only a few milliseconds. However, this work uses this cost model as part of

the index optimization step which can take tens or hundreds of seconds so we have an opportunity to run a model that takes a second or two without significantly hurting the overall performance.

Our analysis starts with an $s \times s$ square-shaped query and tries to estimate its cost. The size s is a parameter in the cost model that we study later in the chapter. Assume that $P = \{p_1, \ldots, p_m\}$ is a partitioning state of a spatial dataset D. Suppose that $w(p_i)$ and $h(p_i)$ are the width and height of a partition p_i, respectively. Figure 4.3 shows the two different possibilities of a query q in relationship to a single partition p_i. As shown in the figure, the query q_1 is disjoint with the partition p_i which means that such query does not have to process the partition p_i. On the other hand, the query q_2 overlaps p_i and hence needs to process that partition to produce the answer. We compute the probability of a query of size $s \times s$ being disjoint or overlapping with the partition p. To do that, we define a *buffer region* B_i that expands p_i with a buffer size of $s/2$ in all directions. We can easily see that if the center of the query falls inside the buffer region (e.g., q_2), it overlaps the partition; otherwise, it is disjoint. If we randomly create a square range query q_i of size $s \times s$ in the domain space $MBR(D)$, the probability that the center of q_i falls inside the buffer region is the ratio of the area of B_i over the area of the domain space $MBR(D)$. Thus the average number of blocks that are contributed from p_i for query q_i is:

$$c_b(p_i, s) = \frac{(w(p_i) + s)(h(p_i) + s)}{w(P)h(P)} \cdot pblocks(p_i) \tag{4.1}$$

In addition, the average amount of data in bytes that is scanned from p_i for query q_i is:

$$c_s(p_i, s) = \frac{(w(p_i) + s)(h(p_i) + s)}{w(P)h(P)} \cdot psize(p_i) \tag{4.2}$$

To process a query, first, there is a fixed cost to load overlapping blocks, which correlates with number of partition's block. Second, there is a cost to scan the entire partition, which correlates with partition size. Therefore, the total cost (running time) to complete a query with size s can be represented as the following:

$$C(p_i, s) = k_b \cdot c_b(p_i, s) + k_s \cdot c_s(p_i, s) \tag{4.3}$$

Overall, the average number of blocks which we need to process to answer a query q_i on the partitions of P can be estimated as follows:

$$C(P, s) = \sum_{p_i \in P} C(p_i, s) \tag{4.4}$$

$$C(P, s) = k_b \cdot \sum_{p_i \in P} c_b(p_i, s) + k_s \cdot \sum_{p_i \in P} c_s(p_i, s) \tag{4.5}$$

4.4.2 Reorganization benefit

This section shows how to use the range query cost model described above to estimate the *benefit* of the reorganization step. We define the benefit as the reduction in the range query cost after the reorganization process, i.e., cost after subtracted from the cost before. To formalize the benefit calculation, suppose that the state of partitioning at timestamp t is P_t and after reorganization it will be P_{t+1} as presented in Figure 4.2. Now, assume that

the partition selection step has selected a group of partitions $G_t = \{p_1, \ldots, p_n\} \subseteq P_t$ to be reorganized. After these partitions are reorganized, they will produce a new set of partitions $G_{t+1} = \{p'_1, \ldots, p'_m\} \subseteq P_{t+1}$. We define the *reorganization benefit* of G_t as the reduction of query cost when the partitions are reorganized from P_t to P_{t+1}.

$$Benefit(G_t, s) = C(P_t, s) - C(P_{t+1}, s) \tag{4.6}$$

We can rewrite P_t as $(P_t - G_t) \cup G_t$ and similarly P_{t+1}.

$$Benefit(G_t,s) = C((P_t - G_t) \cup G_t, s) - C((P_{t+1} - G_{t+1}) \cup G_{t+1}, s) \tag{4.7}$$

Since our cost function C is linear, we can apply super position as follows.

$$\begin{aligned} Benefit(G_t, s) &= C(G_t, s) - C(G_{t+1}, s) \\ &+ C(P_t - G_t, s) - C(P_{t+1} - G_{t+1}, s) \end{aligned} \tag{4.8}$$

But $P_t - G_t \equiv P_{t+1} - G_{t+1}$, which are the set of non-selected partitions. Their cost is the same which means that the benefit or reorganizing G_t is:

$$Benefit(G_t, s) = C(G_t, s) - C(G_{t+1}, s) \tag{4.9}$$

Finally, we can utilize the Equation 4.9 to compute the benefit of the reorganization step that transforms P_t to P_{t+1}. To understand the key idea behind the computation of cost reduction, i.e., benefit, Figure 4.4 illustrates three examples of partitions before and after reorganization. For simplicity, we assume $k_b = 1$ and $k_s = 0$, then the total cost in

Equation 4.5 becomes $C(P,s) = \sum_{p_i \in P} c_b(p_i, s)$. In all three examples, we assume that the area of data space is $w(P) \cdot h(P) = 10$ and query size $s = 0.5$. In Figure 4.4(a), a single partition p_1 with four blocks is reorganized into four single-block partitions. According to the simplified cost model and Equation 4.1, $C(p_1, s) = \frac{(2+0.5)(2+0.5)}{10} \cdot 4 = 2.5$ and $\sum_{i=2...5} C(p_i, s) = 4 \cdot \frac{(1+0.5)(1+0.5)}{10} \cdot 1 = 0.9$. So, we say that the reduction in cost is $2.5 - 0.9 = 1.6$. This value means that if we reorganize p_1 into 4 smaller partitions, we would reduce 1.6 block accesses on average to answer a square query with size 0.5×0.5. This case indicates that partitioning a multi-block partition into several single-block partitions improves the cost.

The second case in Figure 4.4(b) gives an example of partitioning two overlapping partitions. According to our cost model the cost before reorganizing is $C(p_1, s) + C(p_2, s) = 2 \cdot \frac{(2+0.5)(2+0.5}{10} \cdot 3 = 3.75$ while the cost after reorganizing is $\sum_{i=3...8} C(p_i, s) = 6 \cdot \frac{(1+0.5)(1+0.5)}{10} \cdot 1 = 1.35$. The cost reduction is 2.4. This case indicates that partitioning overlapping partitions provides additional cost reduction.

The case in Figure 4.4(c) shows an example of splitting one partition p_1 with some empty regions into two blocks p_2 and p_3 while uncovering that empty region. In this case, we can calculate the cost as $C(p_1, s) = \frac{(2+0.5)(2+0.5)}{10} \cdot 2 = 1.25$ and $C(p_2, s) + C(p_3, s) = 2 \cdot \frac{(1+0.5)(1+0.5)}{10} \cdot 1 = 0.45$. The reduction in the cost is 0.8. Thus reorganizing a partition that contains some *dead space* reduces the cost.

To estimate the benefit of a reorganization scheme (estimated cost reduction), we would like to take these three cases into account. Nevertheless, it would be hard to account for case 3 (Figure 4.4(c)) since it requires extra information about the data distribution inside

the partition which is not available in the master file. Therefore, our benefit computation only accounts for the first two cases. The challenge here is we would not know how G_{t+1} looks like to compute the benefit in Equation 4.9 until we actually reorganized G_t. However, our goal is to use that equation to select the best subset $G_t \subseteq P_t$ in the partition selection process, which promises the maximum benefit. Therefore, we need to be able to estimate $Benefit(G_t, s)$ without physically reorganizing G_t, i.e., without knowing G_{t+1}. To resolve this issue, we compute a *prediction* $\widehat{\Theta}_{t+1}$ of G_{t+1} and use it to calculate a *prediction* (estimate) of the benefit. In this case, $\widehat{\Theta}_{t+1}$ is a set of partitions that we predict to produce after the reorganization step runs. In order to estimate $\widehat{\Theta}_{t+1}$, we assume that the partition reorganization step will produce $m = cblocks(G_t)$ single-block, square-shaped, equi-sized, and non-overlapping partitions. Since all the estimated resulting partitions are identical, their total area is equal to the area of the selected partitions G_t. Hence, the side length of each of the new partitions $p'_i, \forall i = 1 \ldots m$ is calculated as:

$$w(p'_i) = h(p'_i) = \sqrt{\frac{w(G_t) \cdot h(G_t)}{cblocks(G_t)}} \tag{4.10}$$

Although this is an ideal case that might not always happen, it is a good indicator of how far the cost might go down. The actual output of the reorganization step might have a higher cost (if the partitions are overlapping and not square) or a lower cost (if a dead space was uncovered). Finally, the estimated benefit when we reorganize G_t to $\widehat{\Theta}_{t+1}$ is:

$$\widehat{Benefit}(G_t, s) = C(G_t, s) - C(\widehat{\Theta}_{t+1}, s) \tag{4.11}$$

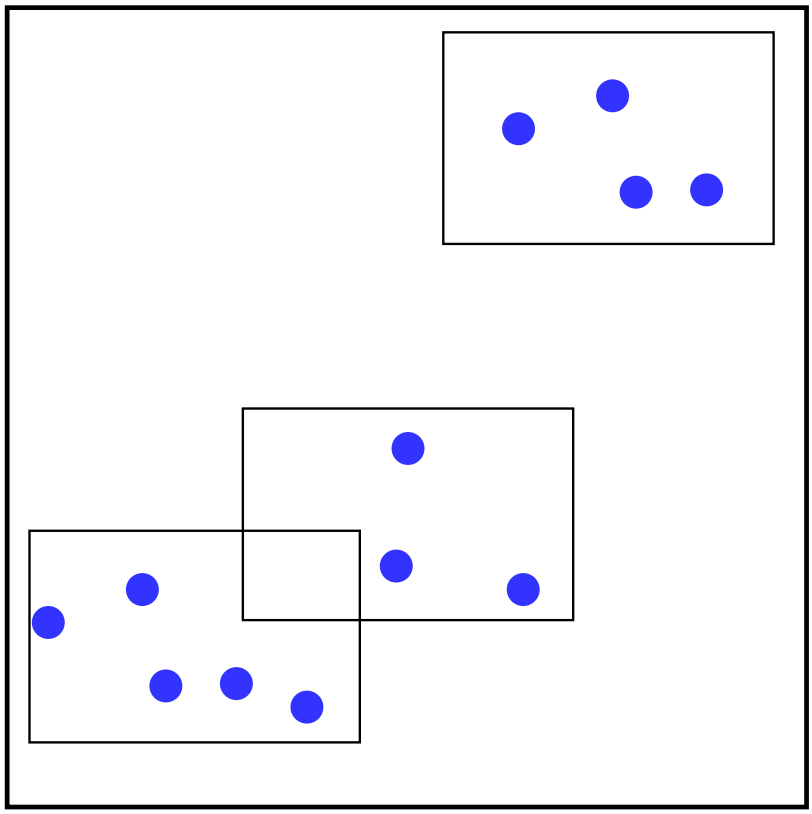


Figure 4.5: Group benefit: (1) Cluster the overlapping partitions (2) Compute total benefit of disjoint groups.

Because we can compute G_t and Θ_{t+1} before the partition reorganization process, the estimated benefit in Equation 4.11 is a good indicator to help us choose the partitions for reorganization with maximum benefit, given a limited disk IO budget. We proved that this is a NP-Hard problem. Therefore, we will use a greedy strategy to solve this problem, which will be discussed in Section 4.5.3.

Grouping Optimization When a large number of partitions is selected for reorganization, we need to decide whether to consider all of them in one group or we can split them into smaller groups. If we consider all of them in one group and apply Equation 4.10, we would assume that the resulting partitions cover the entire space which might be inaccurate

if there is a large gap between partitions. For example, in Figure 4.5, if we compute the benefit of the three partitions together, we would assume that the resulting partitions cover the entire input space. To get a more accurate estimate, we group overlapping partitions together in groups and apply Equation 4.10 to each group separately.

4.5 Proposed Incremental Partitioning Algorithms

This section presents three approaches to instantiate the incremental partitioning framework in Section 4.2. First, we use R*-tree [27], to create a block-based partitioning scheme, termed R*-tree-inspired partitioning (R*P, Section 4.5.1). R*P appends new data to existing partitions and splits overflow partitions into block-sized partitions. Second, we use LSM-tree [150] to build another incremental partitioning scheme, termed LSM-tree-inspired partitioning (LSMP, Section 4.5.2). LSMP creates a new LSM component for each data flushing step, and periodically merges its components based on an LSM merging policy. Third, we use the cost-benefit analysis in Section 4.4.2 to create a partitioning technique, referred as Cost Based Partitioning (CBP, Section 4.5.3), that aims to maximize the benefit of the reorganization process, given a reorganization budget.

Comparing the three approaches, R*P splits overflow partitions into smaller partitions, but it never merges partitions. In contrast, LSMP only merges partitions together based on its merging policy. CBP periodically reorganizes the partitions, essentially both splitting and merging, measuring the benefit using Equation 4.11. We use R*-Grove [190] as the spatial partitioning algorithm in all approaches, but it could be replaced by other spatial

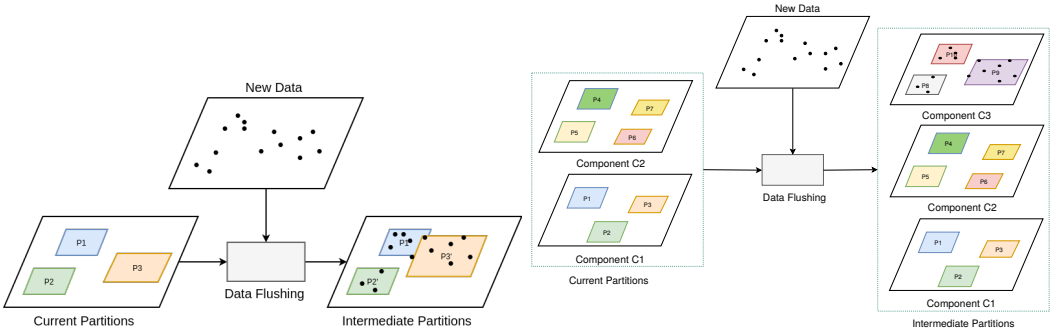
indexing techniques such as Grid File [145], Kd-tree [42], Quad-Tree [165, 89], or Hilbert R-Tree [110].

4.5.1 R*-Tree-inspired partitioning (R*P)

This section describes an adaptation of the traditional R*-tree index [27] to the incremental partitioning problem by implementing the three steps of the partitioning framework: *flushing*, *partition selection*, and *partition reorganization*. The general idea is to treat each partition as a leaf node in the R*-tree. The details of the three steps are described below.

Data Flushing: The flushing step uses the R*-tree insertion algorithm to choose a partition for each record and append it, as shown in Figure 4.6(a). The inputs to this flushing process are the current partition and a non-partitioned batch of new or to-be-deleted records. Based on the MBRs of the current partitions, the flushing process scans the records in the non-partitioned batch and appends each record to one of the partitions following the R*-tree ChooseSubtree method (i.e., choose the partition that requires the least overlap enlargement as detailed in [27]). The output of this flushing phase is a set of intermediate partitions which contains the same number of partitions as the input but the contents of these partitions include the new data records and the markers for deleted records.

Partition Selection: After the flushing phase is complete, the partition selection step identifies the partitions that need to be reorganized. Following the standard R*-tree design, this step simply selects all the *overflow* partitions that go beyond a maximum size M, in this case, 128 MB. For example, in Figure 4.7(a), partition p_1, p_3 is selected for



(a) R*P and CBP Flushing (b) LSMP Flushing

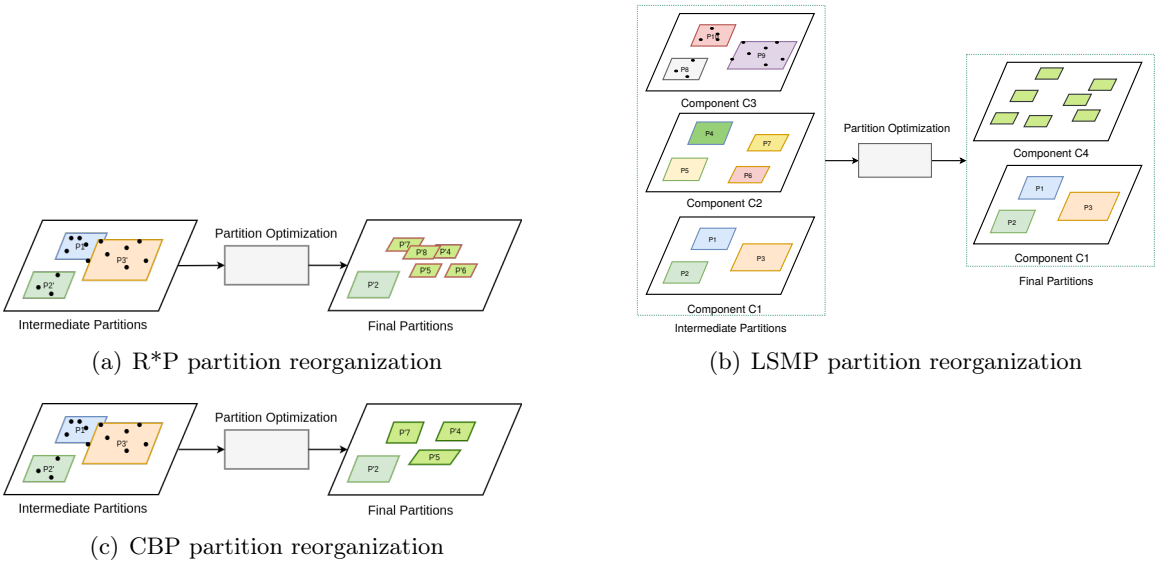
Figure 4.6: Data flushing in different partitions

reorganization because they went beyond the maximum partition size. Notice that we cannot implement the forced reinsert technique in R*-tree since random updates are not allowed in HDFS.

Partition Reorganization: This last step will reorganize the partitions selected by the previous step. In particular, it applies the R*-Grove [190] as the partitioning method for the selected partitions. Overall, R*P is not suitable for datasets which require a high rate of record deletion or update. In particular, R*P only splits partitions into smaller partitions, while there is no merge mechanism for partitions which contain deleted records. As a result, there might be many small partitions when the datasets are updated over time as shown in Figure 4.7(a).

4.5.2 LSM-tree-inspired partitioning (LSMP)

In the traditional LSM-Tree [150], each batch is flushed and indexed as a separate LSM component. An LSM compaction policy merges these components depending on their



(a) R*P partition reorganization

(b) LSMP partition reorganization

(c) CBP partition reorganization

Figure 4.7: Partition optimization in different partitioning techniques

sizes and order of creation. We adapt our generic partitioning framework to support an LSM variation where each component is indexed by an R*-Tree.

Data Flushing: In the *data flushing* phase, the new batch is partitioned as an R*-Tree, similar to [73]. Since the batch can be bigger than an HDFS block, it might consist of multiple partitions. Figure 4.6(b) shows a data flushing process (termed as LSMP Flushing) in which the new batch is partitioned into a new component C_3, besides the current components C_1, C_2 of our LSM partition. In an auxiliary *component file*, we store the component ID and creation order for each component. LSMP flushing requires the same disk IO as R*P Flushing, but it creates new partitions while the number of partitions does not change in R*P.

Partition Selection: This step starts by scanning the component's metadata, i.e., creation order and size, from the master file and the auxiliary component file. The standard LSM compaction policy from H-Base [100] is applied to the LSM components to identify

the components that need to be merged. If there is more than one component to merge, all partitions in those components would be selected.

Partition Reorganization: In this step, a new partition component is reconstructed from all the partitions in all the selected components. This results in replacing all these components with one component which is considered the new merged component. Figure 4.7(b) shows how components C_2 and C_3 are reconstructed into a new component C_4.

The advantage of the LSM-tree-inspired partitioning is that it contains highly optimized partition components. It also works well for the datasets with update workloads, since the merging process always produces the optimized partitions. Thus it will provide a good query performance if the query range completely falls into only one component. However its performance will be negatively affected with large query ranges, when they require scanning multiple components. In order to address this drawback, the number of components should be reduced by the compaction process, with a trade-off of reconstructing time for multiple components.

4.5.3 Cost-based partitioning (CBP)

In this approach, after each flush, we reorganize the partitions given a reorganization budget (we use a budget similar to the one spent by R*P in our experiments). The cost model in Section 4.4.2 is used to estimate the benefit of each candidate reorganzation.

Data Flushing: In general, the data flushing phase for CBP partitioning works in a same way with R*P, which was described in Section 4.5.1 and Figure 4.6(a).

Partition Selection: Based on Equation 4.11, we designed a greedy algorithm to select a group of partitions that will likely lead to high benefit. We start with an empty

set of selected partitions. Then, we scan the set of available partitions and choose the one that maximizes the benefit function if added to the selected group. We repeat this until the allocated budget B is used. Notice that once a partition is selected, the benefit of all other partitions change so the next iteration of the loop will have to recalculate all of them.

Algorithm 4 Greedy Partition Selection Algorithm

```
1:  function PartitionSelection(P = {p_1, ..., p_m}, B)
2:      G = {}                                          ▷ Set of selected partitions
3:      while nblocks(G) ≤ B do
4:          max-benefit = 0
5:          for each p_i ∈ P do
6:              b = max{B̂enefit(G ∪ {p_i}, s),
7:                   B̂enefit(G, s) + B̂enefit(p_i, s)}
8:              if b > max-benefit then
9:                  max-benefit = b
10:                 p* = p_i                            ▷ Update selected partition
11:             G = G ∪ {p*}
12:             P = P − {p*}
        return C
```

Algorithm 5 shows the pseudo code for the proposed partition selection algorithm. Line 2 initializes the set of selected partitions G to the empty set. Then, the loop in Lines 5-12 iterates over all the partitions in P to compute the benefit of each one. Line 7 calculates the benefit of each partition when added to the set of selected partitions. In particular, the benefit of adding a new partition is the maximum of the benefit it promises when it is reorganized together with existing partitions, or the benefit when it is reorganized individually. The partition that results in the maximum benefit increase is chosen (Line 10). The chosen partition $p*$ is then added to the set of selected partitions G and removed from the set of available partitions P. Notice that once a partition is added to G, the benefit of all other partitions change so the next iteration of the loop will have to recalculate all of them. Once

the total number of blocks in G is larger than the budget B the algorithm terminates and returns the set of selected partitions.

Partition Reorganization: We first split the selected partitions into groups, by adding all overlapping partitions in one group, and then partition each group independently. The reason that we reorganize the partitions in groups is to minimize the overlapping of reorganized partitions with existing partitions as shown in Figure 4.5. CBP might be able to create less reorganized partitions than R*P as shown in Figure 4.7(c) since it can merge under-utilized partitions into a single-block partition.

4.6 Experiments

In this section, we perform a comprehensive experimental evaluation to highlight the advantages of proposed work over existing spatial data management systems. For record-level approaches, we two state-of-the-art baselines, namely, AsterixDB [16] (0.9.4.1) and GeoMesa[106] (2.4.0). For block-level approaches, we use SpatialHadoop [73] but we port its partitioning algorithm to Spark for fair comparison and call it Beast. These three baselines are compared to the three techniques that are proposed in this chapter, namely, R*P, LSMP, and CBP. These systems are evaluated based on the ingestion time, partitioning quality, and query performance on the partitioned data. We use range query and spatial join as query workloads.

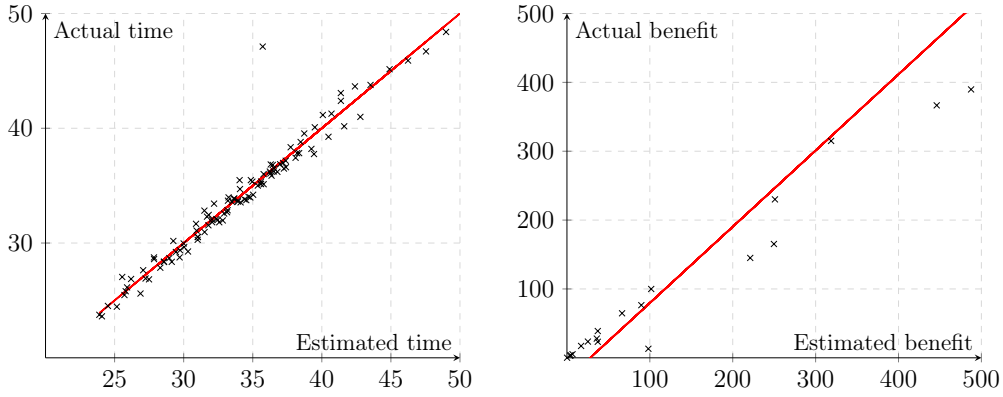
Datasets: We use the following spatial datasets in our experiments: (1) MS-Buildings dataset [140] with size 96GB. This dataset is synthesized from the original MS-Buildings dataset from UCR STAR, in which we extract the rectangles of building's geometries.

In order to increase dataset size, new records are created by shifting the rectangles of existing records. (2) OSM-Parks dataset [79] with size 8 GB (10 million points): a real dataset which represent all the parks on the world map.

Workloads: We evaluate different systems using ingestion/deletion and spatial analytical query workloads. For the ingestion, we split input datasets into batches and keep adding these batches to the comparing systems. There are two type of deletion workload: delete by batch, which deletes an entire batch that was inserted earlier, and delete by sample, which deletes a set of random records from all inserted records. The analytical queries include both range query and spatial join.

System specs: All experiments are executed on a cluster of one head node and 12 worker nodes, each having 12 cores, 64 GB of RAM, and a 10 TB HDD. They run CentOS 7 and Oracle Java 1.8.0_131. The cluster is equipped with Apache Spark 3.0.0. The source code is available as an open-source project [1].

The experimental evaluation is designed as follows. Section 4.6.1 validates the accuracy of the cost model by comparing the estimated and real cost of the range query. Section 4.6.2 compares the three proposed techniques, R*P, LSMP, and CBP, and shows that CBP outperforms that two other techniques. After that, Section 4.6.3 compares the best proposed technique, CBP, to the other three baselines, AsterixDB, GeoMesa, and Beast.



(a) Estimated/actual cost (b) Estimated/actual benefit

Figure 4.8: Cost model and benefit model validation

4.6.1 Cost model and benefit model validation

Cost model validation

The proposed cost model in Section 4.4 mainly estimates the number of accessed file system blocks and uses it as an estimated cost. Based on this model, given a range query with specific size, we can easily compute the estimated number of blocks processed by that query. In addition, we can also compute the actual number of blocks that the query accesses on the index as well as the total running time. If the model is accurate, we expect that the estimated cost (provided by the model) and the actual cost (either actual number of blocks or total time) will have a high correlation. We use MS-Buildings dataset in this experiment. For clarity, we show the results when the data is indexed using the CBP partitioning.

Figure 4.8(a) shows the relationship between the estimated cost and actual cost of range queries for various query sizes. For each query, we compute the *estimated* running time using Equation 4.5 and measure the *actual* running time as we process the query on the

[1]https://bitbucket.org/tvu032/beast-tv/src/inc-update/

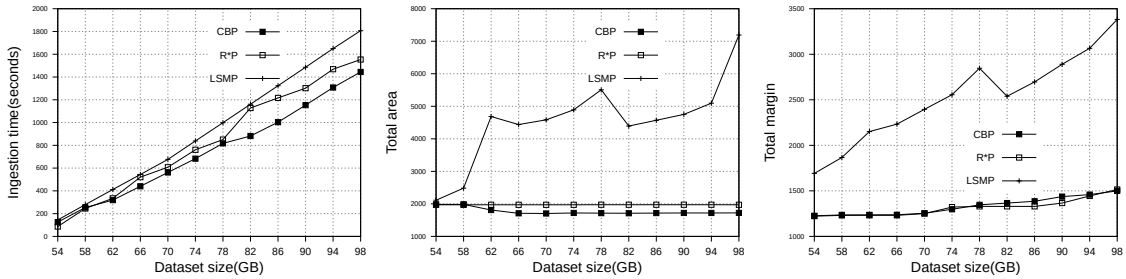
partitioned data. Then, we plot all those queries on a scatter plot with the estimated and actual running time on the x and y-axes, respectively. From the trend lines, we observe that there is a linear correlation between estimated cost and actual cost. This observation indicates that our proposed cost model is accurate in reality. The correlation between the two axes is nearly 94%. In other words, the estimated cost can be used in the partitioning optimization process in order to minimize the actual cost of range query processing. Nevertheless, the cost model still provides a linear correlation for all query sizes which makes it useful for a query of any size.

Reorganization benefit model validation

Figure 4.8(b) shows the the estimated benefit computed in Equation 4.11 and the actual benefit computed in Equation 4.9. The correlation between them is 97%, which indicates that the proposed estimated benefit is reliable to be integrated into our partition selection algorithm.

4.6.2 Performance of proposed partitioning algorithms

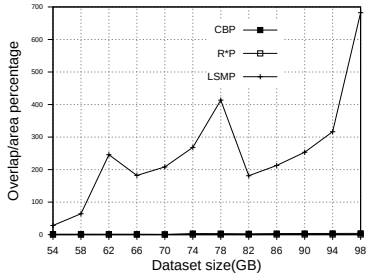
In this experiment, we compare the performance of proposed techniques in two workloads: intert-only and insert-delete. Figure 4.9 and 4.10 show the behavior of our proposed partitioning algorithms. We execute the experiment on MS-Buildings datasets with the comparison in ingestion performance and quality metrics. Figure 4.9(a) and 4.10(a) show the ingestion time for different partitioning techniques. We can observer that CBP outperforms R*P and LSMP in terms of ingestion time. By design, if the reorganization



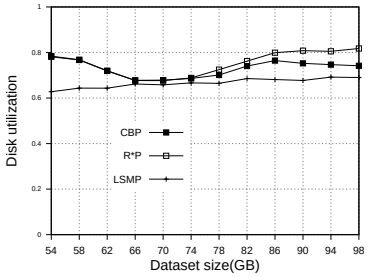
(a) Ingestion performance

(b) Total area

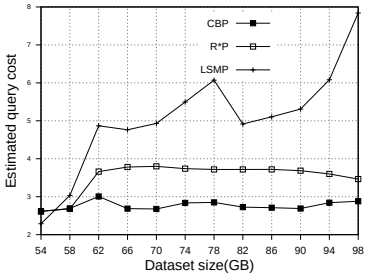
(c) Total margin



(d) Overlap percentage

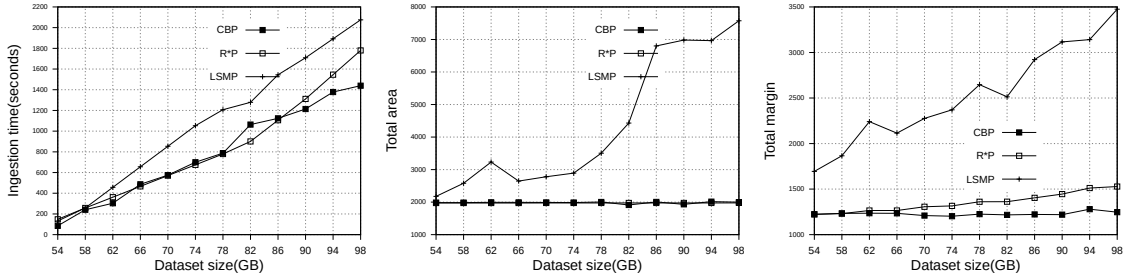


(e) Disk utilization



(f) Range query performance

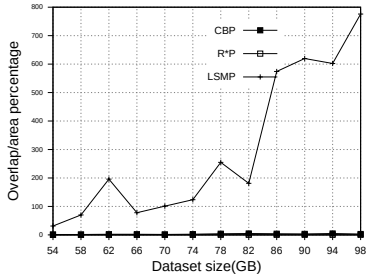
Figure 4.9: Performance of proposed implementations with insert-only workload



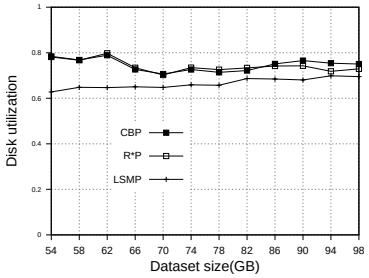
(a) Ingestion performance

(b) Total area

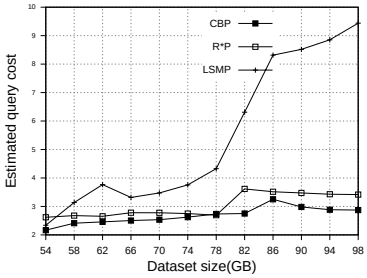
(c) Total margin



(d) Overlap percentage



(e) Disk utilization

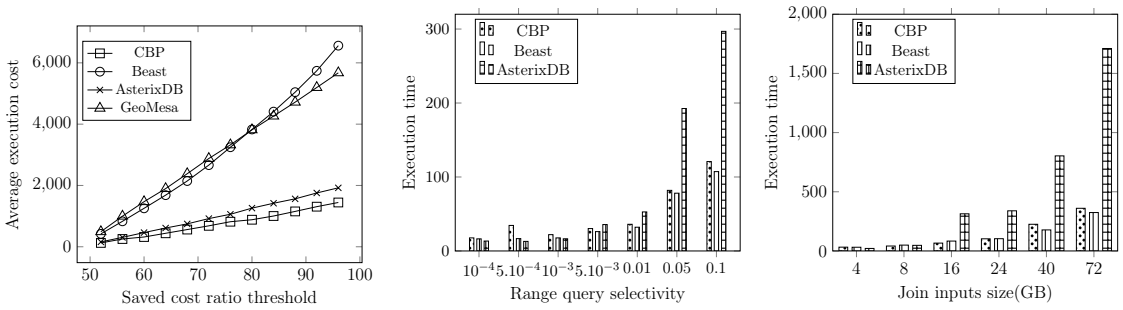


(f) Range query performance

Figure 4.10: Performance of proposed implementations with insert-delete workload

process is trigger, the total number of selected partition in R*P and LSMP is always greater than the selected partitions in CBP. R*P triggers the reorganize operation for all overflow partitions, which require a large number of partitioning jobs. LSMP partition from scratch all the components that meet the merge policy, thus it also requires more time for reorganize operation than CBP.

Figure 4.9(b) and 4.9(c) shows the total area and total margin of different partitioning techniques in the same set of OSM-Nodes batches. Since the CBP is optimized based on the estimated range query cost, it likely creates partitions with good quality. R*P keeps splitting partitions so the total area will increase larger and large when there are new batches coming. LSMP's total area is up and down due to the compaction process. The partitions will be optimized after the compaction, leads to a huge reduction of total area. However, this is not stable overall. Figure 4.9(d) shows that R*P is slightly better than CBP in total overlap area. This is expected since R*P split overflow partition separately, which almost do not introduce new overlapping area between overflow partitions. Figure 4.9(e) show that CBP have a better disk utilization that R*P and LSMP, because it reorganizes partitions in group of partitions. Overall, the partition quality of LSMP is not stable due to its merge policy. In particular, the partition quality will be good after a merge operations but getting worst when the new components are flushed. To choose the best among all proposed partitioning techniques, we use our range query cost model to evaluate which partitioning promise the lower number of processing block for range queries. Figure 4.9(f) indicates that CBP is the winner in terms of range query performance. Therefore, we will use CBP as the representative to compare with other state-of-the-art spatial partitioning systems.



(a) Ingestion performance (b) Range query performance (c) Spatial join performance

Figure 4.11: Performance comparison of CBP with existing techniques: AsterixDB, GeoMesa, Beast

4.6.3 Comparison with existing spatial data systems

In this experiment, we execute some basic analytical queries on the datasets which are partitioned by our proposed CBP and other existing techniques such as GeoMesa, AsterixDB and and Beast [2] - a Spark implementation of SpatialHadoop. GeoMesa is a geospatial database systems that is built on top of distributed databases, which is expected to work well with highly selective queries. Beast is a Spark version of SpatialHadoop, thus it will only work with static data. We choose Beast, GeoMesa and AsterixDB since they are remarkable representatives for record-level and block-level spatial partitioning techniques. To make the experiments being fair, we use HDFS as the storage layer for all systems, except AsterixDB. In addition, all the operations of CBP and Beast are built on Spark. We use XZ2-16bits [44] as the index scheme for spatial objects in GeoMesa. This configuration allows each partitioned file maintain a size of few KBs. We do not include spatial query performance since it is very slow when compared to other systems.

[2]https://bitbucket.org/eldawy/beast/

Update performance

Figure 4.11(a) shows the ingestion performance in different systems. We use MS-Buildings dataset with batches of 8GB. We measure the accumulated time to ingest these batches to the systems we are comparing. We observe that CBP outperforms other systems in terms of ingestion performance. This is expected due to following reason. First, Beast partition all the batches from scratch whenever a new batch is flushed. In the other place, CBP only reorganize a subset of partitions, which requires less partitioning effort. GeoMesa and AsterixDB are built on top of distributed databases, thus their partitioning process require more overhead steps, e.g. convert a multi-dimensional objects to one dimensional objects. There overhead make them working even worst then Beast. Since both GeoMesa and AsterixDB are categorized in a same group of record-level partitioning techniques, we only use GeoMesa as the representative in following experiments.

Range query performance

Figure 4.11(b) shows the performance of range query in different systems. In this experiments, we partition MS-Buildings dataset in all three mentioned systems. After that, we execute a batch of 10 range queries with same query size and measure the total running time in seconds. The query size varies from high to low selectivity with regard to the entire space area. We can observe that AsterixDB is better than CBP and Beast with highly selective queries. However, its performance significantly drops for large size queries. This cause by the index scheme of AsterixDB as we mentioned. Each partition is very small in terms of storage size. Therefore, it will process faster for a small query, when it only needs to

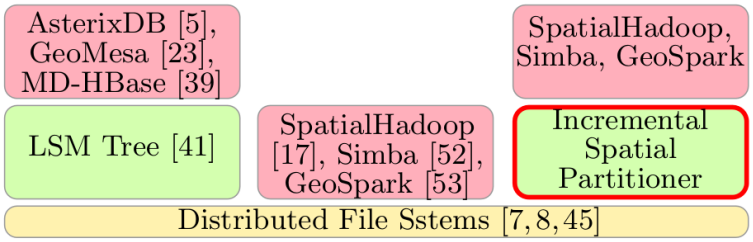
process few partitions. For the large queries, the number of processing partitions significantly increase, which leads to a poor performance. CBP and Beast got a reasonable performance for small queries and much better than AsterixDB and GeoMesa for large queries since they store partitions in block-level, which reduces the total processing jobs for range query.

Spatial join performance

In this experiment, we partition different batches of data which are extracted from MS-Buildings with size from 2GB to 64GB and OSM-Parks with size from 2GB to 8GB. We execute the spatial join algorithm which computes all the intersected pairs of two datasets. Figure 4.11(c) show the spatial join query performance in CBP, Beast and AsterixDB. We do not include the join time on GeoMesa since it is very slow, due to GeoMesa's design. In particular, we measure the total time to complete the query. We can easily observe that CBP and Beast outperforms AsterixDB in all join operations of large datasets, while AsterixDB is only the winner in the join of small datasets. The explanation is similar to the range query performance. Since AsterixDB partitions size is small, they will require more jobs to complete the same query when compared to block-level partitioning techniques.

4.7 Related Work

The existing work for supporting big spatial data can be broadly categorized into two categories, *record-level* systems and *block-level* systems, as depicted in Figures 4.12(a) and 4.12(b).



(a) Record-level with updates (b) Block-level with static data (c) **Proposed design**, block-level with incremental updates

Figure 4.12: Different approaches for accessing big spatial data

Record-level Systems: Systems in this category address the write limitation of DFS by utilizing the LSM-Tree [150] which converts random writes to sequential writes and merges. This idea was originally used to build key-value stores with a single-dimensional index, e.g., BigTable [48], HBase [100], and Accumulo [8]. To extend this idea to spatial data, some systems use space-filling-curves (SFC) to convert the high-dimensional coordinates to a single dimension, e.g., MD-HBase [147], MongoDB [141], and GeoMesa [106]. Alternatively, AsterixDB [1, 16] hash-partitions the records based on their IDs and then builds an R-tree index for each LSM component.

The drawback of this category is the huge overhead in processing the records since records have to be retrieved one-by-one through the index which makes this approach suitable for highly-selective queries that returns only a few records but does not scale for analytic queries that tend to process most or all the records.

Block-level Systems: Figure 4.12(b) indicates how the existing *block level approach* builds spatial partitions at the block level directly on top of a DFS. The work in this category is geared towards analytical queries that process huge amounts of data. Instead of organizing the data at a record level, these systems organize them in large blocks of

128 MB each. Examples include MapReduce-based systems like SpatialHadoop[73] and Hadoop-GIS[12] or Spark RDD systems like GeoSpark[204] and Simba[198], streaming-based systems like Tornado [136], and data warehousing systems like Sphinx [83].

The main limitation of all these systems is that they cannot incrementally maintain the data in a spatially partitioned way. Whenever new records are inserted or old records are deleted, the data has to be *fully repartitioned* to reach the best performance.

Proposed Work: Figure 4.12(c) shows our proposed work which introduces the incremental spatial partitioner into the block-level approach to enable incremental partitioning on DFS. In analogy with record-level systems, the proposed work plays the same role that the LSM-Tree did to enable record-level indexing on the limited DFS, i.e., it enables block-level systems to incrementally maintain spatial partitions of the data without the need to modify files in DFS.

4.8 Conclusion

In this chapter, we proposed a generic framework for incremental partitioning of big spatial data. We used this framework to implement three incremental spatial partitioning techniques to show its feasibility. Then, we provided a deeper study to the partition optimization problem and proved its NP-hardness. Based on this, we split it into two smaller problems, partition selection and partition reorganization. We then showed that the partition selection problem is crucial for the partition quality. To solve the partition selection problem, we proposed a new range query cost model for distributed indexes and used it to build an approximate greedy algorithm for the partition selection problem. Finally, we carried out

an extensive experimental evaluation using large scale real data to evaluate the efficiency of the proposed work. The experiments showed that the proposed partitioning technique can minimize the partition construction time while maintaining a high quality partitions. While here we focus on data ingestion, in future work we plan to enrich the partitions with update and delete operations to support more applications.

Chapter 5

Selectivity estimation with predicted error and response time

5.1 Introduction

Recently, there has been a notable increase in the amounts of spatial data collected by satellites, social networks, and autonomous vehicles. The main method that data scientists use to process this data is through *interactive exploratory queries*; i.e., an ad-hoc query that should be answered in a fraction of a second. Existing studies show that a response time of more than a few seconds to these queries would negatively impact the productivity of the users [128]. Unfortunately, existing big-spatial data systems [66, 202, 198, 24, 177], require way more than that to run even the simplest queries, hence, they cannot answer interactive exploratory queries.

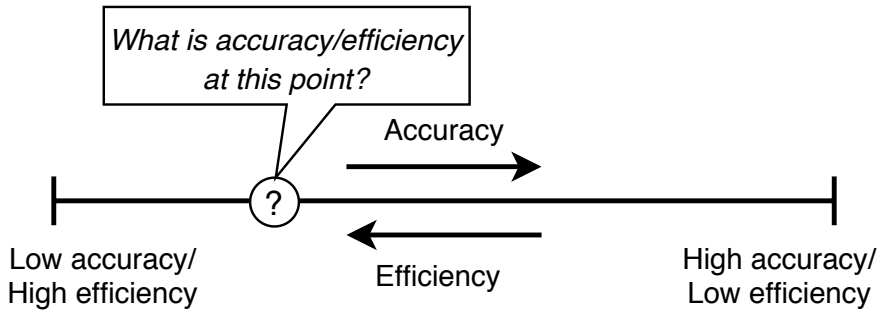


Figure 5.1: Trade-off between accuracy and efficiency in AQP

The most viable solution to the interactive exploration problem is approximate query processing (AQP) which uses a small data synopsis, e.g., a sample, to provide an approximate answer within a fraction of a second. This technique provides up-to three orders of magnitude speedup with a very high accuracy for several fundamental problems, including selectivity estimation, clustering, and spatial partitioning [172]. Figure 5.1 depicts the trade-off between the *accuracy* of the approximate answer and the *efficiency*, i.e., running time, which is highly correlated with the sample size. Unfortunately, this accuracy/efficiency trade-off is very hard to calculate which discourages many users from using AQP systems. Existing solutions either provide answers without any performance guarantee or make unrealistic assumptions such as uniform distribution or independence between dimensions [172, 49, 171, 170, 109, 5, 148, 11, 127, 157, 107]. This problem is particularly challenging due to the intertwined relationship between the sample size, query parameters, algorithm logic, data distribution, and result accuracy.

This chapter proposes DeepSampling, a novel deep learning based model to predict the relationship between accuracy and relative sample size for AQP. The main challenge is how to build a model that works well for any spatial data distribution and query parameters. To solve this problem, we build a deep neural network that takes as input the query parameters

and a histogram that represents the data distribution. This idea can work in two modes: *1) given a sample size, it estimates the expected accuracy, or 2) given a desired accuracy, it calculates the required sample size.* The idea is generic and can work with any approximate algorithm by building a separate model for each one. DeepSampling can be integrated into any existing spatial data system that supports AQP. To the best of our knowledge, DeepSampling is the first system that supports predictable error AQP for spatial data analysis problems. We run an experimental evaluation on both synthetic and real data on the selectivity estimation problem and the results show that the proposed method can accurately model the delicate relationship between accuracy and sample size and is portable to many distributions.

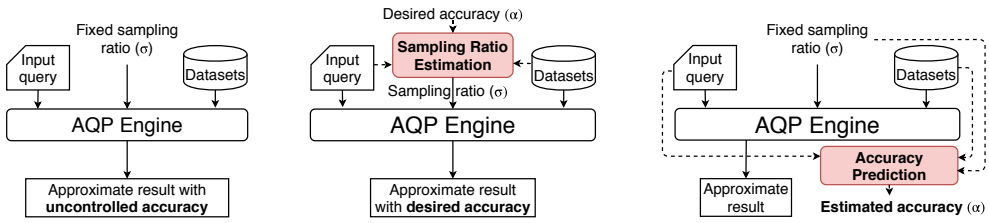
In summary, this chapter makes following contributions. (1) Design a deep learning model for predictable error and response time for AQP in spatial data analysis. (2) Apply this model to the **selectivity estimation** algorithm to solve two problems, sample size estimation and accuracy prediction. (3) Validate the model through experiments and publish the pre-trained model for wide use.

5.2 Related Work

Approximate query processing: AQP is a common method in many spatial data management systems. In AQP, the answer is estimated by executing the query on a small sample of the dataset, instead of scanning entire dataset. AQP is applied on several problems such as selectivity estimation, clustering, and spatial partitioning [172]. For example, SpatialHadoop [66], ScalaGiST [133], Simba [198], SATO [182] use a sample of the input

dataset to compute the minimum bounding rectangles (MBRs) for their spatial partitioning operation. Sampling is also used to cluster very large datasets [31, 205]. Specially, sampling is the fundamental method for many selectivity estimation algorithms for spatial data [10]. The main idea of AQP is the trade-offs between query response time and accuracy as shown in Figure 5.1. The common drawback of existing systems is the lack of a mechanism to choose a suitable sampling ratio to achieve a desired accuracy. For instance, SpatialHadoop just chooses a fixed 1% sample of dataset to compute partition MBRs, which is not always the best choice. DeepSampling addresses this challenge by suggesting the minimum sampling ratio such that the desired accuracy could be achieved. For non-spatial data, BlinkDB [11] provides a bounded errors for standard relational queries. However, BlinkDB assumes the independence of data dimensions, which is not applicable for spatial data.

Deep learning and spatial data: In recent years, the research community has witnessed the rapid growth of research projects in the intersection of big spatial data and machine learning [162]. One of the important research directions is scalable statistical inference systems for big spatial data analysis. For instance, TurboReg [163] is a scalable framework for building spatial logistic regression models. TurboReg is built on top of Markov Logic Network, which is able to predict the presence and absence of spatial phenomena in a geographical area with reasonable accuracy. DeepSPACE [184] is a deep learning-based approximate geospatial query processing engine. DeepSPACE utilize the learned data distribution to provide a quick response for spatial queries with reasonable accuracy. Both TurboReg and DeepSPACE hold the common drawback that they cannot guarantee a required precision of their answers.



(a) Existing AQP engine

(b) Sample Ratio Estimation

(c) Accuracy Prediction

Figure 5.2: DeepSampling addresses critical problems on existing AQP systems

DeepSampling aims to overcome this issue by providing a prediction model such that the required precision is always met with a reasonable of sampling ratio budget.

5.3 Selectivity Estimation with Predicted Error and Response Time

5.3.1 Problem definition

This chapter focuses on the prediction model for the selectivity estimation problem but the proposed approach can be easily generalized to other problems such as K-means clustering or spatial partitioning. The goal is to find the relationship between accuracy and sample size and toward this goal we define two problems, *accuracy prediction* and *sample size estimation* which are both defined in this section. First, we will define the accuracy of an approximate answer in the selectivity estimation (SE) problem.

theoremquery accuracy] In the SE problem, given an approximate answer π and a ground truth Π for query range Q, the accuracy of the approximate answer π is

$$acc(\pi, \Pi) = max(0, 1 - |\Pi - \pi|/\Pi) \tag{5.1}$$

Based on this definition, we define the following two problems:

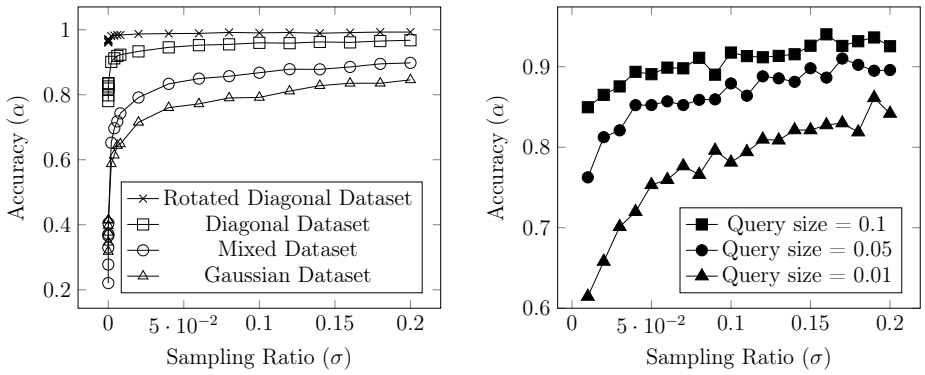
Problem 1 (Sampling Ratio Estimation): *Given a dataset D, a query range Q, and a desired accuracy α, predict the minimum value of sampling ratio σ such that $acc(\pi, \Pi) \geq \alpha$.*

Problem 2 (Accuracy Prediction): *Given a dataset D, a query range Q, and a sampling ratio σ, predict the accuracy α such that $|acc(\pi, \Pi) - \alpha|$ is minimized.*

Both problems are very important in approximate geospatial query processing. If we could address these problems, the existing spatial database systems could minimize the computation effort for sampling process while still achieving a desired accuracy for their answers. Figure 5.2 shows how DeepSampling enhances performance of existing approximate query processing systems. Instead of fixing a sampling ratio as Figure 5.2(a), an AQP engine can use Problem 1 to calculate a suggested minimal sampling size to achieve the used-desired accuracy as shown in Figure 5.2(b). Conversely, if the system has a fixed sampling ratio, it can apply Problem 2 to estimate the result accuracy as shown in Figure 5.2(c).

5.3.2 Prediction with mixed data sources

In general, we know that the accuracy (α) of an approximate answer increases with the sampling ratio (σ). However, we show in this part that this relationship is more



(a) Different distributions

(b) Different query sizes

Figure 5.3: How sampling ratio (σ) relates to accuracy (α)

complex than that. Figure 5.3 shows examples of how these two quantities are related to each other. First, Figure 5.3(a) shows that this relationship highly depends on the dataset distribution. While for all distributions the sampling ratio and accuracy are highly correlated, the relationship is different for each dataset. For example, for the rotated diagonal dataset, the accuracy ranges from 96% to 99% for all sampling ratios while for the mixed distribution dataset, the accuracy ranges from 22% to 90%. Second, Figure 5.3(b) shows the relationship for different query sizes. This time, we see that the relationship highly depends on the query size as well.

These observations show how challenging the problem is. To build an accurate model, we need to take into account the input data distribution and the query size. For other problems, the query size could be replaced with other query parameters, e.g., the number of clusters for the K-means clustering problem, or the number of partitions for the spatial partitioning problem.

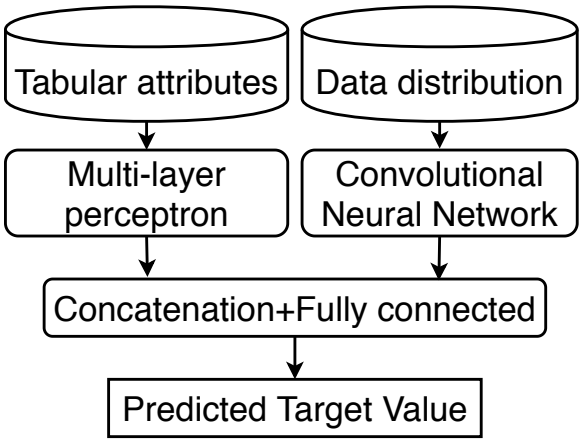


Figure 5.4: DeepSampling architecture

5.3.3 DeepSampling architecture

Figure 5.4 shows an overview of the proposed architecture of the DeepSampling model. This architecture is used to solve both problems described earlier, *sampling ratio estimation* and *accuracy prediction*. To avoid repetition, we write between (parentheses) the changes that need to be made for the *accuracy prediction* problem.

To build an accurate and portable model that accounts for the query size and the data distribution, the proposed model takes two sets of inputs, *tabular data* and *data distribution*.

The **tabular input layer** consists of data taken from the processing logs which includes the query size (q), the sampling ratio (σ), and the resulting accuracy (α). If we need to apply this architecture for other problems, then the query size will be replaced with other query parameters, e.g., number of clusters. Also, the accuracy will be calculated differently. This data is passed to a multi-layer perceptron (MLP) model. MLP is a feedforward neural network with at least three layers of nodes: an input layer, a hidden layer and an output

layer. We chose MLP for tabular input since it can be used to learn complex mathematical models by regression analysis [56].

The **data distribution input layer** catches the distribution of the input dataset. In this chapter, we use a uniform histogram which is expected to accurately catch the dataset distribution if computed at a reasonable resolution. The histogram resolution is a system parameter that we study in the experiments section. Since this histogram is a 2D matrix with spatial relationship between histogram bins, it is fed to a convolutional neural network (CNN) layer.

The **concatenation layer** combines the output of the MLP and CNN layers together and feed them to a fully connected (FC) layer. The final layer of FC is a single node with linear activation so that the model output is the predicted sampling ratio (or accuracy). The **loss function** of the final node provides a feedback on how accurate the predicted value is. Based on the problem definition in Section 5.3.1, we use mean absolute percentage error (MAPE) as the loss function which is the average absolute percentage error of actual value A_t and forecast value F_t for all training points $t \in [1, n]$ as shown in Equation 5.2. MAPE is commonly used in regression models since it is very intuitive interpretation for relative errors.

$$MAPE = \frac{1}{n} \sum_{t=1}^{n} \left| \frac{A_t - F_t}{A_t} \right| \tag{5.2}$$

Table 5.1: Parameters for the selectivity estimation (SE) query

Parameter	Values (Default)
Dataset distribution	Uniform, Gaussian, Diagonal, Sierpinski, Bit, Parcel, Mixed
Sampling ratio (σ)	0.001, 0.0015,...,0.2
Query size (q)	0.01,0.02,...,0.1.
Histogram size (h)	$1 \times 1 \ldots (16 \times 16) \ldots 64 \times 64$

5.4 Preliminary results

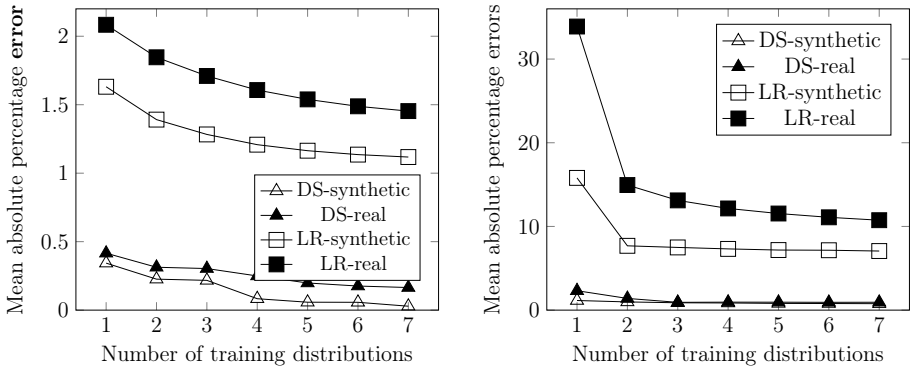
This section gives some preliminary results when applying the proposed approach to the selectivity estimation problem. In particular, we wanted to answer the following questions:

1. How accurately does the model account for the data distribution and query size?
2. Can the model solve both problems efficiently?
3. Is the model portable enough so that we can test it on a new data distribution that was not in the training set?

5.4.1 Experimental setup

We implement the proposed model in Figure 5.4 using Keras [51]. The source code, training data and models are available at [189].

Datasets: We use both synthetic and real datasets in our experiments. We generated a total of 144 synthetic datasets using the open-source spatial data generator [191]. The dataset distributions are listed in Table 5.1 and the detailed distribution parameters are included in the source code [189]. We also used two real datasets: `OSM-Nodes` [76] and `OSM-Lakes` [77]. The real datasets are only used for testing but never for training the model.



(a) Accuracy Prediction problem (b) Sampling Ratio Estimation problem

Figure 5.5: Accuracy of DeepSampling and linear regression

Parameters: In addition to the dataset distribution, we also vary the sampling ratio (σ), the query size (q), and the histogram size (h). The query size is the ratio between the area of the query rectangle and the area of the input minimum bounding rectangle (MBR). Our query workload consists of square queries centered at random locations in the input space. Table 5.1 summarizes all the parameters that we vary in our experiments. In total, our generated dataset contains $54,720$ data points.

Metrics: We use mean absolute percentage error (MAPE) to evaluate the accuracy of a prediction model. The lower the value of MAPE, the better the model is.

Baseline method: We compare the proposed model to a linear regression(LR) model which takes the tabular input and predict a numeric output. The reason behind this choice is that we want to see how the dataset distribution input makes a difference to the baseline which only takes query attributes into account.

5.4.2 Accuracy Prediction

In the first experiment, we build a model to predict the average query accuracy, given the sampling ratio, query size and dataset histogram of size 16×16. In particular, we use the synthetic datasets with $54,720$ data points described in Section 5.4.1 to train and test our proposed model. To observe how training data distribution affects the test accuracy, we organized the training data into different combinations of 1 to 7 distributions in Table 5.1. For each combination, we take a split of 75% data points for training process. We test all the trained models with 25% of the synthetic data points. We also test on 2800 data points that we collected from SE queries on real `OSM-Nodes` and `OSM-Lakes` dataset.

Figure 5.5(a) shows an interesting observation that the more data distributions we used for training process, the more accurate it is. This is expected since some simple distributions might not be able to capture important insights of test datasets. DeepSampling model is doing very well when we tested on both synthetic data and real data (MAPE is around 3% and 16%). This shows the portability of the model. Even though the model was trained only on synthetic data, it still provided good results for the real dataset. In the future, we plan to add more synthetic data to make the model even more accurate with real data. On the other hand, the linear regression baseline, due to its simplicity and the lack of data distribution, did not achieve a good accuracy. For the test on real dataset, its prediction is even more than 100% beyond the actual mean accuracy value.

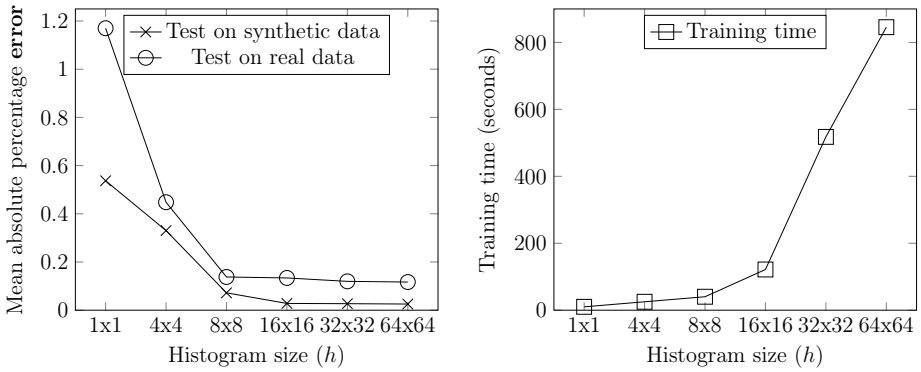
5.4.3 Sampling Ratio Estimation

In this experiment, we build a model based on DeepSampling to predict sampling ratio, given a desired query accuracy, query size and dataset histogram of size 16×16. We use the same set of training and testing split as mentioned in Section 5.4.2.

Table 5.5(b) shows that DeepSampling is still doing better than the baseline when applied on both synthetic and real data. The errors are relatively higher than the accuracy prediction problem in Section 5.4.2. The reason is that the range of the accuracy in the training set is narrow as compared to the range of sampling ratio. For example, in Figure 5.3, the accuracy in some cases stays above 95% while the sampling ratio ranges from 0.1% to 20%. Nonetheless, the DeepSampling approach is consistently more accurate than the linear regression baseline. These results are consistent with existing work that found that the sampling ratio estimation problem is more difficult. For example, in BlinkDB [11] this problem is solved by simply choosing from a predefined set of points, sampling ratio and accuracy, and interpolating between them if needed.

5.4.4 Effect of histogram resolution

To choose a good histogram size, this experiment studies the trade-off between the model accuracy and training time as we vary the histogram size as depicted in Figure 5.6. In this experiment, we vary the histogram resolution from 1×1 (effectively no histogram) to 64×64. Figure 5.6(a) shows the accuracy of the model when tested on both synthetic and real data as the histogram size increases. It is clear from this experiment that the histograms with higher resolutions carry more information that makes the model more accurate. However, the



(a) Effect on accuracy

(b) Effect on training time

Figure 5.6: Effect of histogram resolution

model stabilizes at 16×16 where the histogram is accurate enough to catch the distributions in the training set.

Figure 5.6(b) shows the total time of the training phase, i.e., the time until the model stabilizes. As expected, the model takes more time to train as the histogram resolution increases due to the large input that goes through the CNN model. From this experiment, we choose to set the histogram size to 16×16 which gives a good accuracy in a reasonable time.

5.5 Summary and future work

In this chapter, we introduced DeepSampling, a deep-learning-based system that provides predicted errors for approximate geospatial query processing. The proposed model combines the sampling ratio, the result accuracy, the query parameters, and the input data distribution. We carry some preliminary results when we apply DeepSampling to improve performance of selectivity estimation query. The results show that the proposed model can accurately compute the sampling ratio and accuracy for many synthetic and real distributions.

In the future, we will apply the same model on other important approximate spatial problems such as K-means clustering and spatial partitioning.

Chapter 6

A learned query optimizer for spatial join

6.1 Introduction

The rapid increase in the amount of big spatial data urged the research community to develop many systems to process this huge amount of spatial data, such as SpatialHadoop [63], GeoSpark [201], Hadoop-GIS [13], and many others [67, 146, 198, 132, 196, 136]. One of the most important and challenging operations in all these systems is *spatial join*, which combines multiple datasets together based on their spatial features.

Spatial join is one of the most resource demanding operations in spatial databases and it becomes even more challenging with big data [75, 161, 39]. Since big spatial data systems run in a distributed environment, the best algorithm has to balance the computation across machines, reduce disk access from the distributed file system, and minimize network

overhead. At the same time, the skewed distribution of the inputs and the hardware specification of the machines have to be taken into account. The complexity of the problem encourages the researchers to develop many spatial join algorithms for big data [108, 63, 201]. However, this creates a complex query optimization problem to choose the best spatial join algorithm given the input datasets and hardware resources.

Traditionally, this query optimization problem has been addressed using theoretical cost models [37, 87, 21, 90]. With the rise of big-data, some of these cost models have been ported to MapReduce and similar systems [161, 22]. Unfortunately, this work is limited due to some strong assumptions such as the uniformity of the input datasets or about the query processing engine, e.g., Hadoop MapReduce, which limit their use in practice.

The advancement of machine learning resulted in a new generation query optimizers that rely on data-driven models for database operations including join [200, 138, 120, 115, 139]. However, these models are limited to equi-join and cannot catch the complex logic of spatial join. Further, most of this work assumes that the same datasets are used for training and testing which limit the applicability of the produced models.

In this chapter, we propose the first learned query optimizer for distributed spatial join on big data. This cost model can be abstracted as a complex function that estimates a single value, e.g., selectivity or best algorithm, based on some descriptors for the input. The main challenge with this approach is how to build a model that balances *generality* and *practicality*. On one side, the model should be general so that it can be ported to different datasets, spatial join algorithms, and systems. On the other side, it needs to be practical

by providing a simple output that can be directly used by the query optimizer such as the estimated result size or the best algorithm to run.

To address the challenges above, we propose a machine learning based framework that consists of three levels. The *first level* builds a *cardinality estimation* model that learns the result size independently of the algorithm. The input features used in this level cover individual dataset attributes, e.g., size and some skewness measures, and other features based on the *convoluted histogram*, which catch the joint distribution of the two datasets that is important for spatial join. The *second level* combines the predicted result size with other dataset attributes to build a separate model for each spatial join algorithm that predicts the number of geometric comparison operations, which is an algorithm-specific but hardware-independent feature. Finally, the *third level* builds a classification model which is able to predict the best algorithm.

This proposed architecture gives system designers the flexibility of choosing the level that matches their application needs from the most general (level 1) to the most specific (level 3). Moreover, it allows us to train some of these models once and share them. For example, the cardinality estimation model in level 1 can be used regardless of the algorithm. Similarly, the models in level 2 are hardware-independent so they can be reused from one cluster to another.

We published our works as an open-source project on GitHub [1]. In particular, we implement the proposed models using *scikit-learn* [155] and train/test them on different datasets. To train the model, we use an open-source spatial data generator [191] to generate hundreds of datasets by varying the data distributions. In addition, we train on subsets

[1] https://github.com/tinvukhac/learned-spatial-join

of publicly available real data from UCR-Star [93]. Together, the synthetic and real data generate a rich training set with thousands of training points. The results show that the proposed model can estimate the cardinality of the spatial join with an error of as low as 8% when compared to the ground truth. It can also predict the number of MBR tests for different algorithms with a reasonable error. Finally, we tested the end-to-end framework in predicting the best algorithm in terms of running time.

In summary, we make the following contributions:

1. We design and implement the first machine learning based model which is able to predict the best algorithm for spatial join based on their running times.

2. We break down the time prediction model into separate models, which are optimized for specific join attributes. These models could be the building blocks of many other systems which are trying to address the spatial join cost estimation problem.

3. We carry extensive experiments to validate the advantages of the proposed model over the existing solutions.

The rest of this chapter is organized as follows. Section 6.2 discusses the related work. Section 6.3 describes the process of our proposed cost model. Section 6.4 details the training and test process including data generation and preparation. Section 6.5 gives the results of our experiments. Finally, Section 6.6 concludes the chapters and discusses future works.

Method	Non-spatial	Spatial
Problems	Selectivity estimation, join cost, and join ordering	Selectivity estimation and join cost estimation
Theoretical	[151, 124, 181, 85]	[22, 37, 87, 161, 21, 90]
ML	[200, 138, 120, 115, 139]	**This work**

Figure 6.1: Related work in join optimization

6.2 Related Work

Non-spatial join optimization: Due to its popularity and importance, there has been a large body of work for optimizing non-spatial equi-join. There are mainly three problems related to query optimization, *selectivity estimation*, *join cost estimation*, and *join ordering*. Traditionally, theoretical models were proposed to solve these three problems [124, 181, 85, 151]. One of the key challenges is the correlation between join attributes. With the rise of *machine learning*, it was utilized to build sophisticated data-driven models for selectivity estimation [200, 115], join cost estimation [120, 139], and join order enumeration [138]. These ML-based methods suffer from two limitations. First, they only work with equi-joins and do not support the complex logic of spatial join. Second, the model is limited to a small number of tables and can only work with these tables. For example, they model the input table using a one-hot vector that tells the model which table to work with.

Spatial join optimization: Due to its high cost, spatial join optimization also was rigorously studied through the two problems of selectivity estimation and join cost estimation. Effective formulas have been proposed for uniformly distributed datasets in [22],

but their extension to skewed distributions were not straightforward. For accurate selectivity estimation, a method was proposed that computes the correlation fractal dimension of the considered spatial datasets and applying a power law for self-join [37] and binary join [87]. However, these methods were limited to point datasets with distance join predicate.

To optimize distributed spatial join queries, a cost-based and rule-based query optimizer was proposed for MapReduce [161]. It breaks down the spatial join into two phases, partition and join, and decides which technique to use in each phase. This work has two limitations. First, the cost-based model requires eight parameters to be measured which catch the characteristics of the hardware, algorithms, and data. Second, it does not consider all join algorithms, e.g., block-nested loop join (BNLJ). A more detailed model was proposed in [39] which breaks down the cost into CPU, local and network I/O components. It overcomes the limitations of the earlier work as it uses only simple data statistics and supports more algorithms but it is limited to uniformly distributed data.

This chapter proposes the first ML-based model for spatial join for selectivity estimation and cost estimation. It differs from existing ML-based query optimizers as it supports spatial join and can apply to any input data that was not part of the training process. It also overcomes the limitations of existing theoretical spatial join models as it supports skewed data including real datasets and it works on well-defined data statistics that can be computed with a simple data scan.

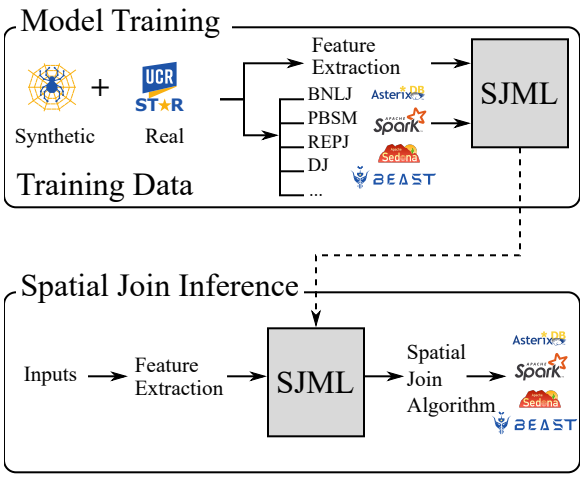


Figure 6.2: Overview of SJML

6.3 Overview of SJML

This section gives an overview of the proposed Spatial Join Machine Learning (SJML) tool illustrated in Figure 6.2. SJML is a supervised model that requires a training phase. Most existing ML-based query optimization models can only be applied to the same data distributions it was trained on. However, SJML breaks from this restriction as it can build a dataset-independent model that can be applied to any dataset. Thus, in the training phase, we use Spider [191], a standard spatial data generator, to generate various datasets with diverse characteristics. For each pair of generated datasets, we extract a set of features that the machine-learning model can train on. We also run all available algorithms on each pair of datasets to measure their behavior with such pair of datasets. This results in a training point that can be fed to the proposed SJML to learn how the existing algorithms behave for a specific pair of input datasets.

This generic model can easily extend to more spatial join algorithms or data processing systems. In this chapter, we focus on four fundamental spatial join algorithms,

namely, block nested-loop join (BNLJ), partition based spatial merge join (PBSM), distributed index-based join (DJ), and repartition join (REPJ). All these algorithms are implemented in both Hadoop and Spark.

After the model is built, we can use it to infer the best spatial join algorithm to run. Given a pair of datasets, we extract their features in the same way done in the training phase. We feed the features to SJML to choose the best algorithm to run.

Data Generation: To obtain a strong and portable model, we need to make sure that we generate diverse and representative data. We generate data using six different distributions and vary the data distribution parameters such as skewness and geometry size. We also create *compound* dataset that combine two or more simple datasets.

Feature Extraction: The most challenging step in the proposed model is the feature extraction step. We need to extract a set of features that are relatively easy to compute and are useful for the spatial join problem. We extract three sets of features. First, we collect statistical-based features such as the cardinality of the input and average geometry area. Second, we collect histogram-based features that represented data distributed and skewness such as box counting. We also introduce a *convoluted histogram* that combines the two datasets together to represent the relationship between the two datasets. Finally, we compute partitioning-based features that represent how the data is partitioned and indexed.

Model Training and Inference: We propose to train three models that work in concert to solve the problem. The first model estimates the size of the spatial join result which is a very important algorithm-independent metric for the cost of spatial join. The second model estimates the total number of rectangle tests (i.e., intersection tests between the MBR

of two geometries) in the algorithm which is an algorithm-specific but hardware-independent metric for the cost of spatial join. The third model combines these two metrics with other features to choose the best algorithm. The first two models are regression models while the third is a classification model.

6.4 Model training and testing

The training process starts by generating synthetic datasets of different distributions and sizes. We use an open-source spatial data generator that supports six different distributions as well as combinations of two or more distributions [191]. This is followed by a feature extraction step that extracts fixed-size features describing the input datasets and any indexes built on them. Finally, we prepare the master dataset that we use for training and testing each of the models that we propose in the chapter. The details of these steps are described below.

6.4.1 Training Data Generation

In this step, we produce the training data that will be used to build the SJML model. Our goal is to build a universal model that can work with any input so it is vital that the training data is diverse and representative of a large swath of distributions. At the same time, we need to make sure that the data generation process will not take too long. Thus, we can summarize our goals as follows:

1. Generate representative data that can train an accurate and universal model.

2. Reduce the overhead of generating the data.

3. Ability to train all the proposed models, i.e., join selectivity, MBR test selectivity, and fastest algorithm.

To generate representative data, we combine synthetic data from the public spatial data generation, Spider [191, 112], and publicly available real data from UCR-Star [93]. For synthetic data, we generate datasets of five different distributions provided by Spider, namely, uniform, parcel, bit, Gaussian, and diagonal. Additionally, we apply various transformations to the generated datasets such as translation, scaling, and rotation to ensure we cover more cases such as when the input datasets are not perfectly aligned. All transformations are represented using a single affine transformation matrix which allows us to control all these transformations and ensure we have diverse transformations. Moreover, we combine some of these generated datasets in pairs to produce complex distributed, e.g., diagonal+Gaussian. In total, we have 320 synthetic datasets. This allows us to produce more than 50,000 data points that represent all pairs.

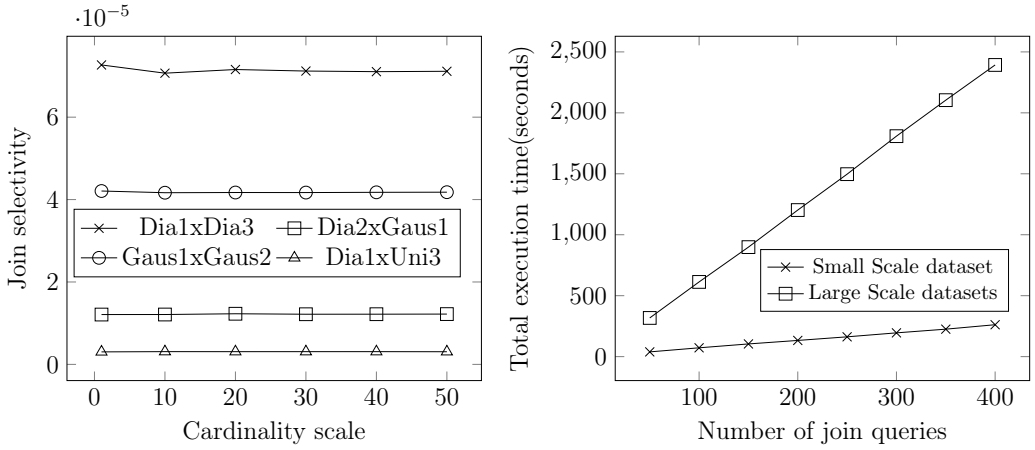
In addition to synthetic data, we also use real datasets from the public repository, UCR-Star. Since the number of real datasets is relatively limited and they are harder to retrieve and process, we generate arbitrarily many real datasets by extracting subsets of the real data with random search windows. This ensures that the distributions of the real data are diverse and representative of many real datasets. We chose the OSM buildings [78], lakes [77], and roads [80] datasets, and US Census linearwater [46], edges [45], and faces [47] datasets. In total, we have 95 real datasets that range from 134 MB to 7 GB.

To reduce the overhead of generating the data, we generate data at three scales: small, medium, and large. These are at the order of 1-2 MB, 10-20 MB, and 1-2 GB,

respectively. The key idea is that smaller datasets are easier to generate and process which allow us to generate thousands of training points in a very short time. However, small datasets might not hold a good representation of performance characteristics, which can only be tested with larger data. Thus, we generate the medium and larger datasets which can better catch performance behavior of spatial join algorithms as described shortly.

To be able to train all models, we divide the training data among the three models as follows. First, all the datasets are used to train the model that estimates the result selectivity since it is not affected by the data size. In other words, if the dataset gets bigger but all data characteristics remain the same, the selectivity will not change. To further clarify that to the reader, we use Spider to generate four groups of datasets. For each group, we fix the dataset characteristics and increase the data size from 1 MB to 50MB. For each dataset size, we run spatial join and measure the join selectivity as the result cardinality divided by the produce of the two input cardinalities. As shown in Figure 6.3(a), the selectivity of spatial join is almost a constant which confirms this point. At the same time, as Figure 6.3(b) shows, joining the 1MB dataset is about 10 times faster than joining the 50MB datasets. Hence, we can use this method to generate thousands of training points in a reasonable time without sacrificing the quality of the model.

Second, to train the model for MBR test selectivity, we use only medium and large datasets. While MBR selectivity might not be affected by cardinality, the way we partition the data is. Therefore, when using the medium dataset (scale of 10-20 MB), we adjust the partition size to be 128 KB. This results in a partitioning that has similar characteristics to partitioning 10-20 GB dataset with a partition size of 128 MB.



(a) Join selectivity

(b) Join execution time

Figure 6.3: Join selectivity and execution time in different cardinality scales

Third, to train the model for fastest algorithm detection, we use the large datasets which show the actual performance of the machines on big data. We use datasets that are big enough to utilize all executor nodes in the cluster. The goal is to avoid the case where only a few executor nodes are working since they do not represent the real performance of the cluster. This is similar to focusing on asymptotic behavior of algorithms when running on large enough input data.

6.4.2 Feature Extraction

The spatial datasets that we generate are not directly usable by the machine learning models that we propose, since they expect a fixed-size set of features. This step extracts these features for each pair of datasets D_i and D_j to prepare the data points for model training. The feature extraction process goes through the following steps.

First, we make one scan over the data to compute a set of statistics for each individual dataset D, detailed shortly. All these values are calculated based on *associative*

and *commutative* aggregate functions that can be computed efficiently in one scan. Some of them can also be obtained by querying the file system (i.e, HDFS in the common case).

Data Statistics

The following statistics are computed individually for each dataset D_* and they are not modified based on the join query, since they describe the dataset itself.

Definition 24 (Data size statistics - P_{Size})**.** *Given a dataset D_*, we define:*

- *#geo(D_*): number of geometries in D_* (i.e., number of records: $|D_*|$).*
- *size(D_*): size of the dataset in bytes. $size(D_*) = \sum_{r \in D_*} size(r)$, where $size(r)$ is the size of a record in bytes.*

Definition 25 (Data density statistics - P_{Dens})**.** *Given a dataset D_*, we define:*

- *`MBR`(D_*): Minimum Bounding Rectangle of D_*. `MBR`$(D_*) = (\min x,\ \min y,\ \max x,\ \max y)$.*
- *$mbrArea^{avg}(D_*)$: given the MBR of geometries r in D_* ($mbr(r)$), it represents the average area of such MBRs: $mbrArea^{avg}(D_*) = \frac{\sum_{r \in D_*} area(mbr(r))}{|D_*|}$*
- *$len_x^{avg}(D_*)$ and $len_y^{avg}(D_*)$: given the MBR of all geometries in a dataset, they represent the average length on the X and Y axis of such MBRs. $len_x^{avg}(D_*) = \frac{\sum_{r \in D_*} len_x(mbr(r))}{|D_*|}$. $len_y^{avg}(D_*)$ is calculated similarly.*

Second, after the above statistics are calculated, the histogram of the input is computed. We always create a high-resolution histogram of size 8192×8192. This histogram

is used to calculate in particular the parameters E_0 and E_2 using the box-counting technique as described in [33]. The histogram is also used to calculate the lower-resolution histograms $H(D_*)$ that will be used for the computation of the convoluted histogram.

Definition 26 (Histogram-based statistics - P_{Distr})**.** *Given a dataset D_*, its high-resolution histogram is generated and the following statistics about data distribution are computed:*

- *E_0 and E_2: these descriptors have been proposed in a previous work [34, 33] with the aim to briefly represent the distribution of the geometries of a dataset in the reference space. They can be easily computed also considering a sample of the dataset content. They derive from the application of the box-counting approach that is usually adopted for computing the fractal dimension of a set of points. Intuitively, E_0 describe the dispersion of the points, and it varies from 0 to the dimension of the embedding space (i.e., two for 2D datasets), thus if the points are clustered around a centroid then E_0 will be closed to zero, while a uniformly distributed set of points will produce a value for E_0 close to two. E_2 has a similar behaviour but is influenced also by the different concentration of the points.*

Since every dataset is partitioned either randomly or by applying a given indexing technique, the following additional parameters are assumed to be known. In particular, any partitioning approach is characterized by the construction of a grid through which geometries are redistributed among nodes. The main difference resides on the way the grid cells are built and their final shape. In case of a uniform grid index, all cells have the same shape and size.

Definition 27 (Partition statistics - P_{Part})**.** *Given a dataset D_* that is partitioned with some technique, the following parameters regarding its partitions can be defined:*

- *#cells(D_*): number of cells partitioning the dataset D_*.*

- *#splits(D_*): number of blocks containing records of the dataset D_*.*

- *Total area of D_*: sum of the area of all partitions.*

$$TT_{area}(D_*) = \sum_{c_i \in \mathcal{I}(D_*)} area(c_i) \times c_i.blocks$$

where $\mathcal{I}(D_*)$ is the set of cells used for partitioning the reference space of D_* and $c_i.blocks$ is the number of blocks stored in cell c_i.

- *Total margin of D_*: sum of the length of the semiperimeter of all partitions.*

$$TT_{margin}(D_*) = \sum_{c_i \in \mathcal{I}(D_*)} sp(c_i) \times c_i.blocks$$

where $sp(c_i)$ is the semiperimenter of the cell c_i, i.e., width + height.

- *Total overlapping of D_*: sum of the area of the overlapping regions produced by intersecting each cell c_i with all other cells.*

$$TT_{overlap}(D_*) = \sum_{c_i, c_j \in \mathcal{I}(D_*)} area(c_i \cap c_j) \times B(c_i, c_j)$$

$$+ \sum_{c_i \in \mathcal{I}(D_*)} \frac{c_i.blocks \times (c_i.blocks - 1)}{2}$$

where $B(c_i, c_j) = c_i.blocks \times c_j.blocks$

- *Load balancing: standard deviation of index cells cardinality*

$$LB(D_*) = \sqrt{\frac{\sum_{c_i \in \mathcal{I}(D_*)} (c_i.card - avgCard)^2}{|\mathcal{I}(D_*)|}}$$

where avgCard is the average cardinality of the cells belonging to the index $\mathcal{I}(D_*)$.

- *Block utility: average percentage of block usage:*

$$BU(D_*) = \frac{\sum_{b_i \in \mathcal{I}(D_*).blocks} (b_i.size/blockSize)}{|\mathcal{I}(D_*).blocks|}$$

These metrics can be easily calculated from the partition information which is typically very small and can be processed by a single machine. For example, for disk-based partitioned data, this information is stored in the master file [22].

Combined statistics

Since spatial join is a binary operation that takes two input datasets, we include additional *combined statistics* that catch the relationship between the two inputs. The following statistics are computed for each pair of datasets that are input of a join query. Notice that we do not need an additional scan over the data to compute these statistics since they are computed based on the summaries computed for each dataset individually.

Definition 28 (MBR overlapping statistics - P_{MBR})**.** *Given a pair of datasets* (D_i, D_j)*, the following statistics describing their overlapping are computed.*

- *Percentage of the area of D_i occupied by the area of the datasets intersections:*

$$Area_i = \frac{area(\texttt{MBR}(D_i) \cap \texttt{MBR}(D_j))}{area(\texttt{MBR}(D_i))}$$

- *Percentage of the area of D_j occupied by the area of the datasets intersections:*

$$Area_j = \frac{area(\texttt{MBR}(D_i) \cap \texttt{MBR}(D_j))}{area(\texttt{MBR}(D_j))}$$

- *Jaccard similarity:*

$$JS_{i,j} = \frac{area(\texttt{MBR}(D_i) \cap \texttt{MBR}(D_j))}{area(\texttt{MBR}(D_i) \cup \texttt{MBR}(D_j))}$$

In order to combine the information stored in the high-resolution histogram computed separately on the datasets D_i and D_j, we introduce the concept of *convoluted histogram*.

The convoluted histogram is simply the result of two histograms for the two datasets overlaid on top of each other. The goal is to give the neural network a picture of how the two datasets overlap with each other, which is extremely important for estimating the cost of the spatial join query.

To compute a *simple* histogram for one dataset, we overlay a uniform grid and count the number of records in each grid cell. If a record overlaps multiple cells, we count it only in the grid cell that contains its centroid to avoid overcounting. Finally, we normalize the histogram by dividing by the largest number to prepare it for machine learning processing.

To compute a *convoluted* histogram for two datasets, we need to apply the same grid for both of them. This ensures that their relative positions in space is taken into account

in the convoluted histogram. To do that, we define the grid based on the minimum bounding rectangle (MBR) of the *union* of the two datasets, i.e., the enlarged MBR that covers both datasets. Since the convoluted histogram requires knowledge of the two datasets before it is computed, it is not possible to precompute before the join query runs, e.g., as a part of data preprocessing and indexing. On the other hand, computing it on the fly when the join query comes would further delay the query and would defy the whole purpose of the spatial join query optimization problem.

To efficiently compute the convoluted histogram, we do it in two steps. The first step, which can be done offline, computes a simple histogram for each datasets based on its own MBR. Then, when the join query comes, we define the grid of the convoluted histogram based on the union MBR of the two datasets. After that, we *transform* the two simple histograms to the new grid to define the convoluted histogram as explained in [173]. The key idea of this transformation is to run a sort-merge-like algorithm that maps each bin in one of the two histograms to the corresponding bin in the convoluted histogram. In addition, it defines a multiplier ratio that is computed based on the amount of spatial overlap between the source and target bins in the two histograms. Note that this transformation is approximate and assumes that the data in each bin is uniformly distributed but it is sufficient for our purpose. Further details about this operation is explained in [173].

Definition 29 (Convoluted histogram-based statistics - P_{CH})**.** *Given a pair of datasets (D_i, D_j), the following statistics describe their mutual distribution.*

- E_0 *and* E_2 *of the convoluted histogram: these descriptors can be computed also on the convoluted histograms in the same way that we described before for the single histogram.*

In this case a value of E_0 close to two means that the two datasets are both uniformly distributed in all regions of the reference space, while a value close to zero means that there is only a small region where the datasets are overlapping and where join pairs can be generated.

This collection of parameters will be used for setting the machine learning model and, in particular, they will be used for generating the input vector of the model.

Learned metrics

The following metrics are extracted from the result of spatial join for each pair of datasets. We propose model that trains on these extracted metrics and estimates them during the inference step as further detailed in Sec.6.4.4).

Join selectivity: the first learned metric is *join selectivity* (L_{JS}), computed as:

$$L_{JS} = \frac{|D_i \bowtie D_j|}{|D_i| \cdot |D_j|}$$

i.e., the cardinality of the spatial join divided by the cardinality of the cross product of the two datasets. To estimate L_{JS}, we train a model starting from a subset of the parameters that we describe in the previous subsections. According previous work about the selectivity estimation of spatial join with uniformly distributed datasets [22], the join selectivity should depend only on the density parameters of each datasets, on the skewness of their distribution and on the overlapping of their MBRs. In the experiments presented in Sec. 6.5 we train a model for join selectivity estimation by feeding such model with different training sets and we tested this hypothesis.

MBR test selectivity: the second learned metric is the *MBR test selectivity* (L_{MS}), computed as:

$$L_{MS} = \frac{\#MBRtest(algo_i, D_i, D_j)}{|D_i| \cdot |D_j|}$$

i.e., the number of MBR tests that each specific join algorithm requires divided by the cardinality of the cross product of the two datasets. For the estimation of MBR test selectivity, we need to train four separate models, one for each algorithm, starting from a subset of the parameters that we describe in the previous subsections. In this case we need to test the impact of the parameters describing the dataset partitions $P_{Part}(D_*)$, and the convoluted histogram between the considered datasets $P_{CH}(D_i, D_j)$.

Join running time: the third metric is an estimation of the running time for a join algorithm in a specific cluster, given the join selectivity and MBR test selectivity.

Best join algorithm: the fourth determines which is the best algorithm in terms of join running time, given four available algorithms: BNLJ, PBSM, DJ and REPJ.

6.4.3 Training Set Preparation

In our system, we design the training data as tabular data. Each row of the training data includes a list of features and the output metric. The metric could be the join selectivity or the MBR test selectivity. In order to highlight the impact of the different features we define three possible training sets as shown in Table 6.1:

1. FS_s: the feature set that includes statistical-based features;

2. FS_{sh}: the feature set that includes FS_s and histogram-based features;

3. FS_{shp}: the feature set that includes FS_s, FS_{sh} and partitioning-based features;

The tabular data would be fed into an efficient regression or classification models, which will be discussed in Section 6.4.4.

2*Name	Features of D_i				Features of D_j				Intersection of MBRs	Convoluted histograms
	P_{Size}	P_{Dens}	P_{Part}	P_{Distr}	P_{Size}	P_{Dens}	P_{Part}	P_{Distr}	P_{MBR}	P_{CH}
FS_s	yes	yes	no	no	yes	yes	no	no	no	no
FS_{sh}	yes	yes	no	yes	yes	yes	no	yes	yes	no
FS_{shp}	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes

Table 6.1: Different possibilities of feature selection.

6.4.4 Definition of the models

In this part, we describe the models that we propose for the spatial join cost estimation problem. We show how we break down the complexity of the spatial join problem and algorithms by proposing an ensemble of models. In particular, we create four models that estimate four different metrics that all help in choosing the best algorithm. The four models estimate, (1) join selectivity, (2) MBR test selectivity per algorithm, (3) the best algorithm by running time, and (4) the running time of the chosen algorithm.

M1: Join selectivity estimation model

The first model is to estimate the join selectivity of the spatial join operation. There are three reasons we are considering the join selectivity for the first model. First, it is a algorithm- and machine-independent metric which makes it relatively easier to estimate. Second, join selectivity is a very important metric for spatial join performance that is

important to estimate by itself. Third, it is a metric that does not depend on the data size which allows us to use small datasets for training.

We build M1 as a regression model that takes as input the features of both inputs except for the size and the partitioning information they are irrelevant. The learned metric join selectivity is the defined as the result cardinality divided by the product of the input cardinalities. Since the join selectivity is independent of the join algorithm, we have only one instance of M1 that can be used for all join algorithms. To speed up the training process, we generate a large training set for small data and we run only one join algorithm to get the result size. We can easily parallelize this process on cluster nodes by letting each executor process one of the datasets.

M2: MBR test selectivity estimation model

We use MBR test selectivity L_{MS} as a stable and machine-independent metric for the performance of each algorithm. There are two reasons for choosing L_{MS} to estimate the performance. First, it is machine independent which makes it easy to port the learned model to a different system without any changes in the training. In addition, since it is system load independent, it is possible to generate the training set in parallel in the cluster without worrying about any conflicts. Second, it is independent of input size which allows us to train it on medium-size data.

We train four M2 models as one for each algorithm. The model is trained as a regression model and takes as input all input features except for input size which is irrelevant for this model. Unlike M1, this model takes the partitioning features into account since the partitioning of the input affects the performance. The training set for this model consists of

medium and large datasets. For medium data (10-20MB), we use a partition size of 128 KB to generate enough partitions to train this model. The resulting partitions are equivalent to partitioning a 10-20 GB dataset with a 128 MB partition size.

M3: Algorithm Selection

The goal of this model is to choose one of the algorithms to run for a specific pair of inputs. We design M3 as a classification model that produces one of four labels for the four algorithms. Since the choice of algorithms is highly dependent on the input size, this model is only trained on large datasets with several gigabytes in size.

M4: Running Time Estimation

This last model estimates the end-to-end running time of a specific algorithm. This model is not necessary for the execution of SJML, but it is added to make the system more user friendly by providing an estimated running time.

Since the running time is hardware dependent, it is not easy to create a universal M4 model for all machines. Therefore, we build a simple regression model for each algorithm that is trained and applied on specific hardware settings. It takes as input the result cardinality and MBR test cardinality and produces an estimated running time. Due to its simplicity, this algorithm can be trained on a very small training set which makes it easy to train when the algorithm is ported to a different hardware. However, we do not expect this model to be accurate enough to rank the algorithms but can still be helpful for producing a rough estimated running time.

6.5 Experiments

This section provides an experimental evaluation to measure the feasibility and accuracy of the proposed models. The experiments are designed to answer the following research questions:

1. Can machine learning models outperform hand-crafted theoretical models ?
2. Do the proposed features catch all aspects of spatial join?
3. Is the proposed model generic enough to be applied on a dataset that was not in the training set?
4. How can the model accurately choose a spatial join algorithm to run?

6.5.1 Experimental Setup

We run the spatial join algorithms on a Spark 3.0 cluster of one master node with 128GB RAM and 2×8-core Intel Xeon CPU E5-2609 v4 @1.7GHz, and 12 executor nodes each with 2×6-core Intel Xeon E5-2603 @1.7GHz and 10TB HDD.

The synthetic data is generated using the open-source Spider generator [191, 112]. The real data is downloaded from UCR-Star [93]. We chose the OSM buildings [78], lakes [77], and roads [80] datasets, and US Census linearwater [46], edges [45], and faces [47] datasets.

As a baseline, we use the theoretical models JS [108], SM [161], and BME [39]. To measure the accuracy of join selectivity model M1 and MBR test selectivity model M2, we use mean absolute error (MAE) and mean absolute percentage error (MAPE) which are calculated as following.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - x_i|$$

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - x_i|}{x_i}$$

where y_i is the estimated value and x_i the true value.

For the algorithm selection model, we use the accuracy metric to evaluate how good the classifier can choose the best algorithm. In some cases, the running time between the predicted algorithm and the actual algorithm is not too far. Thus we also use MAPE value to measure how bad that difference is. A lower value of MAPE indicates a better selection model.

6.5.2 Baseline methods

In order to compare the accuracy of the proposed model with previous work and show the improvements of the proposed approach, we define the following baseline methods.

- *Baseline B1 for the join selectivity model M1:* we consider the well-known spatial join selectivity estimation formula first proposed in [22] and also used in the theoretical model for spatial join algorithm ranking proposed in [39]. This formula is defined for uniformly distributed datasets. In order to evaluate the accuracy of the theoretical model in predicting the join selectivity we report in Table 6.2 the measured accuracy for some groups of datasets. Notice that, while in the uniform distributed datasets the behaviour of the theoretical model is good (column $MAPE_{JS}$), when skewed and non overlapping datasets are considered the accuracy worsts quickly.

- *Baseline B2 for the MBR test selectivity model M2:* we consider the estimation of the theoretical model for ranking spatial join algorithms proposed in [39]. Notice that the theoretical model has a different goal, i.e. to predict the ranking of the spatial algorithms and not the number of MBR tests performed by them. The number of MBR test was used in [39] as an intermediate estimate to evaluate the relative position of the algorithms in the ranking, thus it is very imprecise when compared to the real values computed in the experiments. However, to the best of our knowledge there is no other approach in literature that try to estimate this parameter.

- *Baseline for algorithm selection model M3*: we compare our work with the cost-based spatial join selection model that was proposed in [161] as the baseline $B3_1$. In addition, we consider the estimation of the theoretical model for ranking spatial join algorithms proposed in [39] as baseline $B3_2$. The theoretical model works with uniformly distributed datasets, thus, as shown in Table 6.2 (column Acc_{Rank}), the accuracy in predicting at the top of the ranking the first or the second best algorithm decreases quickly considering non uniform datasets.

6.5.3 Feature Selection

This experiment shows how the selected features affect the accuracy of the proposed models. In general, the more meaningful features will create the more powerful models. In each experiment, we compare three variations of proposed models M1, M2 and M3 with

Size	Distribution	MBR matching	$MAPE_{JS}$	Acc_{Rank}
Small	Uniform	>= 90%	0.021	21.3cm0.72
Medium	Uniform	>= 90%	0.11	
Large	Uniform	>= 90%	0.27	(0.41 1^{st})
Small	Skewed	>= 90%	0.36	2*0.64
Medium	Skewed	>= 90%	0.34	
Large	Skewed	>= 90%	0.33	(0.33 1^{st})
Small	Skewed	< 90%	0.52	3*0.57
Medium	Skewed	< 90%	1.15	
Large	Skewed	< 90%	$\gg 2$	(0.28 1^{st})

Table 6.2: Accuracy of the theoretical model in predicting the join selectivity and the best two algorithms in ranking.

Feature Set	Model	MAPE	MAE
FS_s	B1	0.41	2.84E-05
FS_s	M1	0.2312458636	1.06E-05
FS_{sh}	M1	0.04492620381	2.36E-06

Table 6.3: Effect of feature selection to join selectivity estimation models

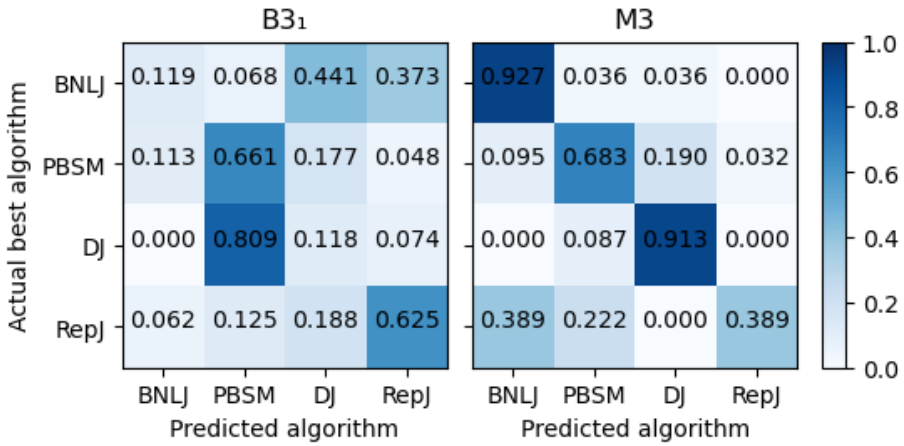


Figure 6.4: The confusion matrix of the algorithm selection model

		BNLJ		PBSM		DJ		RepJ	
Feature Set	Model	MAPE	MAE	MAPE	MAE	MAPE	MAE	MAPE	MAE
FS_s	B2	5.18	1.13E-02	13.32	7.18E-04	3.86	4.36E-04	1.83	7.82E-04
FS_s	M2	0.152	6.64E-04	0.012	3.99E-05	0.068	11.09E-05	0.075	8.53E-05
FS_{sh}	M2	0.0179	6.13E-05	0.014	4.48E-05	0.055	8.25E-05	0.039	4.17E-05
FS_{shp}	M2	0.018	6.22E-05	0.014	4.38E-05	0.044	6.95E-05	0.033	3.45E-05

Table 6.4: Effect of feature selection to MBR selectivity estimation models

Feature Set	Model	Accuracy	MAPE
FS_{shp}	$B3_1$ [161]	0.337	1.228
FS_s	$B3_2$ [39]	0.322	0.805
FS_s	M3	0.712	0.139
FS_{sh}	M3	0.8	0.074
FS_{shp}	M3	0.8	0.077

Table 6.5: Effect of feature selection to algorithm selection models

the three feature sets FS_s, FS_{sh}, and FS_{shp}. The tabular data is collected from 7140 join queries on synthetic datasets with different data distributions.

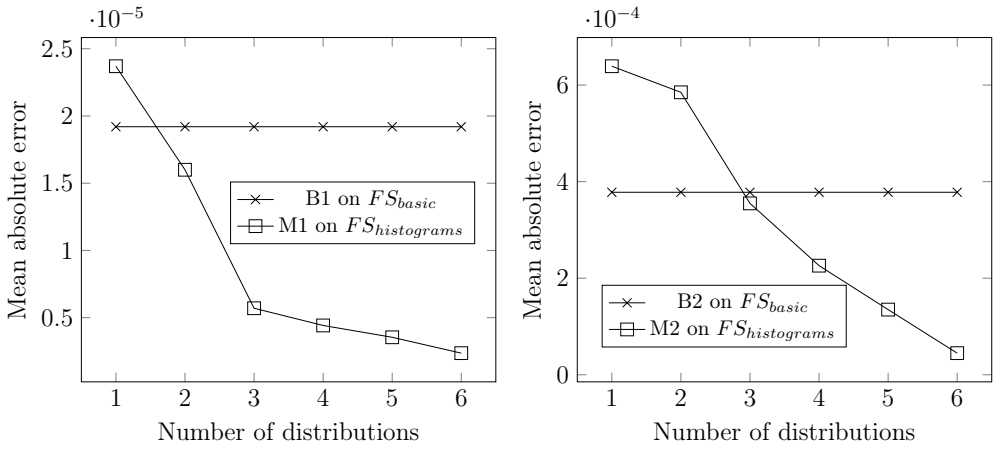
Table 6.3 shows the evaluation of join selectivity model (M1) and the baseline (B1) when using features sets FS_s and FS_{sh}. We do not use FS_{shp} since the partitioning information is not relevant to the join selectivity. First, we evaluate the performance of baseline method B1 from a previous work [22] on the feature set FS_s. Since B1 is designed to work with the join of uniform dataset, its MAPE value is pretty high. In addition, B1 is not applicable for other feature set, which might limits us to add more meaningful features to estimate the join selectivity. Given the same feature set FS_s, a random forest regression model M1 is almost twice better than the baseline B1 in terms of MAPE value. Finally, if we feed FS_{sh} to M1, the MAPE value (4.49%) is significantly reduced when compared to other models. This low MAPE value indicates that a regression random forest model with informative features can efficiently predict the join selectivity of a join query with low error.

Table 6.4 shows the efficiency of MBR selectivity (M2) and the baseline (B2) when using features sets FS_s, FS_{sh}, and FS_{shp}. Overall, the baseline model B2 can not work well with this problem, while its MAPE and MAE value is almost unacceptable. In contrast, the random forest regression model M2 can achieve very good values in terms of both MAPE and MAE values. We can also observe the downtrend of these metrics when there are more meaningful features fed into the model. This observation consolidate our motivation to figure out high impact features of join algorithms to build up our models.

Table 6.5 shows the accuracy of algorithm selection (M3) and the baselines ($B3_1$ and $B3_2$) when using different features sets. We use two metrics to evaluate the efficiency of these algorithm selection models. The accuracy measures the performance of the prediction of the best algorithm. The MAPE value shows how far the difference between the running time of predicted algorithm and the actual best algorithm is. The results show that $B3_1$ and $B3_2$ might not be able to predict the best algorithm well, when their accuracy is too low, and the MAPE value is unreasonable. In the other place, the random forest classifier M3 can provide up to 80% accuracy and 7.7% MAPE value. In addtion, Figure 6.4 show the confusion matrices of the baseline B3 and the proposed M3, respectively. This is a reasonable performance such that we can integrate M3 into existing spatial database systems.

6.5.4 Training Set Generation

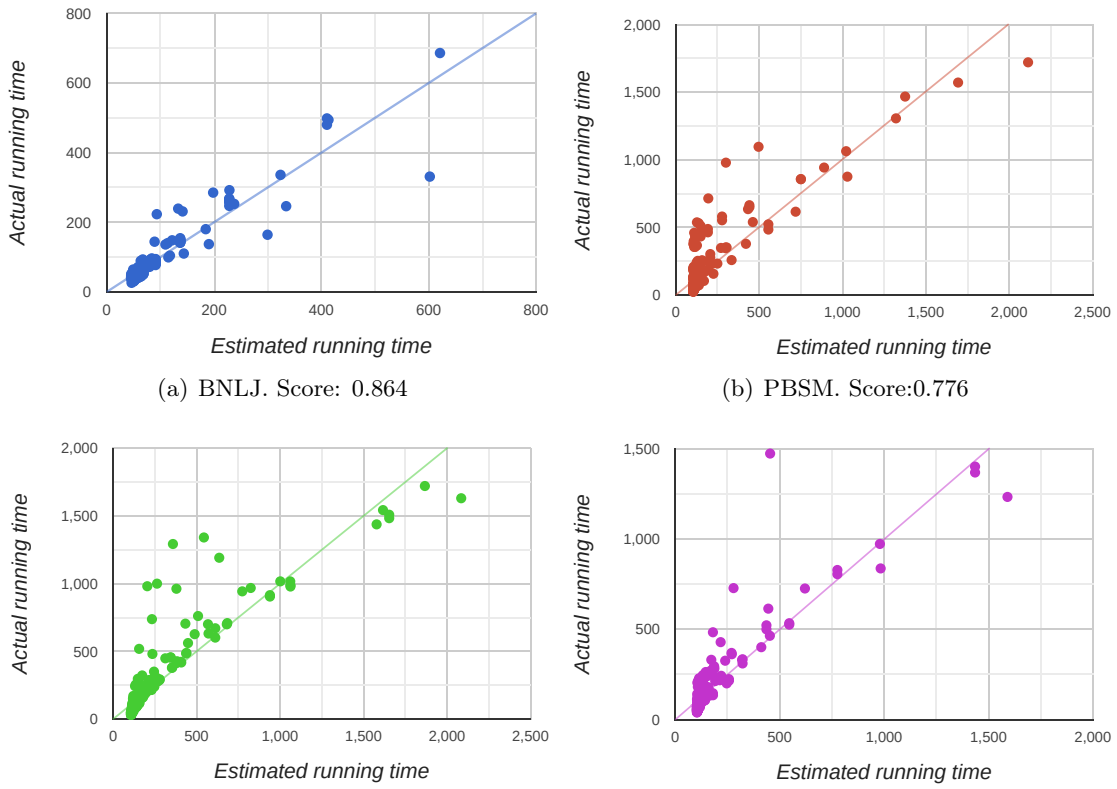
This experiment shows the effect of the training set on the accuracy of the models. Our goal is to show the effect of the number of distributions in the training set on the accuracy of the model. To accomplish that, we gradually increase the number of synthetic



(a) Join selectivity estimation model (b) MBR tests selectivity estimation model

Figure 6.5: Effect of training distributions in M1 and M2 model

distributions in the training set, from 1 to 5, and measure the mean absolute error of join selectivity model M1 and MBR tests selectivity model M2. For fairness, when we limit the number of distributions, we try all combinations and take the average. For example, for two distributions, we try $\binom{5}{2} = 10$ combinations and report their average. Finally, we add the data with real dataset and consider total number of distributions is 6. Figure 6.5 shows the efficiency of the baseline model B2 and the random forest regression model M3 when the number of distributions is varying, while the test dataset is fixed. Since there is no training process for B2, its MAE value is determined. In contrast, the MAE value for M3 is decreasing when the number of distributions is increasing. This behavior indicates that adding more distribution to the training data can help to improve the model's performance. Moreover, there is small difference of MAE value between the training data with all 5 distributions and the training data with real dataset's join results. This small gap shows that we can totally train our models on synthetic data, and still estimate the join algorithm's cost efficiently.



(a) BNLJ. Score: 0.864

(b) PBSM. Score:0.776

(c) DJ. Score:0.819

(d) RepJ. Score:0.804

Figure 6.6: The impact of join result size and number of MBR tests to spatial join cost

6.5.5 Spatial join cost estimation model

This experiment aims to show the high impact of join result size and number of selectivity to the running time of a join algorithm. Given the same join datasets, the number of MBR tests are still different for each join algorithm, due to the difference of algorithm's strategy. Therefore, we build separate linear regression models to estimate the running time for all four join algorithms. Table 6.6 shows that the regression score of a these models are pretty good. In addition, we plot the estimated running time with predicted running time for the different linear regression model of different join algorithms in Figure 6.6. This

Algorithm	Regression Score
BNLJ	0.8639376721
PBSM	0.7758226125
DJ	0.8189442368
RepJ	0.8044394821

Table 6.6: Regression score of the linear model between join result, number of MBR tests and join time

observation indicates that if we can build good models to predict the join result size and number of MBR tests for each join algorithms, we can utilize those models to build another model that predict the actual running time of a spatial join algorithm. This motivated us to proposed the different models M1, M2 and M3 in Section 6.4.4.

6.6 Conclusion

This chapter presented a spatial join cost estimation model based on deep learning. It is able to choose the best distributed spatial join algorithm given two input datasets. It breaks down the cost estimation into three modules. The first module estimates the cardinality of the spatial join result which is an algorithm independent metric. Second, it estimates the number of MBR tests done by each algorithm which is a machine-independent but algorithm-specific metric of performance. Finally, these two metrics are fed together into a classification model that estimates the best algorithm. To train this model, we used a synthetic data generator that generates thousands of datasets with various distributions to train the model on the different aspects of spatial data. We also showed that training on small datasets would be orders of magnitude faster and still produce accurate models. Our experimental evaluation shows the effectiveness of the proposed method in estimating, not

only the cost of the spatial join, but also the output cardinality which can be useful by itself. In the future, we plan to further extend this model by taking into account the hardware specification to estimate a more accurate result.

Chapter 7

Conclusions

The dissertation showed the published and on-going works which solve the fundamental problems in spatial databases: spatial partitioning, spatial query cost estimation. The proposed works could be integrated to existing spatial databases to improve the query optimization process.

There are several directions to go further with the research in this dissertation. First, the idea of DeepSamling could be applied for a wide scope of spatial queries: kNN, k-means clustering, histogram, etc. Second, we can build a generic framework for query optimizer, which provide the following features: (1) Feature selection component: decide which features to be used in DL/ML models. (2) Algorithm selection component: decide which technique to be used to get the best outcome. (3) Generic cost estimation models: build separate models for each kind of operator (JOIN, SELECT, DATASCAN, PROJECT, etc) in a query plan. Then they might be combined to generate a cost model for complex query plans.

Bibliography

[1] Apache AsterixDB. `https://asterixdb.apache.org`.

[2] Spatial Data Generators. `https://github.com/tinvukhac/spatialdatagenerators`.

[3] SpiderWeb: Web-based Spatial Data Generator. `https://spider.cs.ucr.edu/`.

[4] A. Aboulnaga and J. F. Naughton. Accurate estimation of the cost of spatial selections. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, pages 123–134, San Diego, CA, USA, USA, 2000. IEEE, IEEE.

[5] Ashraf Aboulnaga and Jeffrey F. Naughton. Accurate Estimation of the Cost of Spatial Selections. In *ICDE*, pages 123–134, San Diego, CA, February 2000. IEEE.

[6] Patricia S. Abril and Robert Plant. The patent holder's dilemma: Buy, sell, or troll? *Communications of the ACM*, 50(1):36–44, January 2007.

[7] Ildar Absalyamov, Michael J. Carey, and Vassilis J. Tsotras. Lightweight cardinality estimation in lsm-based systems. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018.

[8] Apache Accumulo. `https://accumulo.apache.org/`.

[9] Swarup Acharya, Viswanath Poosala, and Sridhar Ramaswamy. Selectivity estimation in spatial databases. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 13–24, 1999.

[10] Swarup Acharya, Viswanath Poosala, and Sridhar Ramaswamy. Selectivity estimation in spatial databases. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 13–24, "", 1999. ACM.

[11] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42, "", 2013. ACM.

[12] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, 2013.

[13] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *PVLDB*, 2013.

[14] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. Learning-based query performance modeling and prediction. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 390–401. IEEE Computer Society, 2012.

[15] Openstreetmap all objects dataset, 2019. http://star.cs.ucr.edu/dataset=OSM2015/all_objects.

[16] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. Asterixdb: A scalable, open source bdms. *Proceedings of the VLDB Endowment*, 7(14):1905–1916, 2014.

[17] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J Carey, Markus Dreseler, and Chen Li. Storage Management in AsterixDB. *PVLDB*, 7(10):841–852, 2014.

[18] Ahmed M. Aly et al. AQWA: adaptive query-workload-aware partitioning of big spatial data. *PVLDB*, 8(13):2062–2073, 2015.

[19] Ahmed M. Aly, Ahmed R. Mahmood, Mohamed S. Hassan, Walid G. Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. AQWA: Adaptive query workload aware partitioning of big spatial data. *Proc. VLDB Endow.*, 8(13):2062–2073, 2015.

[20] Amazon S3. `https://aws.amazon.com/s3/`.

[21] Ning An, Zhen-Yu Yang, and Anand Sivasubramaniam. Selectivity estimation for spatial joins. In Dimitrios Georgakopoulos and Alexander Buchmann, editors, *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 368–375. IEEE Computer Society, 2001.

[22] W. Aref and H. Samet. A cost model for query optimization using R-Trees. In *Proceedings of the Second ACM Workshop on Advances in Geographic Information Systems*, pages 60–67, 1994.

[23] Microsoft Azure. `https://azure.microsoft.com`.

[24] Furqan Baig et al. SparkGIS: Resource Aware Efficient In-Memory Spatial Query Processing. In *SIGSPATIAL*, pages 28:1–28:10, Redondo Beach, CA, November 2017. ACM.

[25] Michael F Barnsley and Alan D Sloan. A better way to compress images. *Byte*, 13(1):215–223, 1988.

[26] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990.*, pages 322–331, 1990.

[27] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 322–331, 1990.

[28] Norbert Beckmann and Bernhard Seeger. A benchmark for multidimensional index structures, 2008.

[29] Norbert Beckmann and Bernhard Seeger. A revised r*-tree in comparison with related index structures. In *SIGMOD*, pages 799–812, Providence, RI, June 2009.

[30] Norbert Beckmann and Bernhard Seeger. A Benchmark for Multi-dimensional Index Structures. 2019. https://www.mathematik.uni-marburg.de/ rstar/benchmark/distributions.pdf.

[31] Jeremy Bejarano, Koushiki Bose, Tyler Brannan, Anita Thomas, Kofi Adragni, Nagaraj K Neerchal, and George Ostrouchov. Sampling within k-means algorithm to cluster large datasets. *UMBC Student Collection*, 1(1):1–1, 2011.

[32] A. Belussi, D. Carra, S. Migliorini, M. Negri, and G. Pelagatti. What makes spatial data big? A discussion on how to partition spatial data. In *10th International Conference on Geographic Information Science*, pages 1–15, 2018.

[33] A. Belussi and S. Migliorini. Skewness-based Partitioning in SpatialHadoop. *ISPRS International Journal of Geo-Information*, 9(4):201:1 – 201:19, 2020.

[34] A Belussi, S Migliorini, T. Vu, and A. Eldawy. Using Deep Learning for Big Spatial Data Partitioning. *ACM Transactions on Spatial Algorithms and Systems (TSAS)*, page To Appear, 2020.

[35] Alberto Belussi and Christos Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.*, pages 299–310, 1995.

[36] Alberto Belussi and Christos Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 299–310, 1995.

[37] Alberto Belussi and Christos Faloutsos. Self-spacial join selectivity estimation using fractal concepts. *ACM Trans. Inf. Syst.*, 16(2):161–201, 1998.

[38] Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. Detecting skewness of big spatial data in spatialhadoop. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2018, Seattle, WA, USA, November 06-09, 2018*, pages 432–435, Seattle, WA, USA, 2018. ACM.

[39] Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. A Cost Model for Spatial Join Operations in SpatialHadoop. *GeoInformatica*, x(x), 2020.

[40] Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. Skewness-based partitioning in spatialhadoop. *ISPRS International Journal of Geo-Information*, 9(4):201, 2020.

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

[42] Jon Louis Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. Software Eng.*, 5(4):333–340, 1979.

[43] G. E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *Ann. Math. Statist.*, 29(2):610–611, 06 1958.

[44] Christian BÖxhm, Gerald Klump, and Hans-Peter Kriegel. Xz-ordering: A space-filling curve for objects with spatial extension. In *International Symposium on Spatial Databases*, pages 75–90. Springer, 1999.

[45] US Census Bureau. All tiger lines, 2019. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?TIGER2018/EDGES&d`.

[46] US Census Bureau. Linear hydrography, 2019. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?TIGER2017/LINEARWATER&d`.

[47] US Census Bureau. Topological faces (polygons with all geocodes), 2019. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?TIGER2018/FACES&d`.

[48] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[49] Harry Chasparis and Ahmed Eldawy. Experimental evaluation of selectivity estimation on big spatial data. In *Proceedings of the Fourth International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data, Chicago, IL, USA, May 14, 2017*, pages 8:1–8:6, "Chicago, IL, USA", 2017. Acm.

[50] Yong-Jin Choi and Chin-Wan Chung. Selectivity estimation for spatio-temporal queries to moving objects. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 440–451, 2002.

[51] François Chollet et al. Keras. `https://keras.io`, 2015.

[52] Special issue: Digital libraries, November 1996.

[53] Sarah Cohen, Werner Nutt, and Yehoshua Sagic. Deciding equivalances among conjunctive aggregate queries. *J. ACM*, 54(2), April 2007.

[54] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[55] Graham Cormode. Sketch techniques for approximate query processing. In *Foundations and Trends in Databases. NOW publishers*, USA, 2011. NOW.

[56] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[57] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[58] Jens-Peter Dittrich and Bernhard Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 535–546, 2000.

[59] Bruce P. Douglass, David Harel, and Mark B. Trakhtenbrot. Statecarts in use: structured analysis and object-orientation. In Grzegorz Rozenberg and Frits W. Vaandrager, editors, *Lectures on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 368–394. Springer-Verlag, London, 1998.

[60] Ian Editor, editor. *The title of book one*, volume 9 of *The name of the series one*. University of Chicago Press, Chicago, 1st. edition, 2007.

[61] Ian Editor, editor. *The title of book two*, chapter 100. The name of the series two. University of Chicago Press, Chicago, 2nd. edition, 2008.

[62] A. Eldawy, L. Alarabi, and M. F. Mokbel. Spatial partitioning techniques in SpatialHadoop. *Proc. VLDB Endow.*, 8(12):1602–1605, August 2015.

[63] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *Proceedings of the 31st IEEE International Conference on Data Engineering*, pages 1352–1363, 2015.

[64] A. Eldawy and M. F. Mokbel. *Spatial Join with Hadoop*, pages 2032–2036. Springer International Publishing, Cham, 2017.

[65] Ahmed Eldawy, Louai Alarabi, and Mohamed F. Mokbel. Spatial partitioning techniques in spatial hadoop. *PVLDB*, 8(12):1602–1605, 2015.

[66] Ahmed Eldawy et al. Spatial partitioning techniques in spatialhadoop. *Proceedings of the VLDB Endowment*, 8(12):1602–1605, 2015.

[67] Ahmed Eldawy et al. Sphinx: Empowering Impala for Efficient Execution of SQL Queries on Big Spatial Data. In *SSTD*, pages 65–83, Arlington, VA, August 2017.

[68] Ahmed Eldawy, Yuan Li, Mohamed F. Mokbel, and Ravi Janardan. Cg_hadoop: computational geometry in mapreduce. In *SIGSPATIAL*, pages 284–293, Orlando, FL, November 2013.

[69] Ahmed Eldawy, Yuan Li, Mohamed F Mokbel, and Ravi Janardan. Cg_hadoop: computational geometry in mapreduce. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 294–303. ACM, 2013.

[70] Ahmed Eldawy and Mohamed F Mokbel. Pigeon: A spatial mapreduce language. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1242–1245. IEEE, 2014.

[71] Ahmed Eldawy and Mohamed F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363, Seoul, South Korea, April 2015.

[72] Ahmed Eldawy and Mohamed F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1352–1363, Seoul, South Korea, 2015. IEEE.

[73] Ahmed Eldawy and Mohamed F Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1352–1363. IEEE, 2015.

[74] Ahmed Eldawy and Mohamed F. Mokbel. The Era of Big Spatial Data: A Survey. *Foundations and Trends in Databases*, 6(3-4):163–273, 2016.

[75] Ahmed Eldawy and Mohamed F. Mokbel. Spatial join with hadoop. In *Encyclopedia of GIS.*, pages 2032–2036. 2017.

[76] Ahmed Eldawy and Mohamed F. Mokbel. All points on the map as extracted from openstreetmap., 2019. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?OSM2015/all_nodes&d`.

[77] Ahmed Eldawy and Mohamed F. Mokbel. All water areas in the world from openstreetmap, 2019. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?OSM2015/lakes&d`.

[78] Ahmed Eldawy and Mohamed F. Mokbel. The boundaries of all buildings in the world as extracted from openstreetmap., 2019. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?OSM2015/buildings&d`.

[79] Ahmed Eldawy and Mohamed F. Mokbel. Boundaries of parks and green areas from all over the world as extracted from openstreetmap., 2019. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?OSM2015/parks&d#mbr=9qh2s0vm,9qhf060f`.

[80] Ahmed Eldawy and Mohamed F. Mokbel. Roads and streets around the world each represented as a polyline extracted from openstreetmap, 2019. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?OSM2015/roads&d`.

[81] Ahmed Eldawy, Mohamed F. Mokbel, Saif Al-Harthi, Abdulhadi Alzaidy, Kareem Tarek, and Sohaib Ghani. SHAHED: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. In *ICDE*, pages 1585–1596, Seoul, South Korea, April 2015.

[82] Ahmed Eldawy, Mohamed F. Mokbel, and Christopher Jonathan. Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In *ICDE*, pages 601–612, Helsinki, Finland, May 2016.

[83] Ahmed Eldawy, Ibrahim Sabek, Mostafa Elganainy, Ammar Bakeer, Ahmed Abdelmotaleb, and Mohamed F. Mokbel. Sphinx: Empowering impala for efficient execution of SQL queries on big spatial data. In *SSTD*, pages 65–83, Arlington, VA, August 2017.

[84] The Common Metadata Repository: The Foundation of NASA's Earth Observation Data, 2017. `https://earthdata.nasa.gov/the-common-metadata-repository`.

[85] Cristian Estan and Jeffrey F. Naughton. End-biased samples for join cardinality estimation. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 20. IEEE Computer Society, 2006.

[86] Sattam Alsubaiee et al. Asterixdb: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[87] Christos Faloutsos, Bernhard Seeger, Agma Traina, and Caetano Traina. Spatial join selectivity using power laws. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 177–188, New York, NY, USA, 2000. Association for Computing Machinery.

[88] Christos Faloutsos, Bernhard Seeger, Agma Traina, and Caetano Traina, Jr. Spatial join selectivity using power laws. *SIGMOD Rec.*, 29(2):177–188, 2000.

[89] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[90] Miguel Rodrigues Fornari, João Luiz Dihl Comba, and Cirano Iochpe. Query optimizer for spatial join operations. In Rolf A. de By and Silvia Nittel, editors, *14th ACM International Symposium on Geographic Information Systems, ACM-GIS 2006, November 10-11, 2006, Arlington, Virginia, USA, Proceedings*, pages 219–226. ACM, 2006.

[91] Anthony D. Fox, Christopher N. Eichelberger, James N. Hughes, and Skylar Lyon. Spatio-temporal indexing in non-relational distributed databases. In *Big Data*, pages 291–299, Santa Clara, CA, October 2013.

[92] Saheli Ghosh, Ahmed Eldawy, and Shipra Jais. Aid: An adaptive image data index for interactive multilevel visualization. In *ICDE*, Macau, China, April 2019.

[93] Saheli Ghosh, Tin Vu, Mehrad Amin Eskandari, and Ahmed Eldawy. UCR-STAR: The UCR Spatio-Temporal Active Repository. *SIGSPATIAL Special*, 11(2):34–40, December 2019.

[94] Michael F. Goodchild. Citizens as voluntary sensors: Spatial data infrastructure in the world of web 2.0. *IJSDIR*, 2:24–32, 2007.

[95] Rajarshi Guhaniyogi and Sudipto Banerjee. Meta-kriging: Scalable bayesian modeling and inference for massive spatial datasets. *Technometrics*, 60(4):430–444, 2018.

[96] Antonio Gulli et al. *Deep Learning with Keras*. Packt Publishing Ltd, UK, 2017.

[97] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

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

[99] David Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1979.

[100] Apache HBase. `http://hbase.apache.org/`.

[101] Yaobin He, Haoyu Tan, Wuman Luo, Huajian Mao, Di Ma, Shengzhong Feng, and Jianping Fan. MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce. In *ICPADS*, pages 473–480, Tainan, Taiwan, December 2011.

[102] Nicolaus Henke et al. The Age of Analytics: Competing in a Data-driven World. Technical report, McKinsey Global Institute, December 2016.

[103] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range queries in OLAP data cubes. *SIGMOD Rec.*, 26(2):73–88, 1997.

[104] Erik G. Hoel and Hanan Samet. Performance of data-parallel spatial operations. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 156–167, 1994.

[105] Kevin Zeng Hu, Michiel A. Bakker, Stephen Li, Tim Kraska, and César A. Hidalgo. Vizml: A machine learning approach to visualization recommendation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, page 128, UK, 2019. ACM.

[106] James N Hughes, Andrew Annex, Christopher N Eichelberger, Anthony Fox, Andrew Hulbert, and Michael Ronquest. Geomesa: a distributed architecture for spatio-temporal fusion. In *SPIE Defense+ Security*, pages 94730F–94730F. International Society for Optics and Photonics, 2015.

[107] Yannis E. Ioannidis. Universality of serial histograms. In *VLDB*, pages 256–267, Dublin, Ireland, August 1993. VLDB.

[108] Edwin H Jacox and Hanan Samet. Spatial join techniques. *ACM Transactions on Database Systems (TODS)*, 32(1):7, 2007.

[109] Ji Jin, Ning An, and Anand Sivasubramaniam. Analyzing Range Queries on Spatial Data. In *ICDE*, pages 525–534, San Diego, CA, March 2000. IEEE.

[110] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB*, pages 500–509, Santiago de Chile, Chile, September 1994.

[111] Puloma Katiyar, Tin Vu, Ahmed Eldawy, Sara Migliorini, and Alberto Belussi. Spiderweb: A spatial data generator on the web. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, pages 465–468, 2020.

[112] Puloma Katiyar, Tin Vu, Sara Migliorini, Alberto Belussi, and Ahmed Eldawy. SpiderWeb: A Spatial Data Generator on the Web. In *28th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL 2020)*. ACM, November 2020.

[113] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[114] Sheila Kinsella, Vanessa Murdock, and Neil O'Hare. I'm eating a sandwich in glasgow: modeling locations with tweets. In *Proceedings of the 3rd international workshop on Search and mining user-generated contents*, pages 61–68. ACM, 2011.

[115] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[116] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms (3rd. ed.)*. Addison Wesley Longman Publishing Co., Inc., 1997.

[117] Donald E. Knuth. *The Art of Computer Programming*, volume 1 of *Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 3rd edition, 1998. (book).

[118] David Kosiur. *Understanding Policy-Based Networking*. Wiley, New York, NY, 2nd. edition, 2001.

[119] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504, USA, 2018. ACM.

[120] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[121] M. Seetha Lakshmi and Shaoyu Zhou. Selectivity estimation in extensible databases - A neural network approach. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 623–627. Morgan Kaufmann, 1998.

[122] Der-Tsai Lee and CK Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1):23–29, 1977.

[123] Taewon Lee et al. Omt: Overlap minimizing top-down bulk loading algorithm for r-tree. In *CAISE Short paper proceedings*, volume 74, pages 69–72, 2003.

[124] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality estimation done right: Index-based join sampling. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[125] Scott T Leutenegger et al. Str: A simple and efficient algorithm for r-tree packing. In *ICDE*, pages 497–506. IEEE, 1997.

[126] Yuan Li, Ahmed Eldawy, Jie Xue, Nadezda Knorozova, Mohamed F. Mokbel, and Ravi Janardan. Scalable Computational Geometry in MapReduce. *The VLDB Journal*, Jan 2019.

[127] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical Selectivity Estimation through Adaptive Sampling. In *SIGMOD*, pages 1–11, Atlantic City, NJ, May 1990. ACM.

[128] Zhicheng Liu and Jeffrey Heer. The effects of interactive latency on exploratory visual analysis. *Proceedings of the IEEE Transactions on Visualization and Computer Graphics, TVCG*, 20(12):2122–2131, 2014.

[129] Ming-Ling Lo and Chinya V. Ravishankar. Spatial joins using seeded trees. In *SIGMOD*, pages 209–220, Minneapolis, MN, May 1994.

[130] Jiamin Lu and Ralf Hartmut Guting. Parallel secondo: boosting database engines with hadoop. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 738–743. IEEE, 2012.

[131] Peng Lu, Gang Chen, Beng Chin Ooi, Hoang Tam Vo, and Sai Wu. ScalaGiST: Scalable Generalized Search Trees for Mapreduce Systems [Innovative Systems Paper]. *Proc. VLDB Endow.*, 7(14):1797–1808, 2014.

[132] Peng Lu, Gang Chen, Beng Chin Ooi, Hoang Tam Vo, and Sai Wu. Scalagist: Scalable generalized search trees for mapreduce systems [innovative systems paper]. *PVLDB*, 7(14):1797–1808, 2014.

[133] Peng Lu et al. ScalaGiST: Scalable Generalized Search Trees for MapReduce Systems. *PVLDB*, 7(14):1797–1808, 2014.

[134] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *PVLDB*, 5(10):1016–1027, 2012.

[135] Amr Magdy, Louai Alarabi, Saif Al-Harthi, Mashaal Musleh, Thanaa M. Ghanem, Sohaib Ghani, and Mohamed F. Mokbel. Taghreed: a system for querying, analyzing, and visualizing geotagged microblogs. In *SIGSPATIAL*, pages 163–172, Dallas/Fort Worth, TX, November 2014.

[136] Ahmed R. Mahmood, Ahmed M. Aly, Thamir Qadah, El Kindi Rezig, Anas Daghistani, Amgad Madkour, Ahmed S. Abdelhamid, Mohamed S. Hassan, Walid G. Aref, and Saleh M. Basalamah. Tornado: A distributed spatio-textual stream processing system. *PVLDB*, 8(12):2020–2023, 2015.

[137] Tanu Malik, Randal C. Burns, and Nitesh V. Chawla. A Black-Box Approach to Query Cardinality Estimation. In *CIDR*, pages 56–67. www.cidrdb.org, 2007.

[138] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In Rajesh Bordawekar and Oded Shmueli, editors, *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 3:1–3:4. ACM, 2018.

[139] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[140] Microsoft. Computer generated building footprints in all 50 us states., 2020. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?MSBuildings&d`.

[141] MongoDB. `https://www.mongodb.com/`.

[142] P. A. P. Moran. Notes on continuous stochastic phenomena. *Biometrika*, 37(1):17–23, 1950.

[143] Process Earth Science Data on AWS With NASA / NEX Public Data Sets, 2018.

[144] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. Bboxdb-a scalable data store for multi-dimensional big data. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 1867–1870. ACM, 2018.

[145] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[146] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. $\mathcal{MD}$-hbase: design and implementation of an elastic data infrastructure for cloud-scale location services. *Distributed Parallel Databases*, 31(2):289–319, 2013.

[147] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. $\mathcal{MD}$-hbase: design and implementation of an elastic data infrastructure for cloud-scale location services. *Distributed and Parallel Databases*, 31(2):289–319, 2013.

[148] Frank Olken and Doron Rotem. Sampling from Spatial Databases. In *ICDE*, pages 199–208, Vienna, Austria, April 1993. IEEE.

[149] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. Slalom: Coasting through raw data via adaptive partitioning and indexing. *Proceedings of the VLDB Endowment*, 10(10):1106–1117, 2017.

[150] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[151] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 314–325. Morgan Kaufmann, 1990.

[152] Oracle Spatial. `https://www.oracle.com`.

[153] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In Sebastian Schelter, Stephan Seufert, and Arun Kumar, editors, *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 4:1–4:4. ACM, 2018.

[154] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[155] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[156] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 486–495, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[157] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, pages 294–305, Montreal, Quebec, Canada, June 1996. ACM.

[158] PostGIS, 2019. https://postgis.net/.

[159] Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning, 2018.

[160] Ibrahim Sabek and Mohamed F. Mokbel. On spatial joins in mapreduce. In *SIGSPATIAL*, pages 21:1–21:10, Redondo Beach, CA, November 2017.

[161] Ibrahim Sabek and Mohamed F. Mokbel. On Spatial Joins in MapReduce. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '17, 2017.

[162] Ibrahim Sabek and Mohamed F Mokbel. Machine learning meets big spatial data. *Proceedings of the VLDB Endowment*, 12(12):1982–1985, 2019.

[163] Ibrahim Sabek, Mashaal Musleh, and Mohamed F Mokbel. Turboreg: A framework for scaling up spatial logistic regression models. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 129–138, "", 2018. ACM.

[164] Sartaj Sahni. Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM (JACM)*, 22(1):115–124, 1975.

[165] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[166] Manfred Schroeder. *Fractals, Chaos, Power Laws: Minutes From an Infinite Paradise.* W.H. Freeman and Company, New York, 1991.

[167] Salman Ahmed Shaikh, Komal Mariam, Hiroyuki Kitagawa, and Kyoung-Sook Kim. Geoflink: A framework for the real-time processing of spatial streams. *arXiv preprint arXiv:2004.03352*, 1(1):1, 2020.

[168] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[169] A. B. Siddique and Ahmed Eldawy. Experimental evaluation of sketching techniques for big spatial data. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, page 522, USA, 2018. ACM.

[170] AB Siddique and Ahmed Eldawy. Experimental evaluation of sketching techniques for big spatial data. In *SoCC*, page 522, "", 2018. ACM.

[171] AB Siddique, Ahmed Eldawy, and Vagelis Hristidis. Euler++: An improved selectivity estimation for rectangular spatial records. In *IEEE Big Spatial Data Workshop*, page 1, "", 2019. IEEE.

[172] Abu Bakar Siddique, Ahmed Eldawy, and Vagelis Hristidis. Comparing synopsis techniques for approximate spatial data analysis. *Proceedings of the VLDB Endowment*, 12(11):1583–1596, 2019.

[173] Samriddhi Singla and Ahmed Eldawy. Flexible Computation of Multidimensional Histograms. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Spatial Gems (SpatialGems 2020)*. ACM, November 2020.

[174] SpatialLite. `https://www.gaia-gis.it/fossil/libspatialite/index`.

[175] Asad Z. Spector. Achieving application requirements. In Sape Mullender, editor, *Distributed Systems*, pages 19–33. ACM Press, New York, NY, 2nd. edition, 1990.

[176] M Tang, Y Yu, WG Aref, AR Mahmood, QM Malluhi, and M Ouzzani. Locationspark: In-memory distributed spatial query processing and optimization. *CoRR*, 1(1):1–15, 2019.

[177] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. *PVLDB*, 9(13):1565–1568, 2016.

[178] Yannis Theodoridis and Timos Sellis. A model for the prediction of r-tree performance. In *Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 161–171. ACM, 1996.

[179] Twitter Usage Statistics, 2018.

[180] The ucr spatio-temporal active repository (ucr-star), 2019. https://star.cs.ucr.edu/.

[181] David Vengerov, Andre Cavalheiro Menck, Mohamed Zaït, and Sunil Chakkappen. Join size estimation subject to filter conditions. *Proc. VLDB Endow.*, 8(12):1530–1541, 2015.

[182] Hoang Vo, Ablimit Aji, and Fusheng Wang. SATO: a spatial data partitioning framework for scalable query processing. In *SIGSPATIAL*, pages 545–548, Dallas/Fort Worth, TX, November 2014. ACM.

[183] Hoang Vo, Ablimit Aji, and Fusheng Wang. SATO: A spatial data partitioning framework for scalable query processing. In *Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 545–548, New York, NY, USA, 2014. ACM.

[184] Dimitri Vorona, Andreas Kipf, Thomas Neumann, and Alfons Kemper. Deepspace: Approximate geospatial query processing with deep learning. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 500–503, "", 2019. ACM.

[185] Tin Vu. Deep query optimization. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1856–1858, USA, 2019. ACM.

[186] Tin Vu. Spatial join cost estimation using deep learning. `https://github.com/tinvukhac/deep-join`, 2020.

[187] Tin Vu and Ahmed Eldawy. R-Grove: growing a family of R-trees in the big-data forest. In *SIGSPATIAL*, pages 532–535, Seattle, WA, November 2018.

[188] Tin Vu and Ahmed Eldawy. R-grove: growing a family of r-trees in the big-data forest. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 532–535. ACM, 2018.

[189] Tin Vu and Ahmed Eldawy. Deep sampling. `https://github.com/tinvukhac/deep-sampling`, 2020.

[190] Tin Vu and Ahmed Eldawy. R*-grove: Balanced spatial partitioning for large-scale datasets. *Frontiers in Big Data*, 3:28, 2020.

[191] Tin Vu, Sara Migliorini, Ahmed Eldawy, and Alberto Belussi. Spatial data generators. In *1st ACM SIGSPATIAL International Workshop on Spatial Gems (SpatialGems 2019)*, page 7, "", 2019. ACM.

[192] Tin Vu, Sara Migliorini, Ahmed Eldawy, and Alberto Bulussi. Spatial data generators. In *1st ACM SIGSPATIAL International Workshop on Spatial Gems (SpatialGems 2019)*, page 1, Chicago, Illinois USA, 2019. ACM, ACM.

[193] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. Learned index for spatial queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*, pages 569–574. IEEE, 2019.

[194] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. Learning to hash for indexing big data - A survey. *Proceedings of the IEEE*, 104(1):34–57, 2016.

[195] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th edition, 2015.

[196] Randall T. Whitman et al. Spatial indexing and analytics on hadoop. In *SIGSPATIAL*, pages 73–82, Dallas/Fort Worth, TX, November 2014.

[197] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, 2017.

[198] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, pages 1071–1085, San Francisco, CA, July 2016. ACM.

[199] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1071–1085, New York, NY, USA, 2016. ACM.

[200] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Peter Chen, and Ion Stoica. NeuroCard: One Cardinality Estimator for All Tables. *PVLDB*, 14(1):61–73, 2020.

[201] J. Yu, J. Wu, and M. Sarwat. GeoSpark: a cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 70:1–70:4, 2015.

[202] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 70, "", 2015. ACM, ACM.

[203] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. Geospark: a cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL*, pages 70:1–70:4, Bellevue, WA, November 2015.

[204] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. A demonstration of geospark: A cluster computing framework for processing big spatial data. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1410–1413, 2016.

[205] Jian Yu, Miin-Shen Yang, and E Stanley Lee. Sample-weighted clustering methods. *Computers & mathematics with applications*, 62(5):2200–2208, 2011.

[206] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28. USENIX Association, 2012.

[207] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 38–49. ACM, 2012.

[208] Xiaofang Zhou, David J. Abel, and David Truffet. Data partitioning for parallel spatial join processing. *GeoInformatica*, 2(2):175–204, Jun 1998.

Appendix A

Spatial data generator

A.1 Introduction

In many published papers, researchers often need to test their implementations of new index structures or query execution methods on large scale spatial data. While some real datasets exist, the research community also needs to try datasets with specific characteristics to highlight how the proposed research behaves under certain circumstances. Synthetic data generation gives researchers full control over the data characteristics such as data skewness, complexity of geometries, or amount of overlap between datasets.

This chapter proposes a practical tool for generating synthetic spatial datasets with various skewed distributions. These generators have been successfully used in existing research to evaluate index construction, query processing, spatial partitioning, and cost model verification. While the generators are already used in many papers, there is a fact that the researchers rarely describe the details of generating these datasets for two reasons. First, it is not usually a research contribution and the authors do not want to draw attention

to it. Second, it takes a precious space of the paper that authors usually prefer to utilize for other parts.

This work takes the burden of describing, in detail, how to generate synthetic data of six common distributions. As the work is designed to be flexible, it fits very well the generation of synthetic data where other researchers can add more datasets in the future.

Figure A.1 gives an overview of the main parts of the proposed generator. First, the dataset descriptor is a vector that contains information about the dataset to be generated. It acts a unique identifier for the synthetic dataset and it consists of three parts, (1) the distribution ID $i \in [1, 6]$ for six implemented distributions, (2) the model parameters depending on the chosen distribution, and (3) a transformation matrix used later by the transformer. The first two components of the dataset descriptor, i.e., distribution ID and model parameters, are passed to the generator which generates the desired dataset. After that, the transformer applies an affine transformation on the generated data according to the third component of the dataset descriptor.

Since the dataset descriptor fully identifies the generated dataset, researchers who use these generators can simply cite this work and list the descriptors of the datasets they used. Other researchers can then regenerate the same datasets following the same procedure described in this work. For guidance, it provides a reference implementation [2] to all the generators as a supplementary material, but researchers can develop other generators that follow the same guidelines, e.g., a Spark-based generator for big spatial data. For example, Table A.1 at the end of this work defines the six datasets used in this article.

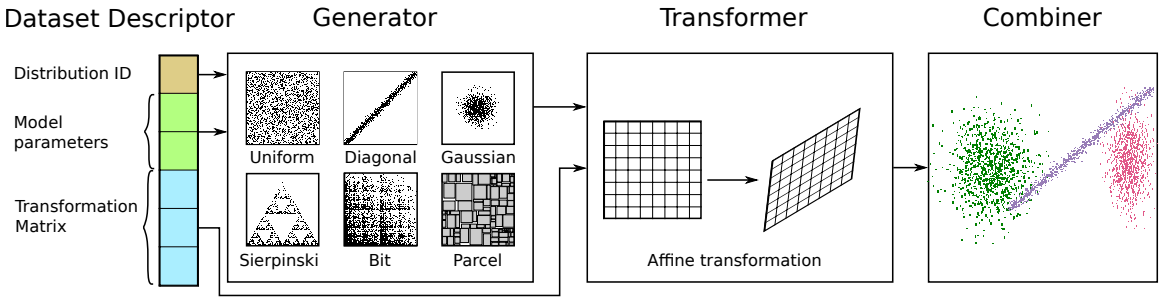


Figure A.1: Overview of the spatial generator

The final component, combiner, can be used to create compound datasets by simply merging two or more simple datasets. In this case, the descriptor of the compound dataset is simply the concatenation of all the descriptors of the simple datasets.

As shown in Figure A.1, this work uses six different distributions for generating the simple datasets, which are all described in the next section.

A.2 Data Generators

This section describes the six synthetic generators which are used in this work, namely, uniform, diagonal, Gaussian, Sierpinski, bit, and parcel. Some of these generators are inspired by a benchmark developed by Beckmann and Seeger [30]. This article provides more details about these generators and defines some additional generators. Each generator G_* takes a list of common parameters $[cp_1, cp_2, \dots]$ and a list of distribution-specific parameters $[sp_1, sp_2, \dots]$. For the family of generators that are considered in this work, there are two common parameters, the DATASET CARDINALITY ($card$) specifying the total number of geometries and the NUMBER OF DIMENSIONS (d). The generator here considers the

generation of two-dimensional geometries; the extension to multi-dimensional datasets is straightforward.

For all these generators, the REFERENCE SPACE that contains the generated data is $[0,1]^d$, where d is the number of dimensions. Additionally, these generators assume the existence of a random number generator RND() which generates random numbers in the range $[0,1)$. This generator can be used to generate random numbers for three popular distributions, Bernoulli, Uniform, and Normal, as follows [1].

$$Bernoulli(p) = \begin{cases} 1 & ; \text{RND}() < p \\ 0 & ; otherwise \end{cases} \tag{A.1}$$

$$U(a,b) = (b-a)\text{RND}() + a \tag{A.2}$$

$$N(\mu,\sigma) = \mu + \sigma\sqrt{-2\ln \text{RND}_1()} \cdot \sin(2\pi \text{RND}_2()) \text{(Box and Muller [43])} \tag{A.3}$$

Algorithm 5 shows the general schemata of the first five generators, namely, uniform, diagonal, Gaussian, Sierpinski, and bit distribution. These five generators can generate both points and rectangles. Points are generated using the GenPNT$_*$ methods, described shortly, while rectangles are generated by using these points as centers. Lines 2,3 initialize the result set $\mathcal{G}$ to the empty set and the random number generator. If desired, the seed of the random number generator can be fixed to generate exactly the same random dataset. The loop in

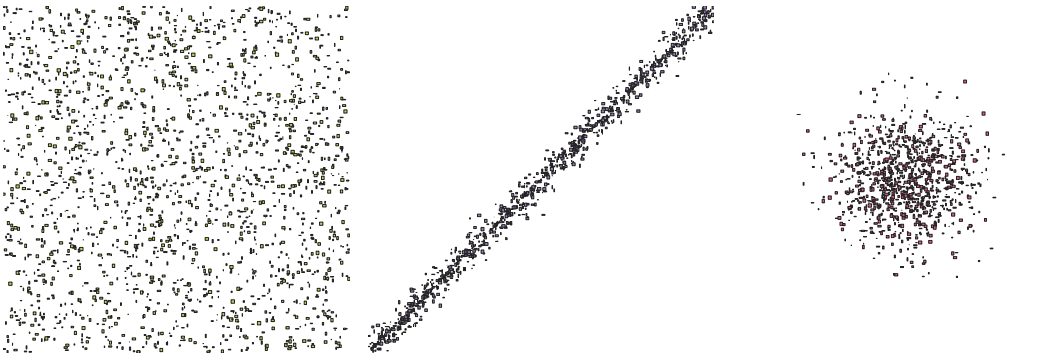
[1] More efficient implementations are usually available in standard packages

lines 4-10 generates one record at a time. Line 5 calls a generic GenPNT$_*$ function that is different for each generator. This function returns a random point (x, y) according to the desired distribution. Then, Line 6 tests if the point is inside the reference space $[0, 1]^d$ as some generators can generate points outside that space, e.g., Gaussian. If the point is inside the reference space, the algorithm continues by generating random width and height for the rectangle by using the parameters sp_1 and sp_2 as the maximum allowed width and height. Finally, a rectangle is generated with (x, y) as the center and (w, h) as its dimensions, i.e., the corner point is at $(x - w/2, y - h/2)$. The notation $Box(x, y, w, h)$ is used to indicate a box with its lower corner point at (x, y) and has dimensions of w and h.

The following parts are the descriptions of the point generators (GenPNT$_*$) for the first five distributions and then the parcel distribution which generates rectangles directly without generating points first.

Algorithm 5 Basic algorithm for the first five generators.

```
1:  function G_*(card, d = 2,SP_1,..., sp_n )
2:      G ← ∅; i ← 0
3:      Initialize the random number generator
4:      WHILE i < card DO
5:          (x, y) ← GenPNT_*(i, sp_3, ..., sp_n)
6:          IF (x, y) ∈ [0, 1]^d THEN
7:              w = U(0, sp_1)
8:              h = U(0, sp_2)
9:              G = G ∪ {Box(x − w/2, y − h/2, w, h)}
10:             i ← i + 1
        RETURN G
```



(a) Uniform distribution (b) Diagonal distribution (c) Gaussian distribution

Figure A.2: Examples of the first three distributions

A.2.1 Uniform

In the uniform distribution, points are generated randomly inside the reference space $[0, 1]^d$ as shown in Figure A.2(a). This distribution models non-skewed data such as data in suburban areas. No additional specific parameters are needed for this generator. The point (x, y) is generated using the following equations.

$$GenPNT_{uni}() = (x = U(0, 1), y = U(0, 1)) \tag{A.4}$$

A.2.2 Diagonal

The diagonal point generator generates points that are concentrated around the diagonal line $x = y$ as illustrated in Figure A.2(b). This distribution models real data concentrated around a line such as a river bank or a highway. The affine transformation, described later, can be used to arbitrarily rotate this line. This generator takes two additional parameters $perc$ and buf, where $perc \in [0, 1]$ is the percentage (ratio) of the points that are

exactly on the line, and $buf \in [0, 1]$ is the size of the buffer around the line where additional points are scattered. The additional points are scattered according to a normal distribution. Algorithm 6 illustrates the generation of a point using the diagonal distribution. Line 2 decides with a probability *perc* to generate a point exactly on the diagonal. Otherwise, with probability $1 - perc$, it generates a point that is shifted with a distance d from the center. The distance is generated from the normal distribution $N(0, buf/5)$ which has almost a 99% probability in generating a number in the range $[-buf, +buf]$. This distance is then divided by $\sqrt{2}$ to calculate the orthogonal offset for x and y. Finally, the point location is generated.

Algorithm 6 Diagonal Point Generator Algorithm

```
1: function GenPNT_dia(perc, buf)
2:     if Bernoulli(perc) = 1 then
3:         x = y = U(0, 1)
4:     else
5:         c = U(0, 1)
6:         d = N(0, buf/5)
7:         x = c + d/√2
8:         y = c − d/√2
       return (x, y)
```

A.2.3 Gaussian

In the Gaussian distribution, the points are concentrated around the center of the input space $(0.5, 0.5)$ as illustrated in Figure A.2(c). This distribution can model real data concentrated around a point such as a metro area. The coordinates follow a normal distribution $x, y \sim N(0.5, 0.1)$. This ensures that almost 99% of the points fall in the box $[0, 1]^d$. If points are generated outside that space, the loop in Algorithm 5 will drop the point and generate another one. This might make the data distribution is a bit different

from original Gaussian distribution. In summary, the Gaussian point generator follows the following formula.

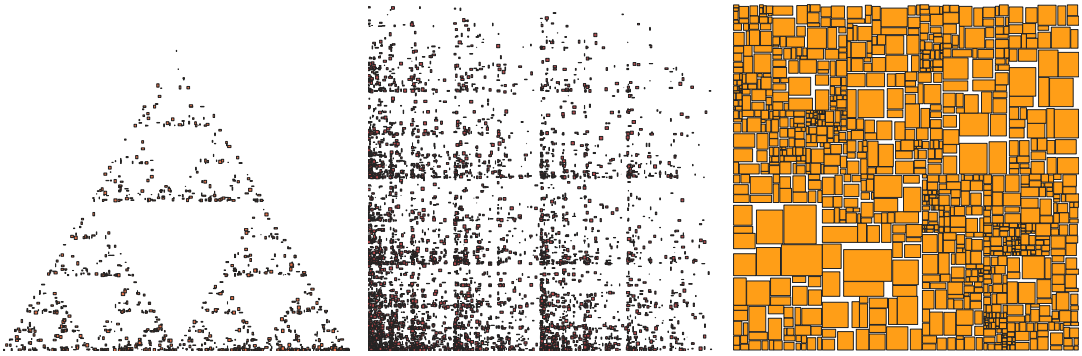
$$\textsc{GenPNT}_{Gaussian} = (x = N(0.5, 0.1), y = N(0.5, 0.1)) \tag{A.5}$$

A.2.4 Sierpinski triangle

In this case, the skewed distribution is obtained by applying a rule for generating points that belong to a fractal (the Sierpinski's triangle) [25]. Figure A.3(a) gives an example of this data. This pattern could be found in many real model such as cellular automata or motors. This rule is based on an iterative approach such that the generation of the next point of the set depends on the current point and a random function. The function $\textsc{GenPNT}_{sie}(pnt, i)$ has two specific parameters: the previous point of the iteration pnt and the iteration variable i. It generates a two-dimensional point at each iteration as follows:

$$\textsc{GenPNT}_{sie}(pnt, i) = \begin{cases} (0.0, 0.0) & \text{if } i = 0 \\ (1.0, 0.0) & \text{if } i = 1 \\ (1/2, \sqrt{3}/2) & \text{if } i = 2 \\ \textsc{MiddlePoint}(pnt, (0.0, 0.0)) & \text{if } i > 2 \wedge \textsc{Dice}(5) \in \{1, 2\} \\ \textsc{MiddlePoint}(pnt, (1.0, 0.0)) & \text{if } i > 2 \wedge \textsc{Dice}(5) \in \{3, 4\} \\ \textsc{MiddlePoint}(pnt, (1/2, \sqrt{3}/2)) & \text{if } i > 2 \wedge \textsc{Dice}(5) = 5 \end{cases}$$

where $\textsc{Dice}(5) = \lfloor U(0, 5) \rfloor + 1$ is a random function producing a number between 1 and 5 and $\textsc{MiddlePoint}(pnt_1, pnt_2)$ computes the middle point between two points, pnt_1 and pnt_2.



(a) Sierpinski distribution (b) Bit distribution (c) Parcel distribution

Figure A.3: Examples of the last three distributions

Notice that, the first three points are the corners of the triangle; the successive points are generated starting from the current point *pnt* and computing the middle point between *pnt* and one of the vertices of the triangle chosen according to the random function DICE(5). The vertices of the base are chosen with a probability of 2/5, the other vertex with a probability of 1/5. Figure A.3(a) shows an example of the resulting set.

A.2.5 Bit distribution

Another approach for generating a skewed point dataset is to introduce a rule for generating the coordinates of the points by assigning higher probability to a subset of coordinates. For instance in the Bit distribution, the point coordinates are generated as a bit string of a fixed length where each bit is set with a fixed probability $p \in [0, 1]$. This generator takes two parameters, p and $digits$, where p represents a fixed probability of setting each bit independently to 1 and $digits$ represents the number of binary digits after the fraction point. It generates a point in a higher dimensional space by setting each dimension independently

in the same method as shown below:

$$\text{GenPNT}_{bit}(p, digits) = (\text{Bit}(p, digits), \text{Bit}(p, digits))$$

where $\text{Bit}(p, digits)$ generates a real number between 0.0 and 1.0 as shown in the Algorithm 7.

Algorithm 7 Bit generator function

```
1: function Bit(p, digits)
2:     n ← 0.0; i ← 0
3:     for i = 1 to digits do
4:         c ←Bernoulli(p)
5:         n ← n + c/2^i
       return n
```

A.2.6 Parcel distribution

This generator directly generates rectangles according to the parcel distribution. The parcel distribution generates geometries that represent boxes of different sizes as illustrated in Figure A.3(c). This distribution can model land sections delineated in urban areas.

In addition to the cardinality *card* of the generated dataset, the parcel generator takes two specific parameters r and d, where:

- $r \in [0, 0.5]$ is the minimum tiling range for splitting a box. $r = 0$ indicates that all the ranges are allowed while $r = 0.5$ indicates that a box is always split into half.

- $d \in [0, 1]$ is the dithering parameter that adds some random noise to the generated rectangles. $d = 0$ indicates no dithering and $d = 1.0$ indicates maximum dithering that can shrink rectangles down to a single point.

Algorithm 8 describes how the parcel generator works. The first loop in lines 4-15 repetitively splits the reference space $[0,1]^d$ along one of the two axes x and y. It always splits along the longer axis of the given box. This loop runs $card - 1$ times as it generates one new box at each iteration.

The second loop in lines 16-18 adds the dithering effect by shrinking each box with a ratio $1 - U(0,d)$. This means that if $d = 0$, the ratio will always be equal to 1.0 which means no shrinking. If $d = 1.0$, the ratio can reach up-to 1.0 which shrinks the boxes all the way to a single point.

Algorithm 8 Generated boxes of the parcel distribution

```
1:  function G_parcel(card, d = 2, r, d)
2:      Initialize the random number generator
3:      G ← {Box(0, 0, 1.0, 1.0)}
4:      while |G| < card do
5:          b ← G.dequeue
6:          if b.width > b.height then
7:              splitSize = b.width ∗U(r, 1 − r)
8:              b1=Box(b.x, b.y, splitSize, b.height)
9:              b2=Box(b.x + splitSize, b.y, b.width - splitSize, b.height)
10:         else
11:             splitSize = b.height ∗U(r, 1 − r)
12:             b1=Box(b.x, b.y, b.width, splitSize)
13:             b2=Box(b.x, b.y + splitSize, b.width, b.height - splitSize)
14:         G.enqueue(b1);
15:         G.enqueue(b2);
16:     for b ∈ G do
17:         b.width = b.width · (1 − U(0, d))
18:         b.height = b.height · (1 − U(0, d))
        return G
```

A.3 Post Transformations

This section describes two methods that can giver further flexibility on customizing the generated data, transformation and compounding. The transformation method applies a simple affine transformation on the generated geometries. The compounding method combines several datasets by simply unifying all their geometries.

A.3.1 Affine Transformation

The generators described above are designed to be very simple on purpose. Instead of complicating each generator, most of the customization is moved to this step. This step simply applies a standard affine transformation to all the generated geometries. In the case of points, the transformation is applied to the coordinates of the point. For rectangles, the transformation is applied to the two opposite corners, i.e., the lower left and upper right corners. This ensures that the rectangle remains orthogonal even after rotation which is usually desired in data structures and algorithms that deal with bounding boxes.

An affine transformation is defined by a fixed-size matrix. For two-dimensional data, the affine transformation transforms a point (x, y) to a transformed point (x', y') according to the following equation.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{A.6}$$

Distribution ID	*card*	*d*	sp_1	sp_2	sp_3	sp_4	a_1	a_2	a_3	a_4	a_5	a_6
Uniform (Figure A.2(a))	1000	2	0.02	0.02			1	0	0	0	1	0
Diagonal (Figure A.2(b))	1000	2	0.01	0.01	0.2	0.1	1	0	0	0	1	0
Gaussian (Figure A.2(c))	2000	2	0.1	0.1			1	0	0	0	1	0
Sierpinski (Figure A.3(a))	1000	2	0.01	0.01			1	0	0	0	1	0
Bit (Figure A.3(b))	5000	2	0.01	0.01	0.3	10	1	0	0	0	1	0
Parcel (Figure A.3(c))	1000	2	0.2	0.2			1	0	0	0	1	0

Table A.1: Identifiers for the six sample datasets shown in this paper. For simplicity, all of them use the identity transformation (affine) matrix

where $a_1 \cdots a_6$ are the parameters of the affine transformation. This formula could be modified to work with higher dimensional data.

A.3.2 Compound Datasets

The final stage is to combine several datasets of either the same distribution but different parameters, different distributions, or with different transformation matrices. Show some examples of how combining datasets can generate new interesting datasets.

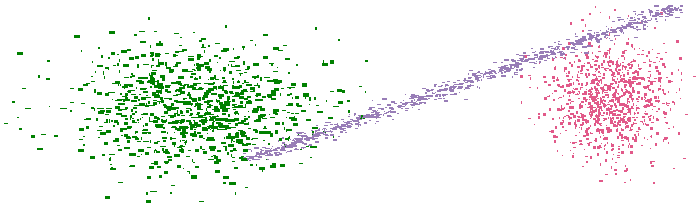


Figure A.4: Example of compound dataset obtained by combining two different Gaussian distributions and one diagonal distribution.

A.3.3 Identifying Datasets

Based on the proposed method, each generated simple datasets, i.e., not compound, can be identified using a fixed vector. This vector is the one illustrated in Figure A.1 and it

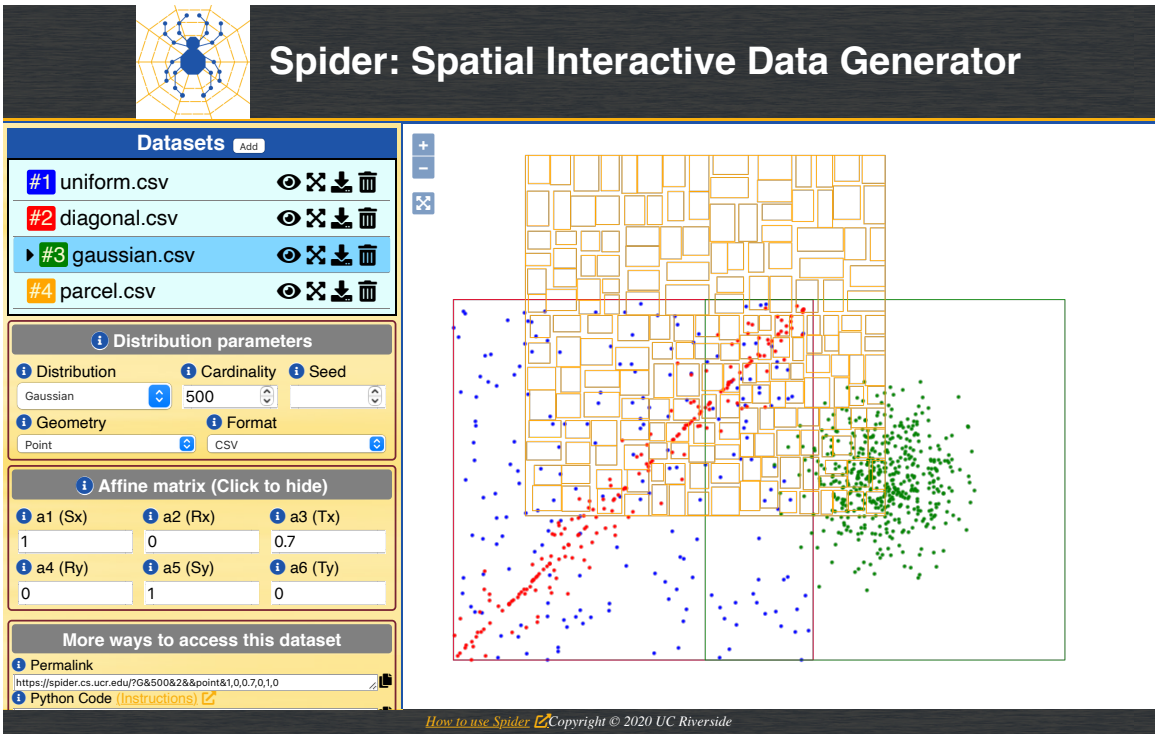


Figure A.5: Spider Web: Online service for spatial data generation

contains the generator model G_*, the common parameters $card$ and d, the specific parameters $sp_1 \cdots sp_n$, and the affine transformation matrix parameters $a_1 \cdots a_6$. Researchers who use the generators described in this paper can simply list all these parameters in a table to allow other researchers to generate datasets of the same characteristics. For example, Table A.1 identifies the six sample datasets illustrated in Figure A.2 and A.3.

A.4 Spider: Web-based Spatial Data Generator

The proposed spatial data generator is available as open source [2] in Python for users with basic programming skills. In order to serve a wider range of users, Spider Web [111, 3] was proposed as a web-based spatial data generator based on the proposed

design. Figure A.5 shows a simple example of data generation using Spider Web. Users can choose the distribution type, parameters, and affine matrix in the panel on the left. The visualization immediately reflects how the dataset will look like. Users can then download the selected dataset with an arbitrary size in standard formats, CSV, WKT, or GeoJSON. Furthermore, as illustrated in Figure A.5, users can generate multiple datasets and visually compare them. Finally, Spider Web provides a permalink for any generated dataset that can be shared between team members or researchers to promote the reproducibility of results. With these comprehensive features, Spider Web aims to be a standard tool to improve the reproducibility of experiments in the spatial research community.