Benchmarking data partitioning techniques in HDFS for big real spatial data

Nikolaos Niopas
niopasn@gmail.com

July 10, 2019, 36 pages

Research supervisor: Adam Belloum, A.S.Z.Belloum@uva.nl
Host/Daily supervisor: Jasper van Rooijen, jasper.van.rooijen@spatial-eye.com
Host organisation/Research group: Spatial Eye, https://www.spatial-eye.com/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Since big data has emerged as central in many applications, efficient ways of storing and processing need to be adopted. Traditional database systems seem not to scale very well, whereas using cloud computing frameworks such as Hadoop becomes a growing trend followed by many big enterprises. This global need for handling big data efficiently creates space for improvements in the already implemented algorithms and frameworks.

In the Hadoop framework, big spatial data, unlike traditional data, have some extra characteristics that need to be taken into account to improve the storing and processing operations. While working in a multinode environment, it is essential that each node has the same amount of computation to do, while grouping together spatial objects that are geographically close to each other may reduce the amount of the required computations depending on the query type.

In this thesis, we investigate how existing spatial partitioning algorithms help to scale up the processing of big spatial data to reduce the runtime. Also, we draw some results about what type of partitioning is more optimal over others regarding the query type and the dataset. Finally, we discuss why spatial partitioning is important if we focus on the efficient processing of the data.

Our experiments show that the quad-tree based partitioning algorithm is the most efficient regarding speed in most of the conducted queries, followed by the R-tree based partitioning algorithm. More specifically, for queries affecting up to 70% of the dataset, the quad-tree based algorithm has 1.5 to 8 times less execution time, while R-tree based one ranges from 1.5 to 2 times, compared with the default partitioner of Hadoop. The universal grid algorithm is quicker to be applied, but it is significantly dependent on the data skew. Finally, the default hashPartitioner of Hadoop, completely ignores any spatial characteristic, rendering it inefficient in processing big spatial data.

Contents

Chapter 1

Introduction

1.1 Spatial Data

Among the specialized data handled by today's databases, spatial data has emerged as central to many applications. These include geographic information systems (GISs), computer-aided design (CAD), robotics, image processing, and VLSI, all of which have at their core spatial objects that must be stored, queried, and displayed[1]. There is an increasing need for specific techniques to handle spatial objects and that gave rise to the area of spatial databases, to make the processing faster[1]. This way, the difficulty regarding the execution time that arises from handling big data needs to be overcome. In a relatively short period, spatial databases have developed a comprehensive technology, including representations for spatial objects, spatial access methods for fast retrieval, specific query languages, and algorithms adapted from adjacent areas such as computational geometry[1].

Spatial Database Management Systems store and maintain large collections of multidimensional data. Past research in the database community focused on the evaluation of big data systems by applying common query types which they are quite often used in many applications, such as range queries[2] (i.e., find all objects that intersect or are contained in a spatial region), spatial joins[3] (i.e., find all pairs of objects from two datasets that satisfy a spatial predicate), and nearest-neighbor queries[4] (i.e., find the closest object to a reference point or object)[5].

Traditional data processing systems cannot process spatial data with the same efficiency as other types of data and the reason behind that is some extra characteristics that the spatial data has [6]. First of all, spatial data is location-based data, which means that the key to reducing the run-time of each query is the efficient processing of the spatial location of each object. Additionally, spatial data has a complex structure, unlike most of the traditional data, which can vary from a single point, to complicated polygons or curves in many dimensions. This extra complexity has to be taken into account, to achieve efficient processing. Moreover, spatial data is at most cases dynamic, leading to multiple insertions and deletions over time, making indexing quite challenging.

1.2 Big Data Systems

Some decades ago, the available data was not sufficient enough to process it and draw results out of it. In recent years though, the problem is quite the opposite. There are numerous sources of data, providing information about various fields, but due to its extent, it is challenging to process it all efficiently and acquire information out of it[1]. Traditional database systems work well when the data is relatively small, but they seem to have a scaling problem [4]. For that reason, to efficiently store and process big data, alternative approaches need to be followed.

According to Rui Han et. al. [7], Big Data Systems can fall into 3 major camps:

Hadoop and its relevant systems: Hadoop is mainly known for its Hadoop Distributed File System (HDFS) along with its MapReduce method for storing and processing data respectively. It also comes with a rich, growing ecosystem containing various support regarding high-level languages and Structured Query Language (SQL). This camp has become the de facto solution for a majority of big data applications. Hadoop and in general MapReduce based systems are increasingly being used for large-scale data

analysis applications[8].

Database Management Systems (DBMSs) and NoSQL Data Stores: Such Systems are widely used in online transactional and analytical applications. While they provide an efficient solution, these systems rely on the high availability of memory.

Specialized System in the big data domain: Such systems deal with specific processing requirements of connected graphs, continuous streams, and complex scientific data. They are used for more specific problems, and they do not provide a generic solution.

Choosing the right big data system has to be done regarding the very nature of each problem. Empirical studies have shown that Hadoop can be efficiently used when the update rate of our data is relatively high and works great even with limited memory sources [9]. Relational database systems work better when we are concerned about reading the data rather than update it, while NoSQL's efficiency may be close to the Hadoop's, but it requires high memory utilization [9].

1.3 Big Data Partitioning

When the amount of data gets too big for a single node to handle, the data is usually partitioned and each group of partitions is stored in a different data block. When a query takes place, these data blocks are given to different calculating nodes, where each of them is executing part of the query to its data block. The partitioning technique should be chosen regarding the type of the data, its complexity and the type of processing. Different queries work better under different partitioning, thus to choose a partitioning technique, one should consider the most common and time-consuming queries that take place.

Traditional and spatial data do not share the same characteristics when it comes to partitioning. While traditional data can be partitioned with simple techniques such as based on their hashmap by using key-value pairs, spatial data needs to be partitioned in a way that its spatial attributes will be taken into account. To illustrate this further, spatial objects that their position is physically close to each other should be -if possible- included in the same data block, if we are interested in spatial location queries[10]. This is very useful since the majority of the processing that one needs to have with spatial data are queries that rely on the position of the objects. Hence, if the partitioning is done regarding the spatial location of the data, queries may have to be conducted in fewer data blocks, saving valuable processing time. Further, in parallel computation, it is essential for each node to have all the objects that fall into an area because this minimizes the communication among other nodes in spatial queries.

1.4 Problem Statement

As mentioned in the above sections, the structure and complexity of spatial data make it more difficult to store and process in the traditional data systems and the problem gets tougher the bigger the spatial data is. For that reason, big data systems have been developed to bridge the gap between processing traditional and spatial data. Such systems should take into account an efficient way of partitioning the data, that respects their location so that spatial queries can be conducted more efficiently.

We have also mentioned in the previous section that when dealing with big data, it is quite common to divide/partition the data into multiple data blocks and assign different nodes to store and process them. Spatial objects adjacent to each other should be saved in the same data block so that each block can have all the objects included in an area. Traditional partitioning techniques are not designed to be efficient with spatial data, thus they completely ignore their spatial location and lead into spatial objects that belong to an area to be scattered around different data blocks [11]. While this may be efficient for processing alphanumeric values, spatial queries cannot process the data efficiently, since they rely on the spatial location of the objects. Given the spatial datatype, the volume and the query type, the selection of a right partitioning technique could reduce the run time and provide quicker results.

1.4.1 Research questions

To tackle these issues, we investigate how the most efficient partitioning techniques for spatial data in traditional databases scale up in a multinode environment conducting queries in parallel. We have chosen to use the Hadoop System along with its relevant systems since we are interested in spatial data that

requires continuous updates and we want to use a limited portion of the memory resources. We will examine how different partitioning techniques affect the efficiency of a parallel computation in Hadoop for spatial data, given the spatial database and query type. More specifically, we will examine how partitioning based on quad and r-trees affect the computational efficiency of a query in the HDFS environment, comparing to the standard grid partitioning of the SpatialHadoop framework[10] and default hashPartitioner of Hadoop, that completely ignores any spatial characteristics. The problem statement can be formalized into the following research question.

RQ: How can spatial partitioning methods help to scale up the processing of big spatial data?

This question is of great importance to be answered since by using the right partitioning methods, we can boost the performance of processing the big data and as a result increase the capabilities of our system. However, to answer the above question we need to provide answers to the following subquestions:

What are the best data structures that are used in traditional spatial databases?

Answering this question can help us understand the core concepts of storing spatial databases, therefore partitioning it in a distributed file system could be based on the best practices, rather than implement something from scratch.

Do these methods scale up well in multinode systems processing data in parallel?

Learning from the past is a good technique to predict similar behaviors in other systems that have some aspects in common. Because the two systems are not identical though, we have to delve into the behavior of these practices and examine if their performance is efficient for the given problem.

1.5 Outline

In Chapter 2 we describe the background of our research along with some core terminology. Next, in Chapter 3 we analyze the partitioning algorithms used for our research and the query types for processing the data efficiently. The results of our experiments are shown and discussed in Chapter 4. Finally, we present our concluding remarks in Chapter 5 together with future work.

Chapter 2

Background

This chapter will present the necessary background information for this thesis. First, we define some basic terminology that will be used throughout this thesis. Next, we explain the importance of efficient partitioning techniques along with the efficient state of the art partitioning methods for traditional spatial databases. Last, we present some use cases and we explain the relevance of this research.

2.1 Hadoop and Subsystems

The Apache Hadoop software library is a framework that eases the processing of large data sets across clusters of computers[12]. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. The library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures[12].The 2 most useful components of the Hadoop, are the Hadoop Distributed File System (HDFS) and the MapReduce under a master/slave architecture (Fig.2.2), which we will describe in the next sections.

2.1.1 MapReduce

MapReduce is a popular programming model for distributed processing of large data sets. [8]. A MapReduce job consists of two main functions the Map and the Reduce along with some other processes that take place in between:

Map Function:

The Map function creates key-value pairs for each input, while many pairs have the same -no unique- key. Then the system sorts the key-value pairs by key, and for each one of them creates a pair consisting of the key itself and a list of all the values associated with it [13].

Reduce Function:

The Reduce function takes each corresponding key and its associated list of values. Then it executes the necessary calculations written by the user and applies them to the list of values. The final result contains the calculation outcome to each key and its list.

In Figure 2.1 we can see a visualization of the structure of the MapReduce model. In the Hadoop environment, since the Map tasks can be executed in parallel as well as the Reduce tasks, we can obtain an almost unlimited degree of parallelism, if we also have the hardware support [13]. Furthermore, one of the core features of MapReduce is the blocking property [14], which reassures that all of the Map or Reduce tasks deliver the outputs when all the work has been finished. As a result, if a hardware or software fails in the middle of a MapReduce job, the system can easily recover by restarting the tasks [13].

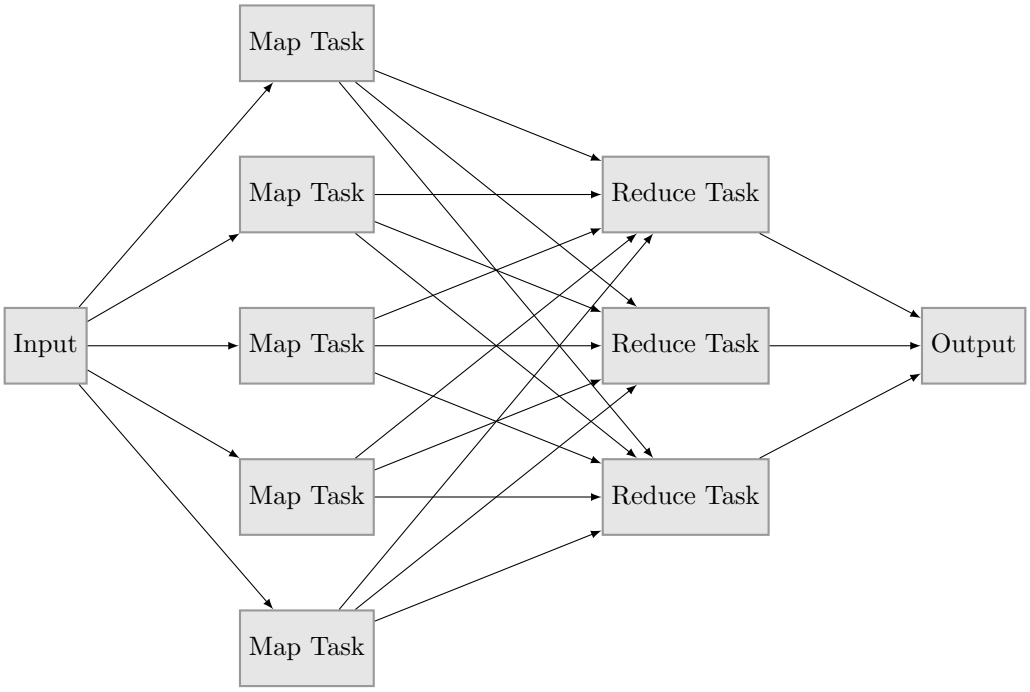

Figure 2.1: The structure of a MapReduce job in Hadoop

2.1.2 Hadoop Distributed File System (HDFS)

The Distributed File system of Hadoop (HDFS) is a highly fault-tolerant distributed file system designed efficient scaling up of big data storage and run on clusters of commodity machines. An HDFS cluster consists of a single NameNode and several Datanodes. The NameNode is a master server that manages the file system and regulates access to files by clients containing all of the metadata. The DataNodes, usually one per node in the cluster, manage storage attached to the nodes that they run on. HDFS exposes a file system and allows user data to be stored in files, where each of them is split into one or more blocks and these blocks are stored in a set of DataNodes. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode [12]. When we want to process the data or conduct a query, the JobTracker which belongs to the MasterNode assigns Map and Reduce tasks to the TaskTrackers (Figure 2.2).

2.2 Spatial Partitioning in Traditional Databases

To store and process big spatial data efficiently, we need to partition the big data into smaller data blocks. As mentioned by Han et. al. [7], two of the most efficient partitioning techniques in traditional databases are based on the Quad-tree and the R/R+ tree. Another empirical study using the spatialHadoop Framework [15] shows that the quad-tree based partitioning outperformed the other 6 partitioning methods that they examined for the range query and the spatial join, while R/R+ tree-based partitioning was the second best.

We hypothesize that these partitioning methods will have the same relevant efficiency in the HDFS since we will have 3 major benefits. First, if the spatial query to be conducted does not refer to the whole data set, but to a specific area, it is quite possible that not all of the data nodes will have to be queried. If the partitioning was ignoring spatial characteristics, then all of the data blocks should have been examined to be sure about the correctness of the query. Second, within each data block, the processing time will require less communication with the other blocks, if each one has all the adjacent objects, for example in a range query. Last, these partitioning methods try to balance the number of spatial objects within each data block, so that roughly all of them will require similar execution tasks. This boosts the efficiency in parallel since no node will have to wait for the others as the processing time should be roughly equal.

Some of the partitioning techniques rely on sampling, while others do not. Sampling allows us to predict factors such as the topology of the objects along with other useful statistics, which makes it

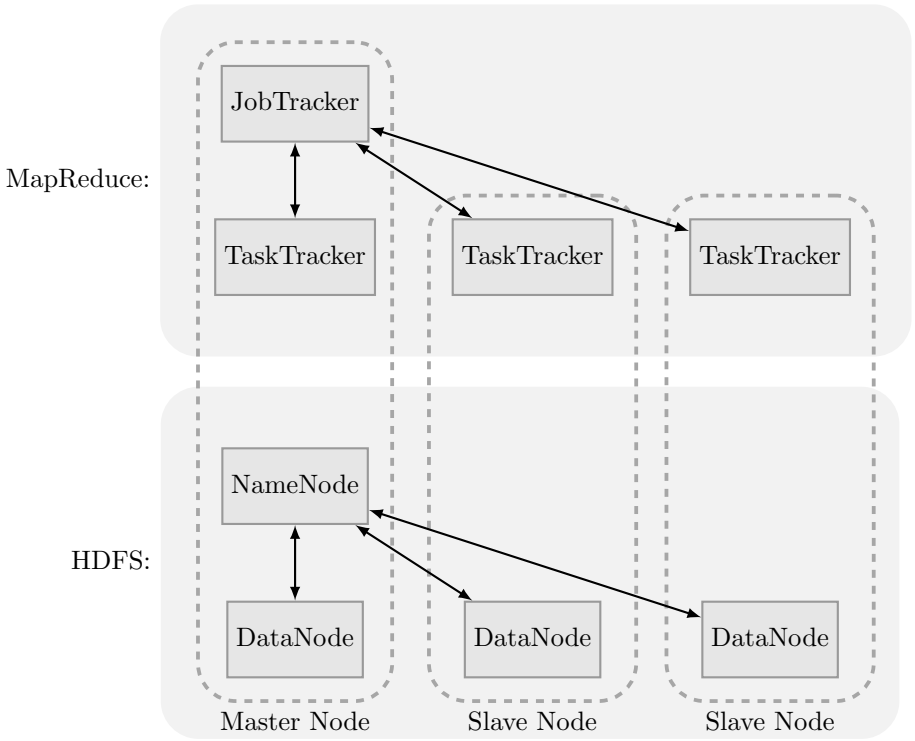

Figure 2.2: High Level Architecture of Hadoop

easier to choose the right thresholds given the partitioning technique. Sampling, however, comes with an extra cost of pre-processing the data, increasing the computational time of the algorithm. Hence, the number of random objects for sampling should guarantee that the overall computational time gets improved. This trade-off is not easy to be thoroughly examined, due to the randomness of sampling, along with the different types of datasets and queries.

2.3 Partitioning in HDFS

When using the Hadoop environment, the traditional databases need to be partitioned into data blocks for parallel calculations. The two major problems that a partitioning technique should overcome are the data skew and the boundary object problem[16]. Regarding spatial data partitioning, the majority of the approaches in the literature are based on a) location of the objects, b) spatial grid cells [17], or 3 space-filling curves[18]. Other factors also affect the partitioning result, namely, the size of the objects, their distribution, their number, and the query type. Thus, different partitioning techniques may have varying performance depending on the dataset and query operation.

2.3.1 Supported Frameworks

HDFS is a popular way of storing big data along with the MapReduce for processing it, but unfortunately, it was not designed to take into account spatial characteristics. For that reason, Hadoop should be used along with a pre-processing of the dataset, and/or other technologies when used for spatial data operations if one is interested in optimization. The Hadoop ecosystem offers some support to the processing and storing of spatial data which we are going to further discuss it in this chapter.

Parallel and distributed GIS systems have been vastly improved during the last years, but there is still a lot of room for improvements, especially in the spatial area. There are a few different frameworks that work in the HDFS as Hadoop extensions and deal with spatial data but they come with some cons. First, Hadoop-GIS [16] uses a post re-partitioning technique to deal with the data skew, which increases the computational time. Second, SpatialHadoop [10], uses random sampling for the partitioning part, which may lead to over/under-sized blocks. Third, Kangaroo[19] has quite a few partitioning techniques based on k-d trees and depending on the query type, the dataset gets re-partitioned again. Yet, only large data blocks based on the spatial grid are taken into account which leads to data balance miscalculations.

Moreover, Eagle-Eyed Elephant(E3)[11] to deal with data splits, it handles only one-dimensional spatial datasets, while 2-dimensional spatial data is more common [18].

2.3.2 Spatial Partitioning over HashPartitioning

The default partitioner that Hadoop uses is the HashPartitioner [20] which splits the data into many child blocks with a fixed block size - most commonly 64MB or 128 MB - to reduce data skew and have a good data balance in HDFS. The HashPartitioner groups together data based on their hash value, which means that in the spatial world, it can group data quite randomly. In figure 2.3 we can see a simulation on how the HashPartitioning could group the objects in comparison with a Spatial Partitioning technique.

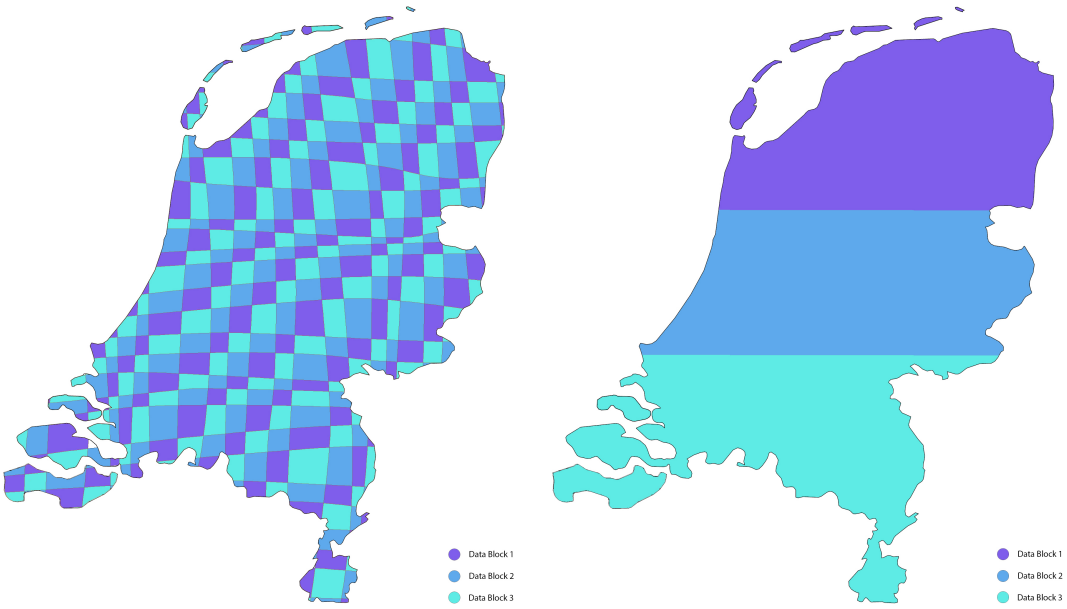

Figure 2.3: Simulation of the HashPartitioner (on the left) and Spatial Partitioning (on the right)

Let's examine the 2 partitioning methods in Figure 2.3 showing the map of the Netherlands. If a spatial query would ask to return all the objects that belong in the northern islands of the Netherlands in the first case we would have to search in every data block, since the objects were not grouped by their location. On the second case, only the data block 1 (purple) would be searched for since the partitioning reassures that all of the objects are within the block 1. With this simple example, it is easy to understand the significance of the partitioning based on object location in spatial storing.

2.4 Data structures

The spatial partitioning techniques respect the location of the spatial objects, hence they group objects if they are adjacent. Though, there are different techniques on how to split the objects and draw the borders between the data blocks. HDFS defines the size of each data block and then the partitioning algorithm has to fill these blocks with the spatial objects. Two of the most efficient partitioning techniques in traditional database systems are based on quad-trees and R-trees [7],[15]. In this chapter, we will give an illustration of these data structures.

2.4.1 Uniform Grid

Having a uniform grid partitioning is an easy way to divide the space into cells of the same length, each of them stored as a data block. This partitioning method requires very little time to be applied, but it works only if the spatial objects are scattered around the space uniformly. If the data skew is big, we will end up with unbalanced cells and the processing time for each of them may differ significantly. This technique can be used as a baseline for more complex algorithms.

In Figure 2.4 we can see the partitioning using a uniform grid. While the uniform grid is a simple implementation and works well with equally distributed objects around the space, in this example we observe the main drawback of the algorithm. The upper-left cell, for example, contains only 2 points, while the upper right one contains 13. Since the object difference in each cell is quite big, this will have an impact if each cell will be given to different nodes for further calculations. Some of them will have less work to do, while others will suffer. The result will be that the overall runtime will be based on the "slowest" nodes. If though, the cells contained the same amount of objects, each node would have undertaken a similar processing task, so there would not be a significantly slowest node and the runtime would be less.

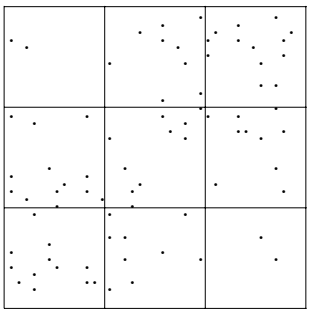

Figure 2.4: Example of dividing the space with Grid Partitioning

2.4.2 Quad-trees

Quad-trees are data structured that each internal node has exactly 4 children. The root of the tree divides the space into 4 quadrants [21]. Quad-trees are often used to decompose space into adaptable cells. If a cell reaches its maximum capacity, it has to split into 4 other sub-cells in order all of the data to fit in respecting the max size. In general, quadtrees are a cheap way to divide the space into cells without requiring many computations for this division. This benefit of quadtrees saves computation time during the partitioning phase before we go to the processing of our data. Thus, if we are interested in the complete run-time including both the partitioning time and the processing time, quadtrees come with an edge.

In Figure 2.5 there is an example of a quadtree. On the left, we can see an example of a space that contains points. In some areas, the points are more dense, while in others they are quite sparse. In our example, the maximum points that a cell can store are 8. The first division will split the space into 4 quadrants. The upper right and bottom left quadrants contain more than 8 points, so they need to split again. Finally, we have 10 cells with a different size, but all of them contain the same amount of points. On the right, we see the representation of our example in a tree form. Notice that some nodes like 2 and 4 may not have any children while others like 1 and 3 have exactly 4 children. No node can have a different amount of children than the 2 aforementioned cases.

Using quad-trees to slice the space is a relatively light procedure, since the only variables that make increase the complexity of the algorithm are the space limit and the number of objects. While the uniform grid only takes into account the space, without minding the number of objects in each cell, the quad-trees add another parameter for optimizing the capacity of objects in each cell. Comparing the complexity of the quad-tree algorithm with some data partitioning algorithms, the latter ones take into account even more parameters like the geometry of the objects and/or the size of them, making them more computationally heavy.

2.4.3 R-trees

R-trees are height-balanced trees based on B+-trees. Each node of the R-tree corresponds to the Minimum Bounded Rectangle (MBR) that bounds its children, whilst the leaves point to the database objects. [22]. Most of the time the MBRs may overlap with each other or be included in many nodes (it is only

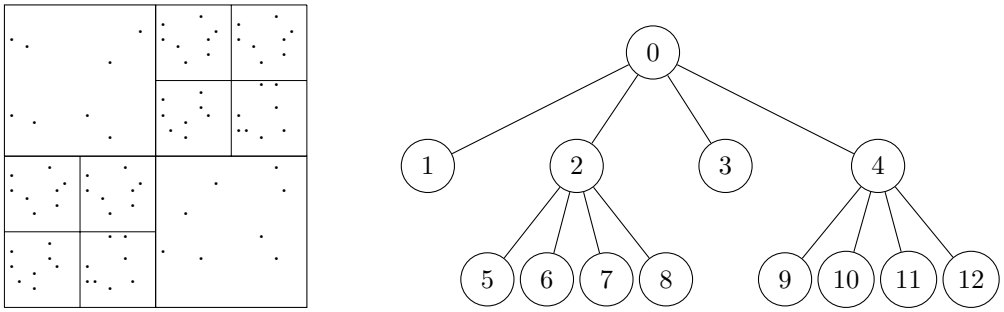

Figure 2.5: Left: Example of dividing the space with Quad-tree partitioning. Right: Representation of the Quad-Tree

associated with only one of them). Thus, a spatial query might visit many nodes when seeking for an object. An R-tree has a maximum number of leaves per node, and each node has to be at least half-filled.

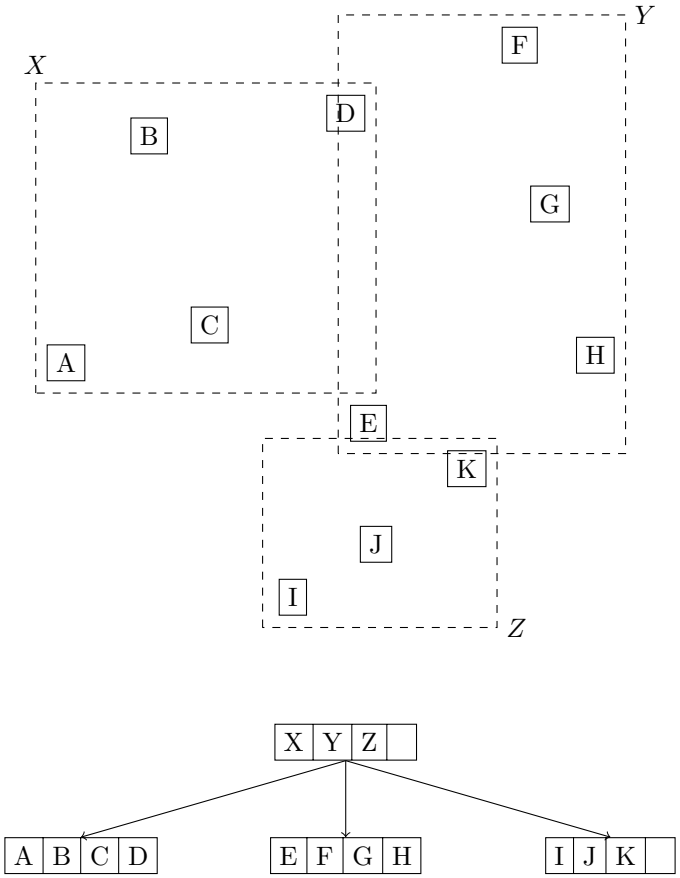

Figure 2.6: Demonstration on how we can use an R-tree to group up and store square-shaped objects.

There are multiple ways to construct an R-tree. The original algorithm creates the tree based on the sequence of the items which is a quick way to calculate the corresponding R-tree. A variation of the R-tree is the R+tree, which is somewhere between an R-tree and a kd-tree. In this variation, the nodes are not guaranteed to be at least half-filled and there is no overlap between internal nodes because we can have an object inserted in multiple leaves. Minimal coverage reduces the empty areas which are covered by the nodes of the R-tree so that efficient search can be achieved.

In figure 2.6 there is an example of squares getting grouped by in an R-tree. In this example, assuming that the maximum number of objects each partition can have is 4 and the minimum 2, we can see the corresponding MBR. Several R-trees can represent the same set of data rectangles since the resulting R-tree is determined by the insertion and deletion order of its entries [22].

Chapter 3

Partitioning Algorithms

In this chapter, we will demonstrate our partitioning methods, explain the algorithms behind each implementation and illustrate the importance of having efficient partitioning methods in spatial data. For these algorithms, we used the SpatialHadoop [10] framework, an extension for Hadoop[12], which contains all the tools for the next algorithms that we are going to describe. The four main algorithms that we are going to put into test are the hashPartitioner (the default partitioner of Hadoop), the grid-based partitioning (space partitioning), the quadtree based partitioning (space and space-filling curve partitioning) and the R-tree based partitioning (data partitioning).

3.1 HashPartitioning

The default partitioning algorithm that Hadoop has is the HashPartitioner. It is a partitioning algorithm that takes the hash code of each data entry and given the capacity of each data block, it groups together objects that have a similar hash code. This means that it completely ignores any spatial characteristic of spatial shapes, resulting in not grouping together with neighboring objects. On the other hand, partitioning takes place quite quickly because almost none preprocessing needs to be done. In figure 3.1 we can see how Hadoop implements its main partitioning method. In the next chapter, we will conduct experiments to find out if the hashPartitioner of Hadoop has efficient results compared to the other three algorithms, but our hypothesis is that since the spatial characteristics are ignored, the processing time under the hashPartitioning will be significantly higher.

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {
    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

Listing 3.1: Hadoop's hashPartitioner

3.2 Grid Partitioning Algorithm

The grid partitioning algorithm is an easy implementation of dividing the space into equal slices, where each of them contains the objects that geographically belong to it. Rather than dividing the space randomly, first, there is a sample pre-processing (approximately 1% of the total objects) where we get information about the number of the points and their relative location on the map. The sampling method, which is provided by the spatialHadoop framework [10], decides the number and the location of the partitions. All the partitions have equal size, but then they can get expanded or reduces, in order to fit whole shapes and/or contain mostly objects instead of empty space.

In Figure 3.1 we can see how the partitioning took place when using the grid algorithm. We can observe 6 partitions that on first sight they don't seem to occupy the same space, but this is because after the improvements some of the partitions got reduced forming an MBR, like the top left one, and

others had to be increased in order to contain full objects and not part of them. As a result of the latter, we have some overlapping partitions but they do not contain the same objects since each object is only stored once to its corresponding partition. We can also conclude that the grid partitioning algorithm is rather a simple implementation and quick to run, but it does not take into account the data skew and the density of the objects resulting in rather unbalanced partitions. It could be efficient for objects scattered around symmetrically, but this is not the case in most datasets.

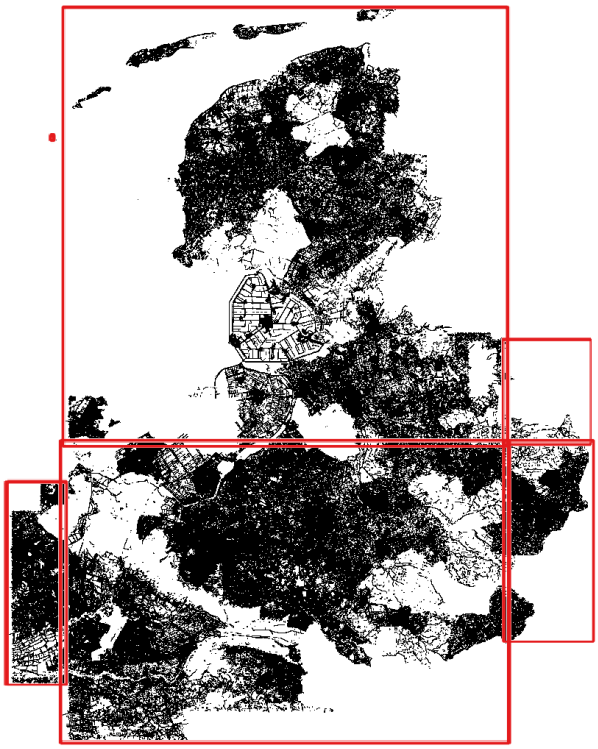

Figure 3.1: Example of how the Grid partitioning algorithm divides a space of spatial objects.

3.3 Quadtree Partitioning Algorithm

Next, we are going to describe the quadtree-based algorithm we used which is a variation of the quadtree partitioning we described in chapter 2. Before the main algorithm is applied, there is a pre-indexing of the data based on the Z-curve of our space. Thus, to explain the quadtree-based algorithm, first, we will explain the Z-curve Partitioner of SpatialHadoop. Below we explain the 3 most common space-filling curves that are used for indexing 2-dimensional spatial objects into one-dimensional ordering.

3.3.1 Space filling curves

Space-filling curves are widely used in the design of indexes for spatial and temporal data. The key metric for a space-filling curve is the clustering, which measures how well the curve preserves locality in moving from higher dimensions to a single dimension [23]. In the 2-dimensional space, spatial coding using space-filling curves is a specific implementation method of spatial data structure in a standard database [24], which indexes the cells regarding their topology in a single number ascending order. The most common space-filling curves that are used in spatial indexing are the Peano, Hilbert, and Z-curve. Among those, the Z-curve is frequently used for space partitioning in literature [10], [18] due to its simple implementation and the quick one-dimension indexing that provides for easier processing.

- **Peano curve:** The Peano curve was the first to be discovered by Giuseppe Peano [25], and some authors use the term "Peano curve" regarding any space-filling curve. An indexing value (1,2...n) is

given to each cell, enabling the indexing of a 2-dimensional space into a growing series of numbers. The construction code of a Peano curve can be seen in Listing A.1 but since it is rather complicated, we will demonstrate an example of Peano's curve which can be seen in Figure 3.2. In this example, the Peano curve starting from the bottom left cell uses trident-shape processing to order all of the cells. This type of a space-filling curve is a continuous, surjective but not injective function, meaning that for each cell will be given an indexing value, while not all cells have a unique one. Through each consecutive step, only the x or the y value changes, but not both at the same time.

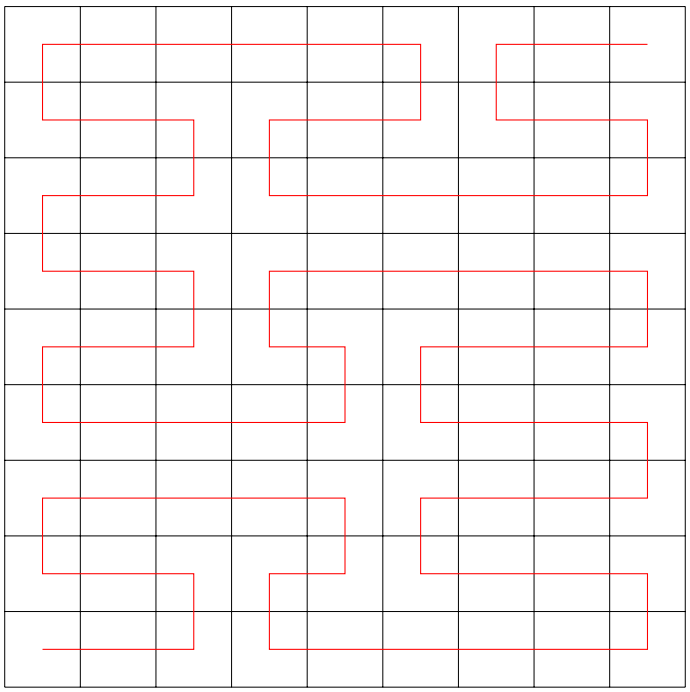

Figure 3.2: Representation of the Peano space filling curve

- **Hilbert curve:** is a continuous fractal space-filling curve first described by the German mathematician David Hilbert in 1891 [26], as a variant of the space-filling Peano curves discovered by Giuseppe Peano in 1890 [25]. The Hilbert curve is also fractal and self-similar. If we zoom in at one section of a higher-order curve, the pattern is the same. Like the Peano curve, the Hilbert one is also surjective and through each consecutive step, only the x or the y value change at a time. The construction code of the Hilbert curve can be seen in Listing reflist:hilbertCode, but since it is complicated as well, we will illustrate an example in Figure 3.3. Here we can see an implementation of the curve in a 2-dimensional space starting from the upper left cell and by using a wrench shape it processes all of the cells.

- **Z-curve:** The Z-curve or Morton space-filling curve was named after Guy MacDonald Morton [27], who first applied this order to file sequencing. The z-curve, like the previous space-filling curves, maps multidimensional data to one dimension while preserving locality of the data points. The z-value of an object is calculated by interleaving the binary representations of its coordinate values. The z-value pre-processes the object ordering them in the one-dimension, making it easy for simple data structures to undertake them such as binary trees, B-trees or in our case quadtrees. In Figure 3.4 there is an illustration of how the z-curve applies the ordering in a space.

3.3.2 Quadtree based partitioning

The idea behind the second partitioning algorithm is that it is based on a quadtree after an object indexing based on the Z-curve, along with some improvements in merging cells if they contain few

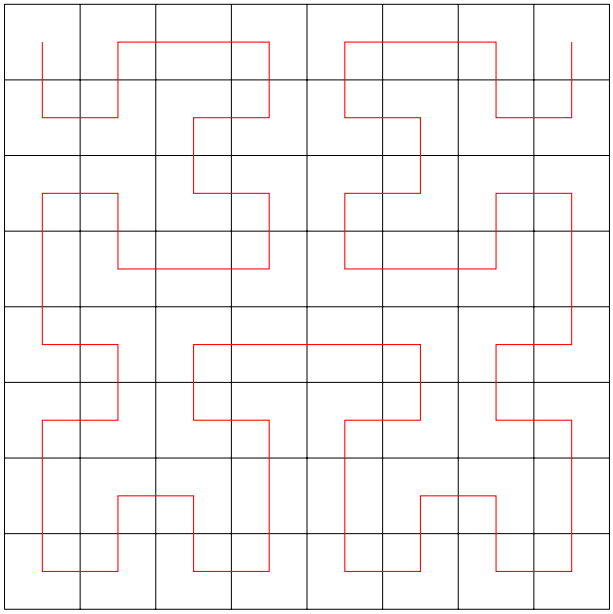

Figure 3.3: Representation of the Hilbert space filling curve

objects or deleting cells if they contain none. Furthermore, some partitions overlap with each other, in order to contain full shapes and no cut some shapes that are close to the border in half. The Z-curve was chosen because it has been extensively used in literature for pre-ordering spatial objects to be stored in a quadtree [15], because the resulting ordering can equivalently be described as the order one would get from a depth-first traversal of a quadtree. All of the sample objects are inserted into a quadtree with node capacity of $\lfloor k/n \rfloor$, where k is the sample size, while the boundaries of all leaf nodes are used as cell boundaries.

In Listing 3.2 we can see some simplified pseudo-code on how the algorithm is implemented. Note that we may have overlapping partitions depending on the data set we are trying to store. In Figure 3.5 we can see an example of a dataset with points, lines and polygons, and how it gets partitioned by the quadtree based algorithm in order to be stored in HDFS.

```
// Order the objects based on the z-curve
for (i=0; i<object_num; i++) {
    z_value[i]=object[i].calculate_z_value();
}
arrays.sort(z_values);

// Create the quad tree
create_quad_tree(z_values);
for (i=0; i<partition_num; i++) {
    // calculate MRB for each partition and expand
    // if required in order to store whole shapes
    partition[i].calculate_MRB();

    // Split partition if exceeds capacity
    if (partition[i].objects > capacity){
        partition[i].split();
        configure_partitions();
    }

    // Delete empty partitions
    else if (partition[i].empty()) {
        partition[i].delete();
    }

    // Merge partitions
```

```
        if (partition[i].objects_num + neighbor_partition[i].objects_num < capacity) {
            merge_partitions();
            configure_partitions();
        }
    }
```

Listing 3.2: Pseudo code for the object ordering in the quadtree partitioning algorithm.

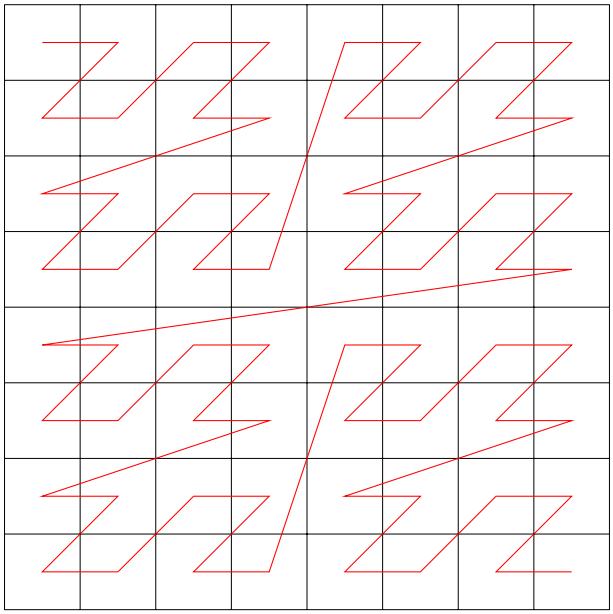

Figure 3.4: Representation of the Z space filling curve

3.4 R-tree Partitioning Algorithm

There are many different ways to create r-trees from a list of objects. The general form of the algorithm, as presented in [28] contains 3 core steps:

1. Pre-processing stage: We pre-process the data so that the total (T) objects are ordered in $\lceil T/b \rceil$ consecutive groups of b rectangles, where each group of b is intended to be placed in the same leaf level node. The b number is chosen based on the capacity of the R-tree and it is possible that the last group can contain less than b rectangles.

2. Calculating Minimum Bounding Rectangles: Next to the grouping follows the Minimum Bounding Rectangles (MBRs), assigned with a page-number. For each group of data, we calculate the borders of the cell by finding the MBR of the containing objects and we assign a page-number which be used as a child pointer in the nodes of the next higher level.

3. Finalize node packing: The final step is to pack the MBRs into nodes at the next level recursively until the root node is created.

In our research, we used a variation of the general algorithm which is also presented in [28]. this variation is called Sort-Tile-Recursive (STR) and has shown better results than the original algorithm and some other variations [28]. In general, in comparison with the Hilbert sort algorithm (HS, sorting according to the Hilbert curve) and the Nearest-X (NX, where the rectangle are sorted by x-coordinate), the STR algorithm gives better results. In detail, STR requires almost half of the disc access comparing with the HS, both for point and region queries [28]. Whilst the NX algorithm performs as well as STR

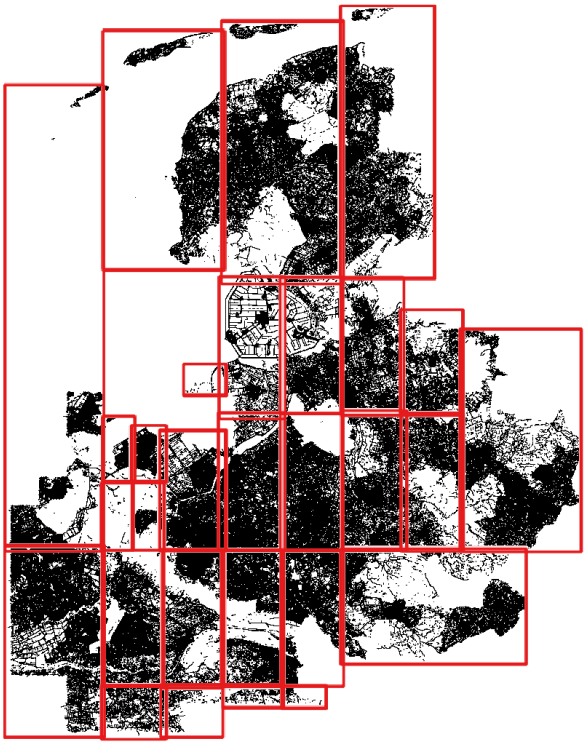

Figure 3.5: Example of how the quadtree based partitioning algorithm divides a space of spatial objects.

for point queries on point data, its performance is significantly worse for point queries on region data or region queries [28]. For these reasons, we chose to use a partitioning based on the STR algorithm.

3.4.1 STR-based Algorithm for R-trees

The idea behind this algorithm is to first divide the space into vertical slices so that each slice contains enough objects (polygons or points) to pack roughly $\sqrt{r/b}$ nodes, where r is the total amount of the spatial objects and b is the capacity of the data block. It follows the pattern of the original aforementioned algorithm, but it differentiates in how the objects are ordered at each level. In Listing 3.3 there is the pseudo-code for the object ordering in the STR algorithm. First, since the number of the leaf level pages P is $\lceil r/b \rceil$ we calculate the number of vertical slices $\sqrt{r/b}$ by having our objects sorted by the x-axis. Each slice consists of $\lceil \sqrt{\lceil r/b \rceil} \rceil$ consecutive objects from the sorted list, except for the last slice which may contain less. After we have created the slices, we sort the objects by the y-axis in each slice, so that we can assign nodes to each slide. The first b objects ordered by the y-axis will belong to the first node, the next b to the second, etc.

```
// Sorting points by the x-axis and calculate the MBR
Points.sort_by_x();
mbr = calculateMBR();

// Calculate the number of rows and columns
splitsNum = ceiling(points.length / capacity);
columns = rows = ceil(sqrt(numSplits));

// Calculate the boundaries of each column
for (i=0; i<columns; i++) {
    column[i] = points[i*capacity].x;
    points_at_column[i].sort_by_y();
```

```
        //Calculate the rows in this slice
        for (j=0; j<rows; j++){
            row[i*rows +j] = points_at_column[i][rows*j];
        }
    }
```

Listing 3.3: Pseudo code for the object ordering in the STR partitioning algorithm.

In Figure 3.6 we can see an illustration of the STR algorithm and the partitions it creates. In this example, the space contains polygons, points, and lines (building installation) and provides a clear visualization on how the algorithm works. We can see that areas with dense objects tend to have more and smaller partitions, while sparse areas have less and bigger ones. Furthermore, we can observe that a lot of the partitions overlap with others, while not all of the space is covered in partitions. This optimization helps in order to have more accurate data stored and save computation time from checking in this direction.

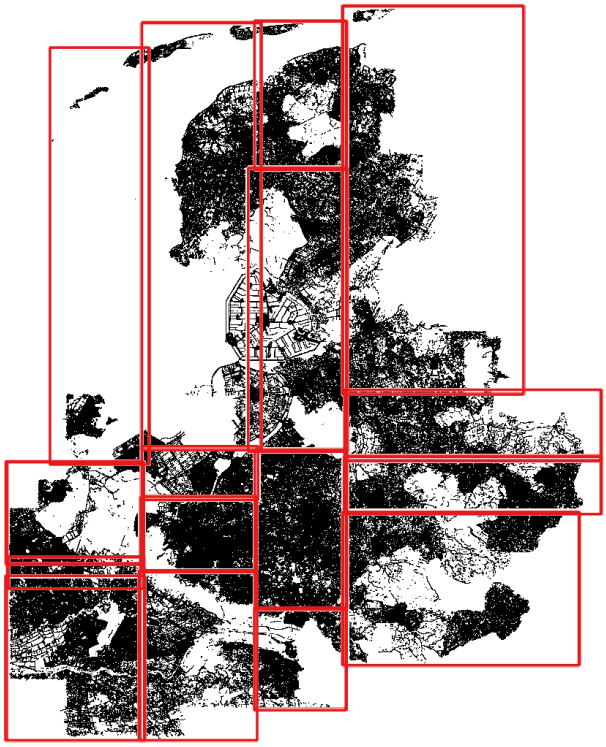

Figure 3.6: Example of how the STR-based partitioning algorithm divides a space of spatial objects.

Chapter 4

Experiments and Discussion

In this chapter, we demonstrate the experiments that were conducted, in order to test the efficiency of the partitioning algorithms that were mentioned in the previous chapters. In the experiments, the 4 partitioning algorithms that were introduced in chapter 3 were put to test; the hashPrtitioner, universal grid, R-tree and quadtree based algorithm. The processing type that was chosen is the range queries and we will further discuss it in this chapter.

4.1 Experimental setup

In this section, we are going to describe the procedure of the experiments were conducted. In Figure 4.1 we can see an outline of the steps that were conducted in order to experiment with the aforementioned algorithms. In the next sections, we will describe in detail these steps and we will give an illustration of the whole process.

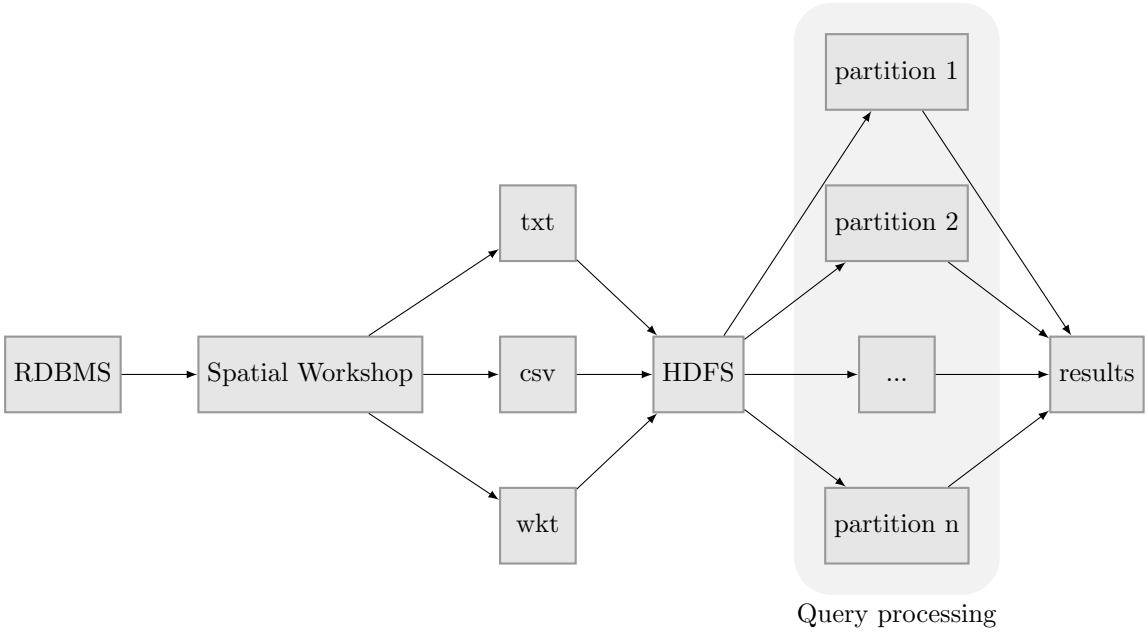

Figure 4.1: The experiments procedure

4.1.1 Data provision

The spatial data that was used for the following experiments was provided by our cooperation with the company Spatial Eye. The company possesses big real data from various customers along with some public spatial data. Such kind of data contains spatial objects of various shapes like points, (multi)lines, (multi)curves, multi(polygons), etc and vary from roads and building installations to complete water or

electricity networks. For our experiments, we used some public data that the company obtains, mostly for sensitivity issues, but also because they provide a variety of objects, shapes and data skew.

Spatial Eye stores all the data into relational databases. The Hadoop ecosystem provides support to directly obtain data from relational databases with Apache Scoop [29]. For our experiments, though we decided to import txt, wkt and csv files to be more flexible in choosing and modifying the data that we want to use. Another reason is that in our theoretical approach we do not aim on a fully automated system that processes data, but on manual experimentation in order to draw some results about how the partitioning technique affects the efficiency of the querying. Thus, we used the product of Spatial Eye called the Spatial Workshop, were we could export multiple data into various format types.

Hadoop and HDFS work better with files that each line can be read individually. For this reason, we chose the format types of txt, wkt, and csv because they can store each spatial object in one line. Hadoop then can read multiple files at the same time and store the items in the HDFS since, the order of the reading does not matter. Hence, after extracting the required data from the product Spatial Workshop into the aforementioned format types, then we imported this data into HDFS. In addition, by using the partitioning algorithms that we put in the test, we create data blocks in Hadoop, where each stores one partition of our datasets. Each different partitioning algorithm creates different data blocks and groups together the objects in a unique way. Finally, after we conduct multiple queries with different parameters, MapReduce gathers all the results.

4.1.2 Query type

The query type that was used for the experiments is the range query. The range query returns all of the items in a specific location. Such a location can be a shape like a circle or a polygon and it returns all of the items that completely fit that area. In Figure 4.2 we can see multiple range queries in the shape of a rectangle, starting from the center of the map and growing until it covers the whole of it.

The range queries are not very computational heavy since they mostly require I/O operations. On the other hand, the spatial joins require a lot of CPU computation along with I/O operations resulting in stressing the system to trigger results under extreme circumstances. We aim to demonstrate if there will be a significant difference in the results if we conduct more complex queries, or if the results will be independent of the CPU usage.

4.1.3 Testing system

At first, the development of this project took place in a single-node Hadoop system in windows 10 Pro, and afterward, it was migrated in a virtual machine under the Linux Ubuntu 17.10 operational system, where it was using a pseudo multinode simulation. However, in order to obtain reliable results, we migrated once more our project to the DAS-4 system [30]. DAS-4 (The Distributed ASCI Supercomputer 4) is a six-cluster wide-area distributed system designed by the Advanced School for Computing and Imaging (ASCI) [31]. We used a Hadoop cluster containing 8 nodes in total, one NameNode, and 7 DataNodes. Each node shares the same specifications:

- **Processor:** dual quad-core, 2.4 GHz,
- **RAM:** 24 GB and
- **Storage:** 30 TB

4.1.4 Datasets

For the datasets to be tested we chose public data over client data, so that there will not be any copyrights issue. We chose datasets that contain points, lines and (multi)polygons all together. More specifically, dataset A includes building installations, overgrowth terrain, barren terrain and supporting water part of the Netherlands, containing 1.1 GB of data. Dataset B contains spatial data about the road and water sections of the Netherlands along with some point-type objects (poles). Its size is 1.58 GB of data and the area that it covers is similar to the first dataset. We made sure that the chosen datasets will contain at least 3 different types of shapes to increase the complexity and examine how the partitioning algorithms behave under diverse spatial objects. Furthermore, we separated the data into two datasets in order to investigate the behavior of the partitions created. We wanted both our datasets to refer to the same area (the Netherlands) and because of the different object numbers between the two datasets, we can examine how the partitioning methods are affected by the density of the data.

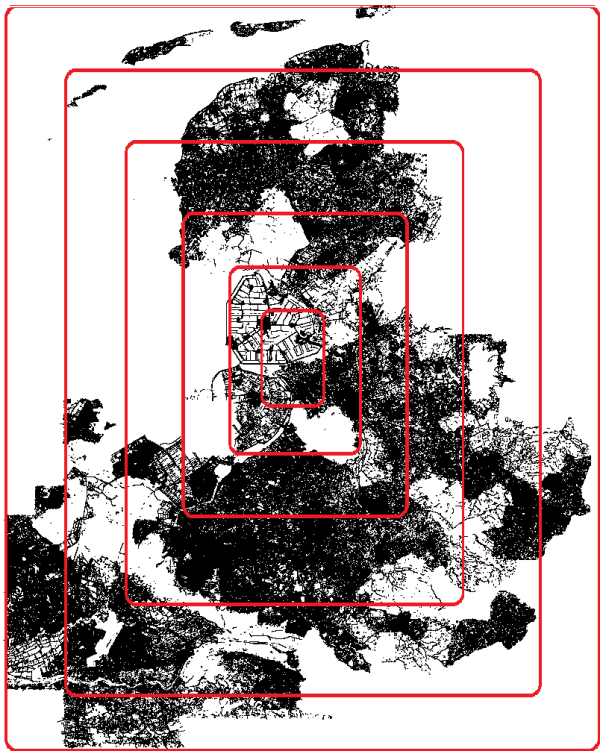

Figure 4.2: Multiple growing range queries strating from the center of a dataset, each time containing more objects.

4.2 Results

In this section, we show the results of our experiments. More specifically, we provide data on how the partitions were formed under different datasets and partitioning methods, performance tests along with the time needed for each partitioning technique.

4.2.1 Partitions

In Figure 4.3 we can see how the Grid-based partitioning algorithm divided the space into partitions. The visualization of the data between the two datasets may seem similar, but they contain a lot of different objects. Hence, the two partitioning results show some diversity. As we can see, the grid partitioning algorithm is highly dependent on the data skew, and in the datasets A and B, it does not create cells with a similar amount of spatial objects. If the objects were equally distributed to the space, each partition would contain roughly the same amount of data.

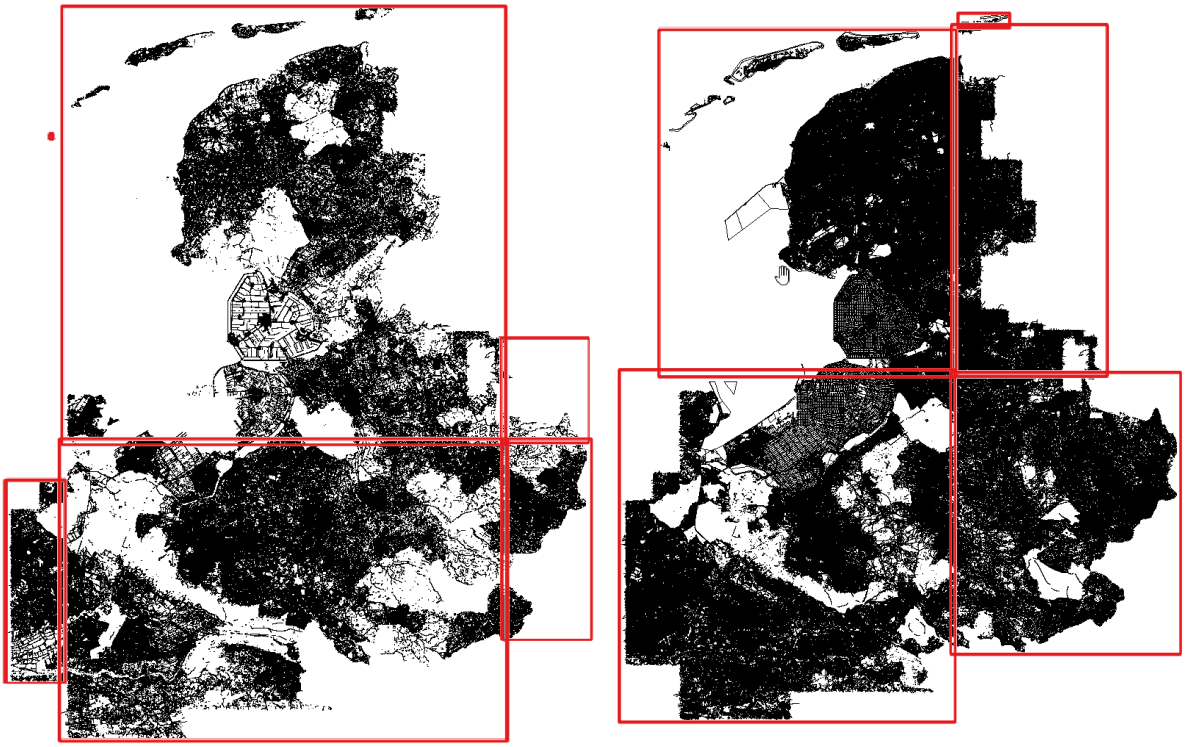

Figure 4.3: Grid partitions of dataset A (on the left) and dataset B (on the right).

In Figure 4.4 we can observe the two datasets being partitioned by the same quadtree based algorithm. We also notice that in dataset B some areas are denser and as a result, we have more partitions. If we pay close attention we will see that the partitioning structure is quite similar, but for dataset B, we have more divisions due to the bigger amount of data. Another observation that one can make is that a lot of the partitions are of the same size. This happens to the dense areas, where the quadtree based algorithm keeps dividing each cell into 4 smaller ones until it reaches a cell size where the objects can be stored to the data block of Hadoop.

Finally in Figure 4.5 we can observe the behavior of the STR partitioning algorithm in the datasets A and B. Similarly, we can notice a similar partitioning structure, but since the datasets differentiate the partitions change accordingly. The amount of cells does not change, but the dimensions of each partition slightly change. After all, both datasets contain data of the Netherlands, hence it is rational that the 2 datasets share a lot of similarities.

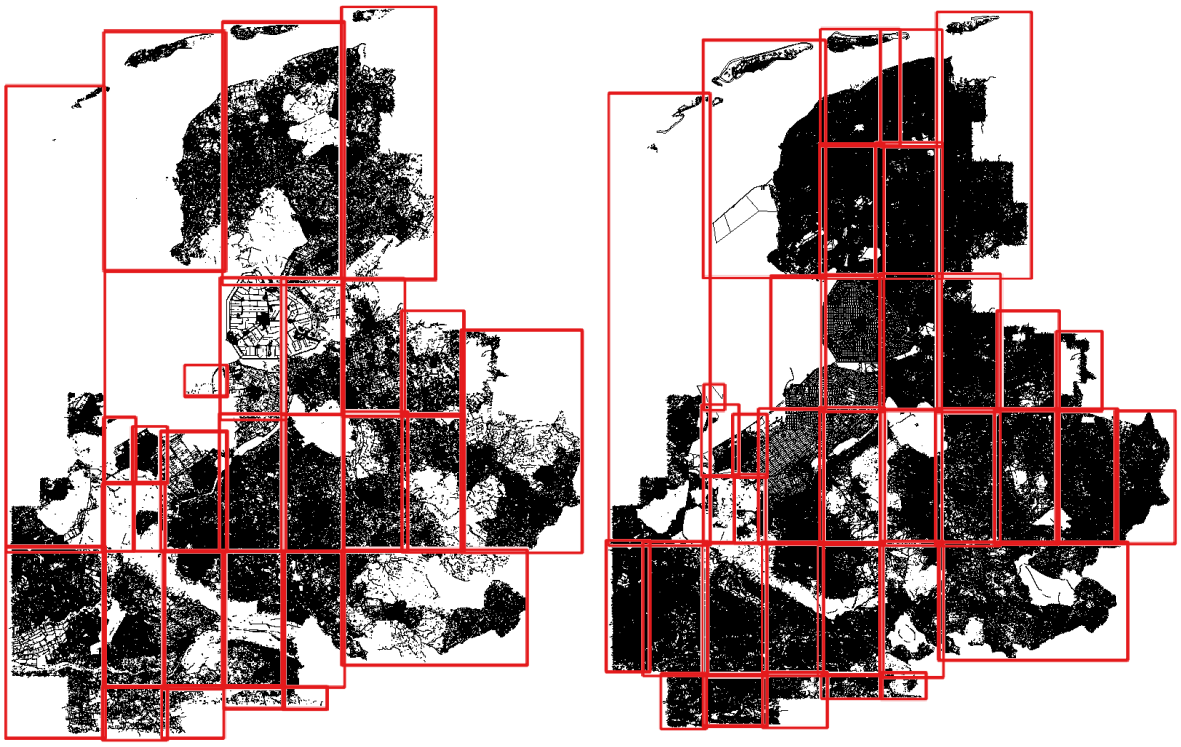

Figure 4.4: Quadtree partitions of dataset A (on the left) and dataset B (on the right).

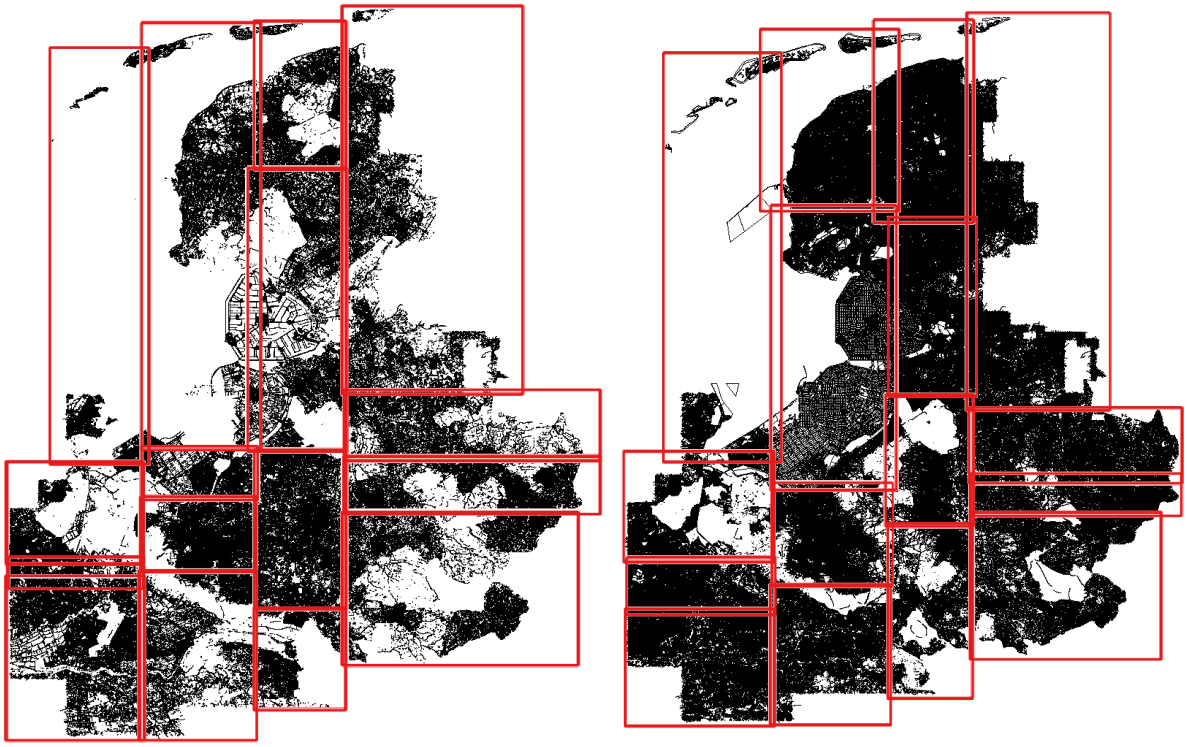

Figure 4.5: STR partitions of dataset A (on the left) and dataset B (on the right).

4.2.2 Performance Tests

In this section, we provide the results of our performance tests and we discuss the important findings after each test.

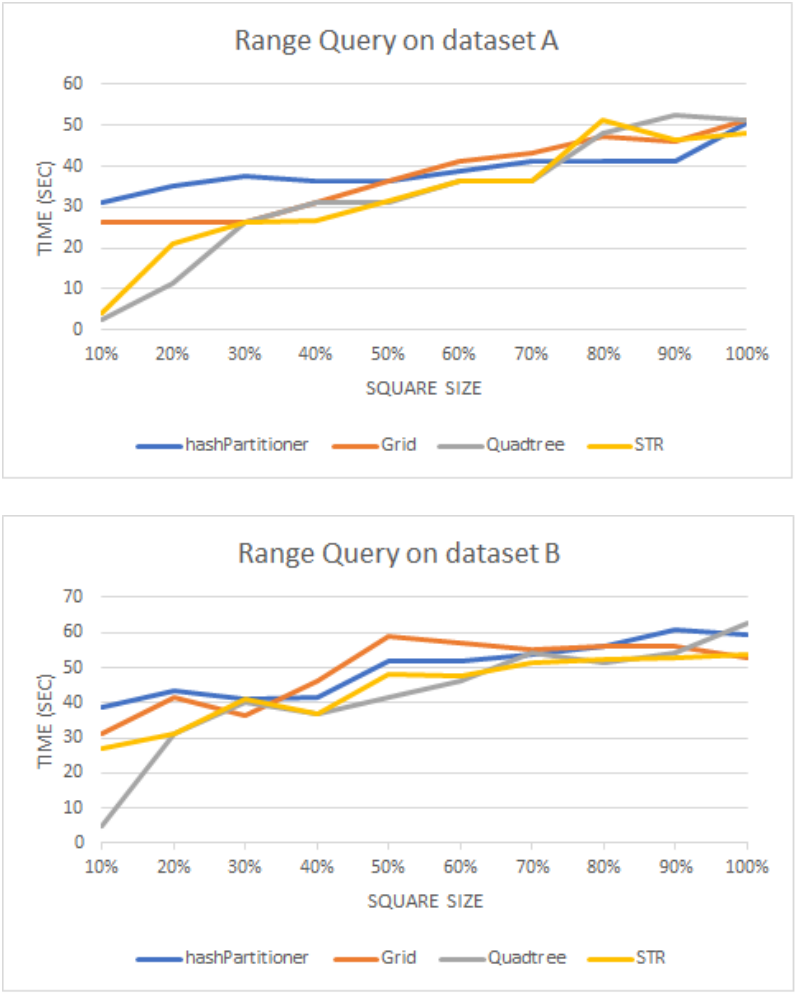

Figure 4.6: Performance test with range queries starting from the center fitting a percentage of the total space.

Quadtree partitioning:

Based on the findings shown in Figure 4.6 we can observe that for range queries that affect up to 70% of the spatial objects the quadtree based algorithm demonstrates the best results regarding the runtime of the processing. The quadtree along with STR partitioning slice the space in many more partitions than the other two methods, saving a lot of computation time if we are concerned about a specific part of the dataset. For queries that affect 70% - 100% of the partitioning it seems that the more partitions, the harder to organize them, resulting sometimes in a greater runtime even than the hashPartitioner. If we want to process (almost) all the data, we cannot reduce the computations required and as a result, all of the objects will be processed. The cells of the quadtree based algorithm are designed to have the ability to skip unwanted computations, but in this case, it seems that it rather incommodes the procedure by having imbalanced partitions. Such partitions will require different processing time, thus some data nodes will have more work to do, while others not.

STR partitioning:

Again, by looking at the 4.6 we can see that the STR method also demonstrates good results, following the quadtree based method. More specifically, for an object coverage up to 70%, the STR algorithm

handles the data efficiently and comes second after the quadtree partitioning with some small exceptions. For queries affecting the 70% until the 100% of the population, the STR algorithm shows the same drawback of the quadtree algorithm, but it behaves slightly better than the latter. We assume that this behavior is because of the fewer partitions it has over the quadtree based algorithm, but also because the very nature of the STR algorithm makes it divide the space into partitions with a roughly equal number of objects.

Grid partitioning:

The grid-based partitioning method displays the most imbalanced results. Since the datasets that we examine do not have objects equally distributed in the space, the grid algorithm does not have the best results that could have. In detail, based on Figure 4.6 for a data coverage up to 30% it allows the processing to be quicker than the hashPartitioner, but slower than the other two algorithms. In addition, for data coverage ranging from 30% to 80% it demonstrates the worst efficiency, while for range queries affecting 80% till 100% of the population, it has an average behavior.

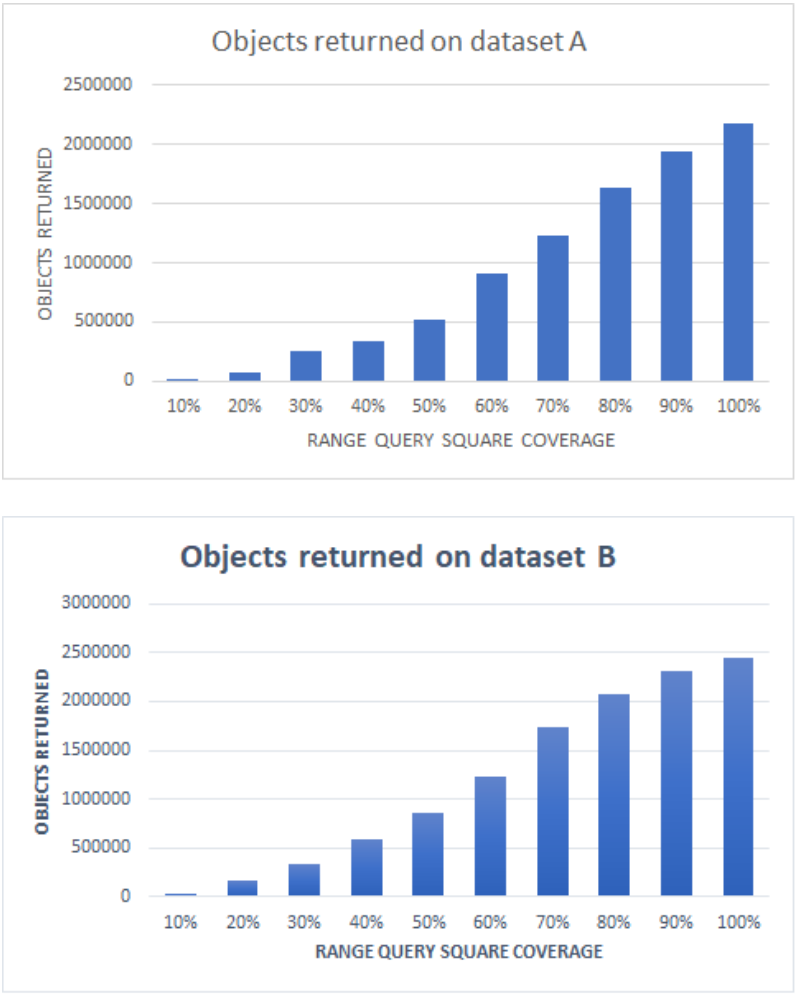

Figure 4.7: Number of objects returned by applying the range queries.

HashPartitioning:

The default partitioning method of Hadoop, the hashPartitioner, exhibits the worst behavior in most cases verifying our hypothesis. Since the hashPartitioner completely ignores the spatial characteristics of the two datasets, every range query has to search in all of the data blocks. This has as a result that

no matter the coverage percentage of the population, the processing of the data to be similar. In Figure 4.6, we detect that throughout the growing coverage percentage, the execution time of the queries for the hashPartitioning method does not differentiate a lot and it is only slowly changing. This happens because of the more I/O operations that are required by the bigger range queries. In Figure 4.7 we can see the number of objects returned for each of the range queries and rationally we can justify the slight increase in the execution time of the hashPartitioning algorithm. Moreover, hashPartitioner makes sure that each partition has roughly the same amount of data. For this reason when we reach 90% - 100% of the coverage it can exhibit similar or even better results than the other algorithms.

4.2.3 Partitions matched

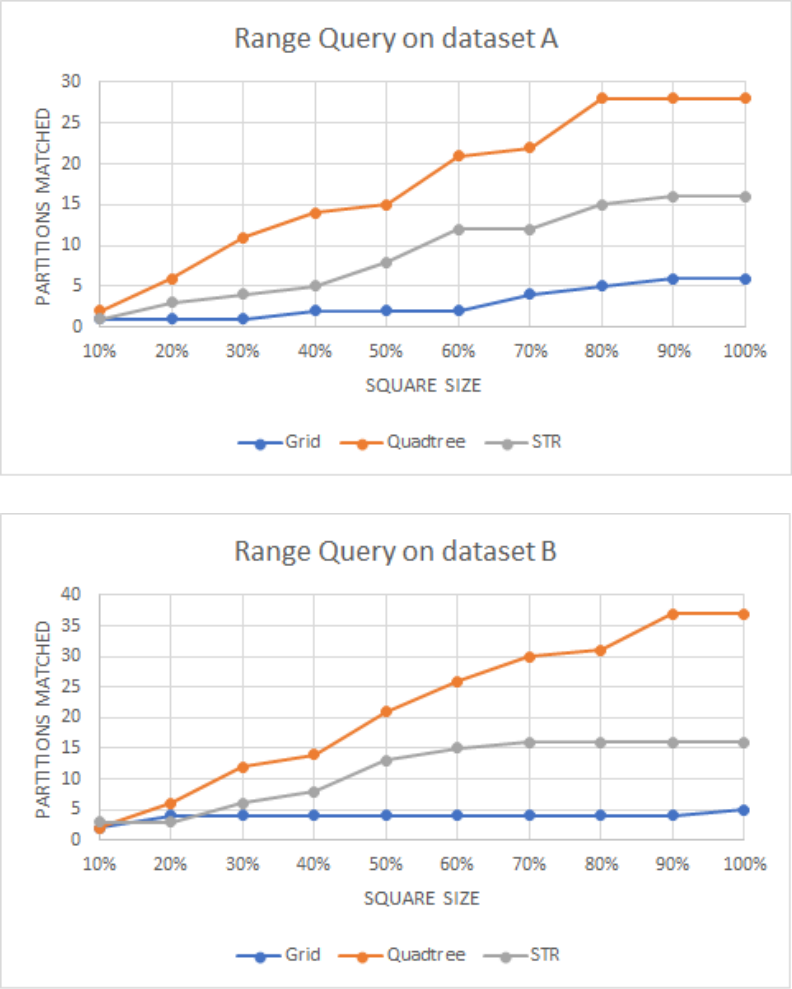

Figure 4.8: Overlapping partitions with range queries starting from the center fitting a percentage of the total space.

Quadtree partitioning:

In Figure 4.9 we can observe the number of partitions that each range query matches. For the quadtree based method, we notice that the partitions matched for each query are roughly twice the number of the STR algorithm. The same conclusion can be drawn from Figure 3.5, since there are many more partitions for the quadtree algorithm than the other ones. This attribute comes with some pros and cons. The main difference is that more partitions can smartly avoid extra computations since they index the space in a better way, but collecting the results of all of them comes more costly. For this reason, we assume that this behavior makes the processing of the high percentages of the population slower since

it includes the coordination of the many partitions. Finally, the time required for applying the quadtree based partitioning is not significantly different than the other two algorithms as we can see in Figure 4.9.

STR partitioning:

The STR algorithm has a relatively average number of partitions (roughly half of the ones that the quadtree based algorithm has). This configuration gives the former the ability to build up a more stable behavior when the object coverage of the query reaches 100%, since the cost for waiting and gathering the results from all the data nodes will be significantly lower. Last, in Figure 4.9 we notice that the STR algorithm requires the most amount of time to be applied, but the difference among the other methods is almost insignificant. By design, the STR partitioning algorithm has a more complex implementation and requires more computations than the other ones.

Grid partitioning:

If we look now at the number of the partitions that the grid-based partitioning algorithm shows in Figure 4.9, we will see that it has the least ones. Since they fit in HDFS' data block, there is no reason for further dividing the cells when applying the grid algorithm, resulting in imbalanced cells with a lot of objects. Finally, the advantage of the grid algorithm over the others is that it can be applied quicker (Figure 4.9) but again the difference in the execution time is not significant.

HashPartitioner:

There is no reason for showing the partitions matched for the hashPartitioner, since for each query all of the data blocks have to be processed. Since this type of partitioning does not group objects together based on their location, no data block will be excluded from the querying.

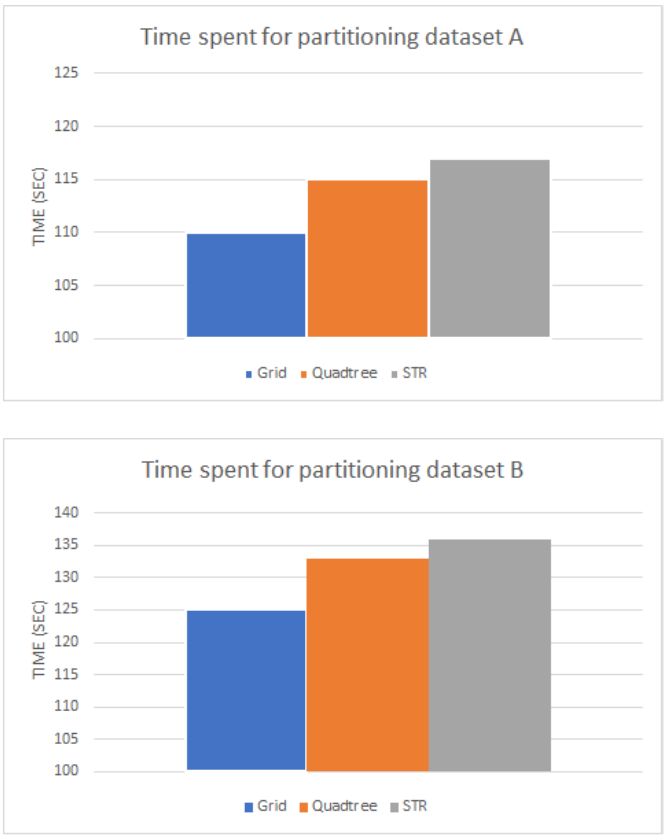

Figure 4.9: Runtime for applying the partitioning methods.

4.3 RQ: How can spatial partitioning methods help to scale up the processing of big spatial data?

In this section, we will answer the research question along with the subquestion that we introduced in chapter 1. We have proven that spatial partitioning is useful over traditional partitioning when dealing with spatial data in the HDFS environment. When the processing coverage of the dataset is less than 70%, spatial partitioning facilitates the query procedure by skipping unwanted calculations. Spatial partitioning can reduce the execution time of processing in multiple cases, and if one's dataset consists of mere spatial objects, indexing it with a spatial partitioning algorithm would possibly boost the performance of the system.

What are the best data structures that are used in traditional spatial databases?

We have answered this question in chapter 2. Two of the most efficient partitioning techniques that exist in traditional database systems are based on quadtrees, r-trees, r+-trees and kd-trees [7], [15]. These data structures demonstrate quite an efficient behavior since they were mostly designed to store spatial data. In our research, inspired by these structures, we examined if they are also efficient in distributed file systems.

Do these methods scale up well in multinode systems processing data in parallel?

In file distributed systems working with many different nodes, it is obligatory to divide the data into smaller parts for parallel computation. We have proven that spatial partitioning of the data based on the best data structures that are used in traditional spatial databases (quad-tree and R-tree) provide better results to the most of the testing cases over partitioning it disregarding its spatial nature. Quad-trees and R-trees showed that they are reliable sources of a baseline for dividing the space into multiple slices and process the data in parallel since in the majority of the cases they outperform the default data partitioning method of Hadoop and the grid partitioning method.

Chapter 5

Conclusion

As the big data emerges in many applications, efficient ways of processing it need to be found. When it comes to spatial data, different rules apply since one needs to also take into account spatial characteristics such as geometry and location of the spatial objects. The growing trend of migrating from traditional database systems to cloud computing is followed by big enterprises. One such solution is the Hadoop environment along with its ecosystem that offers great support for various kinds of data.

We chose the Hadoop Distributed File System (HDFS) and frameworks like spatialHadoop to examine how one can efficiently process spatial data with common queries. The HDFS requires data partitioning to execute computations in parallel, thus partitioning the big data is obligatory. The way that the data will be partitioned may affect the runtime of its processing. We investigated how 4 partitioning methods behave in the HDFS environment regarding the execution time of range queries. The 4 partitioning algorithms that were put to test are the default partitioning method of Hadoop, the hashPartitioner and three other partitioning algorithms using the spatialHadoop framework, the grid-based, quad-tree based and R-tree (STR) based partitioning algorithms.

We have proven that for most of the test cases, a spatial partitioning makes the processing of the data quicker, with some exceptions of the grid partitioning, since it is highly dependent on the uniform data skew. More specifically, we showed that the quad-tree based partitioning demonstrates the best results regarding the runtime of the processing of the data, followed by the STR algorithm, especially when the processing affects less than 70% of the dataset. When the dataset coverage reaches 100%, the partition methods with the more partitions (such as quad-tree) suffer from coordination problems of the multiple resulting sets, increasing slightly the runtime, while the rest of them have an average behavior. For data coverage less than 30% the quad-tree based algorithm outperforms the others, followed by the STR method with a significant difference.

In general, our findings show that for datasets that consist of spatial data, a spatial partitioning method is more efficient in most cases in the HDFS environment. In addition, the data structures that are used in the traditional database systems can efficiently group spatial data in distributed file systems and be used for spatial partitioning. Finally, we showed that even in parallel systems, spatial partitioning outperforms traditional partitioning techniques for big spatial datasets.

5.1 Future work

This research only set a small step in the area of spatial partitioning to efficiently process spatial data. During this research, we have identified several directions for future work. If we had more time throughout this research we would investigate the following points of interest:

Apache Sqoop:

During this research, we focused on the actual research questions rather than on implementing an automated program that could improve the runtime of processing big spatial data. For that reason simple input and output formats were used, instead of dragging the data directly from the relational database using Apache Sqoop [29]. That would be something interesting for further implementation along with other improvements that could turn this project from manual to automated and as a result to be able to be used by individuals interested in migrating from an RDBMS to Hadoop.

Spatial and Alphanumeric data:

Furthermore, we were focused on datasets containing only spatial data, and we proved that spatial partitioning boosts the efficiency of its processing. It would be also challenging to observe the behavior of a dataset containing a hybrid of spatial and alphanumeric data. Queries such as "Find all buildings in the Netherlands that were built after the year 2000" will increase the complexity and a spatial partitioning may not be the best solution. In that case, a mixture of spatial and alphanumeric partitioning techniques could also demonstrate efficient behavior.

More partitioning Algorithms:

Additionally, one could observe the behavior of other partitioning algorithms. We examined 2 of the most promising ones along with the simple implementation of the grid partitioning and the default partitioner of Hadoop. However, there are many other partitioning algorithms in the literature and potentially many more that haven't been implemented.

Furthermore, some other variants would be interesting for someone to take into account and research them as well. The complexity of the outcome though might significantly affect the purpose of this master project.

Multidimensional Data:

It would be an interesting approach if 3-dimensional or multi-dimensional objects were put to test under these algorithms. Partitioning the space in the 3 dimensions can be challenging, while the rules that apply in the 2 dimensions may change. Moreover, it would be interesting to observe the behavior of these algorithms with a dataset that is a mixture of alphanumeric and multi-dimensional spatial data.

Future guide for migration to Hadoop:

Finally, this project could be the start for a future guide on how to migrate one's spatial database from a traditional system to Hadoop, and how to process the data quicker. Regarding the different behavior of the spatial data over the alphanumeric one, one should focus on different things depending on their dataset.

Acknowledgements

First, I would like to thank my thesis supervisor dr. Adam Belloum. The door to his office was always open whenever I ran into a trouble spot or I had questions about my research or writing. His encouragement and guidance was invaluable in the process of this research.

I would also like to thank the company Spatial Eye for having me as an intern providing me with resources and technical knowledge without which the thesis progress would be significantly slower. Moreover, I want to thank Jasper van Rooijen who was always available to help me when I needed help and gave me the freedom to take the path I was most interested in throughout this research.

Finally, I am thankful to my family and friends for their endless support and encouragement that they gave me this whole year and without them, focusing on this research would have been more difficult by far.

Appendix A

Non-crucial information

```
    /*Abhishek Ghosh, 14th September 2018*/

#include <graphics.h>
#include <math.h>

void Peano(int x, int y, int lg, int i1, int i2) {

  if (lg == 1) {
    lineto(3*x,3*y);
    return;
  }

  lg = lg/3;
  Peano(x+(2*i1*lg), y+(2*i1*lg), lg, i1, i2);
  Peano(x+((i1-i2+1)*lg), y+((i1+i2)*lg), lg, i1, 1-i2);
  Peano(x+lg, y+lg, lg, i1, 1-i2);
  Peano(x+((i1+i2)*lg), y+((i1-i2+1)*lg), lg, 1-i1, 1-i2);
  Peano(x+(2*i2*lg), y+(2*(1-i2)*lg), lg, i1, i2);
  Peano(x+((1+i2-i1)*lg), y+((2-i1-i2)*lg), lg, i1, i2);
  Peano(x+(2*(1-i1)*lg), y+(2*(1-i1)*lg), lg, i1, i2);
  Peano(x+((2-i1-i2)*lg), y+((1+i2-i1)*lg), lg, 1-i1, i2);
  Peano(x+(2*(1-i2)*lg), y+(2*i2*lg), lg, 1-i1, i2);
}

int main(void) {

  initwindow(1000,1000,"Peano, Peano");

  Peano(0, 0, 1000, 0, 0); /* Start Peano recursion. */

  getch();
  cleardevice();

  return 0;
}
```

Listing A.1: The Peano curve implementation code in C [32]

```
#include <stdio.h>

#define N 32
#define K 3
#define MAX N * K

typedef struct { int x; int y; } point;

void rot(int n, point *p, int rx, int ry) {
    int t;
    if (!ry) {
        if (rx == 1) {
            p->x = n - 1 - p->x;
```

```
            p->y = n - 1 - p->y;
        }
        t = p->x;
        p->x = p->y;
        p->y = t;
    }
}

void d2pt(int n, int d, point *p) {
    int s = 1, t = d, rx, ry;
    p->x = 0;
    p->y = 0;
    while (s < n) {
        rx = 1 & (t / 2);
        ry = 1 & (t ^ rx);
        rot(s, p, rx, ry);
        p->x += s * rx;
        p->y += s * ry;
        t /= 4;
        s *= 2;
    }
}

int main() {
    int d, x, y, cx, cy, px, py;
    char pts[MAX][MAX];
    point curr, prev;
    for (x = 0; x < MAX; ++x)
        for (y = 0; y < MAX; ++y) pts[x][y] = ' ';
    prev.x = prev.y = 0;
    pts[0][0] = '.';
    for (d = 1; d < N * N; ++d) {
        d2pt(N, d, &curr);
        cx = curr.x * K;
        cy = curr.y * K;
        px = prev.x * K;
        py = prev.y * K;
        pts[cx][cy] = '.';
        if (cx == px ) {
            if (py < cy)
                for (y = py + 1; y < cy; ++y) pts[cx][y] = '|';
            else
                for (y = cy + 1; y < py; ++y) pts[cx][y] = '|';
        }
        else {
            if (px < cx)
                for (x = px + 1; x < cx; ++x) pts[x][cy] = '_';
            else
                for (x = cx + 1; x < px; ++x) pts[x][cy] = '_';
        }
        prev = curr;
    }
    for (x = 0; x < MAX; ++x) {
        for (y = 0; y < MAX; ++y) printf("%c", pts[y][x]);
        printf("\n");
    }
    return 0;
}
```

Listing A.2: The Hilbert curve implementation code in C [33]

Bibliography

[1] P. Rigaux, M. Scholl, and A. Voisard, "Spatial databases: With application to gis.", *San Francisco: Morgan Kaufmann*, 2011.

[2] A. Guttman, "R-trees: A dynamic index structure for spatial searching.", *ACM SIGMOD Intl. Conference on Management of Data*, 1984.

[3] T. Brinkhoff, H. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees.", *ACM SIGMOD Intl. Conference on Management of Data, 1993*, 1993.

[4] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases. acm transactions on database systems (tods).", *ACM Transactions On Database Systems (TODS)*, vol. 24, no. 2, 1999.

[5] J. Zhang, D. Mamoulis, D. Papadias, and T. Yufei, "All-nearest-neighbors queries in spatial databases.", 2004.

[6] V. Gaede and O. Günther, "Multidimensional access methods", *ACM Comput. Surv.*, vol. 30, no. 2, pp. 170–231, Jun. 1998, ISSN: 0360-0300.

[7] R. Han, L. K. John, and J. Zhan, "Benchmarking big data systems: A review", *IEEE Transactions on Services Computing*, vol. 11, no. 3, pp. 580–597, 2018.

[8] D. Glushkova, P. Jovanovic, and A. Abelló, "Mapreduce performance model for hadoop 2.x", *Information Systems*, vol. 79, pp. 32–43, 2019, Special issue on DOLAP 2017: Design, Optimization, Languages and Analytical Processing of Big Data, ISSN: 0306-4379. DOI: `https://doi.org/10.1016/j.is.2017.11.006`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0306437917304659`.

[9] A. W. Yassien and A. F. Desouky, "Rdbms, nosql, hadoop: A performance-based empirical analysis", in *Proceedings of the 2Nd Africa and Middle East Conference on Software Engineering*, ser. AMECSE '16, Cairo, Egypt: ACM, 2016, pp. 52–59, ISBN: 978-1-4503-4293-3. DOI: `10.1145/2944165.2944174`.

[10] A. Eldawy and M. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data.", 2015.

[11] M. Eltabakh, F. Özcan, Y. Sismanis, P. Haas, and H. Pirahesh, "Eagle-eyed elephant: Split-oriented indexing in hadoop.", 2013, pp. 89–100.

[12] *Apache hadoop*, `https://hadoop.apache.org/`, Accessed: 20/04/2019, 2018.

[13] J. Ullman, "Designing good mapreduce algorithms", eng, *XRDS: Crossroads, The ACM Magazine for Students*, vol. 19, no. 1, pp. 30, 34, 2012, ISSN: 1528-4980.

[14] F. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. Ullman, "Map-reduce extensions and recursive queries", eng, ser. EDBT/ICDT '11, ACM, 2011, pp. 1–8, ISBN: 9781450305280.

[15] A. Eldawy, L. Alarabi, and M. F. Mokbel, "Spatial partitioning techniques in spatialhadoop", eng, 12, vol. 8, 2015, pp. 1602, 1605.

[16] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop-gis: A high performance spatial data warehousing system over mapreduce.", 2013.

[17] L. Ma and X. Zhang, "A computing method for spatial accessibility based on grid partition.", *Geoinformatics 2007: Geospatial Information Science*, 2007.

[18] J. Hungershöfer and J.-M. Wierum, "On the quality of partitions based on space-filling curves.", *Computational Science ICCS*, 2002.

[19] A. Aly, H. Elmeleegy, Y. Qi, and W. Aref, "Kangaroo: Workload-aware processing of range data and range queries in hadoop.", 2016, pp. 397–406.

[20] L. Liu, "Computing infrastructure for big data processing", *Frontiers of Computer Science*, vol. 7, no. 2, pp. 165–170, Apr. 2013, ISSN: 2095-2236. DOI: 10.1007/s11704-013-3900-x. [Online]. Available: https://doi.org/10.1007/s11704-013-3900-x.

[21] R. Finkel and J. Bentley, "Quad trees a data structure for retrieval on composite keys", eng, *Acta Informatica*, vol. 4, no. 1, pp. 1, 9, 1974, ISSN: 0001-5903.

[22] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, "Introduction", in *R-Trees: Theory and Applications*. London: Springer London, 2006, pp. 3–13, ISBN: 978-1-84628-293-5. DOI: 10.1007/978-1-84628-293-5_1. [Online]. Available: https://doi.org/10.1007/978-1-84628-293-5_1.

[23] P. Xu, C. Nguyen, and S. Tirthapura, "Onion curve: A space filling curve with near-optimal clustering", *CoRR*, vol. abs/1801.07399, 2018. arXiv: 1801.07399. [Online]. Available: http://arxiv.org/abs/1801.07399.

[24] P. van Oosterom and T. Vijlbrief, "Str: A simple and efficient algorithm for r-tree packing", eng, in *The spatial location code, In: Proceedings of the 7th international symposium on spatial data handling*, Delft, 1996.

[25] G. Peano, "Sur une courbe, qui remplit toute une aire plane", *Mathematische Annalen*, vol. 36, no. 1, pp. 157–160, Mar. 1890, ISSN: 1432-1807. DOI: 10.1007/BF01199438. [Online]. Available: https://doi.org/10.1007/BF01199438.

[26] D. Hilbert, "Über die stetige abbildung einer linie auf ein flächenstück", *Mathematische Annalen*, vol. 38, no. 1, pp. 459–460, 1891.

[27] G. M. Morton, "A computer oriented geodetic data base; and a new technique in file sequencing", *IBM Ltd.*, 1966.

[28] S. Leutenegger, M. Lopez, and J. Edgington, "Str: A simple and efficient algorithm for r-tree packing", eng, in *Proceedings 13th International Conference on Data Engineering*, IEEE, 1997, pp. 497–506.

[29] *Apache sqoop*, https://sqoop.apache.org/, Accessed: 20/04/2019, 2019.

[30] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A medium-scale distributed system for computer science research: Infrastructure for the long term", *IEEE Computer*, vol. 49, no. 5, pp. 54–63, May 2016.

[31] *Das-4 overview*, https://www.cs.vu.nl/das4/home.shtml, Accessed: 8/06/2019, 2016.

[32] G. Abhishek, *Peano curve*, https://rosettacode.org/wiki/Peano_curve, Accessed: 25/06/2019, 2018.

[33] *Hilbert curve*, https://rosettacode.org/wiki/Hilbert_curve#C, Accessed: 25/06/2019, 2018.