UNIVERSITY OF CALIFORNIA
RIVERSIDE

Raptor: Large Scale Processing of Big Raster + Vector Data

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

Doctor of Philosophy

in

Computer Science

by

Samriddhi Singla

June 2022

Dissertation Committee:

Dr. Ahmed Eldawy, Chairperson
Dr. Vassilis Tsotras
Dr. Jiasi Chen
Dr. Elia Scudiero

The Dissertation of Samriddhi Singla is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my advisor, Dr. Ahmed Eldawy, without whose help and guidance, I would not have been here. Thank you for your support, encouragement, and patience. I especially would like to thank my dissertation committee members: Dr. Vassilis J. Tsotras, Dr. Jiasi Chen, and Dr. Elia Scudiero, for reviewing my dissertation. I also would like to thank my collaborators: Tina Diao of Standford University, Ayan Mukhopadhyay, and Michael Wilbur of Vanderbilt University. Finally, thank you to my colleagues at the Big Data Lab: Saheli Ghosh, Tin Vu, Akil Sevim, Xin Zhang, Zhuocheng Shang, and Vinayak Gajjewar for all their support and help.

The text of this dissertation, in part, is a reprint of the material as it appears in ACM SIGSPATIAL 2018 and 2021, ACM SIGMOD 2021, VLDB 2019, IEEE Big Data 2020, NeurIPS 2020 and 2021, and ICDE 2021. The co-author Dr. Ahmed Eldawy listed in that publication directed and supervised the research which forms the basis for this dissertation.

This research was partially supported by Agriculture and Food Research Initiative Competitive Grant no. 2019-67022-29696 from the USDA National Institute of Food and Agriculture. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect those of the USDA National Institute of Food and Agriculture.

To my parents and brother for all the support.

ABSTRACT OF THE DISSERTATION

Raptor: Large Scale Processing of Big Raster + Vector Data

by

Samriddhi Singla

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2022
Dr. Ahmed Eldawy, Chairperson

Advancements in remote sensing technology have resulted in petabytes of remote sensing data being made publicly available. The widespread use of smart devices and GPS technology has also led to the availability of highly accurate geographical features. This increase in the amount of spatial data has allowed for greater research opportunities in many scientific domains including hydrology, political science, environmental science, and agriculture. In these applications, scientists rarely base all their analysis on a single dataset but they usually need to combine multiple datasets in their analysis. Machine learning is a popular tool used by these scientists and often requires combining different datasets into a form usable by the machine learning algorithms. Spatial data is generally available in two representations, raster, and vector. The best data science and machine learning applications need to combine multiple datasets of both representations which is a data and compute-intensive problem.

My dissertation proposes a new system called Raptor that bridges the gap between raster and vector data. It is an end-to-end system for efficiently processing raster and

vector geospatial data concurrently. First, it discusses an initial approach to parallelize the zonal statistics operation called DARaptor. Second, it proposes Raptor Zonal Statistics, a system implemented in Hadoop that can be used to perform the zonal statistics operation for big raster and vector datasets. Third, it proposes Raptor Join which is modeled as a relational join operator in Spark that can be easily combined with other operators, while also offering the advantage of in-situ processing. Raptor Join is flexible to support ad-hoc applications and has been used for various real-world applications such as wildfire modeling, area interpolation, and crop yield mapping. Finally, this work proposes RDPro to add distributed raster pre-processing capabilities to Raptor that can scale to big data. The experimental evaluation on large-scale satellite data with up to a trillion pixels, and big vector data with up to hundreds of millions of segments and billions of points has shown that the proposed system is promising and can scale to big data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The recent decade has seen an explosive increase in the amount of spatial data. The advancements in remote sensing technology have led to a tremendous increase in the amount of remote sensing data. For example, NASA EOSDIS provides public access to more than 33 petabytes of Earth Observational data and is estimated to grow to more than 330 petabytes by 2025 [17]. European Space Agency (ESA) has collected over five petabytes of data within two years of the launch of the Sentinel-1A satellite and is expected to receive data continuously until 2030 [18]. In the meantime, the proliferation of smart devices and GPS technology has led to highly accurate geographical features such as water bodies, city boundaries, roads, agricultural fields, and others.

This increase in the amount of spatial data has allowed for greater research opportunities in many scientific domains including hydrology, political science, environmental science, and agriculture. In these applications, scientists rarely base all their analysis on a single dataset but they usually need to combine multiple datasets in their analysis. Ma-

chine learning is also a popular tool used by these scientists that often requires combining different datasets into a form usable by the machine learning algorithms. This makes it necessary to pre-process data and combine it into a single data representation before it can be used by the algorithm. A few applications that need to combine raster and vector data include the study by ecologists on the effect of vegetation and temperature on human settlement [32, 33], analyzing terabytes of socio-economic and environmental data [29, 30], and studying of land use and land cover classification [57]. It can also be used for areal interpolation [56], crop yield mapping [41], and to assess the risk of wildfires [24, 34].

Spatial data can be classified into two data representations: vector and raster. Satellite imagery is an example of raster data and is usually represented in form of multi-dimensional arrays. Vector data is represented as a set of points, lines, and polygons, and is used to represent geographical features such as regional boundaries, roads, and water bodies. The major differences between these two representations makes combining them difficult. This is why existing systems are designed to either process vector data [80, 14, 45, 37, 52] or raster data [21, 3, 27, 70, 51]. These systems are efficient for combining vector-vector or raster-raster but are limited when the need to combine raster and vector data arises.

In this dissertation proposal, we propose a new system called *Raptor* that can bridge the gap between raster and vector data. It is an end-to-end system for efficiently processing raster and vector geospatial data concurrently. First, it discusses an initial attempt, *DARaptor* to parallelize the zonal statistics operation. Second, it proposes *Raptor Zonal Statistics*, a fully-distributed system implemented in Hadoop that can be used to perform the zonal statistics operation for big raster and vector datasets. Third, it proposes

Raptor Join which is modeled as a relational join operator in Spark that can be easily combined with other operators, while also offering the advantage of in-situ processing. Raptor Join is flexible to support ad-hoc applications and has been used for various real-world applications such as wildfire modeling, area interpolation, and crop yield mapping. Finally, we propose RDPro to add distributed raster pre-processing capabilities to Raptor that can scale to big data.

The rest of this dissertation proposal is organized as follows: Section 2 describes the first attempt to implement a distributed method to tackle the zonal statistics operation, called *DARaptor*. It builds the foundation for Section 3 that proposes a distributed method to tackle the zonal statistics operation, called *Raptor Zonal Statistics*. Section 4 proposes the *Raptor Join* which allows can be used for ad-hoc applications not limited to zonal statistics. Section 5 discusses the work to add distributed raster pre-processing capabilities to *Raptor*. Section 6 details the real-world applications where the proposed systems were used. Section 7 concludes the proposal and discusses the future work.

Chapter 2

DARaptor: Distributed Zonal Aggregation of Big Raster + Vector Data

This chapter talks about *DARaptor*, a distributed implementation in Hadoop that was a first attempt to efficiently compute zonal statistics.

2.1 Introduction

Remote Sensing Data is of vital importance to various research domains, such as agriculture, environmental studies, and oceanography. It has been used to study climate change, model biogeochemical cycles, map land and vegetation change, and has numerous other applications. Recently, there has been a tremendous increase in the amount of this data with the advancements in remote sensing technology. NASA EOSDIS provides public

access to more than 17 petabytes of Earth Observational data, which is estimated to grow to more than 330 petabytes by 2025 [17]. European Space Agency(ESA) has collected over five petabytes of data within two years of the launch of the Sentinel-1A satellite and is expected to receive data continuously until 2030 [18]. Other than the data collected by space agencies, the European XEFL project collects X-ray images of atoms at a rate of up 10 petabytes per year [77].

The remote sensing data is in the raster format, and its use requires it to be often processed in combination with vector data. Zonal Statistics is one such spatial operation that requires to process the combination of raster and vector data to compute statistics for a zone defined by the vector data using the values provided by the raster data. It is used in many applications, including the study by ecologists on the effect of vegetation and temperature on human settlement [32, 33] and by geographers for analyzing terabytes of socio-economic and environmental data [29, 30].

To make use of the ever-growing amount of spatial data, there is a need for scalable distributed techniques that can efficiently process it. The existing systems for big spatial data include SpatialHadoop [14], GeoSpark [80], Simba [78], SciDB [70], RasDaMan [3], and GeoTrellis [36]. The above-mentioned systems are very efficient, however, they focus on either processing big raster data [3, 36, 70] or big vector data [14, 45, 76, 78, 80], and provide a poor performance when the combination of vector and raster data needs to be processed.

Traditional methods to process the zonal statistics problem focused on either vectorizing the raster dataset [86] or rasterizing the vector data [29]. Both suffer from the

drawback of running a costly conversion process which makes them unscalable to high-resolution raster and vector data. To overcome this drawback, the ScanLine [16] method was proposed recently which processes the two datasets in their native format without a need for a conversion process. It proved to be very efficient in producing the best performance on a single machine but it was still limited to the resources available on a single machine.

In this chapter, we study the problem of distributed zonal statistics on high-resolution raster and vector data. We first show that a straight-forward naïve parallelization of the ScanLine method is inefficient due to four limitations. First, and most importantly, since the ScanLine method works on the raw vector and raster data, each machine repeats the same process which imposes a lot of redundant work that slows down the process. Second, distributed execution frameworks, such as Hadoop and Spark, are not designed to parallelize vector and raster datasets efficiently, rather, they work well with text or specific binary formats. Third, it could be inefficient because distributed frameworks are designed to process entire files while in the case of zonal statistics there might be entire regions in the raster files that do not overlap any regions. Finally, the basic ScanLine method relies on loading the entire vector layer in memory and hence cannot scale to large vector datasets.

To overcome the above limitation, we propose DARaptor, an efficient implementation in Hadoop for the zonal statistics problem. The algorithm runs in two phases, namely, preparation and aggregation phases. The preparation phase runs on a single machine and efficiently performs the common logic that is needed by all the machines. At the same time, it gets the chance to look into the metadata of the raster and vector files and decide how

to efficiently split the job across machines. The second aggregation phase is then launched as a MapReduce job that scans the relevant parts of the raster files and computes the desired statistics efficiently. This design overcomes the four limitations described above and scales well to large data. First, the preparation phase carries out the common processing which computes the intersection points between the vector and raster data which allows the worker machines to scan and aggregate the data in parallel without repeating these steps. Second, we introduce a novel RaptorInputFormat which efficiently split the job into tasks that take into account both the raster and vector characteristics. Third, by looking into the metadata of the input file, the proposed algorithm can prune irrelevant parts in the raster files with minimal overhead. Finally, the proposed design splits both vector and raster datasets making it scalable for both big vector and raster files.

We carry out an extensive experimental evaluation on large-scale real raster datasets of around a trillion pixels, and big vector datasets of up to 50 million segments and we show that the proposed method is scalable to high-resolution raster and vector data and outperforms the baseline methods by up-to an order of magnitude.

The rest of this chapter is organized as follows. Section 2.2 covers the related work in literature. Section 2.3 gives a background and problem definition. Section 2.4 describes the Naïve Implementation and the proposed DARaptor Implementation. Section 2.5 provides an extensive experimental evaluation. Finally, Section 2.6 concludes the chapter.

2.2 Related Work

In this section, we cover the relevant work in the literature. First, we give an overview of big spatial data systems and classify them according to whether they primarily target vector data, raster data, or both. After that, we cover the work that specifically targets the zonal statistics problem.

2.2.1 Big Vector Data

In this research direction, some research efforts aimed to provide big spatial data solutions for vector data types and operations. There are several systems in this category including SpatialHadoop [14], MD-HBase [45], ESRI on Hadoop [76], GeoSpark [80], and Simba [78], among others. The work in this category covers (1) spatial indexes such as R-tree [14, 80, 78], Quad-tree [76, 45], and Grid [14], (2) spatial operations such as range query [76, 14, 80, 78, 45], k nearest neighbor [14, 80, 78, 45], spatial join [14, 80, 78], and computational geometry [10], (3) spatial data visualization including single-level and multilevel [15], and (4) high-level programming languages [13].

Vector-based systems can support the zonal statistics problem by utilizing the spatial join operation with the point-in-polygon predicate. Simply, it treats each pixel as a point and tests to find points that are inside the query polygon. The drawback of this algorithm is that it has to test a huge number of pixels which is impractical for large rasters with trillions of pixels and complex polygons with hundreds of thousands of segments.

2.2.2 Big Raster Data

Systems in this research direction focus on processing raster datasets which are represented as multidimensional arrays. Popular systems include SciDB [70], RasDaMan [3], and GeoTrellis [36]. The set of operations supported for raster datasets are completely different than those provided for vector datasets. They are usually categorized into four categories, namely, local, focal, zonal, and global operations [60]. Each operation operates on one or more multidimensional arrays and produces another multidimensional array possibly of a different size.

To support the zonal statistics operation for one polygon, a raster-based system can apply the following operations: (1) Rasterize the polygon by generating a raster layer of the same resolution of the input raster layer where a pixel has a value of one if it is inside the polygon. (2) Apply a mask operation between the input raster and the rasterized layer which is a local operation. (3) Apply the desired statistics function on the resulting masked raster. The drawback of this technique is that the rasterized dataset can be excessively large for very large raster datasets.

2.2.3 Big Raster-Vector Combination

One of the earliest works on combining raster and vector data is done in [49], which proposes a hybrid data structure to store both raster and vector data. It requires a preprocessing offline step that converts both datasets to an intermediate form before it performs any processing. Some of the current systems support both vector and raster data such as PostGIS and QGIS [54]. However, they internally rely on two isolated libraries,

one for vector and one for raster with different sets of operations for each representation. Therefore, they are still stuck with one of the two approaches described above. Wang *et al.* [75] proposed a parallel algorithm for rasterizing vector data which can be used as a preprocessing step for processing the zonal statistics problem using the raster-based method.

2.2.4 Zonal Statistics

The *zonal statistics* problem is a basic problem that is used in several domains including ecology [32, 33] and geography [29, 30]. However, there was only a little work in the query processing aspect of the problem. ArcGIS [2] supports this query by first rasterizing the polygons dataset and then overlaying it with the raster dataset. Zhang *et al* [86, 85] solve the zonal statistics problem using the point-in-polygon query and they rely on GPUs to speed up the calculation. The drawback is that it has to load the entire raster dataset in GPU memory which is a very expensive operation, especially for very large raster datasets. Recent papers by the NSF project Terra Populus [30, 29] demonstrate the complexity of the problem on big raster and big vector datasets. The ScanLine algorithm [16] was a first step in efficiently processing the zonal statistics problem by combining vector and raster data but it was limited to a single machine.

In this chapter, we propose a novel scalable algorithm for processing the zonal statistics problem on big raster and vector data using MapReduce. It is different than the work described above in three ways. 1. It leverages the MapReduce programming paradigm to scale out on multiple machines. 2. It provides a novel work distribution mechanism that combines raster plus vector (Raptor) in one unit of work. 3. It can efficiently prune non-relevant parts in the raster layer to speed up the query processing.

2.3 Background

This section provides a background on some relevant concepts from GIS and spatial databases, and the zonal statistics problem.

2.3.1 Spatial Data Representation

The two common representations of spatial data are *vector* and *raster* representations. The vector representation uses constructs like point, line, and polygon, and operations like intersect, union, and overlaps. The raster representation uses matrices as a common construct and the operations are all performed on these matrices. Each entry in the matrix is called a pixel. To map between pixels and geographic locations, two mappings are used, namely, *world-to-grid* ($\mathcal{W}2\mathcal{G}$) and *grid-to-world* ($\mathcal{G}2\mathcal{W}$). These mappings can also be used to map data between the vector and raster datasets. The algorithm in this chapter relies mainly on these two mappings to map the computation between the two representations without having to convert one of them entirely to the other representation.

2.3.2 Zonal Statistics

The input to the zonal statistics problem is a raster layer r, a vector layer v, and an accumulator *acc*. The vector layer v consists of a set of polygons which are usually disjoint. The accumulator *acc* is a user-provided function which takes pixel values, one at a time, and computes the statistics of interest. For example, one accumulator can compute the average value, while another one can compute the histogram over the spectrum of raster values. The output is a value for the accumulator for *each polygon* in the vector layer. For

example, if r represents the temperature in the world, v represents the 50 US States, and acc is an average accumulator, the output of this problem will be the average temperature for each state.

2.3.3 Single-machine ScanLine Method

The scan-line method [16] is the state-of-the-art algorithm for computing zonal statistics on a single machine. It runs in the following three steps.

Step 1 calculates the Minimum Bounding Rectangle (MBR) of the input polygon(s) and maps its two corners from world to grid co-ordinates using the $\mathcal{W}2\mathcal{G}$ mapping. These two corners help in identifying the lower and upper rows in the grid coordinates that define the range of scan lines to process.

Step 2 computes the intersections of each of the scan lines with the polygon boundaries. It converts each scan line from grid coordinates to world coordinates using $\mathcal{G}2\mathcal{W}$ mapping and stores their y-coordinates in a sorted list. Each polygon is scanned for its corresponding range of scan lines, which are then used to compute intersections with the polygon. These intersections are then sorted by their x-coordinates.

Step 3 finds the pixels that lie inside the polygons and process them. It maps the x-coordinates of the intersections from world to grid coordinates and accumulates the corresponding pixel values. For multiple polygons, all intersections in one row are processed before moving to the next row.

This approach requires a minimal amount of intermediate storage for the intersection points. It also minimizes disk IO by reading only the pixels that overlap the polygons.

2.4 Distributed Zonal Statistics

The state-of-the-art ScanLine method [16] proved to be very efficient in minimizing the memory footprint and disk IO. However, it had a major limitation of running on a single machine which makes it limited to the capabilities of that machine. This section describes the proposed distributed algorithm for zonal statistics. Section 2.4.1 below describes a naive MapReduce implementation which is a straight forward parallel implementation of the ScanLine method. After that, Section 2.4.2 describes our proposed improved MapReduce algorithm which achieves up-to an order of magnitude speedup over the baseline method. Section 2.4.3 provides further implementation considerations about our proposed improved MapReduce algorithm.

2.4.1 Naïve Implementation (NI)

A straight-forward parallel implementation of the ScanLine method is to logically partition the (big) raster file into tiles, let each machine processes its tiles using ScanLine, and finally combine the results to produce the final answer. This algorithm is implemented as a single MapReduce job that runs in three phases. In the first preparation phase, each tile in the raster file is given a unique ID which are all written as a single text file, termed *tile-ID file*, as one ID per line. The tile-ID file is split across mappers as a fixed number of lines (tiles) per mapper. The vector file, which is relatively small, is broadcast to all the machines. Second, each mapper reads a few lines of the tile-ID file and the map function applies the ScanLine method to process the corresponding tiles with the entire vector file. The output of the map function is a set of pairs $\langle p_i, a_i \rangle$, where p_i is a polygon ID and a_i

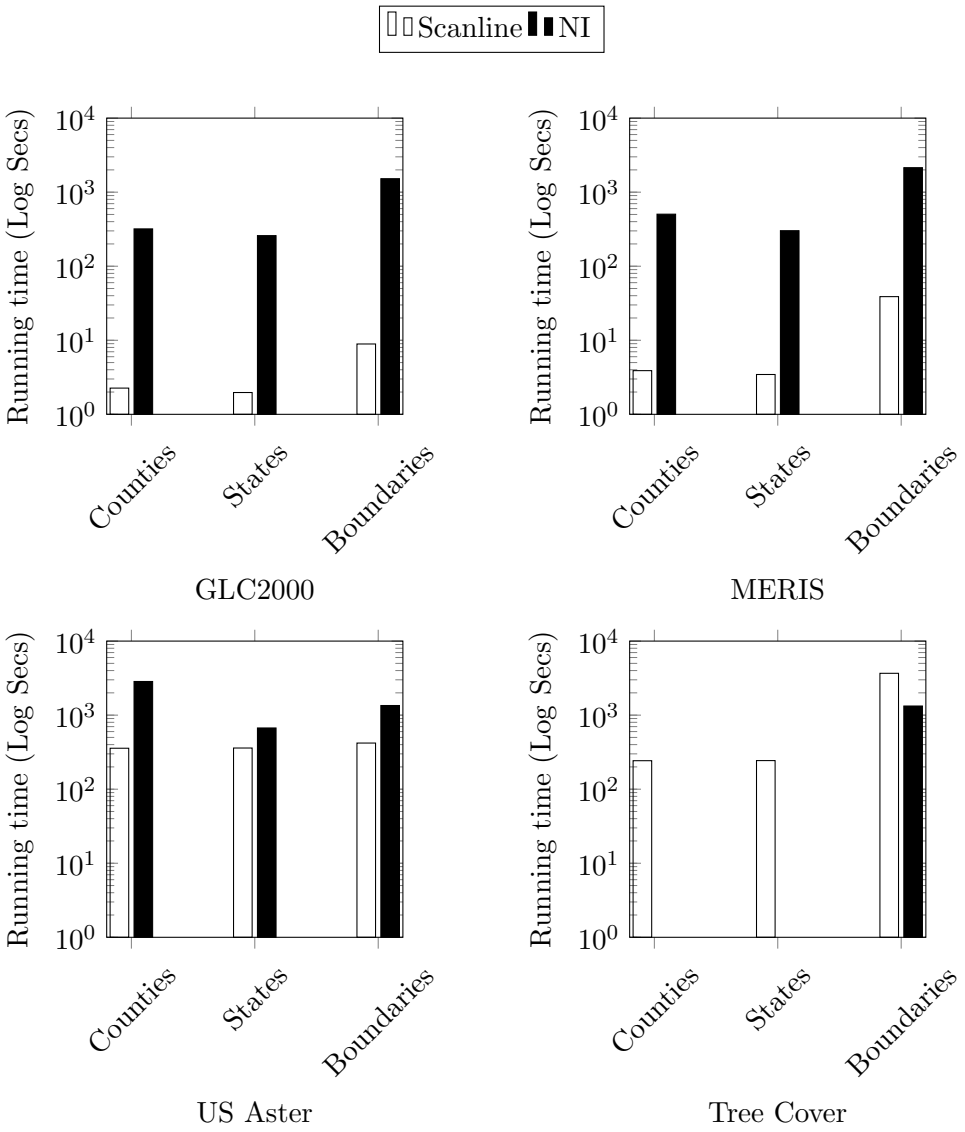


Figure 2.1: Comparison of NI and ScanLine Method

is its aggregate value that the ScanLine method computes. Finally, the reduce function receives a polygon id p_i and a set of aggregate values $A = \{a_i\}$. It aggregates all the partial aggregates and returns the final pair $\langle p_i, \sum a_i \rangle$.

Figure 2.1 compares the performance of this first-cut solution (NI) to the single-machine ScanLine method. As shown, this method is orders of magnitude slower than the single-machine algorithm for small- and medium-sized raster files. It is only *slightly* faster with a very large raster file. We omitted the running times that are more than two hours. As clearly shown in the figure, the naïve algorithm is generally slower than the ScanLine due to the major limitations briefly explained below.

1. The tile-ID file might contain unnecessary tile IDs that do not overlap with any of the polygons.

2. Mappers perform a lot of redundant work by computing the intersection of the vector file with the raster file for each tile.

3. Since each mapper processes the entire vector file, this method does not scale to big vector data due to memory limitations.

4. In the case of multiple raster files, one mapper might process tiles from multiple raster files which adds an overhead for reading and processing all these raster files.

2.4.2 DARaptor

Unlike the naïve implementation (NI) that follows a black-box approach, DARaptor takes a white-box approach to parallelize the ScanLine method. Figure 2.2 provides an

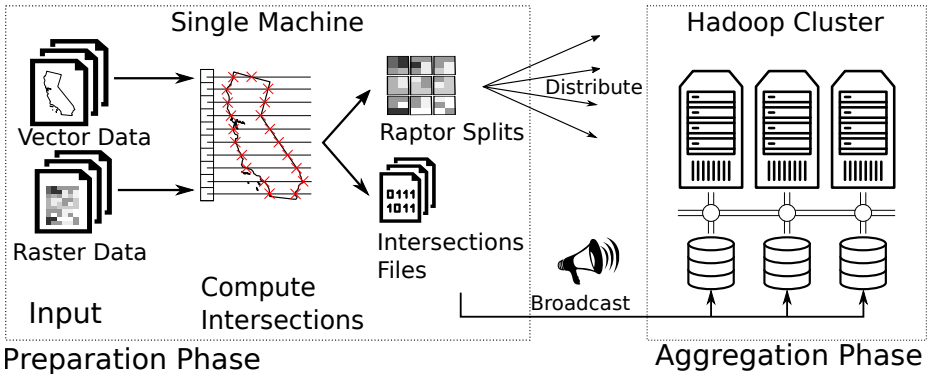


Figure 2.2: System Overview

overview of the proposed algorithm. The proposed algorithm runs in two phases which are outlined below, followed by a detailed description of each phase.

- *Preparation Phase*: As illustrated in Figure 2.2, this phase runs on a single machine and performs two tasks. First, it computes the intersections of the geometries in the vector file with the raster layer similar to the ScanLine method. These intersections are written to binary files called the *Intersection files*, which are broadcast to all the machines. Second, it defines a logical partitioning for the raster and vector files and creates RAster Plus vecTOR splits, termed *Raptor Splits*, which define the smallest units of work and are then distributed to the machines during the second phase.

- *Aggregation Phase*: This phase runs as a MapReduce job, where each mapper receives a *Raptor Split* that defines a subset of raster tiles and polygons to process. The map function reads the specified tiles and the intersection file that corresponds to the polygons and computes a partial answer for the zonal statistics. The partial answers are combined by polygon ID and the reducers combine them to produce the final aggregate.

Preparation Phase

The goal of this phase is to prepare and create the MapReduce job that computes the zonal statistics. It runs on a single machine on the head node of the cluster and performs two tasks, namely, *intersection file generation* and *raptor split generation*. The intersection file generation step computes a common structure, called intersection file, which is broadcast to all the machines to be used in the distributed processing step. The raptor split generation step creates a list of tasks that are distributed among machines to perform the parallel computation. Below, we describe the two tasks in more detail.

Intersection File Generation

The naïve algorithm has the limitation of performing redundant work by computing the intersection of the vector file with the raster file in each mapper for each assigned tile. Furthermore, it cannot scale to big vector data as it processes the entire vector file in each mapper which imposes a huge CPU and memory overhead. The first task, *intersection file generation* overcomes these two limitations by computing the intersections once and broadcasting them to all the machines. In case of a big vector file, it splits the file into small chunks of equal number of polygons which puts an upper limit on the amount of data processed by each machine. Splitting the vector file overcomes the limitation of both the ScanLine and Naïve implementation where they cannot scale to big vector data. From our previous work, we showed that the intersection computation part requires a minimal overhead as compared to the statistics computation phase. Therefore, running it once on a single machine is more efficient than running it thousands of times on multiple machines. To compute the intersections, a chunk of the vector file is loaded into memory and only the

metadata of the raster file is loaded, e.g., resolution and coordinate reference system (CRS). For each chunk of polygons, we run the first phase of the scan line algorithm which computes the intersections between the polygons and each row of pixels in the raster layer. For each row, the intersections are represented as a list of pairs $\langle x, pid \rangle$ sorted by x, where x is the coordinate of an intersection and *pid* is the ID of the polygon intersecting at that position. All these intersections are then written to a compact binary file called the *intersection file*. If multiple raster files are given in the input, this step uses multithreading to compute the intersections with each raster file in parallel. For efficiency, each thread writes its part of the intersection file independently and they are finally concatenated in one intersection file.

Figure 2.3 illustrates the structure of the intersection file. In this figure, a big vector file is split into m chunks and there are n raster files. As shown, a separate intersection file is produced for each vector chunk. In each of these files, there is a separate section for each of the n raster files for a total of $m \cdot n$ sections in all the files. Each section stores a list of intersections between polygons and raster files where one of them is illustrated graphically in the figure. In the intersection file, each section is represented by r lists as one per row in the raster file. Each list stores pairs $\langle x, id \rangle$ sorted by x as described above. In addition, each file has a footer which stores the range of polygon IDs covered by this chunk and a list of offsets in the file for the sections in that file, one for each raster file. To create this file efficiently, multiple threads are created where each thread creates an *interim file* that contains a list of sections. Writing the interim files immediately is crucial to reduce the memory overhead as both the polygons and computed intersections can be evicted from memory right after. Once all threads finish, one thread concatenates the sections that

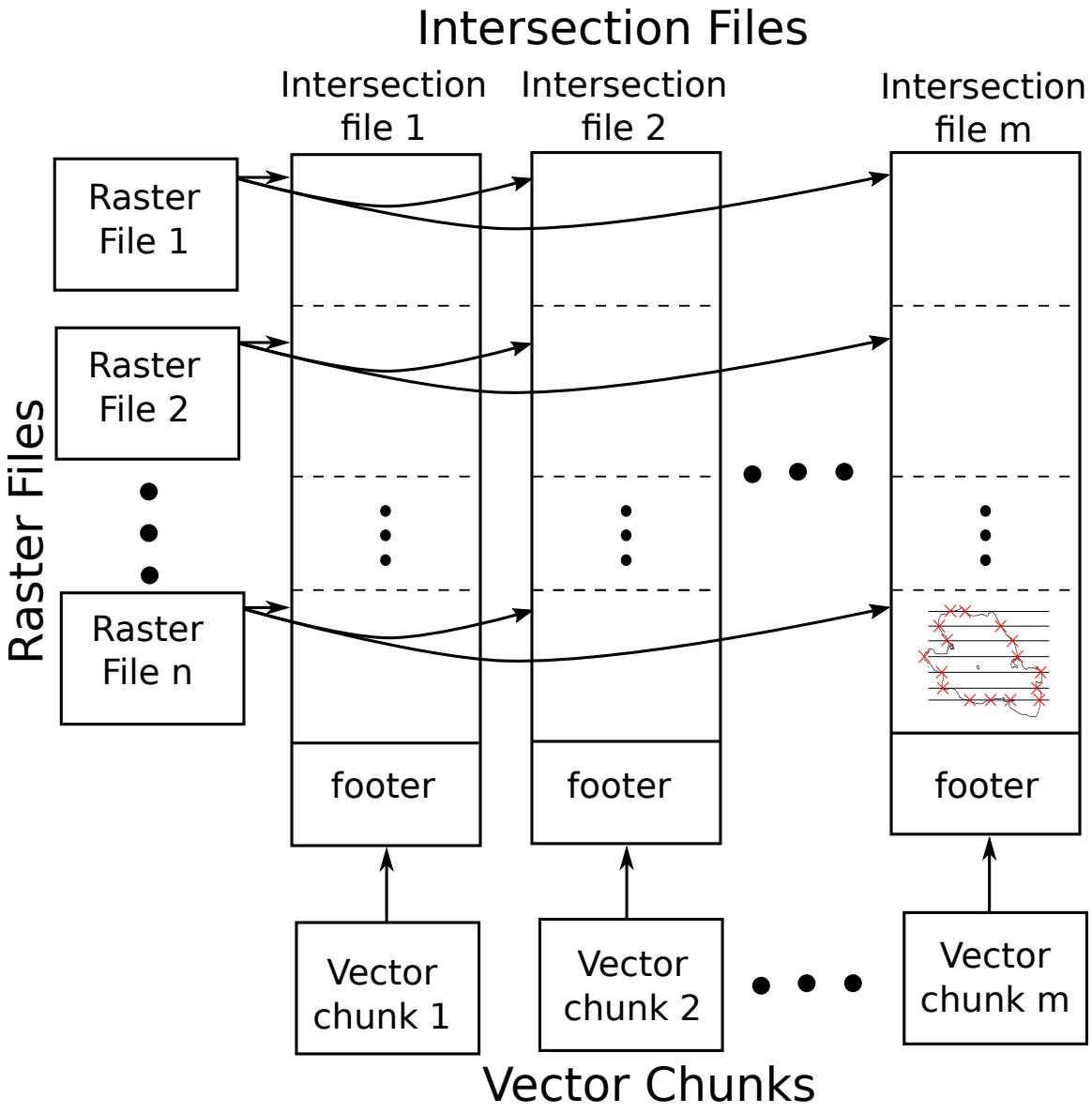


Figure 2.3: Intersection file structure

correspond to one file and adds the footer to it. The concatenation step does not add a huge overhead and it makes the organization of these files easier.

Raptor Split Generation

The second task, *Raptor Split Generation*, performed by this phase, generates *Raptor Splits* using the *RaptorInputFormat*. In Hadoop, the InputFormat is the component that splits the input file into equi-sized splits to be distributed on the worker nodes. These splits are mapped one-to-one to mappers. Therefore, each split defines a unit of work. A corresponding *record reader* uses the split to extract key-value pairs that are sent to the map function for processing. Since our unit of work is a combination of raster plus vector data, we define the new *RaptorInputFormat*, *RaptorSplit*, *RaptorRecordReader*, and *RaptorObject*. Starting with the smallest one, the RaptorObject contains vector chunk ID, a raster file ID, and a tile ID in that raster file. In the next phase, the map function processes one RaptorObject at a time. The RaptorSplit stores a vector chunk ID, a raster file ID, and a *range* of tile IDs in that raster file. The RaptorSplit defines a unit of work given to a mapper. We can control the amount of work given to each mapper by adjusting the number of tiles in the range. The RaptorRecordReader takes one RaptorSplit and iterates over all the RaptorObjects that it represents. Finally, the RaptorInputFormat takes all the input to the problem, i.e., the raster files and all intersection files, and produces a list of RaptorSplits that define the map tasks given to the worker nodes. Notice that the preparation phase only deals with the RaptorInputFormat and generates a list of RaptorSplits out of the input. Therefore, this single-machine step is extremely fast as it does not involve any processing of either vector or raster data.

The information needed to logically partition the raster file is generated when the intersection of a raster file with a *vector chunk* is computed. The definition of the tiles is part of the metadata of the raster file which is loaded to compute the intersections. Moreover, for efficiency, while computing the intersections, we keep track of the tiles that actually overlap the polygons and we make sure to generate RaptorSplits that cover only those tiles. In other words, this step prunes all the tiles that do not contribute to the answer.

The number of generated *RaptorSplits* depends on the total number of Intersection files, raster tiles, and number of tiles. For efficiency, each RaptorSplit is limited to one vector chunk and one raster file. This ensures that each mapper will need to load exactly one section of the intersection file and open one raster file only which overcomes a limitation of the naïve implementation which distributes the tiles regardless of their contained raster files.

The two tasks of the preparation phase make the following contributions:

- The *RaptorInputFormat* eliminates the need of writing a tile-ID file. It generates *Raptor Splits* containing only the *Tile IDs* that overlap with a polygon. This overcomes the first limitation of the Naïve Implementation.

- The second limitation of Naive Implementation is overcome by computing the intersections of the vector files with the raster files once and then broadcasting this information to all the machines.

- The third limitation of the Naïve Implementation is overcome by splitting the vector file into chunks eliminating the need to process the entire vector file at once.

- The intersections are stored in the *intersection file*, and only the intersections corresponding to the given *Raster File ID* are retrieved when required. This decreases the memory footprint of the system, especially in the case of multiple raster files. The mapper may need to process tiles from multiple raster files, and can now retrieve only the required intersections. It does not need to compute intersections for multiple raster files, thus overcoming the fourth limitation of the naive implementation.

- *RaptorInputFormat* limits each *raptorsplit* to one vector chunk and one raster file only which ensures that most machines need to retrieve intersections only once.

Aggregation Phase

This phase is implemented as one MapReduce job, where each machine runs a map function to compute partial Zonal Statistics for the *RaptorSplit* assigned to it. The *RaptorSplit* is a set of RaptorObjects, where each object contains a vector chunk ID, a rater file ID, and a tile ID. The mapper starts by reading the section of the intersection file identified by the chunk ID and raster file ID. A copy of it is cached to process future tiles in the same raster file. Then, the mapper processes the tile identified by the tile ID by loading and aggregating the pixels identified by the ranges in the intersection file. This step is similar to Step 3 in the ScanLine algorithm described in Section 2.3.3. The output is a set of pairs $\langle p_i, a_i \rangle$, where p_i is the polygon ID and a_i is the statistics computed for p_i in the given tile. The reduce function merges the partial statistics a_i belonging to the same polygon p_i and outputs the final aggregation $\sum a_i$.

2.4.3 Other Implementation Details

This section provides further implementation details for the proposed algorithm. In particular, we describe three points, reading raster and vector files in MapReduce, compression of intersection files, and whether to use Hadoop or Spark.

Reading Raster and Vector Files

To read raster and vector files, we use the open source GeoTools Library 17.0. This library requires the data files to be stored on the Local File System and does not provide support to read files from the Hadoop Distributed File System (HDFS). In MapReduce, the files are assumed to be stored in HDFS which makes it incompatible with the GeoTools library. One easy workaround is to let the mapper copy the file from HDFS to the local file system before reading it but it is impractical given the huge sizes of raster files. Another efficient but hard alternative is to modify the GeoTools library to read from HDFS but this would be behind the scope of this work. We follow a third approach where we store the files in a network file system (NFS) which is accessible to all the machines. In this case, the files are not physically replicated to the machines but they can still be accessed as if they are stored locally. A drawback of this method is that the machine that physically stores the file would be a bottleneck. To test the overhead of this reading method, we ran a small experiment that counts the number of text lines in a big text file (roughly 20 GB) using two methods.

- *Local FS*: The file is stored locally in each machine and replicated to all machines.
- *NFS*: Only one copy of the file exists in the network file system.

Then, we executed two MapReduce jobs, one at a time, where each one reads one of the files in parallel and counts the number of lines. We chose line counting due to its minimal processing which allows us to focus on the disk IO. The total times taken for the two MapReduce jobs were 445.24 and 496.53 seconds for the local FS and network FS, respectively. While there is some overhead it was minimal. The reason for observing only a small difference is that our NSF is hosted by a machine with 10 HDD connected through RAID allowing it to serve many requests from different parts of the file at a higher IO throughput than a single disk. Based on this, our experiments read the files from NFS using the standard GeoTools library.

Compressing Intersection Files

The size of the *intersection file* ranges from the order of a few KBs to GBs. Since they are concatenated from *interim files* and simultaneously broadcast to all the machines, we thought it would be a good option to compress them. Compression is expected to reduce the disk I/O in concatenating the interim files and the network I/O in broadcasting them. We tried compressing each intersection written to the *interim file* using both GZIP and Snappy compression. This however marginally slows down the algorithm as can been seen in Figure 2.8. This was caused due to the overhead of compressing and decompressing the file being larger than the time saved in disk and network I/O. This is further discussed in Section 2.5.3.

Hadoop Vs Spark

Even though DARaptor could have been implemented using both Spark and Hadoop, we chose to do it in Hadoop. Spark is known to be efficient for in-memory computation intensive tasks and since our implementation does not require a great deal of in-memory processing but becomes I/O-bound for very large raster layers, we decided to implement this in Hadoop.

2.5 Experiments

This section provides an experimental evaluation of the DARaptor as compared to the Naïve Implementation (NI) and the Scanline Method. We evaluate them on real data and also show the effect of various design decisions on the proposed DARaptor approach. Section 2.5.1 describes the setup of the experiments, the system setup, and the datasets used. Section 2.5.2 provides a comparison of the DARaptor, NI, and the ScanLine method based on the total running time. It also provides a breakdown of total running time for DARaptor between the *Preparation Phase* and the *Aggregation Phase*. Section 2.5.3 shows the effect of various parameters on the total running time of DARaptor. The parameters include the utility of intersections files, size of vector chunks, compression of intersection files, and the effect of RaptorInputFormat.

2.5.1 Setup

We run all the experiments on a cluster with one head node and 12 worker nodes. The head node has Intel(R) Xeon(R) CPU $E5-2609$ v4 @ 1.70GHz processor, 128 GB of

Table 2.1: Vector and Raster Datasets

Vector datasets

Dataset	Polygons	Segments	$\frac{\#segments}{\#polygons}$	File Size
Counties	3,108	51,638	17	978 KB
States	49	165,186	3,370	2.6 MB
Boundaries	284	3,817,412	13,440	60 MB
TRACT	74,133	38,467,094	519	632 MB
ZCTA5	33,144	52,894,188	1596	851 MB

Raster datasets

Dataset	Resolution	File Size
glc2000	40,320×16,353	629 MB
MERIS	129,600×64,800	7.8 GB
US-Aster	208,136×89,662	35 GB
Tree cover	1,296,036×648,018	782 GB

RAM, a 64 GB of SSD, 2 TB of HDD, and 2x8-core processors running CentOS and Oracle Java 1.8.0_131. The worker nodes have Intel(R) Xeon(R) CPU E5-2603 v4 @ 1.70GHz processor, 64 GB of RAM, a 64 GB of SSD, 10 TB of HDD, and 2x6-core processors running CentOS and Oracle Java 1.6.0_31-b04. The File Server used to host the vector and raster datasets has 10 disks of 10 TB each, with a 64 TB formatted size. The methods are implemented using the open source Geotools library 17.0.

In all the techniques, we compute the four aggregate values, minimum, maximum, sum, and count. We measure the end-to-end running time as the performance metric which includes reading both datasets from disk and producing the final answer. Table 2.1 lists the datasets that are used in the experiments. The vector layers represent the US continental counties and US continental states with 3000 and 49 features respectively. The Large-Scale International Boundaries (LSIB) includes geographic national boundaries for 249 countries and disputed areas. The TRACT and ZCTA5 datasets are a part of TIGER 2017 dataset which represents the contiguous US. TRACT represents the US census tracts boundaries and ZCTA5 represents 5-digit ZIP Code Tabulation Areas. The raster datasets come from various government agencies. The GLC2000 and MERIS 2005 datasets are from the European Space Agency with pixel resolutions of 0.0089 decimal degrees (1km) and 0.0027 (300m) respectively. The US Aster dataset originates from the Shuttle Radar Topography Mission (SRTM) and covers the continental US. Hansen developed the global Tree Cover change dataset which covers the entire globe. Both datasets have a spatial resolution of 0.00028 decimal degrees (30m).

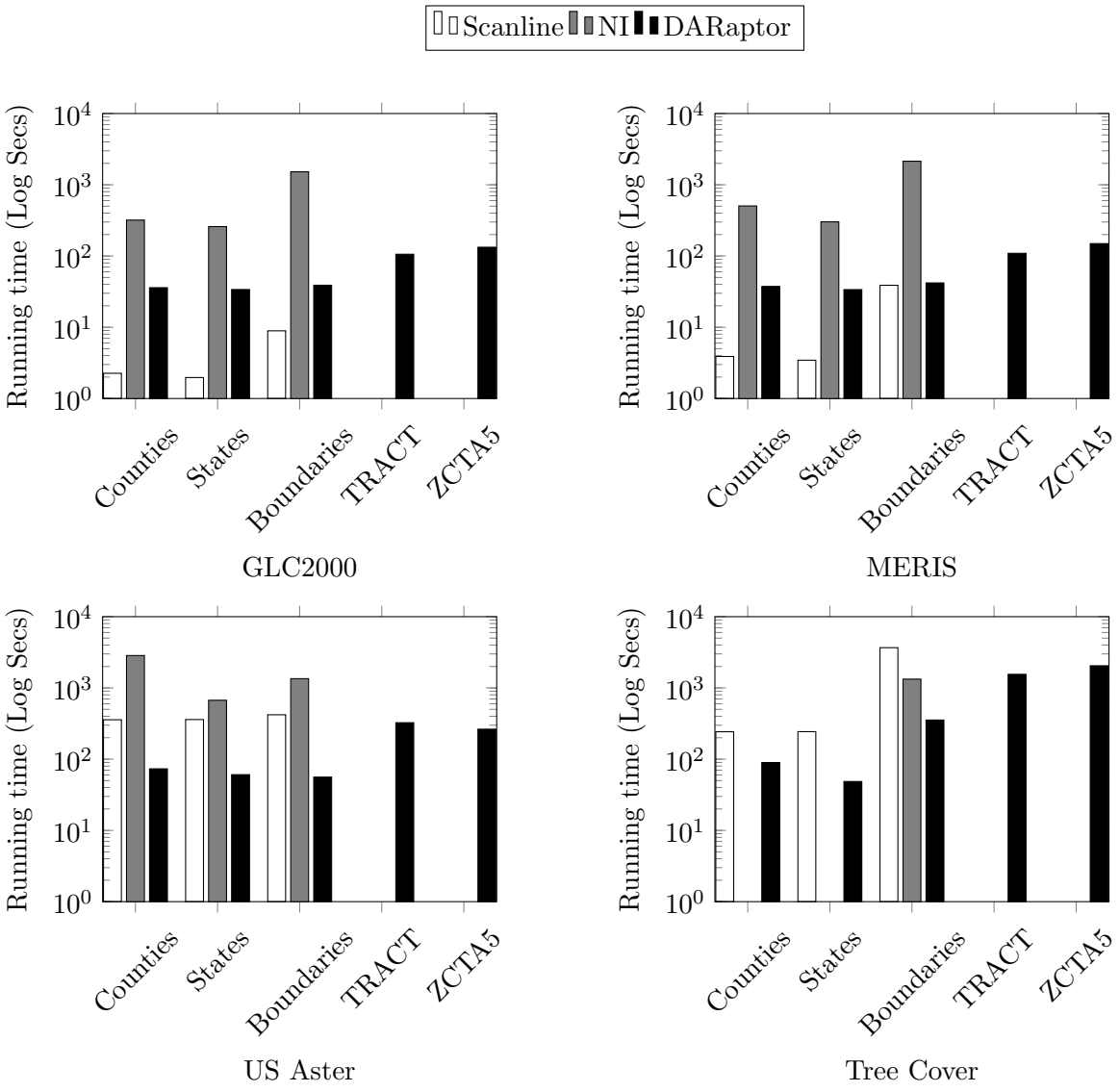


Figure 2.4: Overall comparison of ScanLine, NI and DARaptor algorithms

2.5.2 Overall Comparison

This section provides a comparison of the proposed DARaptor approach, the Naive Implementation (NI), and the Scanline Method [16]. This experiment is run for all the combinations of vector and raster datasets shown in Table 2.1, and its results can be seen in Figure 2.4. We omitted the running times that are more than two hours, or for which the algorithm runs out of memory.

As can be observed from Figure 2.4, 1. DARaptor is scalable for larger vector datasets (TRACT and ZCTA5), while the single-machine ScanLine Method and the Naïve Implementation fail (run out of memory) for them. 2. For large raster datasets, the proposed DARaptor algorithm is much faster than both the Scanline and NI algorithms with up-to an order of magnitude speedup. 3. The combination of MERIS and Boundaries datasets is where the single machine and the proposed DARaptor algorithms provide the same performance. At this point, the overhead of parallelization is equal to its performance gain. 4. The DARaptor algorithm shows a 10x speedup with Tree Cover and Boundaries dataset and continues to scale to larger vector datasets where others fail. 5. The naïve implementation performs worse than the proposed algorithm for all the combinations of raster and vector datasets.

The scalability of the proposed algorithm can be attributed to the decision of creating vector chunks and *interim files*. Its gain in performance over the single-machine ScanLine Method for larger datasets is due to the distributed computation of Zonal Statistics. The decrease in performance for smaller raster datasets is as expected. The proposed distributed algorithm incurs an additional overhead in setting up a MapReduce job in Hadoop

and therefore, performs slower than the single-machine ScanLine Method. Its performance is at par with the ScanLine method for the MERIS dataset and Boundaries dataset and then increases as the size of one of the raster or vector datasets increases. Its performance gain over NI is due to the decision of computing intersections once on a single-machine and then broadcasting them to all machines for distributed computation of Zonal Statistics.

Figure 2.5 shows the breakdown of the total running time for DARaptor into two phases, *Preparation Phase* and *Aggregation Phase*. The *Preparation Phase* computes the intersections of the vector file(s) against the raster layer and writes them to *intersection files*, that are broadcast to all machines for the next phase. It also defines the logical partitioning of raster and vector file(s) and generates *Raptor Splits*. The *Aggregation Phase* computes zonal statistics for each *raptor split*. All the numbers shown in Figure 2.5 are normalized to the overall running time for comparison. The actual numbers are the same as in Figure 2.4.

Breakdown of Total Running Time

It can be observed from the figure that the running time is dominated by *Preparation Phase* for the combination of smaller raster datasets (GLC200 and MERIS) with the larger vector datasets (TRACT and ZCTA5). For all other combinations of raster and vector datasets, the running time is dominated by the *Aggregation Phase*. The reason for the domination of *Preparation Phase* for GLC200 and MERIS against TRACT and ZCTA5 datasets is because of the large number of geometries in the vector files for whom the intersections with the raster file must be computed. Also, the small size of raster files leads to a small number of logical partitions for which the Zonal Statistics must be computed,

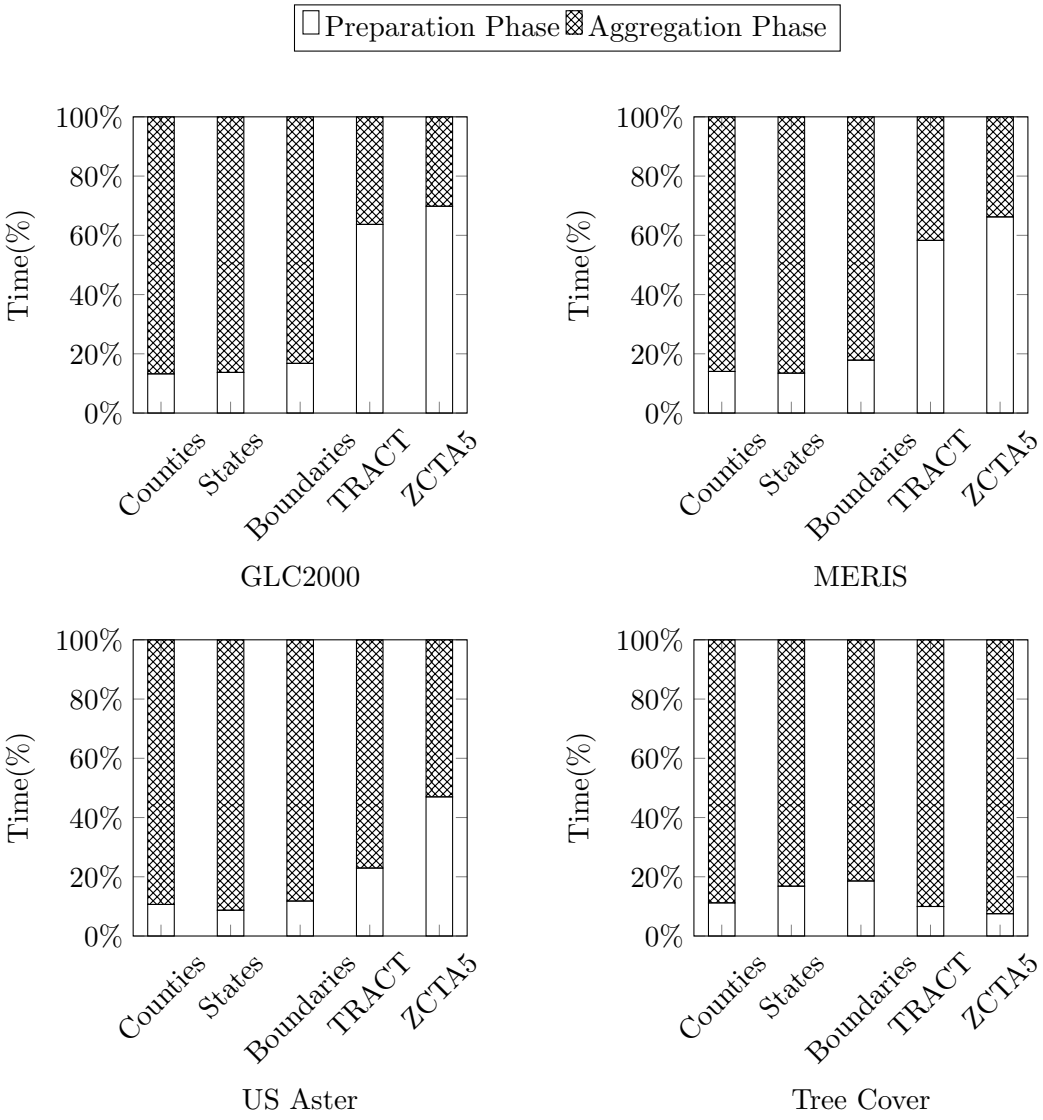


Figure 2.5: Breakdown of running time for DARaptor

hence the *Aggregation Phase* takes less time than the *Preparation Phase*, for these datasets. Moreover, the computation of Zonal Statistics requires reading the required pixel values from the disk for each *Raptor Split*. This makes the *Aggregation Phase* dominated by disk IO for reading only the required pixel values. This leads to the running time for *Aggregation Phase* becoming dominant over that for *Preparation Phase* for large raster files.

2.5.3 Tuning

This section provides a deeper study of the proposed algorithm and examines the effectiveness of each component individually.

Computing Intersections

In this part, we examine the effect of computing the intersections once in the preparation phase Vs computing it repeatedly in each node. This experiment compares three algorithms:1. the Naive Implementation 2. DARaptor and, 3. an improved version of the Naive Implementation (NI-Improved). This improved version of NI takes advantage of the setup function of the mapper in Hadoop, which is used to initialize all the mappers. It computes the intersections in the setup function, hence making them available to all the mappers on their initialization.

Figure 2.6 shows the total running times for the three algorithms. We omit the running times that are longer than two hours or for which the algorithm ran out of memory. As can be seen from the figure, 1. the NI-Improved algorithm shows orders of magnitude increase in performance over NI, but it is still slower than DARaptor. 2. The NI-Improved

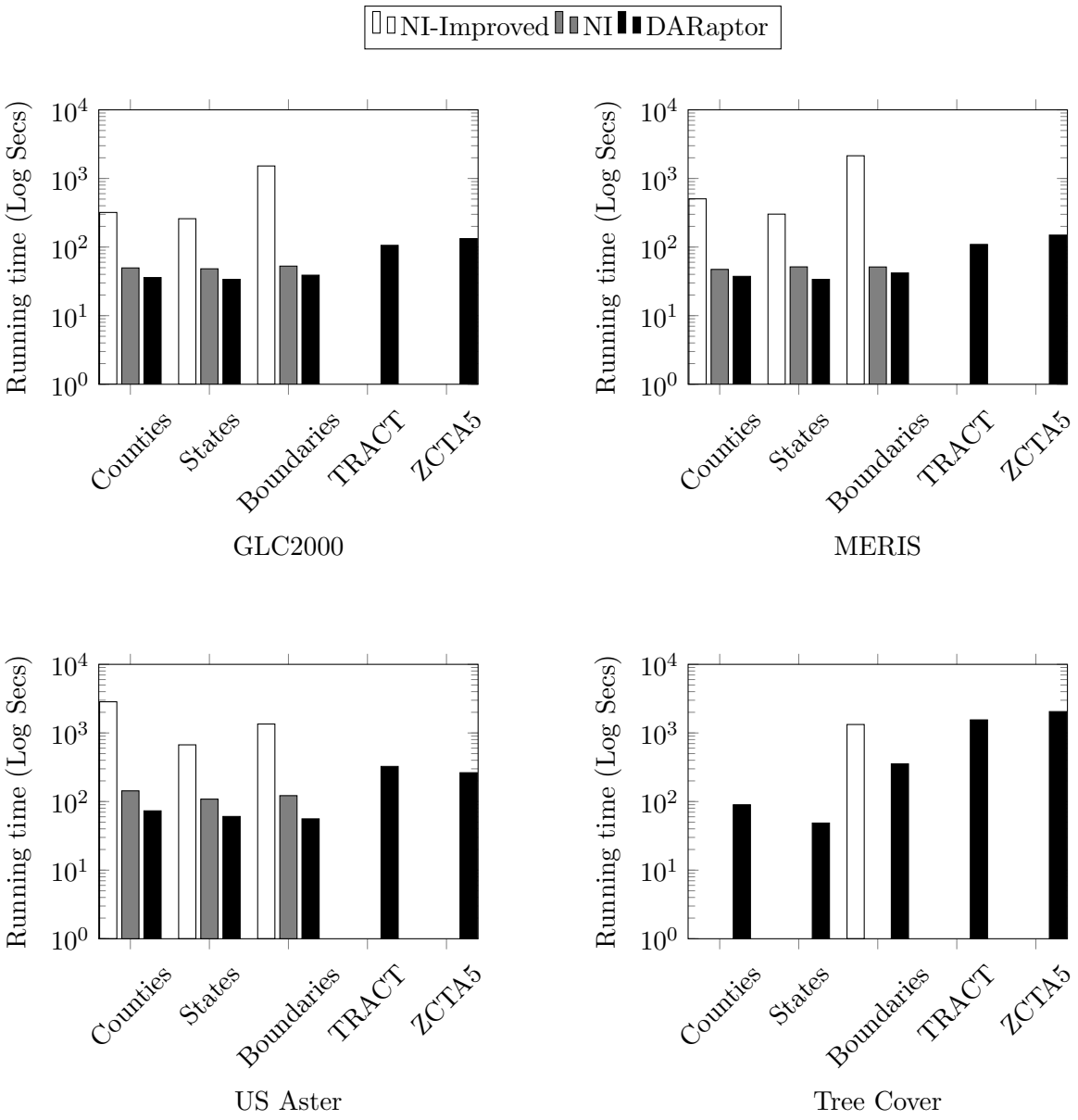


Figure 2.6: Comparison of the computation of intersections

and NI run out of memory for the larger vector datasets (ZCTA5 and TRACT) due to the overhead of computing the intersections of all the vector layers. 3. NI-Improved runs out of memory for all combinations of the Tree Cover dataset with any of the vector datasets due to the large size of the intersections. 4. NI takes longer than two hours to run for the combination of Tree Cover and, Counties and States dataset. The Naïve Implementation is slower than both NI-Improved and DARaptor due to its repeated computation of intersection in each mapper for each tile assigned to it. NI-Improved eliminates this repeated computation of intersections, it still sends each mapper a copy of these intersections, while keeping them in memory, which increases the memory footprint. If there are multiple raster files, like in the case of Tree Cover dataset, each mapper receives the intersections for all the raster files, which makes the algorithm run out of memory to proceed. Each mapper does not require the intersections for all the raster files. NI and NI-Improved cannot compute intersections for vector datasets with large number of geometries because they cannot keep all the intersections in memory and run out of memory. DARaptor is faster than NI and NI-improved because of the other improvements that we add in DARaptor, e.g., the RaptorInputFormat. Also, *intersection files* allows each mapper to retrieve only the required intersections from the file rather than loading all the intersections.

Vector Chunks

This experiment studies the effect of splitting the vector file into chunks. In particular, this experiment varies the sizes of vector chunk used in DARaptor starting with 100, then 1000 to 10,000, incremented in steps of 1000. Figure 2.7 shows the overall running

Table 2.2: Size of Intersection Files

Raster Dataset	Vector Dataset	Intersection File Size		Compression Ratio
		Raw	Compressed	
GLC2000	Counties	1.4 MB	347 KB	4.13%
	States	276 KB	81 KB	3.41%
	Boundaries	4.1 MB	1.6 MB	2.56%
	TRACT	5.6 MB	1.9 MB	2.94%
	ZCTA5	6.9 MB	2.4 MB	2.875%
MERIS	Counties	4.6 MB	1.1 MB	4.18%
	States	888 KB	266 KB	3.34%
	Boundaries	13.3 MB	5.3 MB	2.55%
	TRACT	18 MB	5.6 MB	3.21%
	ZCTA5	22.1 MB	7.1 MB	3.11%
US Aster	Counties	46 MB	10.6 MB	4.34%
	States	8.9 MB	2.7 MB	3.3%
	Boundaries	9.2 MB	2.5 MB	3.68%
	TRACT	171.6 MB	49.7 MB	3.45%
	ZCTA5	212.7 MB	63.7 MB	3.34%
Tree Cover	Counties	52.2 MB	10.0 MB	5.22%
	States	17.0 MB	2.8 MB	6.07%
	Boundaries	1.2 GB	70.5 MB	17.43%
	TRACT	259.6 MB	52.8 MB	4.9%
	ZCTA5	308.6 MB	66.5 MB	4.64%

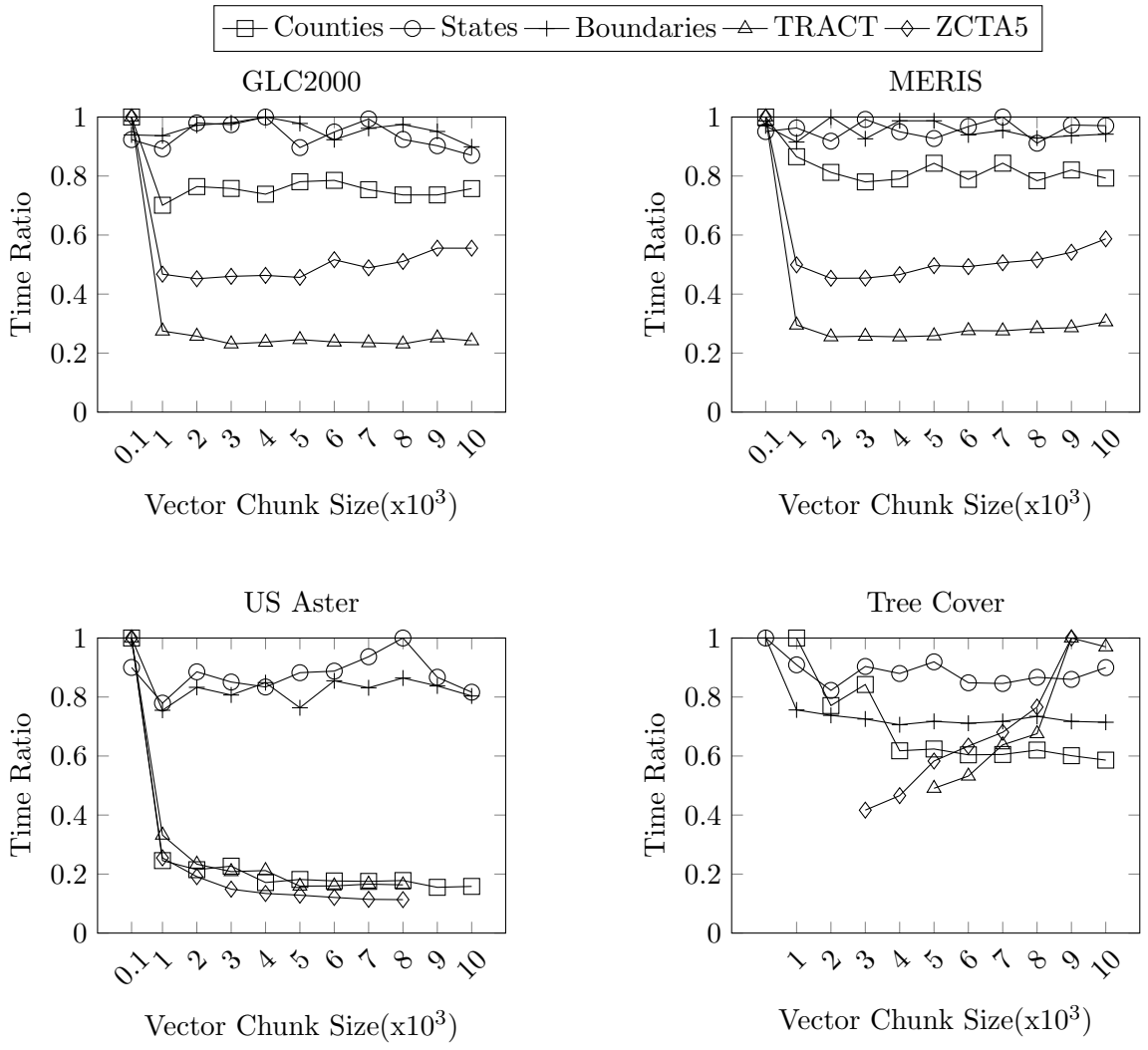


Figure 2.7: Effect of different sizes of Vector Chunks on total running time

time as the chunk size increases. In this experiment, we are only interested in the trend of the lines rather than comparing the different lines. Therefore, each line is normalized independently to fill all of them in one figure. We omit the running times for which the algorithm runs of memory. We observe in this experiment that a very low chunk size of 100 results in a reduction in the performance due to the overhead of creating and running too many RaptorSplits. On the other hand, using a very large chunk size eventually results in some job failures due to the memory overhead. This is equivalent to not splitting the vector file.

After chunk size of 3000, the number of vector chunks generated for Counties and Boundaries, becomes stable which leads to marginal variation in their running times. The variation of chunk size on larger vector datasets and Tree Cover is more prominent than the other vector datasets. This is due to the large size of the Tree Cover dataset. The increase in vector chunk size leads to a decrease in the number of chunks being generated and hence, less number of *raptor splits*. This leads to each machine having more amount of work to do, and a non-optimal distribution of work. It can be concluded that the choice of vector chunk size should neither be too big (10,000) nor too small(100). It should lead to an optimal distribution of work in the *Aggregation Phase* and can depend on the system configuration. We chose it to be 5,000 based on the experiments and according to our system configuration.

Compression of Intersection File

In this experiment, we study the effect and tradeoff of compressing the intersection files. The size of the raw (uncompressed) intersection files ranges from a few kilobytes to a

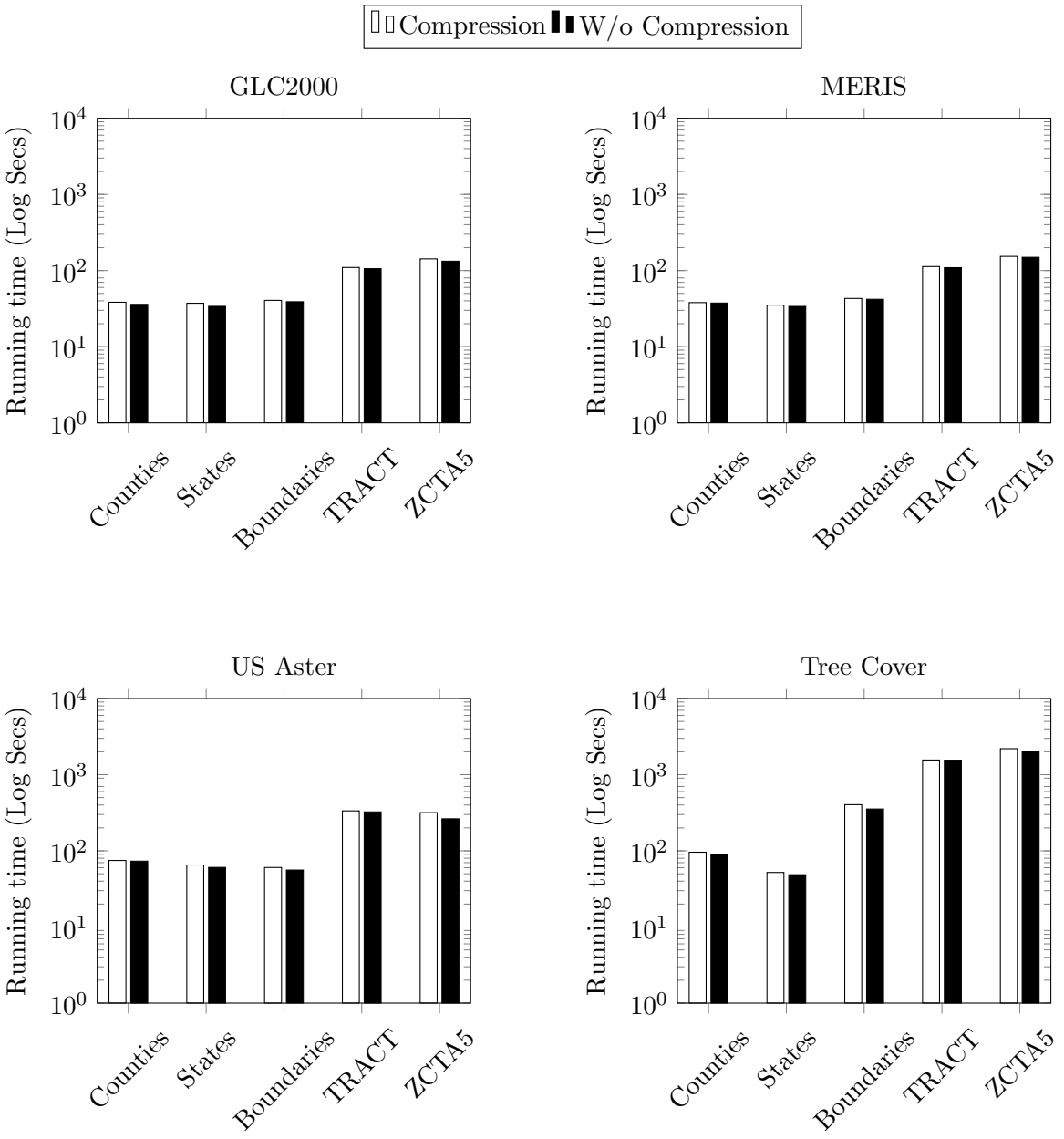


Figure 2.8: Effect of compression of Intersection File

gigabyte as can be seen in Table 2.2. In order to reduce the network overhead to broadcast the file over the network to all the machines and the time taken to concatenate them, we investigated the option of compressing the intersection files using both GZIP and Snappy compression. We did not see a major difference between GZIP and Snappy so we are only reporting the results of GZIP. The comparison of the execution time with and without using compression can be seen in Figure 2.8. It can be observed that there is a marginal increase in the total running time if the intersection files are compressed for all the combinations of the datasets. Although, from Table 2.2, it can be seen that the size of compressed files is far smaller than that of non-compressed *intersection files*. This is because the time saved in concatenating and broadcasting the compressed intersection file is nearly the same as the time taken for compression and decompression of the intersection file. However, compressing the intersection file can be a viable option, in case the network IO becomes a bottleneck.

RaptorInputFormat

This experiment shows the effect of defining our own *RaptorInputFormat* as compared to using a single text file containing all *Tile IDs* like the Naïve Implementation does. Figure 2.9 shows this comparison by using two versions of DARaptor, one uses a *RaptorInputFormat* and another uses a text file just like NI does. The writing of the tiles to the text file is a part of the total running time. The text file also does not contain any unnecessary Tile IDs. The file is however split among machines so that the minimum number of lines per split is at least 10 which allows the MapReduce jobs to take full advantage of the cluster.

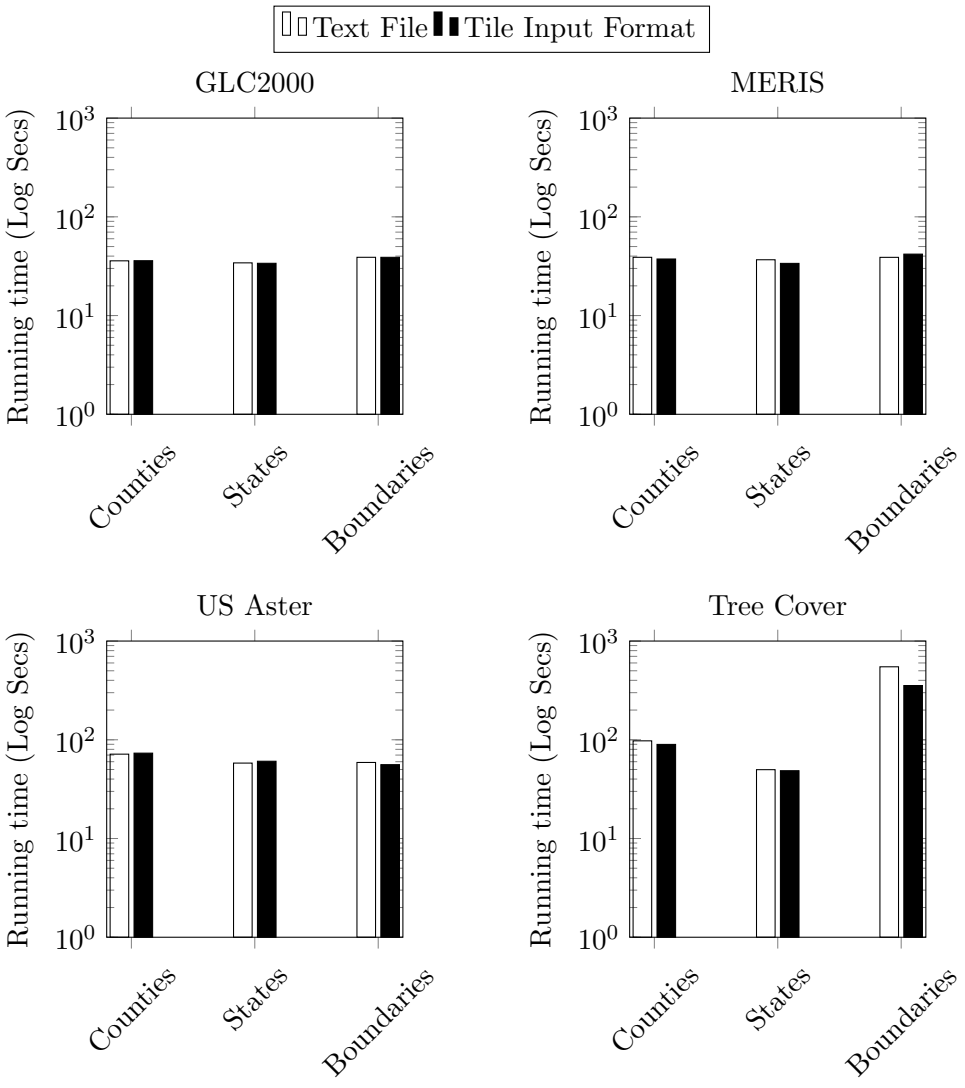


Figure 2.9: The overall effect of input split sizes (in seconds)

As can be seen from the figure, there is either an improvement in the algorithm's running time with the introduction of *RaptorInputFormat* or it is almost equal to the one using text file. The raster datasets GLC2000, MERIS, and US Aster have fewer number of logical partitions as compared to the Tree Cover dataset. The time taken to write the input text files for them is the fraction of a second, and these files lead to the creation of just one intersection file. Also, these datasets contain only one raster file, which leads to *RaptorSplits* being equivalent to the splits that are created using the text file. The introduction of *RaptorInputFormat* therefore, does not show any significant improvement for them.

The Tree Cover dataset consists of multiple raster files and covers the whole globe. The Counties and States vector datasets cover only the US, while the Boundaries dataset covers the whole world. Therefore, writing the text file (tile-ID file) for Counties and States datasets takes a fraction of a second and the overall running time is not much different. However, for the boundaries dataset which covers the entire globe, writing the tile ID file takes approximately eight seconds. Therefore, the speedup is mainly driven by saving the time to write the tile ID file. However, we need to keep in mind that the RaptorInputFormat also allows us to apply other optimizations such as combining relevant tiles in the same RaptorSplit which is not easy to do with a simple tile ID file.

2.6 Conclusion

In this chapter, we presented a distributed MapReduce algorithm for the zonal statistics problem. The proposed algorithm provides several key ideas that can carry on

to other distributed algorithms for processing big vector and raster datasets. First, the proposed framework runs two phases, a single-machine preprocessing step that computes a common data structure to be used in parallel, and defines the tasks that will be executed in parallel. The second phase runs in parallel and aims at reading and processing the big raster files efficiently. The second key idea is introducing the RaptorInputFormat which is the first input format that combines raster plus vector data in one split. The RaptorInputFormat can define the units of work to be executed in parallel and provides an easy way to optimize and balance the load across machines. The RaptorInputFormat combined with the preprocessing step can also prune irrelevant parts of the raster file in order to speed up the parallel processing. Finally, we investigated several improvements to this method such as splitting the vector file into chunks and compressing the intermediate files between the preprocessing and distributed aggregation. Our experiments show that the proposed algorithm can scale to very large data whereas the baselines could not handle big vector or raster data.

Chapter 3

Raptor Zonal Statistics: Fully Distributed Zonal Statistics of Big Raster + Vector Data

This chapter talks about *Raptor Zonal Statistics*, a distributed implementation in Hadoop that can be used to efficiently compute zonal statistics.

3.1 Introduction

Advancements in remote sensing technology have led to a tremendous increase in the amount of remote sensing data. For example, NASA EOSDIS provides public access to more than 33 petabytes of Earth Observational data and is estimated to grow to more than 330 petabytes by 2025 [17]. European Space Agency (ESA) has collected over five petabytes of data within two years of the launch of the Sentinel-1A satellite and is expected to receive

data continuously until 2030 [18]. This big remote sensing data is available as raster data which is represented as multidimensional arrays. Many applications need to combine the raster data with vector data which is represented as points, lines, and polygons. This chapter studies the *zonal statistics* problem which combines raster data, e.g., temperature, with vector data, e.g., city boundaries, to compute aggregate values for each polygon, e.g., average temperature in each city. This problem has several applications including the study by ecologists on the effect of vegetation and temperature on human settlement [32, 33], analyzing terabytes of socio-economic and environmental data [29, 30], and study of land use and land cover classification [57]. It can also be used for areal interpolation [56] and to assess the risk of wildfires [24].

There exist many big spatial data systems which either efficiently process big *vector* data, e.g., SpatialHadoop [14], GeoSpark [80], and Simba [78], or big *raster* data, e.g., SciDB [70], RasDaMan [3], GeoTrellis [36], and Google Earth Engine [28]. Unfortunately, none of these systems are well-equipped to combine raster and vector data together and they all become very inefficient for the zonal statistics problem for big raster and vector data.

Traditional methods to process the zonal statistics problem focus on either vectorizing the raster dataset [86] or rasterizing the vector data [29, 2]. The first approach converts each pixel to a point and then runs a traditional spatial join with polygons using a point-in-polygon predicate [86]. Finally, it groups the pixels by polygon ID and computes the desired aggregate function. This algorithm suffers from the big computation overhead of the point-in-polygon query. Even if the polygons are indexed, this algorithm is still im-

practical. Furthermore, when the vector data is very large, a disk-based index is needed which makes this algorithm even slower. The second approach rasterizes the vector data by converting each polygon to a raster (mask) layer with the same resolution as the input raster layer and then combines the two raster layers to compute the desired aggregate function [29, 2]. This algorithm suffers from the computation overhead of the rasterization step and the disk overhead of randomly accessing the pixels that overlap each polygon.

To further show the limitation of the baseline algorithm, this chapter provides a theoretical analysis of the two existing approaches by making an analogy to traditional join algorithms. First, we show that the vector-based approach resembles an index nested loop join which suffers from the repetitive access of the index. This means that it should be used only if the non-indexed dataset, i.e., the raster dataset is very small. On the other hand, the raster-based approach resembles a hash-join algorithm which suffers from the excessive size of the hashtable, i.e., the number of pixels. This reveals that this algorithm would work well if the number of pixels that overlap the polygons is very small.

This chapter proposes a fully distributed algorithm, termed *Raptor Zonal Statistics* (RZS), which overcomes the limitations of the two baseline algorithms. RZS overcomes the limitations of existing algorithms since it scans the raster data sequentially while aggregating pixels without running any point-in-polygon queries. The key idea is to generate an intermediate structure, termed *intersection file*, which accurately maps vector polygon to raster pixels. Further, this intersection file is sorted in a way that matches the raster file structure which optimizes the disk access to raster data, hence, speeds up the algorithm; this results in a linear-time algorithm for merging raster and vector data that is analogous

to sort-merge. In general, the proposed algorithm runs in three steps, namely, *intersection*, *selection* and *aggregation*. The intersection step partitions the vector data and generates the intersection file that contains a compact representation of the intersection of the vector and raster datasets. The intersection file is then sorted to match the raster file by only reading the *metadata* of the raster file, e.g., resolution and tile size, i.e., without reading the pixel values. The selection step concurrently scans the intersection file and the raster file to produce the set of ⟨polygon ID, pixel value⟩ pairs. To run this step, we introduce two new components, *RaptorInputFormat* and *RaptorSplit*, that allows this step to read both raster and vector data and process them in parallel. Finally, the aggregation step groups these pairs by polygon ID and calculates the desired aggregate function. A previous work (presented as a poster [65]) proposed a straight-forward parallelization of an efficient single-machine algorithm [16, 67] which showed some promising results but was limited since it used the single-machine algorithm as a black-box. In this chapter, we redesign the algorithm to make it fully distributed and we make a cost analysis to theoretically prove its efficiency over the baselines.

Experiments on real datasets, with nearly a trillion pixels and 330 million polygon segments, show that the proposed algorithm (RZS) outperforms all baselines for big data, including Rasdaman and Google Earth Engine, and shows perfect scalability with large data and big clusters. Furthermore, RZS is up-to 100× faster in data loading since it does not require any preprocessing while baseline systems need a heavy data loading phase to organize the data for efficient processing.

The rest of this chapter is organized as follows. Section 3.2 covers the related work in literature. Section 3.3 provides a review of the concepts used in this chapter. Section 3.4 describes the proposed system, Raptor Zonal Statistics. Section 3.5 provides a theoretical analysis and comparison of the proposed approach and the raster-based baseline. Section 3.6 provides an extensive experimental evaluation. Finally, Section 3.7 concludes the chapter.

3.2 Related Work

In this section, we cover the relevant work in the literature. First, we give an overview of big spatial data systems and classify them according to whether they primarily target vector data, raster data, or both. After that, we cover the work that specifically targets the zonal statistics problem.

3.2.1 Big Vector Data

In this research direction, some research efforts aimed to provide big spatial data solutions for vector data types and operations. There are several systems in this category including SpatialHadoop [14], Hadoop-GIS [1], MD-HBase [45], Esri on Hadoop [76], GeoSpark [80], and Simba [78], among others. The work in this category covers (1) spatial indexes such as R-tree [14, 80, 78], Quad-tree [76, 45], and grid [14], (2) spatial operations such as range query [76, 14, 80, 78, 45], k nearest neighbor [14, 80, 78, 45], spatial join [14, 80, 78], and computational geometry [10], (3) spatial data visualization including single-level and multilevel [15], and (4) high-level programming languages such as Pigeon [13].

Vector-based systems can support the zonal statistics problem by utilizing the index-nested loop join operation with the point-in-polygon predicate. Shahed [11] further improves this query by building an aggregate Quad-tree index for the raster layer but it only supports rectangular regions while this work considers complex polygons without the need to prebuild an index. When it comes to complicated polygons and high-resolution raster data, all these techniques become impractical.

3.2.2 Big Raster Data

Systems in this research direction focus on processing raster datasets which are represented as multidimensional arrays. Popular systems include SciDB [70], RasDaMan [3], GeoTrellis [36], and Google Earth Engine [28]. The set of operations supported for raster datasets are completely different from those provided for vector datasets. They are usually categorized into four categories, namely, local, focal, zonal, and global operations [60] and the computational model is based on linear algebra.

To support the zonal statistics operation in these systems, each polygon is first rasterized to create a *mask layer*. Then, the mask layer is combined with the input raster layer to select overlapping pixels which are finally aggregated. This process is repeated for each polygon separately since the polygons might be overlapping in the mask layer. There are two drawbacks to this approach. First, if the raster data has a very high resolution, the size of each mask layer can be excessively large. Second, for nearby and overlapping polygons, this algorithm will need to read the same regions of the raster data many times. A parallel algorithm can be used for efficient rasterization [75] but this does not address the two limitations described above.

3.2.3 Big Raster-Vector Combination

Systems like PostGIS and QGIS [54] can work with both vector and raster data but they internally use two isolated libraries, one for each type, and they are still limited to the approaches described above for the zonal statistics problem. Other research suggests an alternative data representation for vector and raster data [49, 5] that can speed up some queries that combine both data types. However, these methods are impractical since they require an expensive preprocessing phase for rewriting and indexing all the data. On the other hand, the proposed approach does not require any index construction while achieving a better query performance.

3.2.4 Zonal Statistics

Zonal statistics is a basic problem that is used in several domains including ecology [32, 33] and geography [29, 30]. However, there was only a little work in the query processing aspect of the problem. ArcGIS [2] supports this query by first rasterizing the polygons dataset and then overlaying it with the raster dataset which resembles a hash-join. Zhang *et al.* [86, 85] solves the zonal statistics problem using an algorithm that resembles the index nested-loop join. It converts each pixel to a point and relies on GPUs to speed up the calculation. The drawback is that it has to load the entire raster dataset in GPU memory which is a very expensive operation and is impractical for very large raster datasets. Zhao *et al.* [89] aimed at increasing the performance of the existing zonal statistics method using python in a shared memory multi-processor system. That method uses threads by sending each thread a set of raster files but requires rasterizing the polygon dataset.

Recent work in Terra Populus [30, 29] demonstrates the complexity of the problem on big raster and big vector datasets. The Scanline algorithm [16, 67] was a first step in efficiently processing the zonal statistics problem by combining vector and raster data but it was limited to a single machine. In [65] (a poster), we tried a straight-forward parallelization of the Scanline approach that showed some performance improvement but it was limited as it used Scanline as a black-box. In particular, it only parallelized the second phase that reads the raster data but still processes the vector data on a single machine which made it limited to small vector data.

This work proposes a novel scalable algorithm that follows a sort-merge approach for processing the zonal statistics problem on big raster and vector data using MapReduce. It is different than the work described above in four ways. 1. It leverages the MapReduce programming paradigm to scale out on multiple machines. 2. It is a fully distributed approach to the zonal statistics problem which allows it to scale to big raster and vector data. 3. It provides a novel mechanism for parallel task distribution that combines raster-plus-vector (Raptor) in one unit of work. 4. It can efficiently prune non-relevant parts in the raster layer to speed up query processing.

3.3 Review of Raster and Vector Data

This section provides a background on some relevant concepts from GIS and spatial databases, and the zonal statistics problem. Interested readers can refer to [60] for more information.

3.3.1 Spatial Data Representation

The two common representations of spatial data are *vector* and *raster* representations. The vector representation uses constructs like point, line, and polygon, and operations like intersect, union, and overlaps. The raster representation uses matrices as a common construct and the operations are all performed on these matrices. Each entry in the matrix is called a pixel. To map between pixels and geographic locations, two mappings are used, namely, *world-to-grid* ($\mathcal{W}2\mathcal{G}$) and *grid-to-world* ($\mathcal{G}2\mathcal{W}$). The $\mathcal{W}2\mathcal{G}$ mapping takes a coordinate in longitude and latitude and maps it to a position in the matrix and the $\mathcal{G}2\mathcal{W}$ mapping does the opposite. These mappings can also be used to map data between the vector and raster datasets. The algorithm in this chapter relies mainly on these two mappings to map the computation between the two representations without having to convert one of them entirely to the other representation.

3.3.2 Raster File Structure

A raster layer is modeled as a very large dense matrix. One layer can typically contain trillions of entries. Most standard file structures, e.g., GeoTIFF and HDF5, partition this large matrix into smaller equi-sized *tiles*. Each tile is stored as one block and is typically small enough to load entirely in the main memory. Tiles are identified by sequence numbers identifies, t_{id}, starting at zero. The file contains a lookup table that allows locating any tile efficiently. The data in all tiles can be stored in row-major or column-major order but they have to be the same. Finally, if a compression technique is applied, then the data in each tile is compressed separately to allow for the decompression of a single tile. The

proposed algorithm relies on this structure to achieve a highly optimized solution while avoiding any prebuilt indexes.

3.3.3 Zonal Statistics

The input to the zonal statistics problem is a raster layer r, a vector layer v, and an accumulator *acc*. The vector layer v consists of a set of polygons which are usually disjoint, e.g., city boundaries. The raster layer is a large two-dimensional matrix that contains remote sensing data, e.g., temperature values. The accumulator *acc* is a user-provided function which takes pixel values, one at a time, and computes the statistics of interest. For example, one accumulator can compute the average value, while another one can compute the histogram over the spectrum of raster values. The output is a value for the accumulator for *each polygon* in the vector layer. For example, if r represents the temperature in the world, v represents the 50 US States, and *acc* is an average accumulator, the output of this problem will be the average temperature for each state.

3.3.4 Zonal Statistics on Raster DB

The raster database approach focuses on processing raster data by representing them as multi-dimensional arrays and is used in [70, 3, 36, 2, 30]. To solve the problem of zonal statistics, this approach follows a method called *clipping*. It runs in the following three steps.

Step 1 rasterizes each polygon in the vector layer separately, by generating a mask layer of the same resolution of the input raster layer where a pixel has a value of one if it is inside the polygon and a zero if it is outside.

Step 2 applies a masking operation between the input raster and the rasterized mask which is a local operation.

Step 3 computes the desired aggregate function by aggregating all the values in the masked layer.

3.3.5 Zonal Statistics on Vector DB

The vector database approach focuses on processing vector data by representing each feature in terms of a list of coordinates associated with a set of attributes. A feature can be a point, a line, or a polygon. This approach is used in [14, 80, 78]. To solve the problem of zonal statistics, this approach follows a method called *point-in-polygon*. It runs in the following three steps.

Step 1 builds a spatial index for the vector layer to facilitate the point-in-polygon query. That is, given a query point, find the containing polygon.

Step 2 converts each pixel to a point and runs an index lookup to find the containing polygon, if any.

Step 3 groups the points by their containing polygon ID and runs the desired aggregate function on each group.

3.3.6 Single-machine ScanLine Method

The scan-line method [16] is the state-of-the-art algorithm for computing zonal statistics on a single machine. It runs in the following three steps.

Step 1 calculates the Minimum Bounding Rectangle (MBR) of the input polygon(s) and maps its two corners from world to grid coordinates using the $\mathcal{W}2\mathcal{G}$ mapping. These two corners help in identifying the lower and upper rows in the grid coordinates that define the range of scan lines to process.

Step 2 computes the intersections of each of the scan lines with the polygon boundaries. It converts each scan line from grid coordinates to world coordinates using $\mathcal{G}2\mathcal{W}$ mapping and stores their y-coordinates in a sorted list. Each polygon is scanned for its corresponding range of scan lines, which are then used to compute intersections with the polygon. These intersections are then sorted by their x-coordinates for each scan line.

Step 3 finds the pixels that lie inside the polygons and process them. It maps the x-coordinates of the intersections from world to grid coordinates and accumulates the corresponding pixel values. For multiple polygons, all intersections in one row are processed before moving to the next row.

This approach requires a minimal amount of intermediate storage for the intersection points. It also minimizes disk IO by scanning the raster data exactly once and by reading only the pixels that overlap the polygons.

3.4 Implementation

This research started with the aim of implementing a distributed system for solving the zonal statistics operation efficiently for large datasets. To do so, we worked on creating a distributed implementation for the *scanline algorithm* introduced in [16]. Scanline algorithm is a single-machine algorithm, that can work with raster and vector data

in their native formats and can be used to perform the zonal statistics operation. We initially implemented a partially distributed method in Hadoop called DARaptor or the Efficient MapReduce Implementation (EMI) [65, 67]. While EMI was more efficient than the Scanline algorithm and could scale to large datasets, its scalability was limited. We then implemented the fully distributed method, *Raptor Zonal Statistics* [66].

Raptor Zonal Statistics (RZS) follows a *sort-merge* approach to combine raster and vector data and answer the zonal statistics query. By observing the analogy between the two baseline approaches and the two join algorithms, hash join and index nested loop, the reader can see why neither of them scales to big raster and vector data. The rasterization approach does not scale due to the large number of pixels that overlap the polygons which needs to be retrieved from disk and decompressed. The vectorization approach does not scale either due to the overwhelming computation cost of index lookups or the large size of the index which has to be stored on disk. This makes us think of using a sort-merge-like algorithm, however, it requires an expensive sorting phase which outweighs the saving in the merge step. We show below that we can minimize the overhead of the sorting phase using a novel data structure named, intersection file.

Raster data is inherently sorted, the pixels can be indexed based on the tiles in the raster layer to which they belong as well as based on their row and column numbers. Hence, the key idea of the proposed algorithm is that we exploit this internal structure of the raster data, and produce an intermediate compact representation of the vector data, called *intersection file*, which perfectly matches the order of the raster data. Furthermore, to produce the intersection file, we only need to process the vector layer and the *metadata*

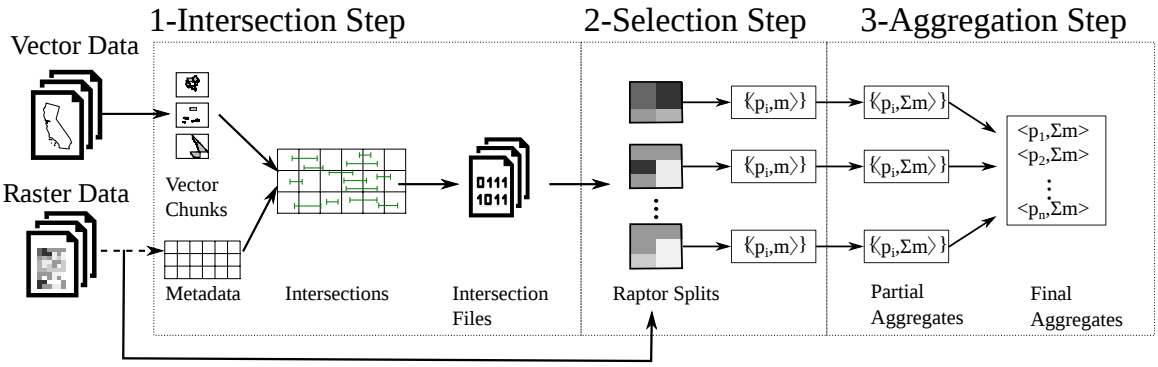


Figure 3.1: Overview of the Raptor Zonal Statistics (RZS) algorithm

of the raster layer which means that the raster dataset needs to be scanned only once. Finally, the intersection file is generated and stored in a distributed fashion which allows the proposed algorithm to be parallelized over a cluster of machines.

This algorithm as shown in Figure 3.1 runs in three steps, namely, *intersection*, *selection*, and *aggregation*. The *intersection* step computes the intersection file which captures the intersections between the vector and raster data and sorts these intersections to match the raster data. The *selection* step uses the intersection file to read the pixels in the raster layer that intersect each polygon in the vector layer. Finally, the *aggregation* step groups the pixel values by polygon ID and computes the desired aggregate function, e.g., average. The details of the three steps are given below.

Step 1: Intersection

This step runs as a map-only job and is responsible for *intersection file generation*. It takes as input the vector layer and the *metadata* of the raster layer and computes a common structure, called *intersection file*, which is stored in the distributed file system to

be used in the selection step. The vector layer consists of a set of polygons each represented as a list of straight line segments. The metadata of the raster layer consists of the dimensions (number of rows and columns), two affine matrix transformations $\mathcal{G}2\mathcal{W}$ and $\mathcal{W}2\mathcal{G}$ (which define mapping from raster to vector layer and vice-versa), and the size of each tile in the raster layer, i.e., number of rows and column, and is only a few kilobytes.

The input vector layer is partitioned into fixed-size chunks, 128 mb by default, and each chunk is assigned to one task. While any partitioning technique works fine, we employ the R*-Grove partitioning technique [74] which maximizes the spatial locality of partitions while ensuring load balance.

For each chunk, this step computes all the intersections between each row in the raster layer and each segment of each polygon in the vector layer. To compute these intersections, each line segment is mapped to the raster layer using the $\mathcal{W}2\mathcal{G}$ transformation to find the range of rows that it intersects. Then, it is a simple constant-time computation to find the x-coordinate of the intersection. Since we only need to know the intersection at the pixel level, the intersection is mapped to the raster space and the integer coordinate of the pixel is computed. We record the intersection as the triple $\langle p_{id}, x, y \rangle$ where p_{id} is the polygon ID to which the segment belongs and (x, y) is the coordinate of the intersection in the raster layer. All these triplets are kept in memory and can be spilled to disk if needed.

After all the intersections are computed, we run a sorting phase which sorts the intersections lexicographically by (t_{id}, y, p_{id}, x)[1], where t_{id} is the raster tile ID that contains the intersection. Notice that the raster tile ID does not have to be explicitly stored since it can be computed in a constant time for each intersection using metadata of raster layer.

[1] the sorting order becomes (t_{id}, x, p_{id}, y) if the raster file is stored in column-major order

Finally, the sorted intersections are stored in the intersection file as illustrated in Figure 3.2. The figure shows multiple intersection files, one for each chunk in the vector data. Since each chunk is processed on a separate machine, the intersection files are computed and written in a fully distributed manner. In each intersection file, the intersections are stored in the sorted order (t_{id}, y, p_{id}, x). As mentioned earlier, t_{id} is not physically stored to save space and is computed as needed. If there are multiple raster files, all intersections for the first raster file are stored first, followed by the second raster file and so on. This imposes a logical partitioning of the intersection file by t_{id} as illustrated by the dotted lines in the figure. In addition, we append a footer to each intersection file which stores a list of polygon IDs that appear in this file, the number of tile IDs for each raster file in this file, and a pointer to the first intersection in each raster tile ID t_{id}. Since intersection files contain intersections in only tiles overlapping the vector chunk, we store the combined extent of the tiles that appear in the file, to map each tile to its raster tile ID as and when required. This footer is only a few kilobytes for a hundred megabyte file and does not impose any significant overhead.

Step 2 : Selection

This step uses the intersection files produced in the first step to select all the pixels that are contained within each polygon. The input to this step is the intersection files and the raster layer while the output is a set of pairs $\langle p_{id}, m \rangle$ where p_{id} is the ID of the polygon and m is the measurement in the pixel. Notice that these values are pipelined to the next step and are not physically stored as there are typically hundreds of billions of these pairs.

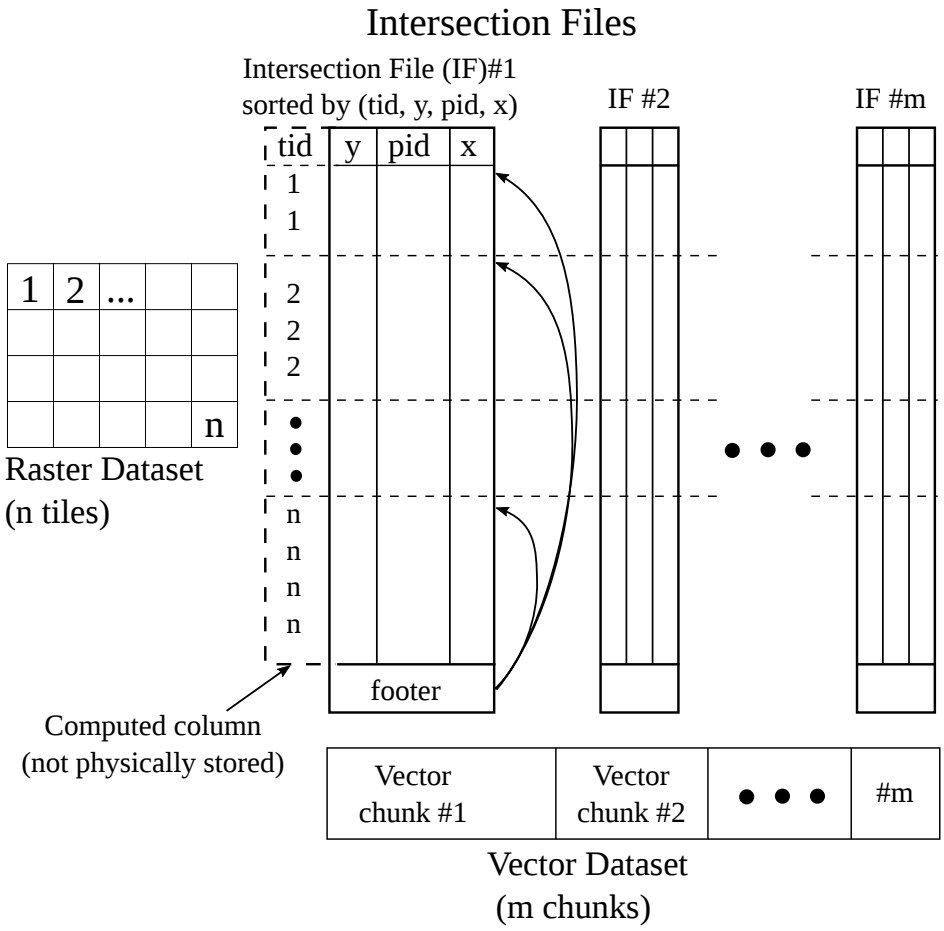


Figure 3.2: Intersection file structure

This step runs in two phases, namely *Raptor Split Generation* and *Raptor Split Processing*. The first phase creates a list of tasks that is distributed among machines to perform the parallel computation. The second phase processes the raster files and intersection files as defined by the Raptor splits.

Raptor Split Generation

The *Raptor Split Generation* phase generates *Raptor Splits* using the *RaptorInputFormat*. In Hadoop, the InputFormat is the component that splits the input file(s) into roughly

equi-sized splits to be distributed on the worker nodes. These splits are mapped one-to-one to mappers. Therefore, each split defines a unit of work called map task. A corresponding *record reader* uses the split to extract key-value pairs that are sent to the map function for processing. Since our unit of work is a combination of raster plus vector data, we define the new *RaptorInputFormat*, *RaptorSplit*, *RaptorRecordReader*, and *RaptorObject*, described as follows. Starting with the smallest one, the RaptorObject contains a vector chunk ID $[1, m]$, a raster file ID $[1, r]$, and a raster tile index $[1, n]$. The RaptorObject defines the smallest unit of work done by the map function. The RaptorSplit stores a vector chunk ID, a raster file ID and a *range* of tile indexes. The RaptorSplit defines the task given to each mapper. We can control the amount of work given to each machine by adjusting the number of tiles in the RaptorSplit. The RaptorRecordReader takes one RaptorSplit and generates all the RaptorObjects that it represents. Finally, the RaptorInputFormat takes all the input to the problem, i.e., the raster dataset and all intersection files, and produces a list of RaptorSplits that define the map tasks given to worker nodes.

The information needed to create the RaptorSplits is contained in the *footers* of the intersection files. As shown in algorithm 1, *Raptor Split Generation* takes as input a 2-d array called *TileSizes* and the number of mappers, *M*. *TileSizes* is an array of size $m \times r$, where the first dimension is a vector chunk ID in the range $[1, m]$, and the second dimension is a raster file ID in the range $[1, r]$. It contains the number of intersecting tiles for each combination of vector chunk and raster file, extracted from the footers of their respective intersection files. **Line 1** calculates the total number of tiles, *TotalTiles* among all the vector chunks. To ensure load balance among workers, we assign each RaptorSplit

Algorithm 1: Raptor Split Generation

Input: TileSizes$[m][r]$: Number of intersecting tiles for each VectorChunkID$\in [1, m]$ and RasterFileID$\in [1, r]$

M: Number of mappers

Output: RaptorSplits : {(VectorChunkID, RasterFileID, StartTileIndex, EndTileIndex)}

```
/* calculate total number of tiles                                    */
```

1 TotalTiles = Sum(TileSizes)

```
/* calculate number of tiles per RaptorSplit based on number of mappers
   available                                                          */
```

2 TilesPerSplit = Max$\left(10, \frac{TotalTiles}{M}\right)$

3 **for** $VectorChunkID \leftarrow 1$ **to** m **do**

4 StartTileIndex = 0

5 **for** $RasterFileID \leftarrow 1$ **to** r **do**

6 NumTiles = TileSizes[VectorChunkID][RasterFile]

7 EndTileIndex = Min(NumTiles, StartTileIndex + TilesPerSplit)

8 RaptorSplits.add(VectorChunkID, RasterFileID, StartTileIndex, EndTileIndex-1)

9 StartTileIndex = EndTileIndex

a maximum number of tiles. **Line 2** calculates *TilesPerSplit* by dividing the *TotalTiles* by the number of mappers for load balance; this value is upper-bounded at 10 to cap the amount of work per split. **Lines 3-9** generates the RaptorSplits. For each vector chunk and raster file, it generates a set of *StartTileIndex*es and *EndTileIndex*es. Note that these are indexes to the raster tile IDs in their respective intersection files. Information in the footer of the intersection file facilitate the mapping of indexes to actual tile IDs. The *VectorChunkID*, *RasterFileID* the *StartTileIndex* and the *EndTileIndex* are then added to a list of RaptorSplits.

The use of tile indexes allows this step to effectively prune all the tiles that do not contribute to the answer. In case of multiple raster files, this phase limits the range of tile IDs in each RaptorSplit to only one file, as can be seen in algorithm 1 This ensures that each machine in this step will need to process only one file. On the other hand, if the raster layer is stored in one big file, we can still split that file among multiple machines for efficient processing.

Raptor Data Processing

This phase takes a RaptorObject, which contains a vector chunk ID, raster file ID, and a tile index, and generates a set of pairs $\langle p_{id}, m \rangle$ where p_{id} is a polygon ID and m is a measurement of a pixel contained in that polygon. In Spark, this phase can be implemented as a `flatMap` transformation and in Hadoop, it can be implemented as part of the `map` function. Algorithm 2 shows the pseudo-code of this phase. **Line 1** maps the tile index to a tile ID t_{id} from the footer of the intersection file. **Line 2** loads the tile t_{id} from the raster file as a two-dimensional array. A tile is typically small enough to fit in main-memory. Then,

Algorithm 2: Raptor Data Processing

Input: RaptorObject: (VectorChunkID, RasterFileID, TileIndex)

Output: $\{\langle p_{id}, m\rangle\}$

```
1  t_id = getTileID(VectorChunkID, TileIndex)
2  Tile = loadTile(RasterFileID, t_id)
3  Intersections = getIntersections(VectorChunkID, RasterFileID, t_id)
4  for i ← 1 to Intersections.length by 2 do
5      y = Intersections[i].y
6      p_id = Intersections[i].pid
7      x1 = Intersections[i].x
8      x2 = Intersections[i + 1].x
9      for x ← x1 to x2 do
10         m = Tile.getPixel(y, x)
11         output.add(p_id, m)
```

Line 3 loads the intersections for the given tile and vector chunk as shown in Figure 3.2. The for loop in **Line 4** iterates over these intersections in order; each pair of consecutive intersections represents a range of pixels in the tile that are inside the polygon. The loop in **Line 9** iterates over these pixels, reads each one, and output the pair $\langle p_{id}, m\rangle$ where p_{id} is the polygon ID and m is the measure value of the corresponding pixel. These pairs are aggregated in the next step as described shortly.

Step 3: Aggregation

In this last step, the set of pairs $\langle p_{id}, m \rangle$ are aggregated to produce final aggregate values $\langle p_{id}, \sum m \rangle$, where $\sum$ is any associative and commutative aggregate function. As in most distributed systems, the aggregate function is computed in two phases, *partial* and *final* aggregates. The partial/final computation supports any function that is both associative and commutative which covers a wide range of aggregate functions. The *partial* aggregates are computed locally in each machine while the *final* aggregates are computed by combining all the partial aggregates for the same group in one machine. In Spark, this can be implemented as a `aggregateByKey` transformation while in Hadoop this can be implemented as a pair of `combine` and `reduce` functions.

3.5 Theoretical Analysis

This section presents a theoretical analysis of the proposed algorithm, *Raptor Zonal Statistics* (RZS) and the baseline *Raster Database Approach* (RDA). For conciseness, we focus only on the disk IO cost of these two algorithms; previous work showed that the vector database approach (VDA) is not competent [16]. Table 3.1 summarizes the parameters used throughout the analysis.

3.5.1 Raster Database Approach (RDA)

The RDA algorithm scans the polygons and for each polygon it clips the overlapping portion of the raster layer and aggregates the pixel values. We first estimate the average disk cost per polygon $\overline{D_{RDA}}$ and the total cost is simply computed by multiplying

Table 3.1: Parameters for Cost Estimation

Symbol	Meaning
r	Number of rows in raster
c	Number of columns in raster
w_t	Tile width in pixels
h_t	Tile height in pixels
p	Pixel size in degrees
n_p	Number of polygons
n_s	Number of line segments in all polygons
$\overline{n_s} = \frac{n_s}{n_p}$	Average number of line segments per polygon
w_p	Average polygon width in longitudinal degrees
h_p	Average polygon height in latitudinal degrees
I	Input size in bytes
B	HDFS block size in bytes
C	Chunk size. Number of polygons per chunk

by the total number of polygons n_p. The key point in this analysis is that the smallest access unit in raster files is a tile of size $w_t \times h_t$ pixels. Raster data is stored in compressed blocks of that size so the entire block has to be decompressed even if only one pixel needs to be processed. On average, each polygon overlaps with n_t tiles as calculated below:

$$n_t = \left\lceil \frac{w_p}{w_t \cdot p} \right\rceil \times \left\lceil \frac{h_p}{h_t \cdot p} \right\rceil \tag{3.1}$$

where w_p and h_p are the average width and height of a polygon in *degrees*. This makes $\frac{w_p}{p}$ and $\frac{h_p}{p}$ the average width and height of a polygon in *pixels*. Next, we can calculate the average amount of disk access per polygon $\overline{D_{RDA}}$ as follows:

$$\overline{D_{RDA}} = \left\lceil \frac{w_p}{w_t \cdot p} \right\rceil \times \left\lceil \frac{h_p}{h_t \cdot p} \right\rceil \times w_t \times h_t \tag{3.2}$$

Hence, the total disk cost for RDA, $D_{RDA} = n_p \times \overline{D_{RDA}}$.

3.5.2 Raptor Zonal Statistics (RZS)

In RZS, the unit of work is a *vector chunk*. The input file is first loaded into HDFS and split into blocks using the R*-Grove [74] partitioning algorithm. Then, each block is further split into chunks of C polygons each. In the following part, we first analyze the characteristics of each chunk, i.e., width and height, and then we present the cost estimate for each one.

Vector Chunks: Given a vector file of size I, R*-Grove will create n_B blocks with roughly equal size. The number of blocks is $n_B = \lceil I/B \rceil$. The average number of pixels covered by each block $= c \cdot r/n_B$. Assuming square-like partitions, the width and height of each block is:

$$w_B = h_B = \sqrt{c \cdot r/n_B} \tag{3.3}$$

Next, each block is split into chunks where each chunk has roughly C polygons each. Since a block is split based on number of polygons not the location of the polygons, the width and height of each chunk is equal to that of a block. Keep in mind that the above width and height are in pixels.

$$w_c = h_c = \sqrt{c \cdot r / n_B} = \sqrt{\frac{c \cdot r}{\lceil I/B \rceil}} \tag{3.4}$$

Disk Access per Chunk: To estimate the disk access per chunk, we follow a similar analysis to the one we did with RDA where we need to read all tiles that intersect the chunk. Unlike RDA where the same tile can be read many times for all overlapping polygons, the proposed intersection file allows us to read each tile at most once. Therefore, the average disk access per chunk for RZS is calculated as follows:

$$\overline{D_{RZS}} = \left\lceil \frac{w_c}{w_t} \right\rceil \times \left\lceil \frac{h_c}{h_t} \right\rceil \times w_t \times h_t \tag{3.5}$$

Assuming balanced partitioning of the n_p polygons in the input into equi-sized chunks of C polygons each, the number of chunks would be $n_c = \lceil n_p/C \rceil$.

Finally, the overall disk access for RZS is:

$$D_{RZS} = \lceil n_p/C \rceil \cdot \overline{D_{RZS}} \tag{3.6}$$

This approach is analogous to *sort-merge join*, which achieves a very efficient computation time by sorting only the smaller dataset, i.e., the vector dataset. Unlike the traditional sort-merge algorithm, the sorted files might be scanned several times due to the overlap between chunks.

3.5.3 Discussion

The disk IO cost of RZS is much lower than that of RDA because the size of vector chunks is much bigger than that of a single polygon. This results in much smaller overlap between chunks, hence, lower disk IO cost. Notice that this is only possible thanks to the intersection file structure which enables RZS to transform the complex computation of multiple, possibly overlapping polygons, to a single scan over the raster data. In the experiments section, we will verify the accuracy of our model using real data.

3.6 Experiments

This section provides an experimental evaluation of the proposed algorithm, Raptor Zonal Statistics (RZS). We compare the distributed RZS algorithm to the single-machine Scanline Method [16], Rasdaman (a RasterDB approach) [3], Google Earth Engine (GEE) [28] and our previous distributed algorithm EMI [65]. We show that the proposed RZS is up-to three orders of magnitude faster than Rasdaman, is twice as fast as GEE while using an order of magnitude fewer machines and is able to scale to much bigger vector datasets than EMI. We evaluate them on real data and also show the effect of vector partitioning on RZS.

Section 3.6.1 describes the experimental setup, the system setup, and the datasets. Section 3.6.2 provides a comparison of the proposed RZS, Scanline method, EMI, Rasdaman, and Google Earth Engine based on the total running time. It is followed by a discussion of the vector and raster dataset ingestion time for each of these methods in Section 3.6.3. Section 3.6.5 gives a verification of the proposed cost model for RZS and RDA methods.

Table 3.2: Vector and Raster Datasets

Vector datasets

Dataset	n_p	n_s	$\overline{w_p}$	$\overline{h_p}$	File Size I
Counties	3k	52k	0.82	0.51	978 KB
States	49	165k	12.18	4.28	2.6 MB
Boundaries	284	3.8m	18.61	8.18	60 MB
TRACT	74k	38m	0.096	0.068	632 MB
ZCTA5	33k	53m	0.19	0.15	851 MB
Parks	10m	336m	0.0067	0.0043	8.5 GB

Raster datasets

Dataset	image size $c \times r$	tile size $w_t \times h_t$	resolution p
glc2000	$40,320 \times 16,353$	128×128	0.0089
MERIS	$129,600 \times 64,800$	256×256	0.0027
US-Aster	$208,136 \times 89,662$	208136×1	0.00028
Tree cover	$1,296,036 \times 648,018$	36001×1	0.00028

Section 3.6.6 discusses two applications where the proposed RZS algorithm has been used. The next three sections discuss the effect of various parameters on the total running time of RZS. Section 3.6.7 shows the effect of the size of vector chunks on the performance of RZS. Section 3.6.8 and Section 3.6.9 shows the effect of compression of intersection files and the effect of partitioning the vector data on the total running time of RZS.

3.6.1 Setup

We run all the experiments on a Amazon AWS EMR cluster with one head node and 19 worker nodes of type m4.2xlarge with 2.4 GHz Intel Xeon $E5-2676$ v3 processor, 32 GB of RAM, up to 100 GB of SSD, and 2×8-core processors. The methods are implemented using the open source GeoTools library 17.0.

In all the techniques, we compute the four aggregate values, minimum, maximum, sum, and count. We measure the end-to-end running time as well as the performance metrics which include reading both datasets from disk and producing the final answer. Table 3.2 lists the datasets that are used in the experiments along with their attributes using the terminology in Table 3.1. The vector layers represent the US continental counties and US continental states with 3,000 and 49 features respectively. The Large-Scale International Boundaries (LSIB) includes geographic national boundaries for 249 countries and disputed areas. The TRACT and ZCTA5 datasets are a part of TIGER 2017 dataset. The *parks* datasets is derived from OpenStreetMap and contains park boundaries over the entire world. The raster datasets come from various government agencies. The GLC2000 and MERIS 2005 datasets are from the European Space Agency with pixel resolutions of 0.0089 decimal degrees (1km) 0.0027 (300m) respectively. The US Aster dataset originates from the Shuttle Radar Topography Mission (SRTM) and covers the continental US. Hansen developed the global Tree Cover change dataset which covers the entire globe. Both datasets have a spatial resolution of 0.00028 decimal degrees (30m).

RZS is implemented in Hadoop 2.9. We chose to implement it in Hadoop rather than Spark for two reasons. First, it was simpler because Hadoop can easily support custom

input format, i.e., the RaptorInputFormat while Spark does not have a specialized method to define a new input format; it just reuses Hadoop's InputFormat architecture. Second, Spark is optimized for in-memory processing while RZS is a disk-intensive query that does not have a huge memory footprint.

For RDA, we used Rasdaman 10.0 running on a single machine since the distributed version is not publicly available. We also used Google Earth Engine (GEE) which runs on Google Cloud Engine. GEE is still experimental and is currently free to use. The caveat is that it is completely opaque and we do not know which algorithms or how much compute resources are used to run queries but it uses up-to 1,000 nodes according to their published report [28]. We run each operation on GEE 3-5 times at different times and report the average to account for any variability in the load. For large vector data, we hit the limit of GEE of 2GB vector file. To work around it, we split the file into 2GB smaller files, run on each file separately, and add up the results. All the running times are collected as reported by GEE in the dashboard.

3.6.2 Overall Execution Time

This part compares Raptor Zonal Statistics (RZS), Scanline, EMI, Rasdaman, and Google Earth Engine (GEE) based on the end-to-end execution time. This experiment is run for all the combinations of vector and raster datasets shown in Table 3.2, and its results can be seen in Figure 3.3. For the cases when Rasdaman takes more than 48 hours, we extrapolate the results based on our cost model and mark them with a dotted line. All experiments on RZS run on a cluster of 20 machines except the TreeCover dataset which

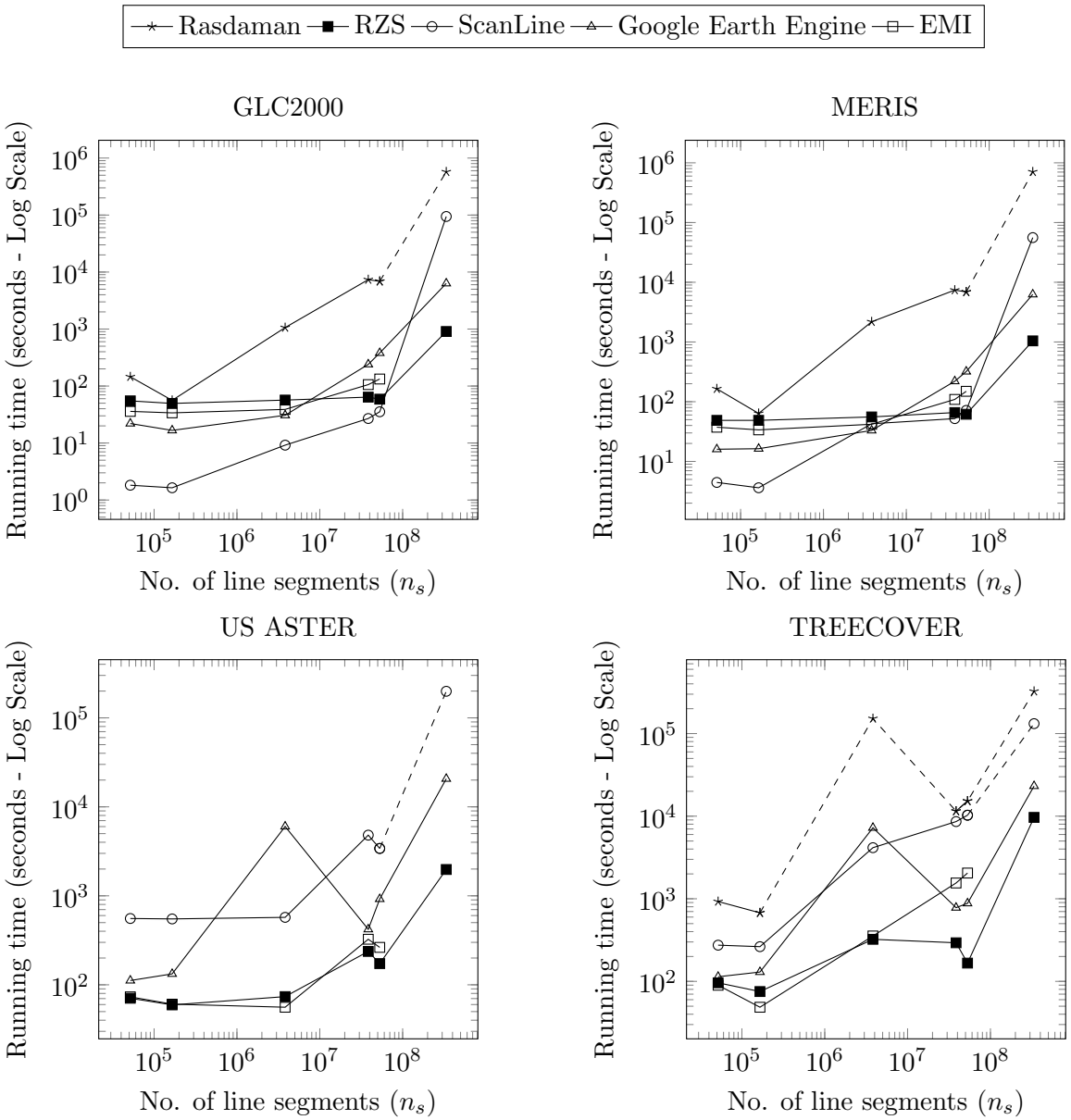


Figure 3.3: Comparison of total running time of RZS, Scanline, EMI, GEE and Rasdaman

runs on 100 nodes due to its huge size. This is done in order to provide a comparison to GEE which may run on from 100 nodes to up-to 1,000 nodes [28].

As can be observed from Figure 3.3, the proposed distributed RZS algorithm is orders of magnitude faster than Rasdaman for all combinations of raster and vector datasets. RZS is up-to two orders of magnitude faster than GEE and twice as fast for the largest input (Parks⋈TreeCover), despite using an order of magnitude less machines.

Rasdaman failed to ingest the US-Aster file due to its huge size (48GB as a single BigGeoTIFF file). In addition to being a single machine, Rasdaman does not scale due to using the RDA method which scans the polygons, clips the raster layer for each polygon, and aggregates the clipped values. Based on the overlap of polygons and raster tiles, the same tile could be read tens of times. RZS overcomes this problem by generating intersection files that are ordered based on the raster file structure to reduce the number of scans of the raster file.

When compared to **GEE** for large datasets, RZS is at least 2x faster for the large datasets and up-to two orders of magnitude faster. This is an impressive result knowing that GEE runs on up-to a 1,000 machines. In particular, the speedup of RZS is much higher for large vector datasets since GEE is a raster-oriented system that does not handle big vector data well [28]. While GEE is faster for small vector datasets, this is due to the overhead of using Hadoop for small inputs. Indeed, the single machine Scanline [16] algorithm is an order of magnitude faster than both for these small datasets, e.g., GLC2000 and MERIS and small vector data. Therefore, if we want to be always faster than GEE, we just need to switch to Scanline for small datasets but we leave this for a future work. GEE is a free

tool (for now) but the knowledge of how it implements the zonal statistics operation[2] is not public. Also, the amount of resources available to users at any time can vary and this is why we report the average of 3-5 runs.

When compared to the single-machine **Scanline** we observe two orders of magnitude speedup of RZS due to the parallel implementation. We also observe that Scanline cannot scale to large vector data due to the limitations of the intersection step which runs on a single machine and that is why the straight-forward parallelization of Scanline on Hadoop [65], **EMI** does not scale either. RZS shows performance on par or better than the previously proposed EMI, however, EMI runs out of memory for the parks dataset, which is the largest vector dataset.

3.6.3 Ingestion Time

Figure 3.4 shows the ingestion time of the raster and vector datasets for the proposed RZS algorithm, Rasdaman, and GEE. Scanline method can read data from disk and hence does not have the overhead of ingestion time. RZS algorithm requires both raster and vector datasets to be stored into HDFS, while GEE requires them to be uploaded to its web interface as well. We do not upload US Aster and Treecover to GEE as they are available in its data repository. Rasdaman only requires to ingest the raster datasets, although it failed to ingest the US Aster dataset as explained earlier.

As can be observed from the Figure 3.4 RZS has a lower raster data ingestion time as compared to Rasdaman and GEE. Figure 3.4 also shows that RZS has an order of magnitude lower vector data ingestion time when compared to GEE. The reason for that

[2]reduceRegions function in Google Earth Engine (GEE)

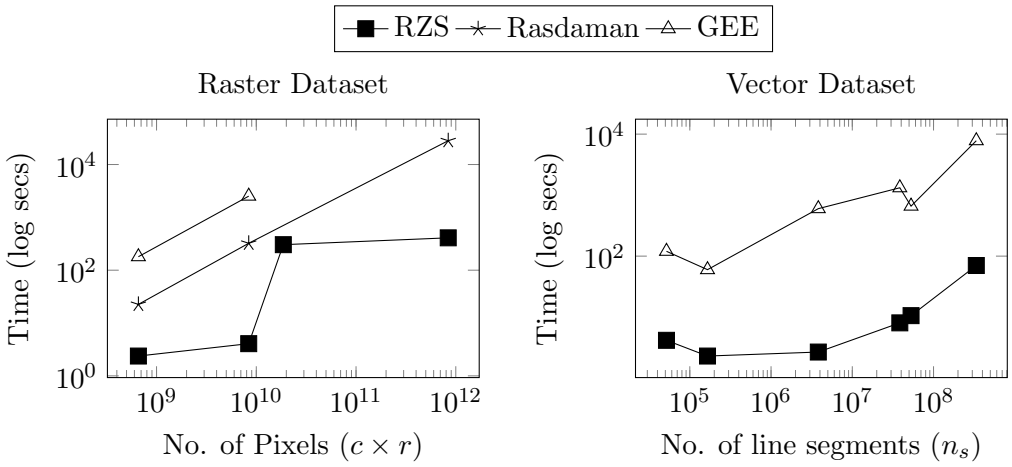


Figure 3.4: Ingestion time

is that RZS follows an in-situ data processing methodology which does not need to read the data until the query is executed. This makes it a perfect choice for ad-hoc exploratory queries.

3.6.4 Closeup Scalability of Rasdaman

Since Rasdaman preloads the raster data but not the vector data, it could be a good choice if only a few polygons need to be processed on a large raster dataset. Figure 3.5 shows the results of the zonal statistics query using the MERIS dataset and varying the number of polygons in the parks dataset. It can be observed that Rasdaman is optimized for a very few polygons. However, its running time has a steep ascent as the number of polygons increases. This happens because Rasdaman processes each polygon in a separate query which results in overlapping work. RZS is able to scale more steadily as it can combine the overlapping work using the intersection file structure.

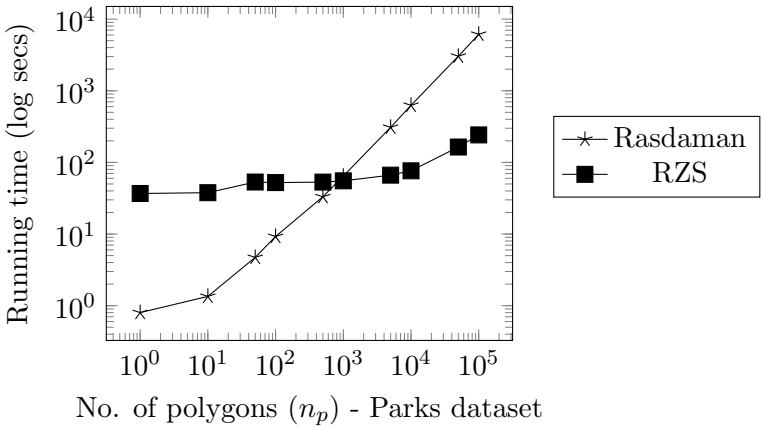


Figure 3.5: Scalability of Rasdaman and RZS on MERIS dataset

3.6.5 Verification of Cost Models

This part verifies our cost models described in Section 3.5 and uses that analysis to better explain the results that we got earlier. Since zonal statistics is a disk IO-intensive operation, we only use the disk cost estimation. Also, to be able to compare the actual running time to the estimated cost, we normalize all values and compare the trends rather than the absolute values. All cost models are computed based on the parameters in Table 5.1, the system parameters $B = 128MB$ and $C = 5,000$, and the equations in Section 3.5.

Figure 3.6 compares the actual running time of RZS to the estimated cost (both normalized) when processing GLC2000 and TreeCover datasets. As shown in the figure, the trends generally match for both the small and the large datasets showing the robustness of the cost model. Notice that there is still some deviation due to our assumptions of uniform distribution of the vector data which does not hold in reality. To quantify the relationship, we calculated Spearman's correlation coefficient for both cases and it turned out to be 0.94 and 0.77 for GLC2000 and TreeCover respectively. Notice that we did not use the

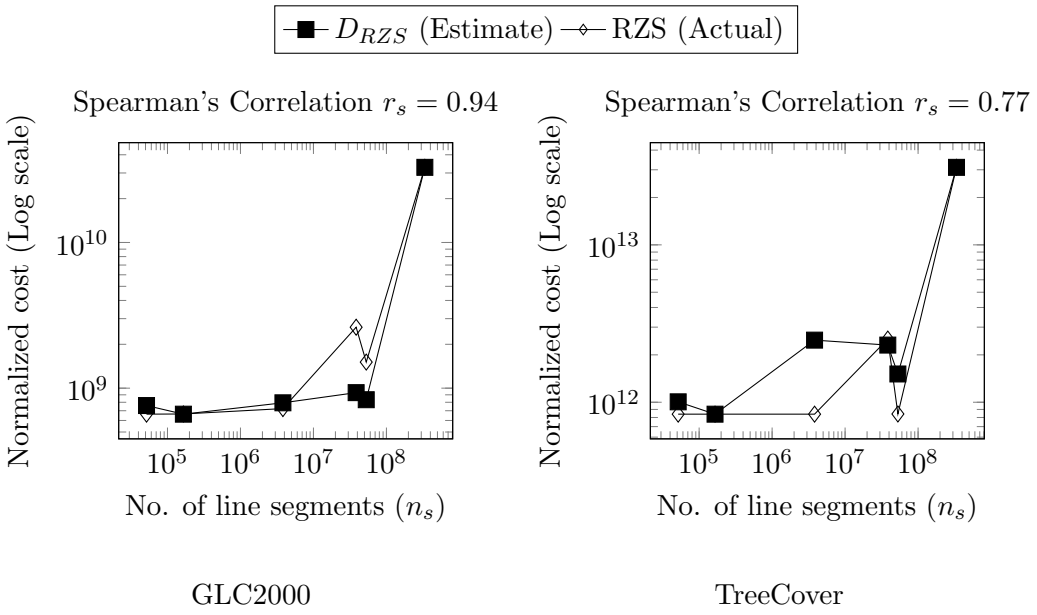


Figure 3.6: Verification of the cost model of RZS

Pearson's correlation coefficient as it will result in a *misleading* value of almost 0.999 due to the exponential increase on the x and y values.

Figure 3.7 shows the estimated cost of Raster Database Approach (RDA) and the actual cost of both Rasdaman and GEE. It is evident from the chart that the trend of the cost model and the actual times are very similar. While we do not have definite information about GEE, we highly predict that they use the RDA algorithm given the almost perfect match in trends. The correlation coefficient with Rasdaman in GLC2000 is perfect, and for GEE 0.9 and 0.83. We did not have enough data points for Rasdaman with the TreeCover dataset to plot the figure or calculate the coefficient. We believe the cost model for RDA is more accurate since it does not make an assumption about the uniformity of the data as we do with RZS. Still, it is amazing how accurate the results are given that we only relied on the statistics shown in the Table 3.2.

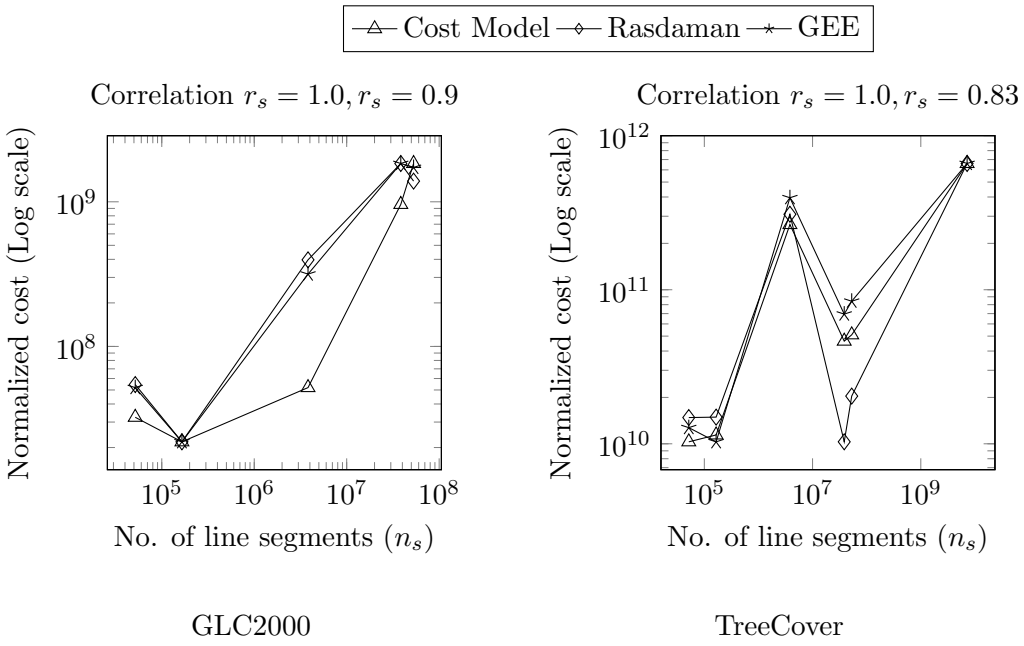


Figure 3.7: Verification of the cost model of RDA

3.6.6 Applications

This section discusses two real-life applications of our proposed system, *population estimation* and *wildfire combating*.

The first application of RZS is to estimate the population of arbitrary regions using landcover data [56]. The problem is that the US Census Bureau reports the population at the granularity of *census tracts* which are regions chosen by the Bureau to keep the privacy of the data. Areal interpolation transforms these counts from source polygons, i.e., tracts, to target polygons, e.g., ZIP Codes, with unknown counts. One accurate method [56] uses the National Land Cover database (NLCD) [46] raster dataset as a reference to disaggregate the population counts into pixels and then aggregate them back into target polygons. To speed up the process, we apply RZS to compute the histogram for each polygon in the

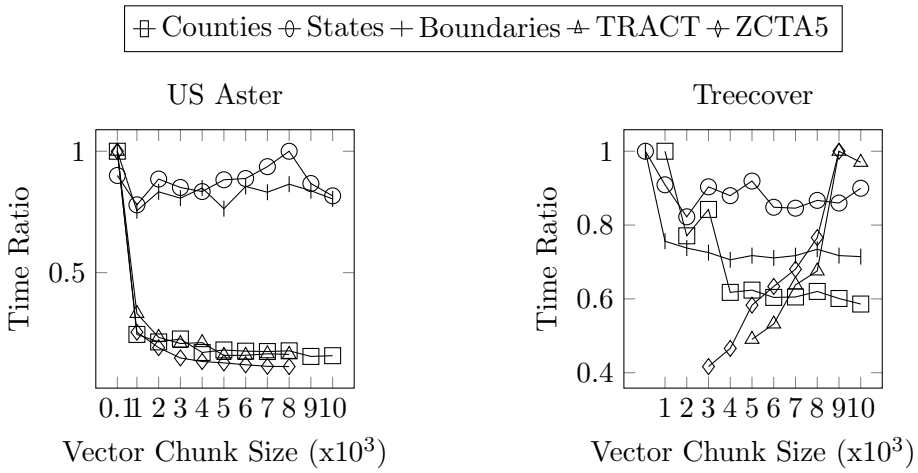


Figure 3.8: Effect of vector chunk size on total running time

TRACT dataset on the NLCD dataset. We compared our single-machine implementation to the original Python implementation used by the developers of that algorithm, which was a vector database approach (VDA). Using RZS, the entire process completed in 10 seconds for the state of Pennsylvania while the python-based script took over 100 minutes to complete. Given that impressive speedup, the authors were able to scale their work to the entire US which took under 2 hours on a single machine.

The second application of RZS is in combating wildfires. The goal of this application is to co-create probabilistic decision theoretic models to combat wildfires, which would use RZS for pre-processing satellite wildfire data. The raster data used has a size of 60 GB and the vector data has over 3 million polygons, both spanning over California. We have been able to compute zonal statistics for it in under 2 hours using an AWS cluster of 20 machines.

3.6.7 Vector Chunks

This experiment studies the effect of splitting the vector file into chunks. In particular, this experiment varies the sizes of vector chunk used in RZS starting with 100, then 1000 to 10,000, incremented in steps of 1000. Figure 3.8 shows the overall running time for the two larger raster datasets as the chunk size increases. Since each line in this graph represents a different vector dataset, we are only interested in the trend of the lines. Therefore, each line is normalized independently to fit all of them in one figure. We omit the running times for the extreme cases that run out of memory. We observe in this experiment that a very low chunk size of 100 results in a reduction in the performance due to the overhead of creating and running too many RaptorSplits. On the other hand, using a very large chunk size eventually results in some job failures due to the memory overhead. This is equivalent to not splitting the vector file.

After chunk size of 3000, the number of vector chunks generated for Counties and Boundaries becomes stable which leads to marginal variation in their running times. The variation of chunk size on larger vector datasets and Tree Cover is more prominent than the other vector datasets. This is due to the large size of the Tree Cover dataset. The increase in vector chunk size leads to a decrease in the number of chunks being generated and hence, less number of *raptor splits*. This leads to each machine having more amount of work to do, and a non-optimal distribution of work. It can be concluded that the choice of vector chunk size should neither be too big (10,000) nor too small(100). We chose it to be 5,000 based on the experiments and according to our system configuration.

Table 3.3: Compression Ratio of Intersection Files

	GLC2000	MERIS	US Aster	Tree Cover
Counties	4.13	4.18	4.34	5.22
States	3.41	3.34	3.3	6.07
Boundaries	2.56	2.55	3.68	17.43
TRACT	2.94	3.21	3.45	4.9
ZCTA5	2.88	3.11	3.34	4.64

3.6.8 Compression of Intersection File

In this experiment, we study the effect and trade-off of compressing the intersection files. The size of the raw (uncompressed) intersection files ranges from a few kilobytes to a gigabyte. In order to reduce the network and disk IO when storing this file in the distributed file system, we investigated the option of compressing the intersection files using both GZIP and Snappy compression libraries. We did not see a major difference between the two libraries so we are only reporting the results of GZIP. The speedup of the approach without using compression to the approach with the use of compression can be seen in Figure 3.9. It can be observed that the speedup is either equal to one or marginally less than it. This means that the total running time without the use of compression is less than or almost equal to that with compression. Table 3.3 reports the compression ratio of intersection files, which is defined as the ratio of the size of uncompressed file to that of the compressed file. Although, from Table 3.3, it can be seen that the size of compressed files is far smaller than

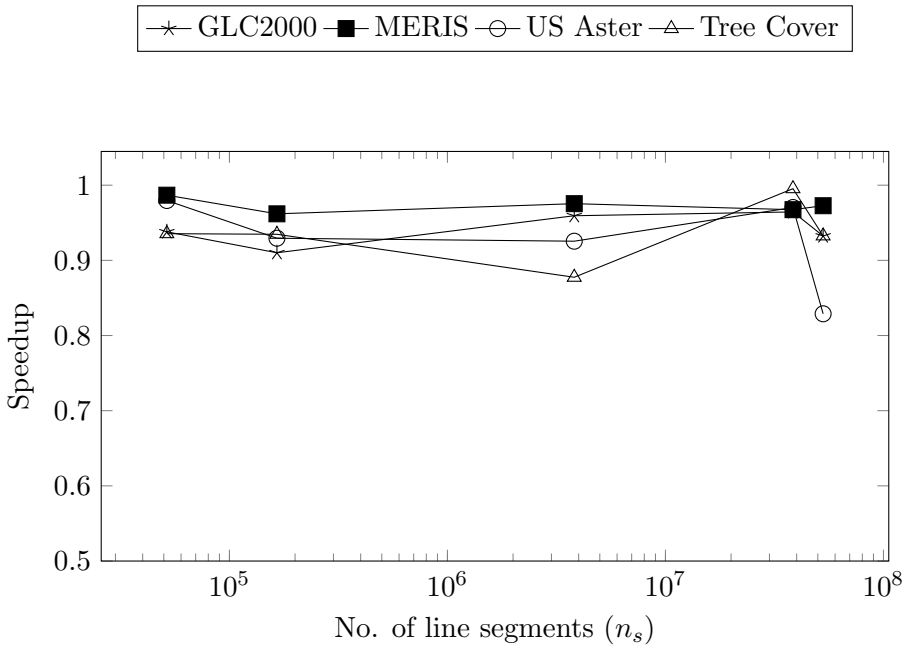


Figure 3.9: Effect of compression of Intersection File

that of non-compressed *intersection files* this did not provide a significant improvement in the running time. This is because the time saved in writing the compressed intersection file is nearly the same as the time taken for compression and decompression of the intersection file. However, compressing the intersection file can be a viable option, in case network IO becomes a bottleneck.

3.6.9 Spatial Partitioning of Vector Data

This experiment shows the effect of spatial partitioning input vector data on the total running time. We use R*-Grove [74] algorithm to partition the vector data. The results for the total running time of the proposed algorithm with partitioned and non-partitioned vector data are shown in figure 3.10. As can be observed from the figure,

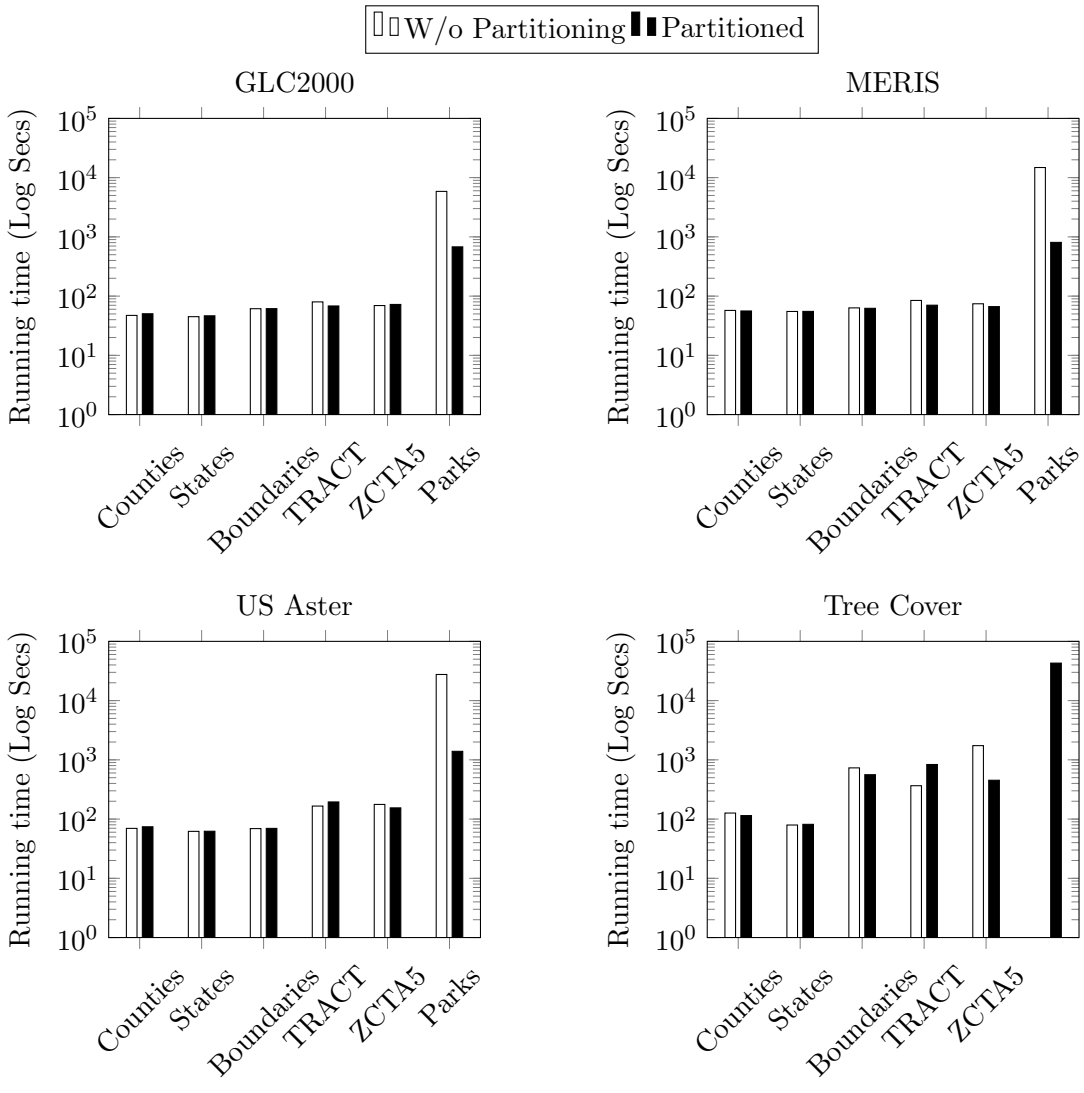


Figure 3.10: Effect of spatially partitioning vector data

Table 3.4: Time taken to Partition Vector Datasets

Vector Dataset	Partitioning Time
Counties	22.97s
States	38.92s
Boundaries	121.09s
TRACT	118.14s
ZCTA5	126.85s
Parks	209.40s

spatially partitioning the vector data helps achieve orders of magnitude of speedup for the large vector datasets (ZCTA5 and Parks). In addition, the largest combination of datasets, Parks×TreeCover took more than 48 hours with non-partitioned data and we had to terminate it. The total running time for other vector datasets either shows a marginal improvement or remains the same.

Spatial partitioning reduces the spatial extents of the contents of each HDFS block which also reduces the overlap with raster data. Not using spatial partitioning means that the contents of each block would cover the entire input space which results in reading the entire raster file when processing each intersection file. Our cost model can further explain the results of this experiment by setting the average width and height of a block to the width and height of the raster dataset, respectively. For small vector datasets with one block, there will be no difference between spatial and non-spatial partitioning. However, as the number of blocks increase the effect of spatial partitioning will be huge.

Table 3.4 reports the time taken by R*-Grove [74] algorithm to partition the vector datasets. As can be seen from the table, The partitioning though fast takes hundreds of seconds to partition the vector dataset. Parks was the only dataset, where the difference between running time of partitioned and non-partitioned input vector data is much larger than the partitioning time. It can thus be concluded that partitioning vector data is of advantage for very large vector datasets.

3.7 Conclusion

This chapter proposes a novel distributed MapReduce algorithm, Raptor Zonal Statistics (RZS) to solve the zonal statistics problem. First, RZS runs an intersection step that computes the *intersection file* which maps vector polygons to raster pixels. Second, the selection step concurrently scans the intersection file and the raster file to find the join result. To process the two files in parallel, the chapter introduces two new components RaptorInputFormat and RaptorSplit which define the smallest unit of work for each parallel task. Third, it runs an aggregate phase that computes the desired statistics for each polygon. Our experiments with large scale real data show that the proposed algorithm is up-to two orders of magnitude faster than the baselines including Rasdaman and Google Earth Engine (GEE). We also presented a cost model which helped us in explaining the results of both RZS and the baseline techniques.

Chapter 4

The Raptor Join Operator for Processing Big Raster + Vector Data

Through our collaborations, we realized that applications that combine raster and vector data use various operations that are not limited to zonal statistics. They might also need to use geometry types other than polygons. These applications would also require to process input data before combining them and then process output data once it is generated. Therefore, to increase the usability of the system and meet the demands of the ever-increasing size of data, we decided to implement the *Raptor Join* operator. This chapter talks about *Raptor Join*, a distributed implementation in Spark that can be used to efficiently process the combination of raster and vector data.

4.1 Introduction

Machine Learning has become a popular tool to analyze and utilize spatial data for research applications such as areal interpolation [56], wildfire risk assessment [34, 24], crop yield mapping [41], studying the effect of vegetation and temperature on human settlement [32, 33], and land use classification [57]. These applications often use spatial data from various sources and in different data representations. This makes it necessary to pre-process data and combine it into a single data representation before it can be used by the algorithm. Spatial data can be classified into two data representations: vector and raster. Vector data includes points, lines, and polygons, while raster data, such as satellite images, is represented as multidimensional arrays. The major differences between these two representations make combining them difficult. This is why existing systems are designed to either process vector data [80, 14, 45, 37, 52] or raster data [21, 3, 27, 70, 51]. These systems are efficient for vector-vector join or raster-raster join but do not implement a raster-vector join. In this chapter, we propose a new type of join, *Raptor Join* that can efficiently combine raster and vector data to implement complex spatial data analysis pipelines.

Existing systems support raster-vector join operation by converting one of the datasets so that both datasets become of the same representation. Figure 4.1 illustrates how raster- and vector-based systems support raster-vector join. First, raster-based systems rasterize the vector data to the same resolution as the raster data and then process them using raster-raster join. Similarly, vector-based systems vectorize the raster data by converting each pixel to a point and then join both datasets using a vector-vector join. These two approaches run fine for small and medium resolution data since the conversion

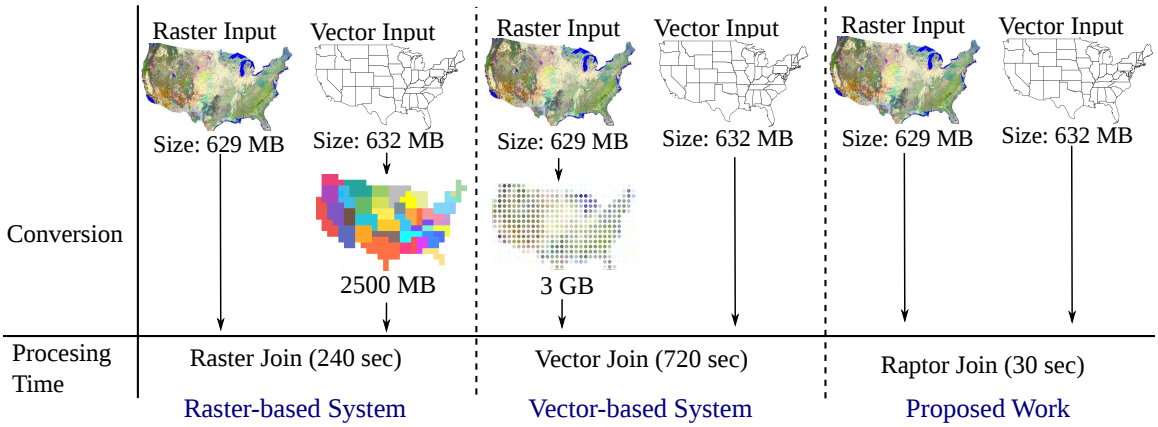


Figure 4.1: Comparison of raster, vector and raster+vector based systems

process does not dramatically increase the data size. However, with the recent availability of high-resolution satellite data, these approaches no longer scale. In fact, the converted data size increases *quadratically* with the resolution [66].

To further highlight the limitations of existing approaches, we start with raster-based systems. Existing raster-based approaches for the raster-vector join problem can be broadly categorized into two methods. First, on-the-fly method iterates over vector records, rasterize each record on-the-fly, and combine it with the raster data to retrieve the relevant pixels. This method suffers from the overhead of the rasterization step and the redundant access to raster data when geometries are close to each other. Second, the materialized method materializes all the rasterized data to increase the computation efficiency but it suffers from the huge size of the rasterized data. For example, a 632MB dataset that represents the 74k US Census tracts is rasterized to nearly 2.5GB of data at 1km resolution.

On the other hand, vector-based systems convert raster pixels to points and process them with the vector data as illustrated in Figure 4.1. Methods can be similarly categorized

as *on-the-fly* and *materialized* methods. The on-the-fly method first indexes the vector data and then scans the pixels, convert each pixel to a point, and process it with the index. This method suffers from the large index size and the overhead of accessing the index for each pixel. The materialized method first converts and materializes all pixels to points, and then runs an efficient distributed spatial join algorithm on the result. This method suffers from the huge size of the materialized data. For example, a 34MB compressed GeoTIFF file with 600 million pixels will be converted to nearly 3 GB of decompressed vector representation.

The proposed *Raptor Join* can concurrently process raster and vector data. Raptor Join ($\text{RJ}_{\bowtie}$) overcomes the limitations of existing systems as follows. First, $\text{RJ}_{\bowtie}$ is implemented using an in-situ approach in Spark [82] which does not require an expensive data loading phase. Second, it directly processes raster and vector data in their native representations and does not require any data conversion. Third, $\text{RJ}_{\bowtie}$ is modeled as a relational operator which makes it easier to combine with other relational operators in Spark such as selection, join, grouping, and aggregation. Previous work showed that a similar approach is efficient for the zonal statistics problem [66, 65, 67] but that work is still limited and cannot support complex analytical queries [68]. The $\text{RJ}_{\bowtie}$ operator is the first general-purpose operator that can build complex distributed processing pipelines for raster and vector data.

The proposed $\text{RJ}_{\bowtie}$ operator models both raster and vector data as relational data. Vector data is represented as a set of geometries while raster data is *virtually* represented as a set of pixels. To join them, we first define three predicates that define the logic of matching a pixel with either a point, a line, or a polygon. Based on that, we define the Raptor join output and show how it can be combined with standard operators to perform

arbitrarily complex query pipelines for real scientific applications. To implement the Raptor join operator efficiently, we propose a novel distributed index structure termed *Flash* index which is stored as a set of integer arrays that represent ranges in the raster data that join with the vector data. *Flash* index has a very small memory footprint which allows it to efficiently process terabytes of data. We compare the proposed system to GeoTrellis [21], Google Earth Engine [27], Rasdaman [3], Sedona [80], Adaptive Cell Trie (ACT) [37], and Beast [88], and show that it has up-to three orders of magnitude performance gain over them while being perfectly able to scale to big data and use fewer resources.

To summarize the contributions, this chapter: 1. Defines a logical relational data model that represents both raster and vector data. 2. Proposes a new operator Raptor Join ($\text{RJ}_{\bowtie}$) that joins raster and vector data. 3. Formulates three predicates for joining points, lines, and polygons, with raster data. 4. Proposes a new data structure, *Flash* Index, and uses it to implement $\text{RJ}_{\bowtie}$ efficiently in Spark. 5. Runs a comprehensive experimental evaluation on real datasets to show the efficiency of $\text{RJ}_{\bowtie}$.

The rest of this chapter is organized as follows: Section 4.2 covers the related work in literature. Section 4.3 describes the data and query model of the proposed system. Section 4.4 details the algorithm used to implement the proposed operator and the *Flash*-index construction and processing. Section 4.5 runs an extensive experimental evaluation of the proposed system. Section 4.6 concludes the chapter and discusses future work.

4.2 Related Work

4.2.1 Non-spatial Joins

The join operation [55] is a fundamental relational database query operation that brings together two or more relations. Logically, it can be modeled as a Cartesian product followed by a filter on the join predicate. Non-matching attributes from the two relations can be included in the output depending on the join type, inner, left outer, right outer, or full outer join. The most common join operation in relational databases is equi-join which uses the equality join predicate. Traditional join algorithms [44] are block nested loop, index nested loop, hash join, and sort-merge join. Sort-merge join is usually the most efficient algorithm if the inputs are already sorted. Otherwise, hash join is most commonly used. Finally, index nested loop join is preferred if one dataset is very small and the other one is indexed. In this chapter, we make an analogy between traditional join algorithms and raster-vector-join algorithms to explain their limitations.

4.2.2 Spatial Join on Raster Data

Raster data is usually represented as multidimensional arrays and it is analysed using map algebra [60, 43]. To join two raster layers, they must have the same dimensions. If the dimensions mismatch, a regridding operation is applied on one dataset to match the other dataset. Systems such as SciDB [70], RasDaMan [3], GeoTrellis [36], ChronosDB [83], and Google Earth Engine [27] implement algorithms for raster operations that can process large amounts of raster data. However, none of these systems provide an efficient join operation for raster and vector data. They usually rasterize the vector data with a matching

resolution and apply the raster operation. [90, 9] optimize raster joins by utilizing its tiled structure. However, they focus on either similarity joins or skewed data, which generally do not apply to raster-vector joins.

4.2.3 Spatial Join on Vector Data

Vector data is represented as a set of points, lines, and polygons, which are all represented as a set of coordinates. A spatial join on vector data can be defined as a join that finds pairs of geometries that satisfy a spatial predicate, such as intersection, overlap, or contains. Spatial join algorithms for vector data include R-Tree join [4], Spatial Hash join [40], Partition Based Spatial Merge join (PBSM) [47], index-based in-memory joins [63, 37], and many more [39, 38]. Efforts have also been made to implement these spatial join algorithms in a distributed environment in order to process big data [87, 14, 1, 80, 78]. None of these systems support raster-vector join efficiently. They can only convert raster pixels to points to apply one of the spatial predicates. For raster datasets with billions of pixels, this method does not scale.

4.2.4 Raster-Vector Joins

There have been some efforts to combine raster and vector data efficiently at the data representation, indexing, and query processing levels. At the data representation levels, a new *vaster* model was proposed [49] which converts both vector and raster data to a common representation. However, it was not practical as it needs to convert both datasets prior to doing any processing. At the indexing levels, the k^2-raster index is proposed [5, 62] to index raster data that can be combined with an R-tree vector index. However, this work

is limited to top-k and range queries and is limited to small data as it is a main-memory index. At the query processing level, the single-machine Scanline algorithm [16, 67] was proposed to solve the zonal statistics problem on raster and vector data. The algorithm was also ported to Hadoop for scalability [65, 66]. However, this work was limited to the zonal statistics computation on polygons and had a limited scalability when it comes to big vector data.

This work is the first to define and implement a general-purpose join operator for raster and vector data that can be used to build any complex relational query plan that runs on the distributed Spark framework. It does not require a conversion process like [49]. The proposed Flash index is lighter than [5, 62, 37] and can scale to terabytes of data and support any relational operation. Finally, the proposed work is more scalable and flexible than [16, 67, 65, 66] since it can support any relational query plan and not only zonal statistics.

4.3 Problem Formulation

This section defines the new Raptor Join ($\text{RJ}_{\bowtie}$) operator which joins raster and vector data. We begin by defining the data model for both raster and vector inputs, followed by the output of the $\text{RJ}_{\bowtie}$ operator for three types of geometries: points, lines, and polygons.

4.3.1 Input Data Model

A key advantage of our system is its ability to process both raster and vector data in their native representation. In other words, it can directly process raster data represented

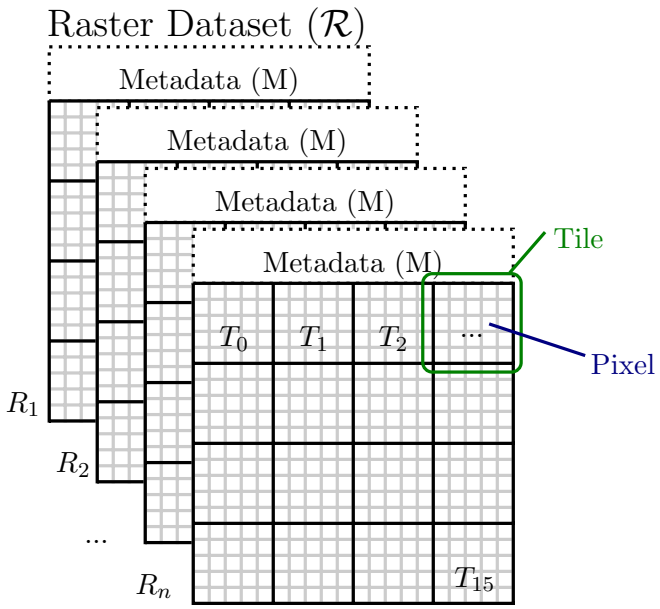


Figure 4.2: Raster file structure

as 2D arrays in compressed GeoTIFF or HDF files, and vector data represented as sequences of coordinates in CSV or binary Shapefiles. This poses a challenge on how to combine these two representations without data conversion. Our idea is to propose a common *logical* model that allows users to write queries but without actually doing any conversion. We choose the relational model as a common data model since it naturally integrates with other Spark operations to produce powerful query plans. We reiterate that even though we propose a common relational model for formalization purposes, the proposed system neither requires raster data to be input in a tabular form nor does it internally convert it into a relational form.

Raster dataset $\mathcal{R}$: Figure 4.2 illustrates the physical structure of the raster data. The input consists of a collection of raster files. Each file $R \in \mathcal{R}$, identified by a unique ID R_{id}, is a matrix of pixels organized into rows and columns. Each pixel $px = (R_{id}, x, y, m)$ in file R_{id} is located at column x, row y, and has a numeric value m, e.g., temperature or vegetation. For efficient storage and access, raster file formats group pixels into equi-sized non-overlapping sub-arrays called tiles, and each tile is assigned an identifier, T_{id}. For example, in Figure 4.2, each file contains 16 tiles and each tile contains 25 pixels. Typically, each tile contains tens of thousands of pixels and is compressed to reduce storage. These tiles are defined by the raster file format and our system uses this information as input to optimize the data access.

We use the array model as a physical representation when data is loaded in memory for its efficiency. In addition, we provide a relational *logical* representation to simplify the query processing for users. The logical model represents the raster dataset, $\mathcal{R}$ as a set of *(R_{id}, x, y, m)* tuples by simply flattening all the pixels and rasters in it.

$$\mathcal{R} = \{(R_{id}, x, y, m)\}$$

Raster Metadata $R.M$: Each raster image R is associated with metadata that consists of the following information:

- Number of columns (c) and rows (r) of pixels in the entire raster image.
- Tile width (tw) and tile height (th) in pixels.
- The grid-to-world ($\mathcal{G}2\mathcal{W}$) affine transformation matrix that converts a pixel location

on the grid to a point location in the world. The world-to-grid ($\mathcal{W}2\mathcal{G}$) transformation is simply the inverse affine matrix.

- Coordinate Reference System (CRS) which describes the projection that maps the Earth's surface to the world coordinates as defined by the ISO-19111 standard [31]. We assume that all $R \in \mathcal{R}$ have the same CRS. If not, we can easily group them by CRS and process each group individually.

We use the above information to calculate the following attributes, as needed, for each raster layer R or pixel px:

- Pixel bounding box $bb(px)$: is the bounding box of a pixel in world coordinates. The two corner points of $bb(px)$ are computed by applying the $\mathcal{G}2\mathcal{W}$ transformation for the pixel locations (x, y) and $(x + 1, y + 1)$.

- Pixel resolution p_x, p_y: is the width and height of a pixel bounding box. This value needs to be calculated only once since all pixels have the same resolution. For square pixels, we use $p = p_x = p_y$ to refer to either of them.

- Tile ID $T_{id}(px)$: is the tile that contains any given pixel.

$$T_{id} = \left\lfloor \frac{y}{th} \right\rfloor \cdot \left\lceil \frac{c}{tw} \right\rceil + \left\lfloor \frac{x}{tw} \right\rfloor \tag{4.1}$$

- Number of tiles in the file:

$$numTiles = \left\lceil \frac{c}{tw} \right\rceil \cdot \left\lceil \frac{r}{th} \right\rceil \tag{4.2}$$

We reiterate that the metadata is *not* replicated for each pixel, but is stored only once in memory and is associated with pixels through the raster ID that contains the pixel.

Vector Dataset V: is defined as a collection of geometric features that comprise points, lines, or polygons. Points represent discrete data values using a pair of longitude and latitude (lon, lat). Lines or linestrings represent linear features, such as rivers, roads, and trails. Each line is represented by an ordered list of at least two points. Polygons represent areas such as the boundary of a city, lake, or forest. Polygons are represented as an ordered collection of closed linestrings, i.e., rings, which constitute the boundary of the polygon and optionally holes inside it. In this work, we represent a vector dataset, V as a set of (g_{id}, g) tuples, where g_{id} is a unique identifier for the record and g is the geometry.

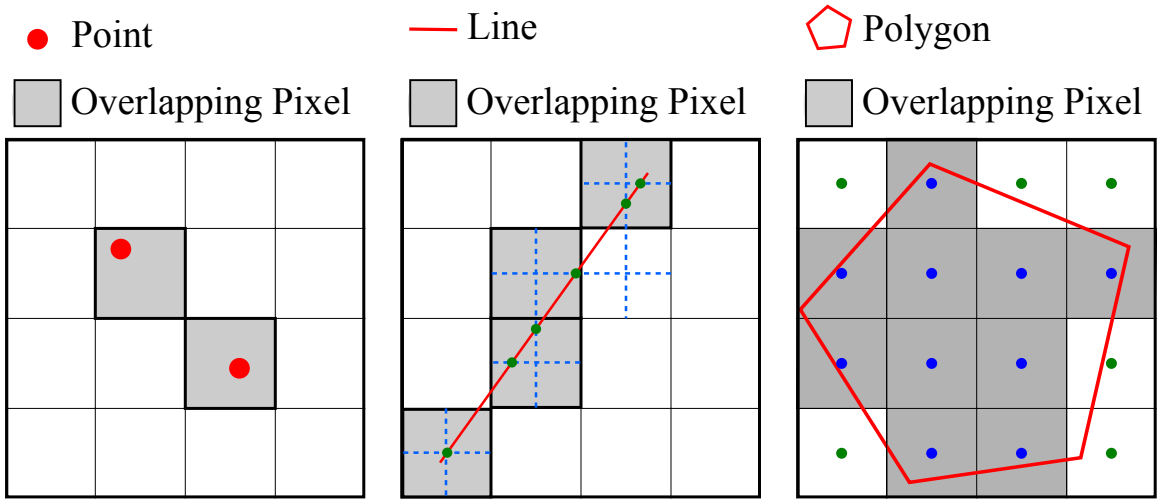
$$V = \{(g_{id}, g)\}$$

If V has a different CRS than the raster dataset $\mathcal{R}$, we convert V to match $\mathcal{R}$ on-the-fly as the data is loaded.

4.3.2 RJ$_{\bowtie}$ Output Definition

This section formally defines the output of the RJ$_{\bowtie}$ operator that brings together raster and vector data.

Raptor Join RJ$_{\bowtie}$: is a spatial join operator that takes as input a vector dataset V, a raster dataset $\mathcal{R}$, and a predicate θ. It produces the set of (geometry, pixel) pairs which satisfy the predicate. Based on our collaboration with domain scientists in various fields [68], we define three predicates θ_{point}, θ_{line}, and $\theta_{polygon}$ for the three types of geometries defined shortly. Each predicate θ takes two input records, a geometry g and a pixel $px = (R_{id}, x, y, m)$, and



(a) θ_{point} (b) θ_{line} (c) $\theta_{polygon}$

Figure 4.3: Three predicates θ for Raptor Join

returns *true* if the geometry and pixel match. Unlike vector-based systems, we show later in this work that we do not need to test this predicate for individual pixels but we can still find the correct result using the proposed *Flash* index.

Point Predicate (θ_{point}) returns true if the point location lies inside the bounding box ($bb(px)$) of the pixel as illustrated by the two shaded pixels in Figure 4.3(a).

Line Predicate θ_{line} returns true if the line intersects the *crosshair* of the pixel, which is defined as the two lines splitting the pixel bounding box in half, horizontally and vertically, as depicted by dotted lines in Figure 4.3(b). In this figure, shaded pixels are the ones that match the line according to this definition. This predicate can be used in hydrology applications to measure the altitude profile along hillslopes to detect watersheds [19].

Polygon Predicate $\theta_{polygon}$ returns true if the center of the pixel bounding box is inside the polygon boundary. Figure 4.3(c) depicts an example where the pixel centers are marked

as points. Blue (green) points mark the centers of the pixels that are inside (outside) the polygon. The shaded pixels are the ones whose centers lie inside the polygon. This predicate can be used in agriculture to calculate the average vegetation per farmland [58] where vegetation values are represented using a raster.

These three predicates can also be used to express other predicates. For example, to match a linestring with all pixels within a distance d, we can compute a buffer of that distance around the linestring and use $\theta_{polygon}$. Conversely, to match a polygon with pixels around its perimeter, we can first compute the polygon boundary as a linestring and then use θ_{line}. These transformations are computed on-the-fly as the data is loaded and do not incur a significant overhead on the overall computation time.

For any of the three predicates, $\text{RJ}_{\bowtie}$ outputs a set of (g_{id}, R_{id}, x, y, m) tuples for all geometries and pixels that match. Notice that the user does not explicitly set the predicate but it is automatically chosen by the system based on the geometry type.

$$\mathcal{R} \bowtie_{\theta} V = \{(g_{id}, R_{id}, x, y, m)\} \tag{4.3}$$

This allows us to define $\text{RJ}_{\bowtie}$ as a new Spark RDD (Resilient distributed dataset) [81] operator. Keep in mind that the $\text{RJ}_{\bowtie}$ operator can be combined with other operators as needed by the application to satisfy the desired query logic. For example, it can be followed by an equi-join with the vector dataset V on the attribute g_{id} if the geometric feature is needed. Also, it can be followed by a *group by* operator on g_{id} to group all pixels that match a single geometry. Similarly, the result can be grouped by the pixel value m if it represents a raster object, e.g., contour or land type. We can extend our $\text{RJ}_{\bowtie}$ definition for outer joins but we omit these definitions for brevity.

4.3.3 Integration with Spark

This part describes how the $\text{RJ}_{\bowtie}$ operation is integrated with the Spark RDD API [81]. We use the Scala *implicit classes*[1] feature to extend the SparkContext and RDD classes without touching the internal code of Spark. In SparkContext, we add two sets of functions for loading *vector* and *raster* data in various formats, e.g., `geoTiff`, `shapefile`, and `geojson`. Vector data is loaded as `RDD[IFeature]` where IFeature represents a geometric feature that contains a geometry and any additional non-spatial attributes in the file. Raster data is loaded as `RDD[ITile]` where each ITile is an interface for accessing pixel values and raster metadata. We also extend `RDD[ITile]` with the `raptorJoin` method that takes `RDD[IFeature]` and returns `RDD[(Long, Int, Int, Int, Float)` which represents a set of tuples $(g_{id}, R_{id}, x, y, m)$. Finally, `raptorJoin` is implemented as a transformation so it can be preceded or followed by other transformations and they will be compiled into one Spark job. For example, it can be preceded by a filter on the vector data and followed by a grouped aggregation on the geometry ID or the pixel value.

4.4 Implementation

This section describes the proposed algorithm that implements the $\text{RJ}_{\bowtie}$ operator. The key design objectives for $\text{RJ}_{\bowtie}$ are: 1. **In-situ:** $\text{RJ}_{\bowtie}$ does not require a preprocessing phase for converting the input data. 2. **Efficiency:** $\text{RJ}_{\bowtie}$ handles high-resolution raster data and big vector data. 3. **Fully distributed:** $\text{RJ}_{\bowtie}$ runs in a fully distributed mode for scalability.

[1] https://docs.scala-lang.org/overviews/core/implicit-classes.html

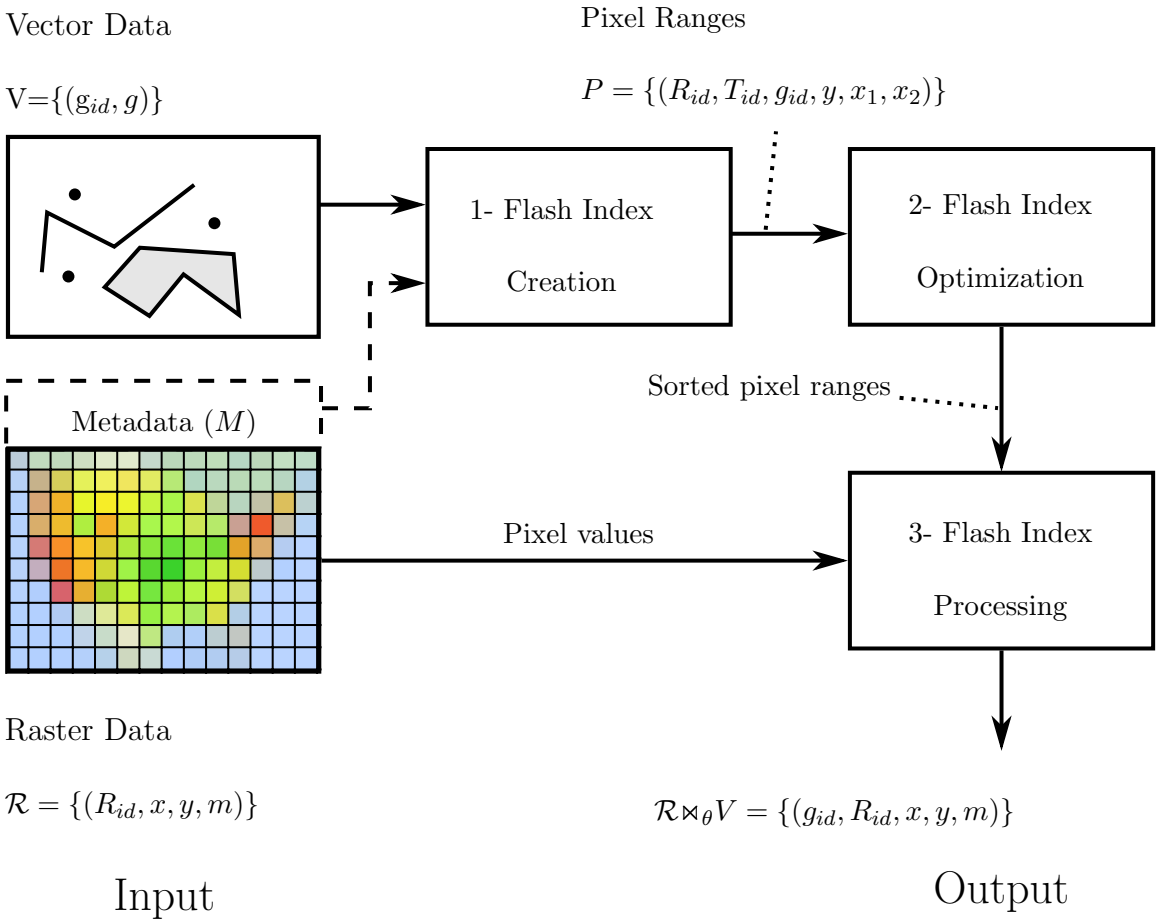


Figure 4.4: Implementation Overview of Raptor Join

The key idea of $\text{RJ}_{\bowtie}$ is to resemble a sort-merge join algorithm which scans the input datasets only once. In contrast, raster-based systems resemble a hash-join where each geometry is hashed to pixels (i.e., buckets) which are then joined with input raster data. This will have a cost of $O(|V| \cdot |\mathcal{R}|)$, since each geometry in V can be rasterized into a layer with as many pixels as the raster layer, $|\mathcal{R}|$. On the other hand, vector-based systems resemble a nested-loop join where each pixel is compared to geometries to find the matching ones. If an index is built on geometries, it will resemble an *index* nested loop join with a

running time of $O(|V| \log |V| + |\mathcal{R}| \cdot \log |V|)$, where the first term is for building the index and the second term is for searching the index for each pixel in $\mathcal{R}$. The proposed algorithm has a running time of $O(|V| \log |V| + |\mathcal{R}|)$, where the first term resembles a sort step for the vector dataset (in our case, building the *Flash* index, described shortly) and the second term resembles the linear merge step.

To accomplish this idea, $\text{RJ}_{\bowtie}$ exploits the inherent structure of the raster data and builds an intermediate index structure, termed *Flash* index. The *Flash* index is built on the vector data and has three main novelties to satisfy our design objectives. 1. **In-situ:** The *Flash* index is built as needed which makes it suitable for in-situ data processing. 2. **Efficiency:** Since it is built as needed, it is adjusted according to the input data size and resolution to produce a compact and highly-efficient index. 3. **Fully distributed:** The index is constructed, optimized, and processed in parallel which allows it to scale to big raster and vector data.

Figure 4.4 gives an overview of the three phases of the $\text{RJ}_{\bowtie}$ algorithm, namely, *Flash* index *creation*, *optimization*, and *processing*. The *creation* phase takes the input vector data and only the *metadata* of the raster data to produce an initial *Flash* index that consists of a set of unordered pixel ranges. The second phase, *optimization*, repartitions, groups, and orders the pixel ranges to match the structure of the raster data. The goal is to ensure that the third phase can process the entire join query in a single scan over the raster data. The final *processing* phase uses the *Flash* index to scan the raster dataset and produce the final output. The output is produced in parallel and is streamed into the next operator depending on how the Spark job is structured.

4.4.1 Flash Index Creation

The input to this phase is the vector dataset (V) and the metadata ($R_{id}.M$) of all raster files ($R_{id} \in \mathcal{R}$) and the output is a set of pixel ranges P in the format $(R_{id}, T_{id}, g_{id}, y, x_1, x_2)$, where, R_{id} is the ID of the raster file, T_{id} is the id of the tile, g_{id} is the geometry ID, and y is a row index in the raster file R_{id} and $[x_1, x_2]$ is a range of pixels in row y that match with the geometry g_{id}. This initial version of the *Flash* index represents all matching ranges between the vector and raster data. This proposed representation produces a compact index by matching the resolutions of the raster and vector data. For example, if a complex geometry overlaps only a few pixels, only those pixels are encoded regardless of the size of the geometry. On the other hand, if the geometry overlaps a large number of pixels, those pixels will be grouped in ranges to reduce the size of the *Flash* index. The pixel ranges also prune any non-relevant parts of either dataset. For example, if the vector dataset covers farmlands, then any non-relevant parts in the raster data, e.g., water areas or deserts, will be excluded from the *Flash* index.

Preparation: Before the vector data can be processed, it might need some of these preparation steps.

1. If the geographical coordinate reference system (CRS) of the vector and raster data do not match, we convert the vector data to match the raster one using a map transformation as defined by the ISO 19111:2019 standard [31].

2. If the geometric data does not already have a unique ID, we use the Spark operation `zipWithUniqueID` to generate a unique ID for each geometry.

3. If the input vector RDD has fewer partitions than the number of cores in the Spark cluster, we randomly repartition the records to have at least one partition per core. This ensures that we fully utilize the cluster during index creation.

Pixel Ranges: Once the vector data is ready, the next step is to create the pixel ranges. The computation of these ranges differs for the three predicates, θ_{point}, θ_{line}, and $\theta_{polygon}$ as detailed below. The following description is given for one raster file R but the process is simply repeated for all raster files and the output of all of them is merged in no particular order.

Pixel ranges for points

The intersection of a point with raster layer is defined as the pixel whose bounding box bb contains the point. Given a point at location (lon, lat), the matching pixel location can be found using the following equation.

$$(x', y') = \mathcal{W}2\mathcal{G}(lon, lat) \tag{4.4}$$

$$(x, y) = (\lfloor x' \rfloor, \lfloor y' \rfloor) \tag{4.5}$$

First, we apply the $\mathcal{W}2\mathcal{G}$ transformation to find the grid coordinates and then use the floor function to find the pixel coordinate. Since each point covers a single pixel, the range is formed as $(R_{id}, T_{id}, y, x, x)$, where R_{id} is the ID of the raster file and T_{id} is calculated using Equation 4.1. Even though there is some redundancy in repeating x, we use this format to maintain uniformity in the output for all geometry types.

Pixel ranges for lines

Our definition for the θ_{line} predicate matches a line segment with those pixels whose centers are closest to the line, either horizontally or vertically. This definition is analogous to the traditional mid-point line drawing algorithm that is used in the field of computer graphics. To compute the pixel intersections, we iterate over each line segment and convert its end points from world to grid. Then, we apply the mid-point algorithm to find pixel intersections. However, unlike the original algorithm that deals with *integer* coordinates, our algorithm must deal with floating point geographical coordinates. This results in a collection of $(g_{id},\ y,\ x)$ tuples, where y and x are the row and column identifiers of the intersecting pixel and g_{id} is the geometry ID.

The next step groups these pixel *intersections* into pixel *ranges*. Simply, if several pixel intersections are in the same tile (T_{id}), belong to the same geometry g_{id}, on the same row (y), and have consecutive x or overlapping coordinates, they are combined into a single pixel range. If there is only one pixel intersection with no adjacent intersections to be combined with, a range with a single pixel is created (similar to the case of θ_{point}).

Algorithm 3 computes pixel ranges from line-pixel intersections. It takes as input a list of pixel intersections represented in three arrays g_{id}, x, and y. The output is five arrays that together represent the pixel ranges. Line 1 sorts the intersections to bring the ones that can be merged next to each other in the sort order. It then scans the list of pixel intersections from the end to allow new intersections to be appended without affecting the processing of the loop. Inside the loop, the tile ID is computed using Equation 4.1 and then the two adjacent pixel intersections at i and $i+1$ are compared to check if they are adjacent

Algorithm 3: LinePixelIntersectionsToRanges

Input: A list of pixel intersections represented by three arrays $g_{id}[], x[], y[]$ with equal length n

Output: A list of pixel ranges represented by five arrays $t[], g_{id}[], y[], x_1[], x_2[]$ with equal length $n_{out} \leq n$

Sort (g_{id}, x, y) lexicographically by (y, g_{id}, x)

Let $t[]$ be an array of tile IDs with length n initialized to zeros

numDeletions $\leftarrow 0$

for *i=n-1* **to** *0* **do**

    $t[i] \leftarrow T_{id}(x[i], y[i])$ /* Use Equation 4.1 */

    if $i \neq n-1$ **and** $(t[i+1], g_{id}[i+1], y[i+1]) = (t[i], g_{id}[i], y[i])$ **and** $(x[i+1] = x[i]$ **or** $x[i+1] = x[i]+1)$ **then**

        $t[i+1] \leftarrow$ numTiles /* Mark for deletion */

        numDeletions++

    /* Form a range by appending a similar pixel */

    else

        $(x[n], y[n], g_{id}[n], t[n]) \leftarrow (x[i], y[i], g_{id}[i], t[i])$

        $n++$

Sort (t, g_{id}, x, y) lexicographically by (t, y, g_{id}, x)

$n_{out} \leftarrow n - numDeletions$ /* Arrays are zero-based */

$x_1 = x[i : \text{i is even} \wedge i < n_{out}]$

$x_2 = x[i : \text{i is odd} \wedge i < n_{out}]$

$t = t[i : \text{i is odd} \wedge i < n_{out}]$

$g_{id} = g_{id}[i : \text{i is odd} \wedge i < n_{out}]$

$y = y[i : \text{i is odd} \wedge i < n_{out}]$

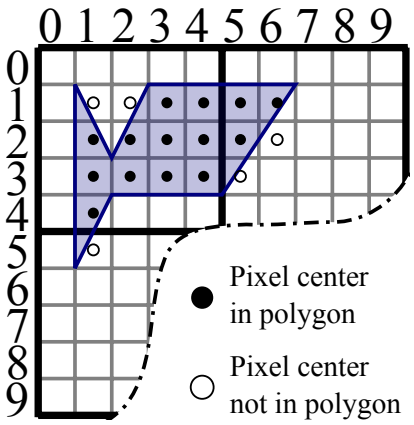
return $(t, g_{id}, y, x_1, x_2, n_{out})$

or overlapping. If they are, we remove the latter one at $i + 1$. For efficiency, we only mark it for deletion by setting its tile ID to $numTiles$ which is larger than all valid tile IDs. Otherwise, if the two pixel intersections cannot be merged, we insert a new intersection at the tail of the list to form a range. Notice that the newly inserted intersection is not in the sort order but this will be fixed after the loop finishes.

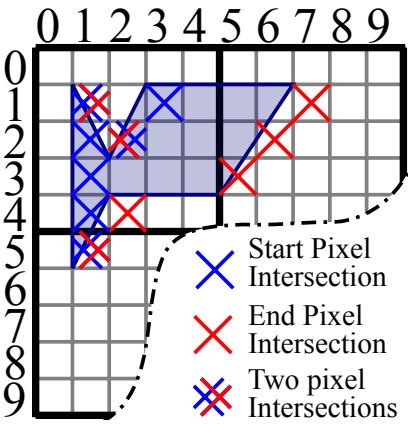
After the for loop, Lines 12-19 sort the list of intersections again by (t, y, g_{id}, x) which has two goals. First, it will push all the pixels marked for deletion to the end of the list since they all have a maximum tile ID. Second, it will form pixel ranges from every two consecutive pixel intersections. The final output is formed by combining every two consecutive intersections into a single range as depicted in the algorithm. The running time of this algorithm is $O(n \log n)$ for the sort steps. It can insert and delete at most n intersections with a constant time for each insert or delete.

Pixel ranges for polygons

The intersection of a polygon with the raster layer is defined as the pixels whose bounding box center is inside the polygon. Similar to linestrings, we first compute pixel intersections and then combine them into ranges. However, unlike the linestrings, the set of pixel intersections already define an initial set of ranges that are further adjusted to match the structure of the output for points and linestrings. Figure 4.5 illustrates the computation of pixel intersections. In this example, the raster file has 10 rows and 10 columns of pixels and is organized into four tiles, each with 5×5 pixels. The solid and hollow circles indicate centers of pixels that are inside and outside the polygon, respectively.



(a) Polygon raster join



(b) Pixel intersections

#	x	y	#	x	y
0	1	1	7	6	2
1	1	1	8	1	3
2	3	1	9	5	3
3	7	1	10	1	4
4	1	2	11	2	4
5	2	2	12	1	5
6	2	2	13	1	5

(c) Sorted pixel intersections

#	t	y	x_1	x_2
0	0	1	3	4
1	0	2	1	4
2	0	3	1	4
3	0	4	1	1
4	1	1	5	6
5	1	2	5	5

(d) Sorted pixel Ranges

Figure 4.5: Pixel intersection computation for polygons. (a) A sample polygon and the pixels that satisfy the $\theta_{polygon}$ predicate. (b) The pixel intersections computed using Algorithm 4. (c) The pixel intersections sorted by (y, x) (g_{id} is omitted for brevity). (d) The pixel ranges produced by Algorithm 5

Algorithm 4: ComputePolygonSegmentIntersections

Input: One polygon segment $(lon_1, lat_1) \rightarrow (lon_2, lat_2)$ and raster metadata M

Output: A list of pixel intersections

1 $(x_1, y_1) \leftarrow M.\mathcal{W}2\mathcal{G}(lon_1, lat_1)$

2 $(x_2, y_2) \leftarrow M.\mathcal{W}2\mathcal{G}(lon_2, lat_2)$

/* xs and ys are real numbers */

3 row1 $\leftarrow$ max(0, round(min(y_1, y_2))) /* row1 and */

4 row2 $\leftarrow$ min(M.r, round(max(y_1, y_2))) /* row2 are integers */

5 **for** *row* $\leftarrow$ *row1* **to** *row2-1* **do**

6 xIntersection $\leftarrow$ round($x_2 - (y_2 - (row + 0.5))\frac{x_2 - x_1}{y_2 - y_1}$)

7 Output $\ll$ (max(0, min(xIntersection, M.c)), row)

The first step is to compute the pixel intersections between the boundary of the polygon and the center lines of raster rows. Algorithm 4 describes how to compute all intersections between a single polygon segment and centers of raster rows. It starts by converting the two end points from the model to grid space. Notice that we keep the grid coordinates as real numbers, not integers, for correctness. After that, it computes the range of raster rows that intersect with the line segment without going out of raster boundaries. For each row, it computes the intersection between the horizontal line at $row + 0.5$ and the polygon segment. The computed intersection is added to the list of intersections.

Figure 4.5(b) depicts the computed intersections with a cross. Pixels with double-crosses indicate two intersections at the same pixel. A blue cross depicts the start of an

intersection range while a red cross depicts its end. Figure 4.5(c) lists all pixel intersection locations computed by Algorithm 4. Notice that we omit g_{id} for brevity since the example shows only one polygon but the computation is repeated for all polygons in the input. The algorithm starts by converting the two end points of each segment to the grid space while keeping it as a real number. Then, it calculates the range of rows that intersect the line segment while keeping in mind that the valid range of rows is $[0, r]$. For each row, the segment is intersected with the horizontal line at the center of the row, i.e., $row + 0.5$ while keeping in mind that the valid range of intersections is $[0, c]$.

The next step is to convert the pixel intersections into ranges in the same structure of points and lines. By inspecting Figure 4.5, one can realize that each row must have an even number of intersections for closed polygons. Each pair of consecutive intersections represent a range of pixels. However, these ranges have three issues that are fixed using a simple algorithm. First, ranges are open-ended, i.e., last pixel in the range is excluded which can be easily fixed by decreasing the position of the range end by one pixel. Second, some ranges are empty, e.g., the first range at rows 1 and 5, and some consecutive ranges can be merged, e.g., the two ranges in line 2. Both of these can be fixed by removing every pair of pixel intersections with the same exact coordinates. Third, some ranges span two tiles which would result in an efficient disk access pattern since each tile is stored in a different disk location. These are fixed by breaking each range that spans multiple tiles at tile boundaries by creating two new intersections one at the end of the first tile and one at the beginning of the next tile. All these adjustments are done in one scan over the list of pixel intersections.

Algorithm 5: PolylgonPixelRanges

Input: A list of pixels intersections represented by three arrays $g_{id}[], x[], y[]$ with equal length n

Output: A list of pixel ranges represented by five arrays $t[], g_{id}[], y[], x_1[], x_2[]$ with equal length n_{out}

1 Sort (g_{id}, x, y) lexicographically by (y, g_{id}, x)

2 Let $t[]$ be an array of tile IDs with length n initialized to zeros

3 numDeletions $\leftarrow 0$

4 **for** *i=n-2* **to** *0* **step** *-2* **do**

5 **if** *x[i]* $\geq$ *x[i+1]* **then**

6 $t[i] \leftarrow t[i+1] \leftarrow numTiles$ /* Mark points for removal */

7 numDeletions $\leftarrow$ numDeletions + 2

8 **else**

9 $t[i] \leftarrow T_{id}(x[i], y[i])$ /* Use Equation 4.1 */

10 $x[i+1] \leftarrow x[i+1] - 1$ /* Make the range inclusive */

11 $t[i+1] \leftarrow T_{id}(x[i+1], y[i+1])$

12 **if** $t[i] \neq t[i+1]$ **then**

/* Break the range into two at tile boundary */

13 $(x[n], y[n], t[n]) \leftarrow (tw \cdot t[i+1] - 1, ys[i], t[i])$

14 $(x[n+1], y[n+1], t[n+1]) \leftarrow (tw \cdot t[i+1], ys[i], t[i+1])$

15 $n \leftarrow n + 2$

16 **if** $(g_{id}[i+1], y[i+1], t[i+1]) = (g_{id}[i+2], y[i+2], t[i+2])$ **and** $x[i+1] = x[i+2] - 1$ **then**

17 $t[i+1] \leftarrow t[i+2] \leftarrow$ numTiles

18 numDeletions $\leftarrow$ numDeletions + 2

19 Similar to lines 12-19 of Algorithm 3

Algorithm 5 describes how to compute pixel ranges from pixel intersections for polygons. Similar to the case of linestrings, it starts by sorting all pixel intersections lexicographically by (y, g_{id}, x). This directly creates ranges as illustrated in Figure 4.5. After that, it makes one scan over those ranges from end to start. Line 5 checks if the range is empty and marks the both ends for removal by setting the tile ID to a value bigger than all valid tile IDs. If not empty, it computes the tile ID for both pixel intersections and decrements the end to convert it from an open-ended to closed interval. Line 12 checks if the range spans two tiles. If so, it breaks it into two ranges at the tile boundary. This is done by inserting two new pixel intersections at the end of the first tile and the beginning of the second tile. While this part assumes that a range can intersect at most two tiles, it can be easily extended to ranges that span several tiles but we omit this part for brevity. Line 16 tests if two consecutive ranges can be merged into one. Merging two ranges is done by removing the end of the first one and the start of the second one. After all pixel intersections are processed, the final pixel ranges are computed in the same way as lines 12-19 in Algorithm 3. This algorithm has a running time of $O(n \log n)$ for the two sorting steps.

Figure 4.5(d) shows the computed ranges using the above algorithm. The empty intersections formed by pixel intersections (#0, #1) and (#12,#13) were removed in the output. Additionally, the range formed by intersections (#2,#3) is split into two ranges at two tiles. Finally, notice that the lower two tiles do not contain any ranges which means they do not need to be processed or even read from disk.

RangeIterator: We store the intersections in memory in a column format, i.e., arrays of integer values. This reduces the memory overhead and improves the cache per-

formance. It also gives an opportunity for efficient column compression to reduce memory overhead which we leave for future work. However, Spark is a row-oriented system and cannot directly process column-oriented data in memory. To solve this mismatch issue, we create a component, termed RangeIterator, that iterates over the ranges and streams them into the next phase.

4.4.2 Flash Index Optimization

The pixel ranges generated in the first phase can be directly used to process the data. However, the performance will not be optimal since some input raster tiles might need to be processed several times. The reason is that the *Flash*-Index is created in parallel for vector partitions so it is most likely that two vector partitions overlap the same set of tiles. In this phase, we perform *global* and *local* optimizations to ensure minimum disk access in the processing phase. At the global level, we repartition pixel ranges by tile ID so that all ranges that belong to one tile are processed in a single task. At the local level, we sort all ranges within each tile to match the order of the pixels in the raster file which maximizes the cache hit while processing each tile. We efficiently perform both global and local optimizations using the Spark operation `repartitionAndSortWithinPartition`.

Global optimization: We use the pair (R_{id}, t_{id}) as a partitioning key which moves all pixel ranges with the same tile ID to the same partition. While all tiles have the same number of pixels, the workload across tiles can differ based on the number of pixel ranges within each tile. The distribution of pixel ranges is expected to follow the distribution of the vector data which means that the workload will have a spatial locality, i.e., nearby tiles will have similar workload. Therefore, we use hash partitioning to distribute nearby

tiles across machines. We also adjust the number of partitions so that each partition will have a pre-configured number of tiles k. The goal is to adjust the processing time for each task to be a few seconds which balances the trade-off between parallelization overhead and load skewness.

Local optimization: Within each partition, we sort the pixel ranges lexicographically by $(R_{id}, t_{id}, y, g_{id}, x_1)$ which ensures that tiles are processed in order and that pixels are accessed row-by-row which matches the in-memory matrix representation of the tile. This will maximize the cache efficiency since each row will be loaded in cache, processed in full, and then evicted when no longer needed.

4.4.3 Flash-Index Processing

This is the only phase where the raster tiles are read from disk. It takes as input a stream of pixel ranges sorted by $(R_{id}, t_{id}, y, g_{id}, x_1)$ and it reads the pixel values m to output the final result as a stream of tuples in the format $(g_{id}, R_{id}, x, y, m)$. The order of the input ensures that only one tile needs to be loaded in memory at a time which minimizes both disk access and memory requirement. For each new range in the input, if the tile is not the one currently loaded, the current tile is replaced with the new one. Then, all pixels in the range are read one-by-one and the value is processed. If the value indicates an invalid pixel, i.e., fill value, the pixel is skipped to the next one. Otherwise, if it is a valid value, a tuple $(g_{id}, R_{id}, x, y, m)$ is output. The output tuples are generated in a streamed way to avoid keeping all of them in memory at the same time. For example, if the $\text{RJ}_{\bowtie}$ operator is followed by an aggregate operator, the values are directly processed and never kept in memory.

4.5 Experiments

This section provides an extensive experimental evaluation of the proposed algorithm for the Raptor Join ($RJ_{\bowtie}$) operator. We compare $RJ_{\bowtie}$ to three vector-based approaches, Adaptive Cell Trie (ACT) [37], Sedona [80] (formerly GeoSpark), and Beast [88]; and three raster-based approaches, Rasdaman [3], Geotrellis [36], and Google Earth Engine (GEE) [27]. When applicable, we also compare to RZS [66] which supports only the zonal statistics problem on polygons using Hadoop. Zonal statistics aggregates pixel values within polygonal regions.

The experiments show that the proposed $RJ_{\bowtie}$ is up-to three orders of magnitudes faster than the baselines. Additionally, $RJ_{\bowtie}$ is two to three orders of magnitude faster in the data loading step. Finally, $RJ_{\bowtie}$ is the only system that is able to perform all the experiments in one run over the big inputs while for GEE and GeoTrellis we needed, in some cases, to manually split big files into smaller ones, process each one separately, and combine the results, to work around system limitations.

4.5.1 Setup

We run $RJ_{\bowtie}$, GeoTrellis, Sedona, and Beast on a cluster with one head node and 12 worker nodes. The head node has Intel(R) Xeon(R) CPU $E5-2609$ v4 @ 1.70GHz processor, 128 of GB RAM, 2 TB of HDD, and 2×8-core processors running CentOS and Oracle Java 1.8.0_131. The worker nodes have Intel(R) Xeon(R) CPU E5-2603 v4 @ 1.70GHz processor, 64 GB of RAM, 10 TB of HDD, and 2×6-core processors running CentOS and Oracle Java 1.8.0_31-b04. The methods are implemented using the open source GeoTools library 17.0.

Google Earth Engine runs on the Google Cloud Platform on up-to 1,000 nodes [27] but it does not reveal the actual resources used by each query. Rasdaman and ACT are run on a single machine with Intel(R) Core $i5-6500$ CPU @ 3.20GHz $\times$ 4, 32 of GB RAM, 1 TB of HDD running Ubuntu 16.04.

For Rasdaman, we use version 10.0 running on a single machine since the distributed version is not publicly available. For GeoTrellis, we use the *geotrellis-spark* package version 1.2.1, as described in its documentation. We used Sedona v1.3.2-SNAPSHOT as described on its website. GEE is still experimental and is currently free to use. The caveat is that it is completely opaque and we do not know which algorithms or how much compute resources are used to run queries. Therefore, we run each operation on GEE 3-5 times at different times and report the average to account for any variability in the load. All the running times are collected as reported by GEE in the dashboard.

For each experiment, we perform the zonal statistics query that performs a raster-vector join and then compute the four aggregate values, minimum, maximum, sum, and count for the resulting tuples. We measure the end-to-end running time as well as the performance metrics which include reading both datasets from disk and producing the final answer. Table 4.1 lists the datasets that are used in the experiments along with their attributes. All vector datasets are available on UCR-Star [25, 72]. All raster datasets except Planet Data are also publicly available. They come from various government agencies. The GLC2000 and MERIS datasets are from the European Space Agency with pixel resolutions of 0.0089 decimal degrees (1km) and 0.0027 (300m) respectively. The US Aster dataset originates from the Shuttle Radar Topography Mission (SRTM) and covers the continental

Table 4.1: Vector and Raster Datasets

Vector datasets

Dataset	$\|V\|$	Points	File Size	Type	Coverage
all_nodes	2.7b	2.7b	257.2 GB	Points	World
Linearwater	6m	292m	5.8 GB	Lines	US
Roads	18m	349	9.1 GB	Lines	US
Edges	68m	759m	33.6 GB	Lines	US
States	49	165k	2.6 MB	Polygons	US48
Counties	3k	52k	978 KB	Polygons	US48
ZCTA5	33k	53m	851 MB	Polygons	US
TRACT	74k	38m	632 MB	Polygons	US
Boundaries	284	3.8m	60 MB	Polygons	World
Parks	10m	336m	8.5 GB	Polygons	World

Raster datasets

Dataset	# pixels	Resolution	Size	Coverage
GLC2000	659M	1 km	629 MB	World
MERIS	8.4B	300 m	7.8 GB	World
US Aster	187B	30 m	35 GB	US48
Tree cover	840B	30 m	782 GB	World
Planet Data	4.2B	3 m	31 GB	California

US. Hansen developed the global Tree Cover change dataset which covers the entire globe. Both datasets have a spatial resolution of 0.00028 decimal degrees (30m). Planet data is sourced from Planet Labs [71] and has a temporal range of a month. The coverage of these datasets is either California, the contiguous 48 states (US48), the entire US, or the world.

4.5.2 Vector-based Systems

In this section, we compare to three vector-based baselines, Adaptive Cell Trie (ACT) [37], Sedona [80], and Beast. Since ACT is a single-machine algorithm, we compare it separately to a single-machine version of $\text{RJ}_{\bowtie}$. Then, we compare Sedona and Beast to the Spark-based $\text{RJ}_{\bowtie}$ implementation.

On-the-fly method: ACT is a highly-efficient in-memory index for point-in-polygon queries. We use it as a representative for the on-the-fly method that can avoid materializing the converted data. To adapt it to our problem, we first create an ACT-4 index (which gives the best result) and we set its precision to match the raster resolution to minimize the index size and maximize the throughput without significantly reducing the accuracy. Then, we use GDAL library to load the pixels that are within the minimum bounding rectangle (MBR) of the vector data directly from the GeoTIFF file. After that, we convert each pixel location to a point, and search for overlapping polygons in the index. For a fair comparison, we compare ACT to $\text{RJ}_{\bowtie}$ when both are running on a single-thread. The single-thread $\text{RJ}_{\bowtie}$ implementation skips the index optimization phase since the *Flash*-index is built on one machine. Since ACT is not optimized for disk access, we did not include the raster or vector loading times.

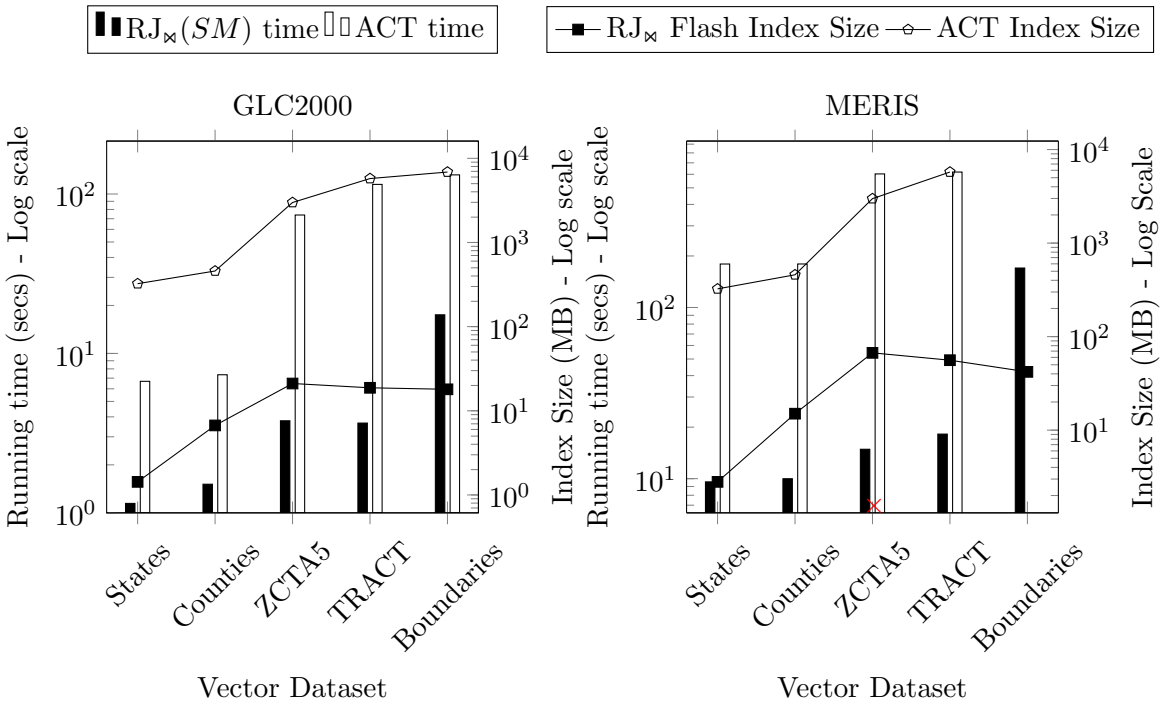


Figure 4.6: Comparison of running time (bars) and index size (lines) of ACT and RJ$_{\bowtie}$ for small raster data on a single machine

Figure 4.6 shows the results of ACT and RJ$_{\bowtie}$ to join the two smallest raster datasets with all vector datasets. ACT ran out of memory for larger datasets on a machine with 32GB of RAM. For the smallest raster dataset, GLC2000, RJ$_{\bowtie}$ is up-to three orders of magnitude faster than ACT. This can be explained by the index size which reaches nearly 6GB for ACT while it barely reaches 20MB for the proposed *Flash* index. Keep in mind that ACT index needs to be searched for each pixel while the *Flash* index is scanned only once. For the medium raster dataset, MERIS, we can see a similar behavior for both running time and index size. Furthermore, ACT runs out of memory for the boundaries dataset while the *Flash* index peaks at 60MB of memory.

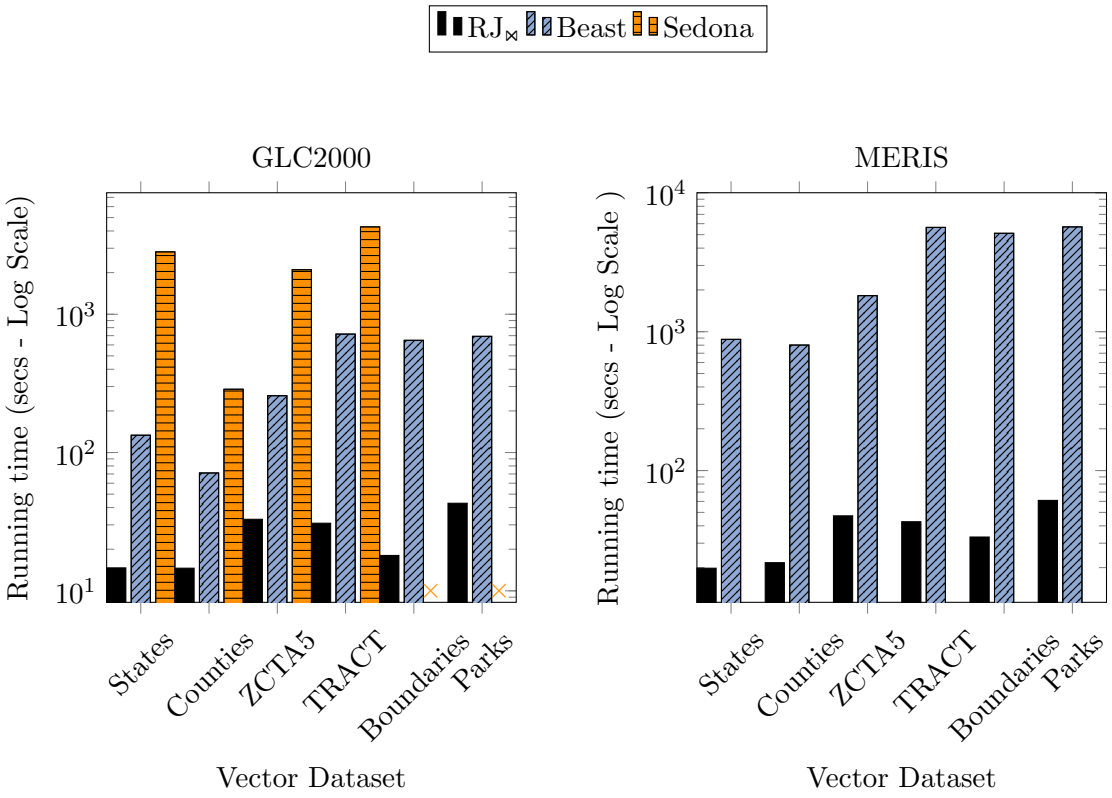


Figure 4.7: Running time of vector-based systems

Materialized method: Sedona and Beast are used to test the performance of the materialized method. To use them, we first convert the raster dataset to points in the format (lon, lat, m), which encodes the pixel location and value. We do not consider the overhead of the conversion process. We show the results on only the smallest raster datasets since none of the baselines was able to finish for the larger datasets. Figure 4.7 shows that RJ$_{\bowtie}$ outperforms all baseline systems hands down. Furthermore, as the raster resolution increases, from GLC2000 to MERIS, the gap grows to more than three orders of magnitude. The reason is that the number of pixels increases quadratically with the resolution which incurs a huge overhead on partitioning and processing this data.

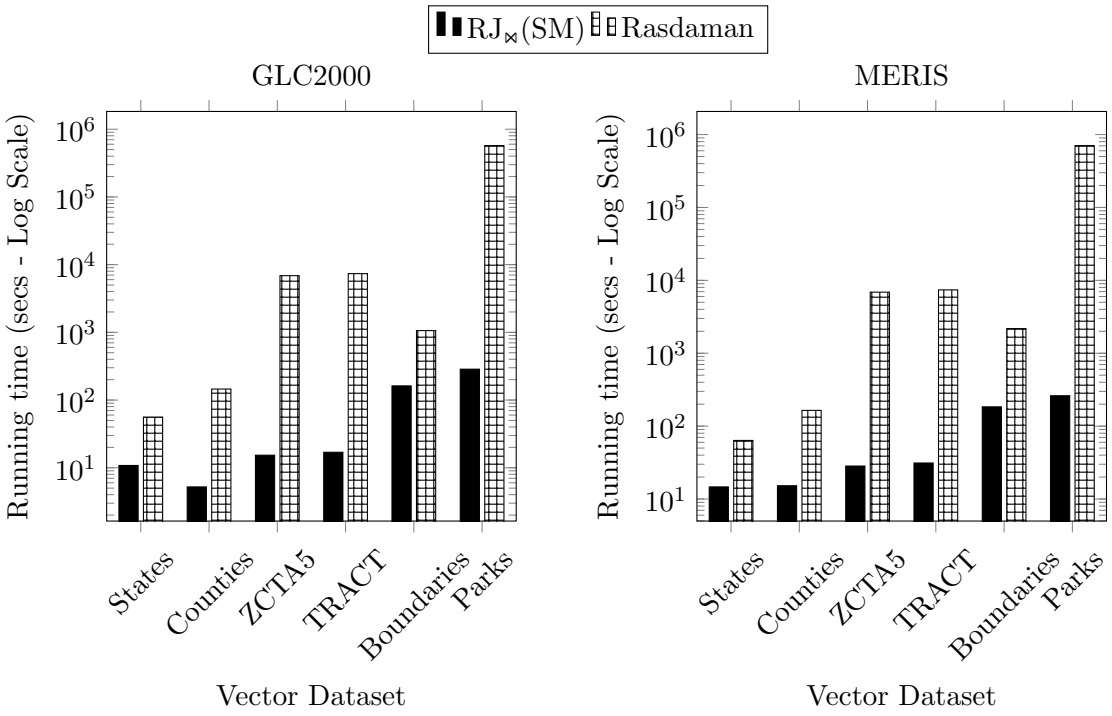


Figure 4.8: Single machine performance of Rasdaman and RJ$_{\bowtie}$

The previous experiments confirm that vector-based systems are not suitable for this problem. The on-the-fly method suffers from the large index size and the excessive index access. On the other hand, the materialized method suffers from the partition and processing overhead of the vectorized pixels.

4.5.3 Raster-based Systems

In this part, we compare RJ$_{\bowtie}$ to four baselines, Rasdaman, GeoTrellis, Google Earth Engine (GEE), and Raptor Zonal Statistics (RZS). The latter is our previous work which is designed only for the zonal statistics problem between polygons and raster data. Since the free version of Rasdaman runs on a single-machine, we compare it to a single-

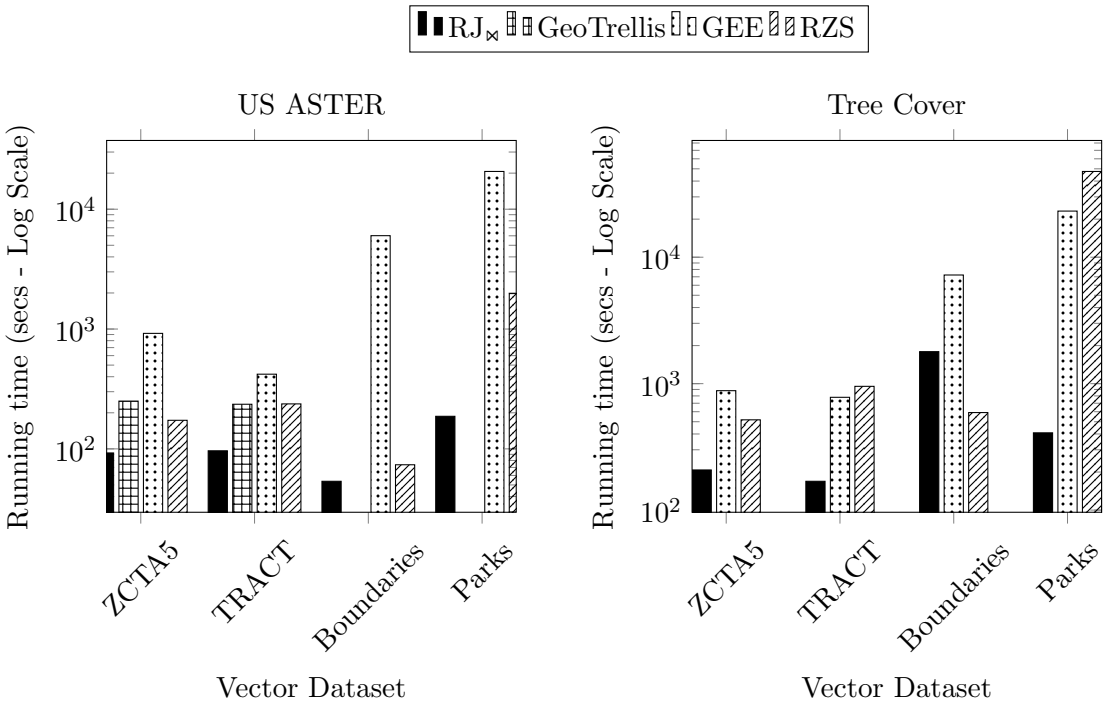


Figure 4.9: Running time of raster-based systems

thread implementation of RJ⋈. We compare all other baselines to the Spark version of RJ⋈. The experiments in [68] show that the ingestion time for raster and vector datasets for the baselines. The in-situ approach of RJ⋈, that is, uploading data to HDFS, can be observed to be two to three orders of magnitude faster than the baselines. Since, they also show that the ingestion performance of all raster systems is very low, we omit the results of the materialized method and show only the on-the-fly method.

Single-machine Systems: Figure 4.8 shows the performance of Rasdaman as compared to the single-machine RJ⋈ algorithm. Rasdaman iterates over polygons, clips, and aggregates the raster data for each one. This would result in some redundant access to the raster data for nearby or overlapping polygons. On the other hand, RJ⋈, even when running on a single machine, ensures that the raster tiles are accessed only once. Rasdaman

would still be helpful for processing a single polygon or a very few polygons but it does not scale for large vector data.

Distributed Systems: Figure 4.9 shows the overall running time for $RJ_{\bowtie}$ as compared to RZS, GeoTrellis, and GEE. Since all these systems are more scalable than the previous ones, we only try on the two bigger raster datasets, US-Aster and TreeCover. $RJ_{\bowtie}$ is still the fastest algorithm in almost all cases. The only case where RZS is faster is when joining Boundaries with TreeCover. Since the Boundaries dataset is small, RZS would broadcast it to all machines where it then processes each file locally. $RJ_{\bowtie}$ would still partition the *Flash* index which might result in some overhead as multiple machines might process different parts of the same file. However, for big vector and raster data, $RJ_{\bowtie}$ is more than 50 times faster than RZS. Additionally, $RJ_{\bowtie}$ is more flexible since the RZS algorithm solves only the zonal statistics problem and runs only with polygons.

Breakdown of $RJ_{\bowtie}$ Running Time Figure 4.10 shows the breakdown of the total running time for $RJ_{\bowtie}$ into three steps, *Flash Index Creation*, *Flash Index Optimization*, and *Flash Index Processing*. It can be observed from the figure that the running time is dominated by the *Flash Index Processing* step. This is because this step is dominated by disk IO for reading the required pixel values from disk. *Flash Index Creation* takes about 10%-40% of the running time and depends on the size of vector data. Treecover dataset is made up of multiple raster files. This is why for this dataset, *Flash Index Creation* step takes more time as it needs to compute the *Flash Index* for each raster file separately. *Flash Index Optimization* takes the least amount of time and depends on the distribution of intersections across worker nodes.

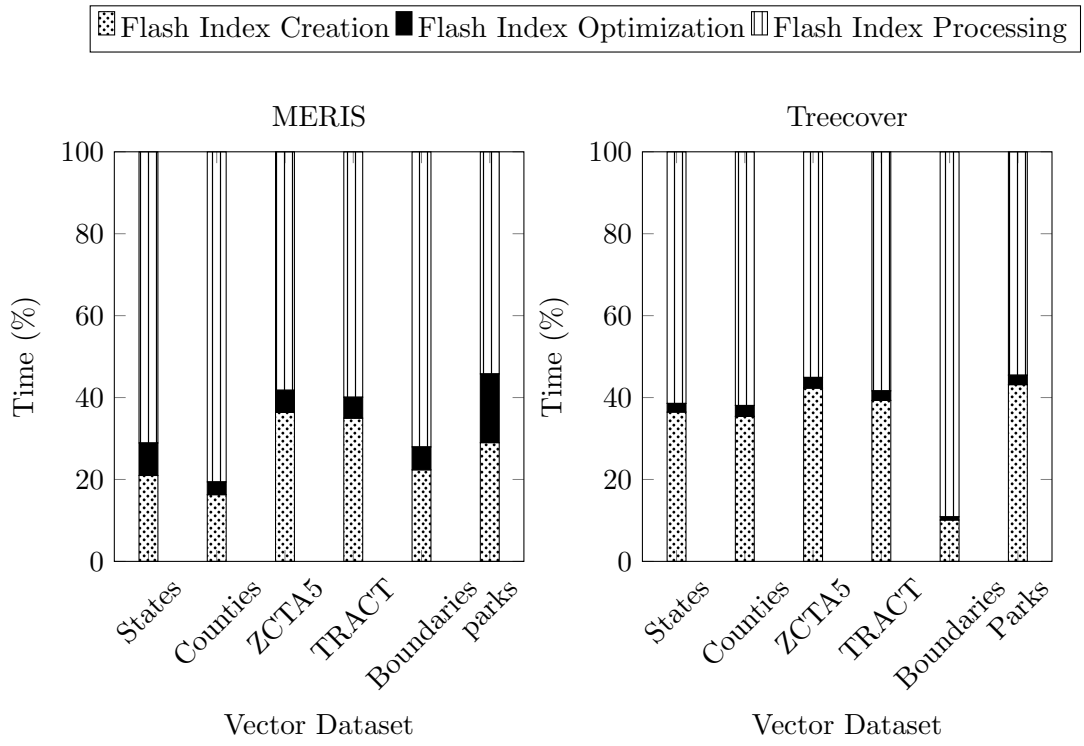
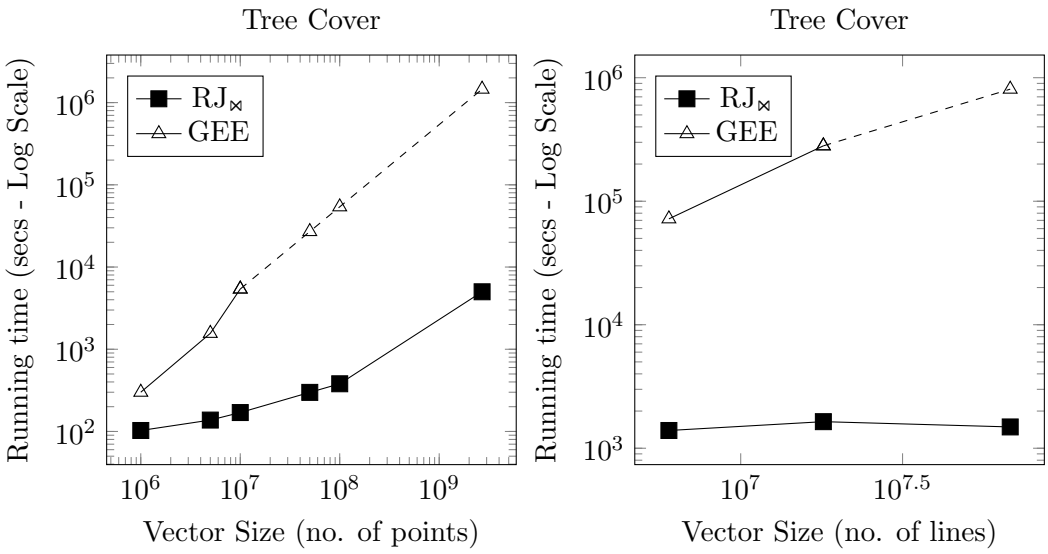


Figure 4.10: Breakdown of $RJ_{\bowtie}$ running time

4.5.4 Flexibility of $RJ_{\bowtie}$

Non-polygon joins: To show the flexibility of $RJ_{\bowtie}$, we test its performance on non-polygonal joins, i.e., point and linestring joins. An application of point joins is agricultural applications that combine ground sensors at fixed points with remote sensors and build a regression model between them [58]. Linestring joins can be used in hydrology applications to compute the elevation model along water paths to measure the water flow and delineate watersheds [19]. Figure 4.11 shows the performance of point and line joins for the vector datasets in Table 4.1 and the largest raster dataset Treecover for $RJ_{\bowtie}$ and GEE.



(a) Point Features

(b) Line Features

Figure 4.11: Performance on non-polygon joins with big raster data. Dotted lines represent extrapolated values.

We chose GEE as a baseline as we observed it to be the most scalable during the experiments for polygon joins. As can be observed, RJ$_{\bowtie}$ outperforms GEE for both linestring and point joins with over two orders of magnitude performance gain. The dotted lines for GEE indicate an extrapolation that we did to estimate the running time for larger vector datasets, since it was not able to process the entire dataset in one run.

Applications: Figure 4.12 shows the results for applications discussed in [68]. As can be observed, RJ$_{\bowtie}$ is at least 10x faster than GEE for the first two applications. For the first application, wildfire combating, RJ$_{\bowtie}$ is used to calculate statistics for both California (3 million polygons) and the entire US (55 million polygons). It takes as input 23 rasters from *landfire.gov* each containing over a billion pixels. The second application, crop yield mapping, takes as input 360,000 agricultural fields and over 1.8 TB of medium

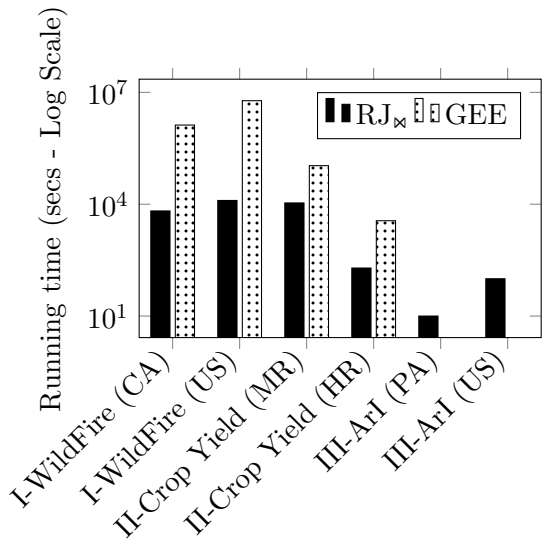


Figure 4.12: Applications

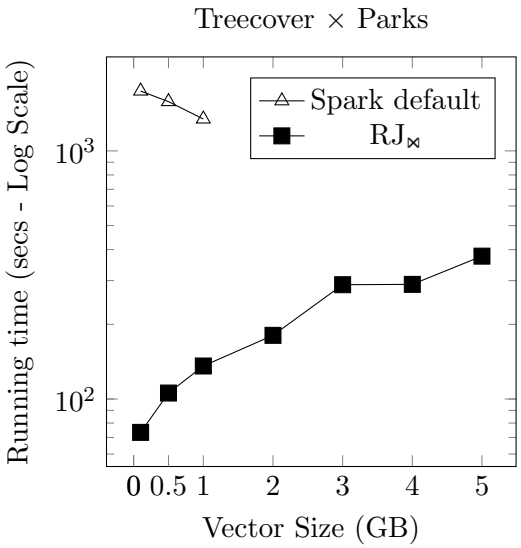


Figure 4.13: Vector Partitioning

resolution(MR) raster data and 31 GB of high resolution(HR) Planet Data. For the third application of areal interpolation(ArI), the results are for the single machine implementation of RJ⋈. It was used to join the National Land Cover (NLCD) raster dataset, with 16 billion pixels, with 74k TRACT polygons to estimate their population.

4.5.5 Optimizing RJ⋈

***Flash* Index Construction:**

This experiment studies the effect of partitioning vector data during the *Flash* index construction step. We compare the default Spark partitioning, that creates a partition for each 128 MB block, against the one proposed in RJ⋈ that partitions the file into blocks of 16 MB each, followed by another partitioning if the number of blocks is less than the number of workers in the cluster. The experiment is conducted using the raster dataset,

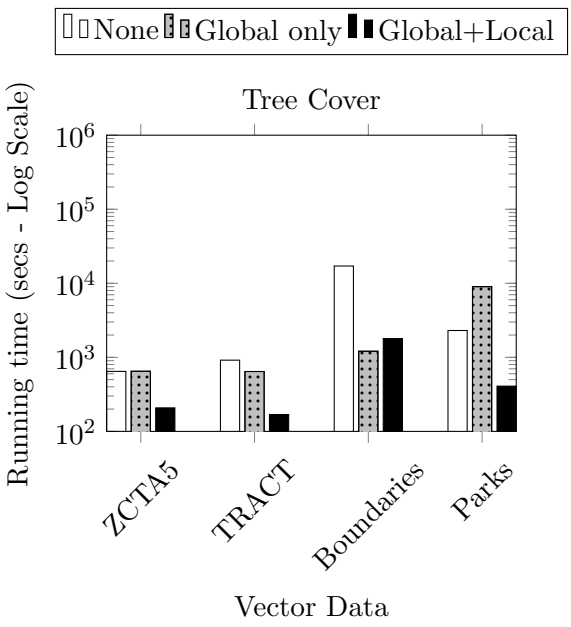


Figure 4.14: Optimization

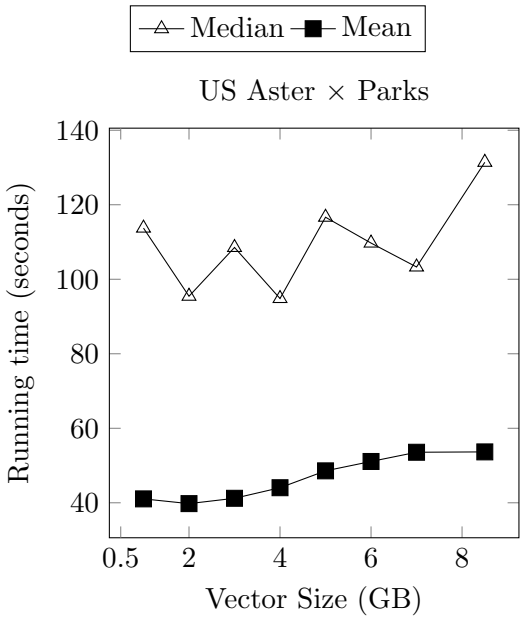


Figure 4.15: Aggregation

Treecover and subsets of vector dataset, Parks. As can be seen in Figure 4.13, the proposed partitioning runs 100 times faster for small datasets due to the additional re-partitioning step that ensures that all executors participate in the processing. For large datasets, the default partitioning method fails with out of memory exception due to the large number of intersections per 128 MB partition. Using a 16 MB partition reduces the overall memory overhead per executor.

***Flash* Index Optimization:**

This experiment studies the effect of global and local index optimizations on the running time of RJ$_{\bowtie}$ algorithm. As can be observed in Figure 4.14, using only the global optimization helps with small vector datasets, i.e., boundaries, but it does not help much with medium-scale datasets, i.e., ZCTA5 and TRACT, and reduces the performance for

large datasets, parks. The reason is that for a small dataset, the *Flash* index is small enough that only partitioning by tile ID (global optimization) is enough while sorting within partition might not help much. For large vector data, local optimization is critical due to the large number of pixel ranges. This can be observed with the parks dataset where global+local optimizations achieve an order of magnitude speedup.

Efficient Aggregation:

There are two types of aggregate functions, *algebraic* and *holistic*. Algebraic functions can be computed efficiently using `reduce` or `aggregate` operation. local and then global aggregation, e.g., min, max, and average. These can be computed in Spark using the `reduce` or `aggregate` operations. Holistic functions, on the other hand, may require collecting all values in one machine and are thus less efficient to compute, e.g., median and percentile. They can be implemented in Spark using the less efficient `groupBy` operation. Since $\text{RJ}_{\bowtie}$ is integrated in Spark, it can compute both types of functions by simply following the $\text{RJ}_{\bowtie}$ operation with the appropriate Spark operation. This design breaks from the limitation of RZS which can only support a limited number of algebraic aggregate functions. Figure 4.15 compares the computation of mean and median as an example of algebraic and holistic functions, respectively. As shown, $\text{RJ}_{\bowtie}$ is three times faster when computing the mean due to the more efficient calculation method. This experiment also shows the flexibility of $\text{RJ}_{\bowtie}$ with any function that the users want to compute.

4.6 Conclusion

The chapter proposes a new raster-vector join algorithm, Raptor Join ($RJ_{\bowtie}$). It overcomes the limitations of the existing systems by combining raster-vector data in their native formats by using a novel index structure *Flash Index*. This algorithm is modeled as a relational join operator and uses an in-situ approach, hence, making it attractive for ad-hoc queries. It runs in three steps, namely, *Flash Index creation*, *Flash Index optimization* and *Flash Index processing*. The *Flash Index creation* step computes a mapping between raster and vector in the form of pixel ranges. The *Flash Index optimization* step partitions and reorganizes this data structure across machines in such a way that each tile in the raster dataset is scanned by only one machine. The *Flash Index processing* step processes the partitioned pixel ranges to read the required pixel values from the raster dataset. We run extensive experiments for the system against Rasdaman, GeoTrellis, Google Earth Engine, Adaptive Cell Trie, GeoSpark, and Beast on large raster and vector datasets to show its scalability and performance gain.

Chapter 5

Distributed Raster Pre-processing

Raster data pre-processing is an important part of a spatial query pipeline. Researchers may need to pre-process raster data before it can be combined with vector data using Raptor. In this section, we propose *RDPro* that adds distributed raster processing capabilities to Raptor.

5.1 Introduction

There is an ever-increasing amount of geospatial data, which has been made possible due to the advancements in remote sensing technology. There are currently more than 1000 [73] active satellites in-use for collecting Earth Observational Data with resolutions varying from 50 cm to 1 km per pixel. Another source of spatial data is aerial imagery, which refers to images taken from drones, balloons, or airplanes. It is a relatively newer source and offers higher spatial resolution than satellites, which is up-to 1-5 cm per pixel. Both public organisations such as NASA, USGS, and European Space Agency (ESA) and private

organisations such as Planet Labs, Hexagon Geosystems, and NearMap capture satellite and aerial images. Due to their efforts, today we have petabytes of earth observational data available for use.

Satellite and Aerial imagery is an example of raster data which is represented using multi-dimensional array of values. Raster data is an important component of research in fields such as disaster response and monitoring [26, 6], management of energy and natural resources [48, 50, 8], agricultural monitoring [59, 61, 79], and marine biology [35, 23]. The increased availability of data has allowed for significant progress to be made in these research applications. However, this has also created the challenge of efficiently processing such large amounts of raster data.

There exist systems for processing raster data such as GDAL [20], PostGIS [53], SciDB [70], GeoTrellis [22], Rasdaman [3], Sedona [80], Google Earth Engine [27] and ChronosDB [84]. However, these systems suffer from one or more of the following limitations: 1. single machine [20, 53], 2. successive disk I/O [84, 20], 3. expensive data ingestion [70, 3, 22, 80, 27], and 4. limited functionality [70, 80].

A detailed explanation of how existing systems suffer from these limitations is provided below:

1. **Single Machine:** There are various raster-based systems [53, 20] that run on a single machine. These systems are not efficient when working with large raster datasets. They either fail because they run out of memory or it may take them days to perform a query on the rasters. In comparison, distributed systems are able to scale to larger datasets and can perform the same query in significantly less amount of time.

2. **Successive Disk I/O:** Some systems [84, 20] are modeled in such a way that they can only perform one query at a time on the dataset. These systems require to read data from disk, perform the computation on the data and then write the data back to disk. If an application needs to perform a series of queries on a raster dataset, these systems would need to read and write data after every query. As the size of raster data increases, these successive disk I/O operations become time expensive and affect the performance of these systems.

3. **Expensive Data Ingestion:** Raster-based systems often implement their own data model which allows them the capability to process huge amounts of raster data. These systems either suffer from the limitation of an expensive data ingestion phase [70, 3, 27] or need to ingest the whole data in memory before processing it [22, 80]. Systems that suffer from the limitation of an expensive data ingestion phase serially read in the data and re-structure it according to their data model. This data loading phase often takes longer than the actual analysis. On the other hand, systems that need to read data in memory before processing are not able to scale to large datasets. This is because the size of large datasets often exceeds the amount of memory available.

4. **Limited Functionality:** The data model used by raster-based systems sometimes limits the type of operations that can be performed on the raster data. For example, [70] implements an array data model and its users can only perform array operations on the data. [80] converts the raster data into vector format and only allows pixel-wise operations to be performed on it. Neither of these systems can perform operations such as reprojection which is necessary when working with datasets in different co-ordinate reference systems.

In this chapter, we propose a novel distributed system, *RDPro*, implemented in Spark that can efficiently perform analysis on big raster data. As mentioned earlier, raster data is represented using multi-dimensional arrays of values. Each value in this array represents a measurement, such as vegetation, and can be identified based on its location in the array. The array is further divided and stored using equi-sized subarrays called *Tiles* that can be randomly accessed. The proposed system derives its data model from how the raster data is stored on disk. It uses a custom RDD, *RDD[ITile]* to represent raster data. It overcomes the limitations of existing systems as follows: 1. The proposed system is a distributed system implemented in Spark and implements the *RDD[ITile]* interface. This allows it to distribute computation across machines and scale to larger datasets. 2. Because the proposed system is implemented in Spark and uses an RDD to model the raster data, it allows the users an advantage to combine multiple operations and run a complex spatial query pipeline on their datasets. The raster data for each of these operations will only be read when required and only the final output needs to be written to disk. 3. The proposed system has the advantage of in-situ processing. It does not need to ingest and re-structure data before performing analysis on it. *RDPro* only reads metadata (a few KBs in size) information from the raster data and initializes the *RDD[ITile]*. It only needs to read the required raster data when performing the analysis and does not require to keep input data in memory. 4. The proposed system implements map algebra operations which include local, focal, zonal, and global operations. These operations provide an exhaustive list of operations that users may want to use to analyze raster data.

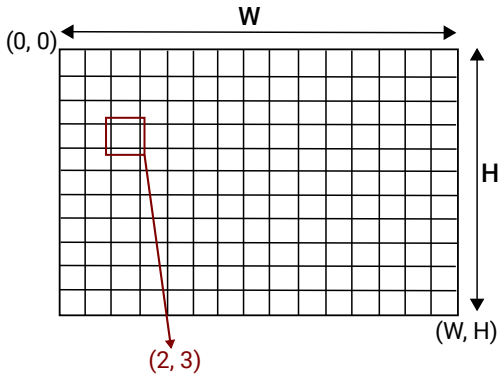


Figure 5.1: Raster Data Model

We compare the proposed system to GeoTrellis [22], GDAL and Sedona [80], and show that it has significant performance gain over them while being perfectly able to scale to big raster data and use fewer resources.

The rest of this chapter is organized as follows: Section 5.2 describes the data and query model of the proposed system. Section 5.3 details the architecture of the proposed system. Section 5.4 runs an extensive experimental evaluation of the proposed system. Section 5.5 concludes the chapter.

5.2 Problem Formulation

In this section, we formulate the problem of raster data processing solved by the proposed system *RDPro*. First, we define logical data models for the two types of spatial data, raster and vector, and then we define the operations for raster analysis using the two data models. Figure 5.1 provides an illustration related to these definitions.

5.2.1 Raster Data Model

Definition 1 (Raster Grid, G) *A raster grid $G = (W, H)$ is a two-dimensional grid that consists of W columns and H rows.*

Definition 2 (Grid Space) *Grid space is defined as the two-dimensional Euclidean space that covers the range $[0, W[\times[0, H[\in \mathbb{R}^2$. The origin of the grid space $(0, 0)$ is always at the top-left corner as shown in Figure 5.1. The location of a raster in the grid space bears no resemblance to what geographical area it represents.*

Definition 3 (Pixel, p) *A pixel $p = (i, j)$ represents the cell in the grid at column $0 \leq i < W$ and row $0 \leq j < H$. According to the definitions above, a pixel $p = (i, j, m)$ occupies the grid subspace $[i, i + 1[\times[j, j + 1[$.*

Definition 4 (Measurement, M) *The measurement is a function that defines a value for each pixel.*

$$M : (i, j) \rightarrow \mathbb{R}^b$$

where $0 \leq i < W$ and $0 \leq j < H$ are integers and $b \geq 1$ is an integer that represents the number of bands for the measurement. For example, RGB rasters contain three bands for red, green, and blue. We use $M(i, j)$ to indicate the value of the pixel at location (i, j)

Definition 5 (Non-geographical Raster Dataset) *A non-geographical raster dataset is defined by a grid G and a measurement function M.*

A non-geographical raster dataset can represent an image but it is not associated with any geographical location. The next set of definitions will help in defining a geographical raster dataset that is associated with a location on the earth surface.

Definition 6 (World Space) *The world space represents a rectangular space on the Earth surface defined by four geographical coordinates* (x_1, y_1) *and* (x_2, y_2) *that define the space* $[x_1, x_2[\times[y_1, y_2[$. *The world space is associated with a coordinate reference system (CRS) that defines how the values in world coordinates map to earth surface.*

Definition 7 (Coordinate Reference System (CRS)) *A Coordinate reference system (CRS) defines how a point in the world space maps to the earth surface. Each CRS is defined by a unique spatial reference identifier (SRID). The details of all types of CRS are outside the scope of this work but interested readers can refer to the ISO-19111 standard [31] for the details. What matters for this work is that there is a standard method to transform a point coordinate from one CRS to another CRS.*

An example of CRS is the world geodetic system, WGS84 with SRID=4326, which defines a location by its longitude and latitude degrees and is used by GPS.

Definition 8 (Grid-to-World, $\mathcal{G}2\mathcal{W}$**)** $\mathcal{G}2\mathcal{W}$ *is a 2D affine transformation that transforms a point from the grid space to the world space.*

$$\mathcal{G}2\mathcal{W} = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ 0 & 0 & 1 \end{bmatrix}$$

The inverse of this matrix is called world-to-grid, $\mathcal{W}2\mathcal{G} = \mathcal{G}2\mathcal{W}^{-1}$ *and can be used to map locations from world space back to grid space.*

Definition 9 (Geographical Raster Dataset, R**)** *A geographical raster dataset is defined by a grid space,* $G = (W, H)$*, a measurement function* M*, a grid-to-world* $\mathcal{G}2\mathcal{W}$ *transformation, and a CRS defined by its unique SRID.*

In a geographical raster dataset, termed *raster dataset* from this point on, each pixel occupies a rectangular space in the world defined by transforming its occupies grid space using the associated $\mathcal{W}2\mathcal{G}$. The measure value $M(i, j)$ of that pixel indicates a physical value measured for that area, e.g., temperature or vegetation. Although, the pixel width and height in grid space is one unit of measurement, in world space the pixel width may not be equal to pixel height. For example, the pixel width in world space may be 80 cm and height may be 30 cm.

5.2.2 Vector Data Model

A vector dataset, V, is defined as a set of geometric features that comprise of points, lines, or polygons. Points represent discrete data values using a pair of longitude and latitude (lon, lat). Lines or linestrings represent linear features, such as rivers, roads, and trails. Each line is represented by an ordered list of at least two points. Polygons represent areas such as the boundary of a city, lake, or forest. Polygons are represented as an ordered collection of closed linestrings, i.e., rings, which constitute the boundary of the polygon and optionally holes inside it. Similar to the raster dataset, a vector dataset is associated with a CRS that defines its world coordinates.

5.2.3 Raster Operations

Raster analysis may require processing one or more raster datasets to produce either a value or another raster dataset. The set of operations that are required to analyze raster data are called map algebra and are broadly classified into four categories: 1. local,

2. focal, 3. zonal, and 4. global.[1]. The definition of the operations implemented by the proposed system is as follows:

1. *MapPixels*: This operation takes as input a raster dataset R_1 and a function f, and outputs the raster dataset R_2 with modified pixel values. R_1 and R_2 share the same dimensions W and H, the same CRS, and the same $\mathcal{G}2\mathcal{W}$. They only differ in the number of bands m_1 and m_2. The function f is a used-defined function that maps a measure value in R_1 to R_2, i.e., $f : \mathbb{R}^{m_1} \rightarrow \mathbb{R}^{m_2}$. The MapPixels operation applies the user-defined function f to each pixel measurement in R_1 and the output of the function is set as the pixel value in the output raster R_2. This is a type of local operation and can be used to, for example, add a constant value to each pixel in the raster. It is defined as:

$$MapPixels(R_1, f) \rightarrow R_2$$

where

$$R_2[i, j] = f(R_1[i, j]) \forall 0 \leq i < W, 0 \leq j < H$$

2. *Overlay*: This operation takes as input two raster datasets R_1 and R_2, and returns a new raster dataset R_3. All the three raster datasets share the same dimensions, W and H, the same CRS, and the same $\mathcal{G}2\mathcal{W}$, but they might have different number of bands m_1, m_2, and m_3. This operation concatenates the measurement values of corresponding pixels in R_1 and R_2 to produce one measurement in the output R_3. This is a type of local operation and can be used to output a raster with multiple bands. This operation is defined as follows:

$$Overlay(R_1, R_2) \rightarrow R_3$$

[1]More information about map algebra can be found at [42]

where

$$R_3[i,j] = R_1[i,j]||R_2[i,j] \forall 0 \leq i < W \wedge 0 \leq j < H$$

and $||$ is the concatenation operator for two arrays.

3. *Convolution*: This operation takes as input a raster dataset R_1, a window size w, and a function f. It output a raster dataset R_2 with the same size, CRS, and $\mathcal{W}2\mathcal{G}$ as R_1. This operation computes the value of each pixel (i,j) in R_2 by processing all values in R_1 in locations $(i+d_i, j+d_j)$ where $-w \leq d_i, d_j \leq w$ using the function f. This operation is an example of focal operation and can be used to apply a range of analytical functions such as smoothing, sharpening, and edge detection. It is defined as:

$$f : \mathbb{R}^{b_1 \cdot (2w+1)^2} \rightarrow \mathbb{R}^{b_2}$$

$$Convolution(R_1, f, w) \rightarrow R_2$$

where

$$R_2[i,j] = f\left(\cup_{-w \leq di, dj \leq +w} R_1[i+di, j+dj]\right)$$

4. *Reprojection*: This operation takes as input a raster dataset R_1, target CRS CRS_2, target raster width W_2 and target raster height H_2. It reprojects the input raster to the target CRS with the specific raster size. Both R_1 and R_2 will have the same number of bands. The value of each pixel in R_2 is equal to its nearest pixel in R_1 *in world space*. This is an example of a focal operation and is often used when the user wants to analyze multiple rasters that do not have the same CRS. It is defined as:

$$Reprojection(R_1, CRS_2, W_2, H_2) \rightarrow R_2$$

Where

$$R_2[i_2, j_2] = R_1[i_1, j_1] : (i_1, j_1) = \mathcal{W}2\mathcal{G}_2(T(\mathcal{G}2\mathcal{W}_1(i_1, j_1), CRS_2))$$

If $0 \leq i_1 < W_1 \wedge 0 \leq j_1 < H_2$. Otherwise, $R_2[i_1, j_2] = null$.

A special case of this function is the resampling function when the target CRS is the same as the source CRS.

5. *Resampling*: This operation takes as input a raster dataset R_1, target raster width W_2 and target raster height H_2. It outputs a raster dataset R_2 of the target width and height. Both R_1 and R_2 share the same CRS and number of bands. It computes its value of each target pixel based on the values of its nearest neighboring pixel of the input raster in world space. This function is a special case of the reprojection function when the target CRS is same as that of the input.

6. *Raptor Join*: This operation takes as input a raster dataset R and a vector dataset V. It is used to select pixels from the raster that overlap the geometries in the vector dataset. It outputs a set of (g_id, i, j, m) tuples. In this tuple, g_{id} represents a unique identifier for each geometry in the vector dataset. (i, j) is the position of the matching pixel in grid coordinates, and m is the measure value of the matching pixel. This is an example of a zonal operation and can be used to compute statistics of pixel values for each geometry (zone). It is defined as:

$$RaptorJoin(R, V) \rightarrow \{(g_id, i, j, m)\}$$

7. *Reduce*: This operation takes as input a raster dataset R and a reduce function f and outputs a value v. This operation aggregates all the pixel measure values into a single

value with the same number of bands as the input. The reduce function f takes as input two pixel values,m_1 and m_2 and outputs another value m_3 as formalized below.

$$f : (\mathbb{R}^b, \mathbb{R}^b) \rightarrow \mathbb{R}^b$$

It is applied recursively on all original and resulting values until a single value v is produced. This is an example of a global operation and can be used to find the minimum, maximum, or average, of all pixels. It is defined as:

$$Reduce(R, f) \rightarrow v$$

5.3 RDPro Architechture

This section describes the proposed system **RDPro** (Raster Distributed Processing). The goals of this system are: 1. **Distributed Processing:** *RDPro* implements distributed reading, writing and processing of raster data. 2. **Efficiency:** *RDPro* can handle high-resolution raster data. 3. **Comprehensiveness:** *RDPro* implements a wide range of operations for raster analysis which can be combined to form a complex spatial query pipeline.

Most existing systems are implemented based on two assumptions. First, that a raster dataset is physically represented as a set of files that are small enough to fit in memory. Second, that each of these files can be processed separately. They implement raster-based operations by loading each raster file into memory and then performing the required analysis on it. Thus, these systems fail to process raster files whose size exceeds the memory size. For example, the US Aster dataset is 35 GB in size and is stored as a

single raster file. It is not small enough to fit in memory and therefore cannot be processed by existing systems unless manually split into several small files. Moreover, the files used to physically represent a raster dataset are logically a part of the same raster grid. Therefore, when these files are processed separately, their logical connection is not taken into account. For operations such as reprojection, it is important to reproject the raster grid as a whole. Reprojecting each file separately may result into a grid that is not well aligned and has overlapping pixels.

In order to implement a distributed system for processing raster data, the proposed system needs to devise methods to partition raster data across machines, load and process this partitioned data, and write the partitioned back to disk in the raster format. However, partitioning raster data is not that straightforward. Usually, for textual file formats, the data can be partitioned line-wise, but this cannot be done for raster data. To overcome this challenge *RDPro* takes advantage of how raster data is stored on disk. Raster files are stored by dividing the raster grid into smaller equi-sized grids called *tiles*(more details in next section), each of which is stored separately. Moreover, these tiles are small enough to load into memory for data processing. Therefore, *RDPro* chooses a tile to be the smallest processing unit and implements algorithms to work with it.

Figure 5.2 shows the architecture of *RDPro*, which consists of four main components: 1. Raster Data Model (RDD[ITile]), 2. Data Loading, 3. Data Writing, and 4. Raster Query Processing. *RDPro* is implemented in Spark and extends the RDD programming interface to implement a (physical) data model whose processing unit is a *tile*. It stores raster data on a distributed file system. The *RDPro* data model is implemented in such a way that

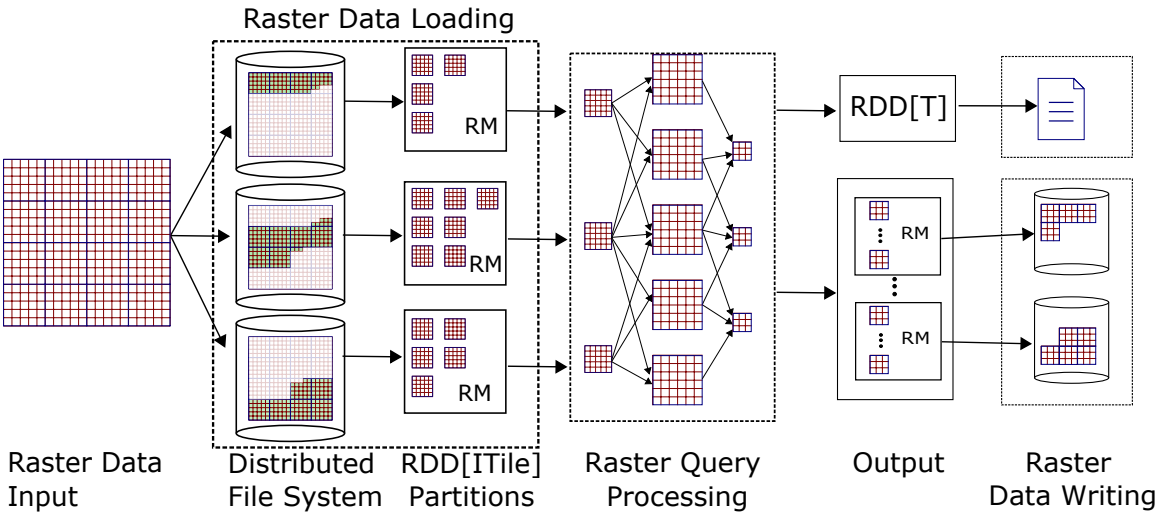


Figure 5.2: RDPro Architecture

it can read, write and represent the distributed raster data. The data loading component is responsible for reading this partitioned raster data from disk and initializing the Raster RDD, while the data writing component is responsible for writing the processed raster data to disk. The raster RDD component handles the tasks associated with the *RDPro* data model while the raster query processing component implements raster operations. We provide details about the proposed data model, data input, data output, and how various raster data operations are implemented in the following sections.

5.3.1 RDPro Data Model

The main goal of the proposed system *RDPro* is to efficiently process raster data in a distributed manner. This means that a part of the raster data should be assigned to each machine for processing. However, unlike text file formats raster data can't be partitioned based on the number of lines in it. In order to partition raster data, RDPro takes advantage

of the way raster data is stored on disk. Most standard raster file structures, e.g., GeoTIFF and HDF5, physically store the raster dataset by partitioning the raster grid into tiles.

Definition 10 (Raster Tiles) *Raster tiles are smaller equi-sized partitions of the raster grid.. The tiles can be identified by sequence numbers identifiers, t_{id}, starting at zero and typically are tens of kilobytes in size.*

Each tile is stored as one block on disk and is typically small enough to load entirely in main memory. The raster file contains a lookup table that allows locating any tile efficiently. This makes tiles the perfect choice of data unit to process raster data in a distributed environment. However, in order to process each tile separately, certain auxiliary information called raster metadata is needed.

Definition 11 (Raster Metadata, RM) *Each raster is associated with auxiliary information called metadata, which is stored in the header of the raster file and consists of the following information:*

1. *Number of columns (c) and rows (r) of pixels in it.*
2. *Tile width (tw) and tile height (th) in pixels.*
3. *The grid-to-world ($\mathcal{G}2\mathcal{W}$) affine transformation matrix that converts a pixel location on the grid to a point location in the world, and the inverse, world-to-grid ($\mathcal{W}2\mathcal{G}$) transformation.*
4. *Coordinate Reference System (CRS) which describes the projection that maps the Earth surface to the world coordinates as defined by the ISO-19111 standard [31].*

RDPro implements its own (physical) data model which allows it to process partitioned raster data in form of tiles. Since the proposed system is implemented in Spark, it extends the RDD programming interface to implement it. RDD (Resilient Distributed Dataset) is defined as a fault-tolerant collection of elements that can be operated on in parallel. The proposed system creates a custom RDD, represented as *RDD[ITile]*, whose each element is a raster tile which in turn has the raster metadata associated with it. This allows the proposed system to partition raster data into a set of tiles which can then be processed separately on each machine.

As shown in figure 5.2, the raster data is stored on a distributed file system. The users do not need to pre-process raster data into tiles before storing it on the distributed file system. The data loading component (described in next section) can load the raster data and initialize the *RDPro* data model. The data model is initialized using only the raster metadata, which is a few KBs in size at most and the location of the tiles on disk. The actual tile data is not loaded until needed during raster operations. Since, the proposed system does not load the data in memory until required, this allows the system to scale for high-resolution raster data. Moreover, the input and output of each raster operation is an RDD which allows the user to pipeline multiple operations in Spark and run complex spatial query pipelines. The raster data for each of these operations will only be read when required and only the final output needs to be written to disk. This allows the system to save on a lot of unnecessary disk I/O and speed up processing.

5.3.2 Raster Data Loading

In this section, we describe the raster data loading component of the proposed system. *RDPro* does not require the users to do any pre-processing and allows them the advantage of in-situ processing. As can be seen from figure 5.2, the users only need to store data on a distributed file system. *RDPro* can also process data stored locally on disk and not on a distributed file system. However, doing so will keep users from taking full advantage of *RDPro*'s distributed processing capabilities.

The raster data loading component of *RDPro* loads data from disk and initializes its data model, RDD[ITile]. *RDPro* implements its own custom raster data loader due to the complex structure of raster files. Unlike existing distributed data loaders in Spark that mostly work with text files, raster data cannot be easily split at text line boundaries. This makes it challenging to load big files in parallel. When a large raster file, e.g., GeoTiff, is stored in a distributed file system, it is split into fixed-size blocks which are distributed among the compute nodes. To efficiently load the file, each machine should load the tiles that are locally stored in its block. The challenge is that some tiles span multiple blocks and they should be read exactly once. Furthermore, to parse the file correctly, the header of the file is needed which is usually located in the first block.

RDPro is implemented in Spark, and therefore, the raster data loading component is implemented in similar to other distributed file parser. It works in two steps: 1. splitting, 2. and reading. The *splitting* step runs on a single machine and splits the input file into smaller partitions that can be processed independently. The *reading* step runs in parallel on one split at a time and reads the tiles within the given split.

Details about these steps are as follows:

Step1: Splitting The splitting step runs on a single machine before the Spark job starts. It produces a set of partitions that define the raster data model, RDD[ITile]. This step reads only the header of the raster file, which is typically a few kilobytes, so it does not incur a huge overhead. From the header, it determines where each tile starts in the file and uses this to correctly define split boundaries. It defines one split for each block in the file which contains the start and end offset of the split and the metadata of the raster file.

Step2: Reading In the reading step, a machine takes one split and should read all the tiles within that split. The problem is with tiles that span two partitions. To ensure a correct result, we define an *anchor point* as the offset of the first byte in a tile. Then, the reading step will read only the tiles that have an anchor point within the split range. The information for defining the anchor point of each tile and how to parse each tile is all included in the metadata within the split.

Please note that keeping in line with Spark's lazy evaluation model, the reading step is only executed when raster data needs to be processed. This allows *RDPro* the advantage of in-situ processing. The data loading component of *RDPro* also saves it from having an expensive ingestion and restructuring phase which many existing systems suffer from.

5.3.3 Raster Data Output

In this section, we discuss the raster data writing component of *RDPRo*. As shown in figure 5.2, the output of a raster operation can be a raster RDD, RDD[ITile]. This creates

the need of a distributed writer that can write raster data to disk in a distributed file system. To do so, *RDPro* implements its own distributed raster data writer. It writes raster data in standard raster file formats, e.g., GeoTiff, so that users can use the output in another system.

There are two main challenges with writing raster files to a distributed file system. First, a distributed file system can write a file only in a sequential mode and with a very large output file, it would generally be impractical to cache the data in memory before writing to disk. Second, the tiles that form a single file might be distributed across multiple machines and bringing them together in one machine would require an extra network overhead.

To overcome the first challenge, the proposed writer runs in two rounds. The first round writes the majority of the file contents in a sequential file without the file header. The second round compiles the file header and concatenates it with the main contents to form a correct file.

First round: The first round writes the majority of the file contents to disk but not in a well-formed raster file. It takes one tile at a time and appends all tile data to a temporary file. While writing, it keeps track of the start and end offset of each tile within the temporary file. In addition to decompressed format, the proposed writer supports two popular compression format for raster files, deflate and LZW compression schemes which can significantly reduce the written disk size. The output of this step is a collection of files, one for each output partition, and the associated information of each tile ID, the start, and length of each tile data in the file.

Second round: The second round adds the appropriate header to write a set of well-formatted output files.

To overcome the second challenge, *RDPro* proposes two modes for writing the output, compatibility mode and fully distributed mode. The compatibility mode produces a file that is backward compatible with existing libraries but it requires an additional step to weld all pieces into one file. The fully distributed mode is more efficient but will produce multiple files that should be individually loaded using an external system or library. The difference between the two modes is in the second round as detailed below.

Compatibility mode: In the compatibility mode, the second round collects all tile information produced in the first round into one machine which contains the list of written tile IDs and their location on files. Notice that since the first round runs in parallel, the tiles are written to several files. This step will concatenate all these files into one file with all tile data. Then, it will form the file header in memory which stores information about the dimension of tiles and the start and length of each tile. During this process, the tile offsets are updated again to account for the size of the file header. Finally, the header file and tile data file are concatenated to produce the final output. If the output file is very large, this step might be a bottleneck since it needs to write a single file with all data but it is still faster than traditional libraries since all the heavy computations including tile compression is done in parallel.

Fully distributed mode: The fully distributed mode avoids the bottleneck in the concatenation step by writing the files in a completely distributed manner. Instead of collecting all metadata in one machine, it writes a separate file for each partition. However,

this would result in *patched* raster files which might contain many missing tiles. To overcome this problem, this step performs two additional steps. First, it writes one additional tile that is completely empty and update the tile offset of all missing tiles to point to this single tile. This way, if another application tries to read this file, it will not fail but will detect that all the tiles are empty. Second, it adds a special tag in the header that indicates which tiles are empty as a bit mask. This additional field is only readable by our system to allow the distributed output to be read efficiently.

5.3.4 Raster Query Processing

The operations for raster analysis are called Map Algebra and can be divided into four main categories, namely, local, focal, zonal and global. They may be used to join a pixel in one raster dataset to either one or many pixels in other raster datasets. We define each of these operations below and detail how they are implemented in the proposed system. The raster operation discussed in section 5.2 can easily be implemented using the methods described below.

Map Algebra : Local Operations A local operation takes as input a raster dataset and a function and outputs another raster dataset. The function is applied to each pixel value in the input raster dataset. The resulting value of the function is then assigned to the corresponding pixel location in the output raster dataset. Examples of local operations include threshold operation where the value of a pixel is set to zero if its above or below the specified threshold value.

Since the function is applied to each pixel value and is not dependent on any other information or pixel value, each tile in the raster dataset can be processed separately.

This operation would apply the function to each raster tile on separate worker nodes in a distributed manner. The resulting raster dataset can then be either written to disk or used as input for another operation.

Map Algebra : Focal Operations A focal operation is used to change the value of each pixel in the raster based on neighboring values of the pixel. It takes as input a raster dataset, a function and a window size that defines the neighboring pixels. It applies the function to the value of each pixel and its neighboring pixels defined by the window. Then it assigns the resulting value of the function to the corresponding pixel in the output raster dataset. For example, the smoothing operation assigns the mean of the neighboring pixel values to the specified pixel.

This operation is not localized to a tile. Depending on the window size, the function may need to use pixel values from other tiles. RDPro implements this operation in two steps. It first computes the neighboring pixel values defined by the window size for each pixel in a tile and assigns the set of pixel values to the corresponding pixel in a new tile. However, each input tile results in nine output tiles. Although each tile is processed separately and does not contain information about other tiles, it can compute the information required by its neighboring tiles. It creates an empty tile for each neighboring tile, and assigns the pixels in the empty tile the pixel values the neighbour might need from itself. In the second step, all the tiles are merged based on the identifier for the tile and the function applied to the set of values contained in the pixel.

Map Algebra : Zonal Operations A zonal operation takes as input a raster layer and a vector layer. The operations then finds the pixels in the raster layer that

overlaps each geometry(zone) in the vector dataset. A function may then be applied to the resulting pixels. Zonal statistics is an example of zonal operations. This method in RDPro is implemented using our previous work called $\text{RJ}_{\bowtie}$ [69].

Map Algebra : Global Operations A global operation or function takes as input a raster dataset and a function and outputs a set of values. The function is applied to the value of all the pixel in the input raster dataset and reduces them to a set of values. Examples of global operations include finding the minimum value of all the pixels in the raster dataset.

This operation is implemented in *RDPro* in two steps. In the first step, the function is applied to each tile and outputs a set of values corresponding to each tile. In the second step, the output values from all the tiles are combined to produce the final output. Please note that a global operation does not require any data from its neighboring tile, which allows the proposed system to process each tile separately and combine the results.

5.4 Experiments

This section provides an experimental evaluation of the proposed algorithm, RDPro. We compare the distributed RDPro algorithm to the single-machine GDAL package [20], and the distributed systems, Apache Sedona [80] and GeoTrellis [22]. We show that the proposed RDPro is faster than the baselines for big raster datasets.

Section 5.4.1 describes the experimental setup, the system setup and the datasets. Section 5.4.2 provides a comparison of the reading time for the proposed RDPro and GDAL. Section 5.4.2 provides a comparison of the writing time for the proposed RDPro and GDAL.

Table 5.1: Vector and Raster Datasets

Raster datasets

Dataset	# pixels	Resolution	Size	Coverage
GLC2000	659M	1 km	629 MB	World
MERIS	8.4B	300 m	7.8 GB	World
US Aster	187B	30 m	35 GB	US48
Tree cover	840B	30 m	782 GB	World
Landsat8	1.27T	30 m	3.4 TB	World
Planet	4T	3 m	16 TB	USA

Section 5.4.3 to 5.4.5 provides a comparison of the running time of the proposed RDPro and the baseline systems for the various map algebra operations. We do not show experiments for zonal operations as we implemented these operations using our previous work which constitutes Chapter 4. The readers may refer to Chapter 4 for more details and experiments about zonal operations.

5.4.1 Setup

We run *RDPro*, Apache Sedona, and GeoTrellis on a Amazon AWS EMR cluster with one head node and 20 worker nodes of type m5.4xlarge with 2.5 GHz Intel Xeon Platinum 8175M series processor, 64 GB of RAM, up to 1024 GB of EBS, and 16-core processors. GDAL is run on a machine with Intel(R) Xeon(R) CPU $E5-2609$ v4 @ 1.70GHz processor, 128 of GB RAM, 2 TB of HDD, and 2×8-core processors running

CentOS. For GeoTrellis, we use the *geotrellis-spark* package version 3.0.0, as described in its documentation. We used Sedona v1.2.0-incubating as described on its website.

Table 5.1 lists the datasets that are used in the experiments along with their attributes. All raster datasets except Planet Data are publicly available and come from various government agencies. The GLC2000 and MERIS datasets are from the European Space Agency with pixel resolutions of 0.0089 decimal degrees (1km) and 0.0027 (300m) respectively. The US Aster dataset originates from the Shuttle Radar Topography Mission (SRTM) and covers the continental US. Hansen developed the global Tree Cover change dataset which covers the entire globe. Both datasets have a spatial resolution of 0.00028 decimal degrees (30m). The Landsat8 data is a 30m pixel resolution multi-dimensional dataset sourced from USGS (United States Geological Survey). It spans over the world and has three bands. Planet data which spans over the US is sourced from Planet Labs [71]. It is a multidimensional raster dataset with a pixel resolution of 3m.

5.4.2 Data Loading

This experiment compares the data loading times for the proposed system RDPro against the baselines GeoTrellis and GDAL. For this experiment, each of the systems was used to load the data and compute a histogram for each of the raster datasets. The results for this experiment are shown in Figure 5.4.2. As can be observed from the experiment, GDAL is faster in loading data than the proposed system for the smallest dataset, GLC2000. However, RDPro gains in performance as the size of datasets increases. This is because RDPro reads and processes data in parallel while GDAL reads data in serially.

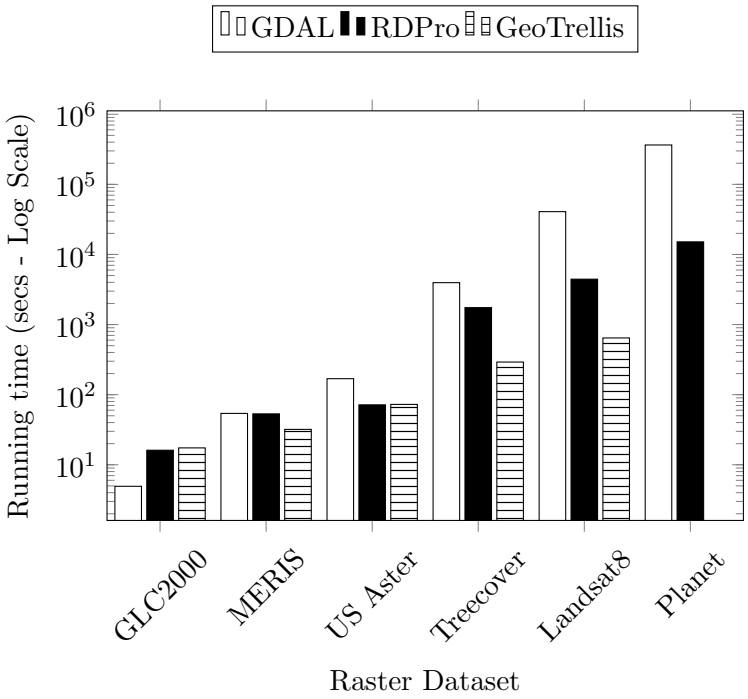


Figure 5.3: Comparison of reading time for GDAL, GeoTrellis, and RDPro

When compared to GeoTrellis, RDPro is either on par in terms of performance or slightly slower than GeoTrellis. However, it is able to scale to the larger dataset Planet, while GeoTrellis fails to do so. This is because GeoTrellis requires to read the whole data in memory before processing it. The Planet dataset is 16 TB (uncompressed) in size and can't be loaded into memory, whereas RDPro takes advantage of its data model to load data only when required and does not need to keep the whole raster dataset in memory. This allows RDPro to scale to larger datasets.

We do not show results for Apache Sedona as it was not able to load any of the raster datasets. Apache Sedona can work with at most 200 million pixels while the smallest dataset GLC2000 has over 650 million pixels.

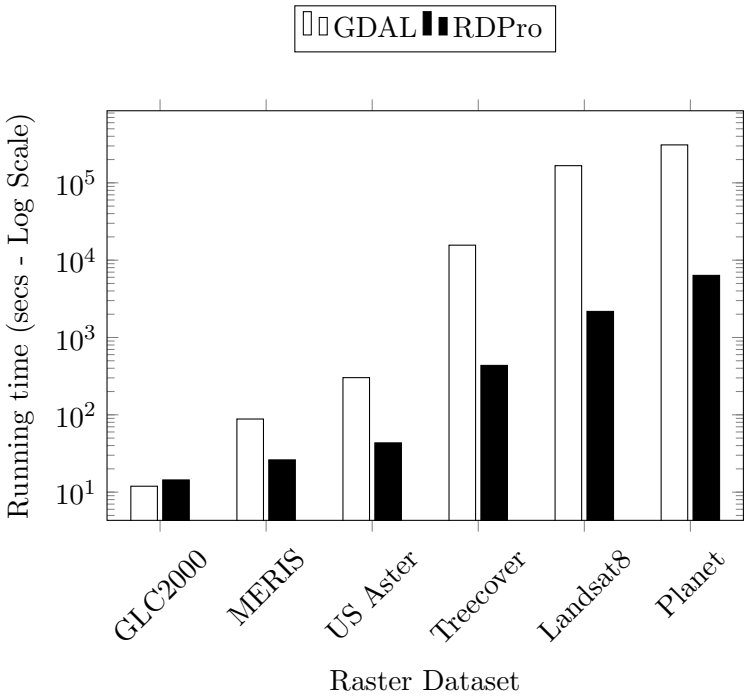


Figure 5.4: Comparison of writing time for GDAL and RDPro

5.4.3 Data Writing

This experiment compares the data writing times for the proposed system RDPro and GDAL. The results for this experiment are shown in Figure 5.4.2. As can be observed from the experiment, GDAL is comparable in writing data to the proposed system only for GLC2000. However, RDPro gains in performance as the size of datasets increases. This is because RDPro implements its own data writing component which writes data in parallel while GDAL writes data serially to disk. We do not show results for Apache Sedona as it was not able to load any of the raster datasets. Apache Sedona can work with at most 200 million pixels while the smallest dataset GLC2000 has over 650 million pixels. Also, we were not able to run this operation for Geotrellis as it requires the raster datasets to be

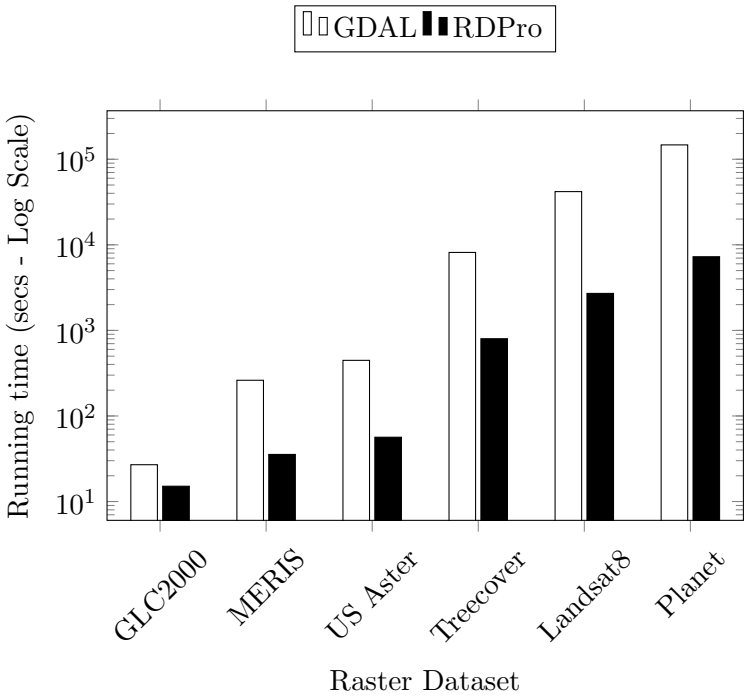


Figure 5.5: Comparison of local operation running time for GDAL and RDPro

stitched into a single tile (represented as an array in Geotrellis) before they can be written to output. However, the number of pixels in the datasets used makes it impossible for all the values to be stored into a single tile(array).

5.4.4 Map Algebra: Local Operations

This experiment compares the running time for a local operation for the proposed system RDPro and GDAL. The map algebra operation that was used was thresholding, where pixel values smaller than a target value are set to zero. The results for this experiment are shown in Figure 5.5. As can be observed from the experiment, GDAL is faster in loading data than the proposed system only for GLC2000 as it is a small file and easier to process

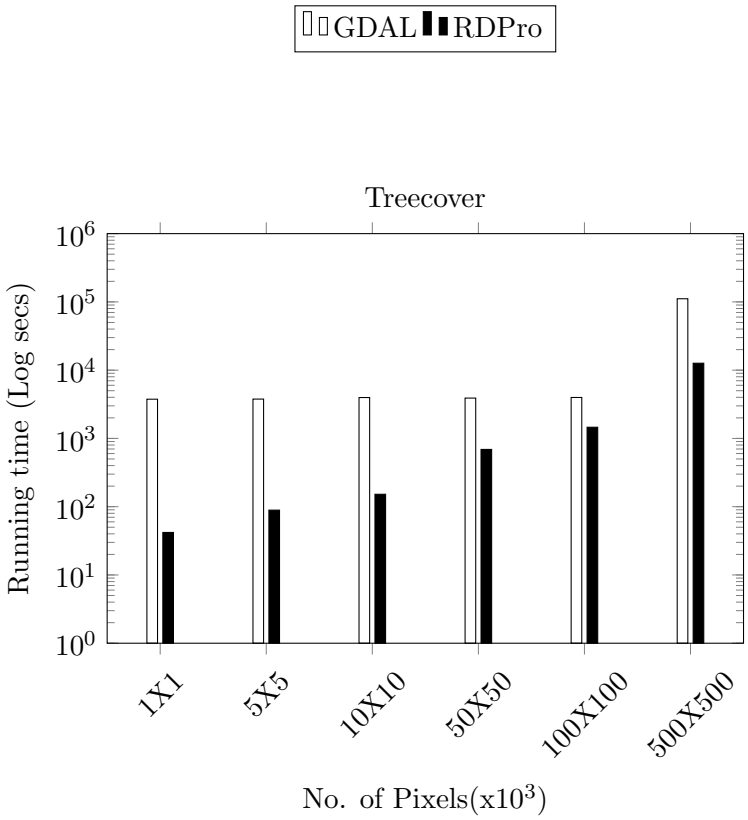


Figure 5.6: Comparison of focal operation running for GDAL and RDPro

on a single machine. However, RDPro gains in performance as the size of datasets increase. This is because as size of data increases, the offset of setting up a distributed job diminishes as compared to the performance gain. We do not show results for Apache Sedona as it was not able to load any of the raster datasets. Apache Sedona can work with at most 200 million pixels while the smallest dataset GLC2000 has over 650 million pixels. GeoTrellis on the other hand fails to run this operation as it tries to combine the whole raster dataset into one single tile before writing it to disk. This is not possible for the datasets used as their size exceeds the size of memory on a single machine in the cluster.

5.4.5 Map Algebra: Focal Operations

This experiment compares the running time for a focal operation for the proposed system RDPro and GDAL. The map algebra operation that was used was reprojection, where pixel values were reprojected from source projection of datasets to EPSG:4269. The results for this experiment are shown in Figure 5.6. We show results for the Treecover dataset. As can be observed from the experiment, RDPro is faster than GDAL. This is due to the distributed processing and writing component of RFPro. We do not show results for Apache Sedona as it was not able to load any of the raster datasets. Apache Sedona can work with at most 200 million pixels while the smallest dataset GLC2000 has over 650 million pixels. Also, we were not able to run this operation for Geotrellis as it requires the raster datasets to be stitched into a single tile (represented as an array in Geotrellis) before they can be written to output. However, the number of pixels in the datasets used makes it impossible for all the values to be stored into a single tile(array).

5.4.6 Map Algebra: Global Operations

This experiment compares the running time for a global operation for the proposed system RDPro, GDAL, and GeoTrellis. The map algebra operation that was used was Statistics, where the statistics such as mean, sum, count, minimum, maximum, and standard deviation of all the pixel values was calculated. The results for this experiment are shown in Figure 5.7. As can be observed from the experiment, GDAL is faster in loading data than the proposed system only for GLC2000 as it is a small file and easier to process on a single machine. However, RDPro gains in performance as the size of datasets increases. This is

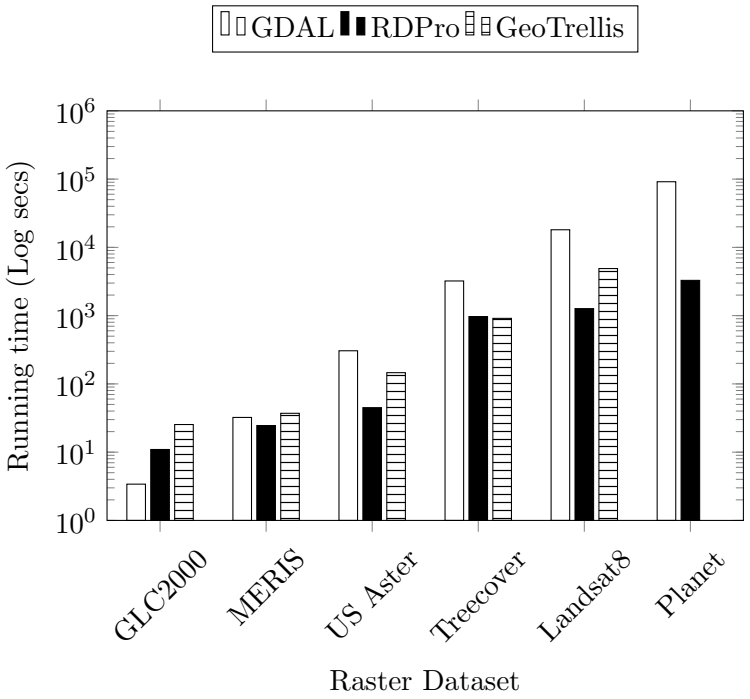


Figure 5.7: Comparison of global operation running for GDAL, GeoTrellis, and RDPro

because as the size of data increases, the offset of setting up a distributed job diminishes as compared to the performance gain. When compared to GeoTrellis, RDPro is either on par in terms of performance or faster than GeoTrellis. It is also able to scale to the larger dataset Planet, while GeoTrellis fails to do so. This is because GeoTrellis requires to read the whole data in memory before processing it. The Planet dataset is 16 TB (uncompressed) in size and can't be loaded into memory, whereas RDPro takes advantage of its data model to load data only when required and does not need to keep the whole raster dataset in memory. This allows RDPro to scale to larger datasets.

We do not show results for Apache Sedona as it was not able to load any of the raster datasets. Apache Sedona can work with at most 200 million pixels while the smallest dataset GLC2000 has over 650 million pixels.

5.5 Conclusion

In this chapter, we propose the distributed system *RDPro* which is used to add distributed raster processing capabilities to Raptor and can scale to big raster data. RDPro is implemented in Spark and uses a custom RDD, RDD[ITile] to represent and process raster data in a distributed environment. It uses RDD[ITile] to implement the operations required for raster analysis. Since the proposed system is implemented in Spark and uses an RDD to model the raster data, it allows the users an advantage to combine multiple operations and run a complex spatial query pipeline on their datasets. We also show how each operation required for raster analysis is implemented using RDD[ITile]. At last, we compare the proposed system to GeoTrellis, GDAL and Sedona, and show its performance gain and scalability.

Chapter 6

Applications

In this section, we discuss three real-world applications that process the combination of raster and vector data.

6.1 Combating Wildfires

Wildfire is a natural disaster which causes massive damage to property and human life. It is a recurring phenomenon, especially in North America, which has led to research in ways to prevent, detect, and combat the spread of wildfires. In the current wildfire season in California so far, more than four million acres have already burned due to more than 8,000 wildfires. At one point in August 2020, the entire northern half of the state had been instructed to prepare for evacuation. This has made it crucial to model the spread of wildfire and predict how to efficiently allocate resources to minimize loss of life and property.

Data-driven modelling of wildfire spread needs to combine the information about occurrences of fire with their corresponding geographical factors [24, 7]. The occurrences

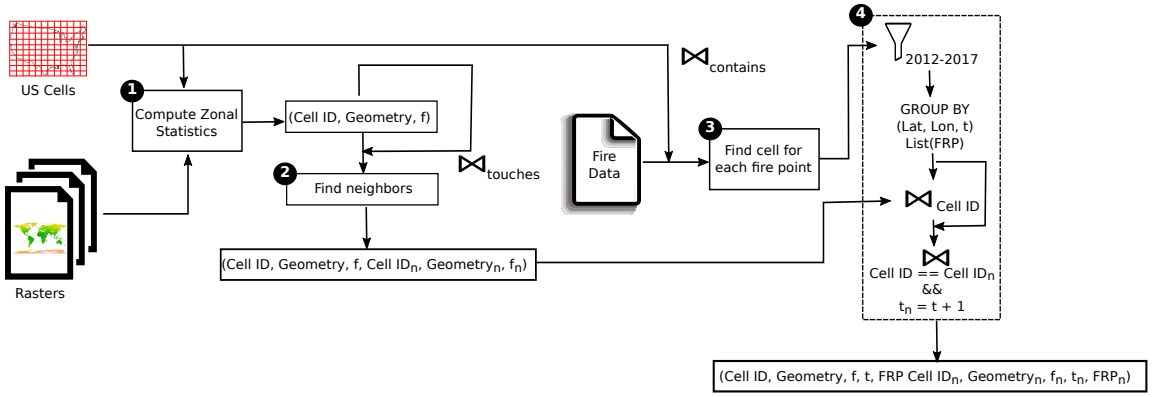


Figure 6.1: Data Generation Process

of *fire* are available in vector format as a collection of geographical points where the fire occurred. Each point also has a variety of attributes associated with it such as the timestamp of fire, fire radiative power (FRP), etc. Geographical factors such as vegetation, elevation, wind direction, and fuel levels, that affect the duration and direction of wildfire spread are available as rasters in form of satellite imagery. Apart from these two datasets, data-driven modelling also requires to divide the target geographic region into numerous polygons called *zones*, which are available as polygons in vector format. At the core of this application, the fire *zones* need to be joined with several raster layers to calculate various statistics such as mean, median, standard deviation, min, and max, of contributing geographical factors for each zone. The computed statistics may or may not have associative and commutative properties, e.g., median is not associative or commutative. After that, it joins the result with the *fire* dataset.

The goal of this application was to build a wildfire dataset for the continental US. The data generation process for it is depicted in Figure 6.1 and includes the following steps:

1. Compute Zonal Statistics: For each cell (fire zone) in the continental USA and for each raster dataset, we want to compute aggregated feature vectors. To compute zonal statistics, we use *Raptor Join*, which outputs a collection of tuples $(g_i, Geometry_i, f_i)$ where $Geometry_i$ and f_i refer to the actual spatial geometry and the set of feature values calculated for cell g_i respectively.

2. Find neighbors: The neighbors for each cell in the spatial grid are computed by doing a spatial self-join using the predicate *touches* on the *Geometry* values of the tuples generated in the previous step. The predicate *touches* returns true, if and only if the boundaries of the cells intersect. We implement the spatial join using Beast [12]. It outputs a collection of tuples $(g_i, Geometry_i, f_i, g_n, Geometry_n, f_n)$ where each tuple in the previous step is appended by the tuples of one of its neighbors (recall that we use subscript n to denote variables that refer to the neighbors of the cell in consideration).

3. Find cell for each fire point: For specific points (latitude-longitude pairs) in fire occurrence data and the cells in our spatial grid, a spatial join using the predicate *contains* is performed to find the cell that each fire point is contained in. The predicate *contains* returns true, if and only if the fire point lies in the interior of the cell. This step is implemented using Beast eldawy2021beast.

4. Generate tuples: To generate the final tuples for *WildfireDB*, we start by filtering the tuples in the fire occurrence data for the years 2012 to 2017. The fire occurrence dataset may contain multiple tuples for the same fire point having the same time-step yet different Fire Radiative Power(FRP) values. We group all such tuples by the fire point and time-step and create a list for the FRP values to generate a single tuple. The resulting

VIIRS tuples are then joined with tuples from Step 2. Finally, we filter information about each neighbor of the cell under consideration at the next time step. This results in tuples of the form $(g_i, Geometry_i, f_i, t, x_{it}, g_n, Geometry_n, f_n, t+1, x_{n(t+1)})$. If the condition on the neighbor's time-step is not satisfied, the value of $x_{n(t+1)}$ is set to zero, i.e. no fire.

This dataset can be used utilized by machine learning algorithms to build a model for wildfire spread [7, 24, 64].

6.2 Crop Yield Mapping

The problem of crop yield mapping in agriculture [41] studies the crop yield of various agriculture fields using Normalized Difference Vegetation Index (NDVI), which can be used as a proxy for crop health, growth status, and yield. NDVI is calculated using the red and near-infrared spectral reflectance (SR) measurements captured by satellites which are available as rasters. To study crop yield, NDVI needs to be calculated for all pixels that overlap the agricultural fields under study for a period of time ranging over multiple years. The agricultural fields are available in vector format as a set of polygons, where each polygon represents one agricultural field. Since crop yield is affected by various environmental conditions, it makes it necessary to study various statistics about the crop yield both in-field and across all the fields under study. These statistics also need to be calculated across different windows of time to make sure whether the factors contributing towards the decrease in crop yield are local or global (such as drought).

This application starts by sorting the NDVI layers by time to logically form a three-dimensional cube where each pixel contains a time series of NDVI values in one year.

Then it calculates the standard deviation of each time series which represents the temporal variability in vegetation in that location. After that, it combines this result with the agricultural fields, represented as polygons, and computes the average and 90^{th} percentile value per field. This whole process is repeated for each year in the study and these values are then used to classify the agriculture fields according to their crop yield and stability over the years.

6.3 Areal Interpolation

Areal Interpolation is the problem of estimating a function in arbitrary areas, e.g., city boundaries, based on values in other non-aligned areas, e.g., census tracts. One application of this problem is to estimate the population of arbitrary regions using land-cover data [56]. The problem is that the US Census Bureau reports the population at the granularity of *census tracts* which are regions chosen by the Bureau to keep the privacy of the data. Areal interpolation transforms these counts from source polygons, i.e., tracts, to target polygons, e.g., ZIP Codes, with unknown counts. One accurate method [56] uses the National Land Cover Database (NLCD) [46] raster dataset as a reference to disaggregate the population counts into pixels and then aggregate them back into target polygons.

This application starts by calculating the histogram of NLCD values of each known region, e.g., census tract. This step estimates how much of each region is covered by each land type, e.g., road, urban area, and water. After that, it uses Poisson regression to estimate the contributing factor of each land type to the true population in these regions. This is called the disaggregation step since it breaks down the population into pixels. In

the next step, it processes the unknown regions, e.g., ZIP codes, with the NLCD dataset to compute the histogram of each known region. Finally, it uses the regression parameters to estimate the population of each known region. The most time consuming step in this process is to calculate the histogram of the (raster) NLCD dataset for each (vector) region.

Chapter 7

Conclusions

In this dissertation proposal, we propose a new system called *Raptor* that can bridge the gap between raster and vector data. It is an end-to-end system for efficiently processing raster and vector geospatial data concurrently.

In the first chapter, we introduce and motivate the problem of efficiently processing raster and vector geospatial data concurrently. We talk about the increase in the amount of spatial which has facilitated several research applications but has also led to the demand for systems that can efficiently process big spatial data.

In the second chapter, we presented an initial approach called *DARaptor* to parallelize the zonal statistics operation. This algorithm provided several key ideas for processing big vector and raster datasets in a distributed environment. First, this framework runs in two phases, a single-machine preprocessing step that computes a common data structure to be used in parallel, and defines the tasks that will be executed in parallel. The second phase runs in parallel and aims at reading and processing the big raster files efficiently. The

second key idea is introducing the RaptorInputFormat which is the first input format that combines raster plus vector data in one split. The RaptorInputFormat can define the units of work to be executed in parallel and provides an easy way to optimize and balance the load across machines. The RaptorInputFormat combined with the preprocessing step can also prune irrelevant parts of the raster file in order to speed up the parallel processing. Finally, we investigated several improvements to this method such as splitting the vector file into chunks and compressing the intermediate files between the preprocessing and distributed aggregation. The experiments show that the proposed algorithm can scale to large raster data whereas the baselines could not handle big vector or raster data.

However, this approach did not scale well for very large vector datasets. Therefore, in the third chapter, we proposed *Raptor Zonal Statistics*, a fully distributed system implemented in Hadoop that can be used to perform the zonal statistics operation for big raster and vector datasets. RZS runs in three steps: 1. First, RZS runs an intersection step that computes the intersection file which maps vector polygons to raster pixels. 2. Second, it runs the selection step that concurrently scans the intersection file and the raster file to find the join result. To process the two files in parallel, the chapter re-introduces and modifies the two components RaptorInputFormat and RaptorSplit which define the smallest unit of work for each parallel task. 3. Third, it runs an aggregate phase that computes the desired statistics for each polygon. The experiments with large-scale real data show that the proposed algorithm is up-to two orders of magnitude faster than the baselines including Rasdaman and Google Earth Engine (GEE). We also presented a cost model in this chapter which helped us in explaining the results of both RZS and the baseline technique.

In the fourth chapter, we propose a new raster-vector join algorithm *Raptor Join* which is modeled as a relational join operator in Spark that can be easily combined with other operators, while also offering the advantage of in-situ processing. It overcomes the limitations of the existing systems by combining raster-vector data in their native formats by using a novel index structure called Flash Index. It runs in three steps, namely, Flash Index creation, Flash Index optimization, and Flash Index processing. The Flash Index creation step computes a mapping between raster and vector in the form of pixel ranges. The Flash Index optimization step partitions and reorganizes this data structure across machines in such a way that each tile in the raster dataset is scanned by only one machine. The Flash Index processing step processes the partitioned pixel ranges to read the required pixel values from the raster dataset. We run extensive experiments for the system against Rasdaman, GeoTrellis, Google Earth Engine, Adaptive Cell Trie, GeoSpark, and Beast on large raster and vector datasets to show its scalability and performance gain. We also show that Raptor Join can be used to support real-world applications such as wildfire modeling, area interpolation, and crop yield mapping.

In the fifth chapter, we propose the distributed system *RDPro* which is used to add distributed raster processing capabilities to Raptor and can scale to big raster data. RDPro is implemented in Spark and uses a custom RDD, RDD[ITile] to represent and process raster data in a distributed environment. It uses RDD[ITile] to implement the operations required for raster analysis. Since the proposed system is implemented in Spark and uses an RDD to model the raster data, it allows the users an advantage to combine multiple operations and run a complex spatial query pipeline on their datasets. We compare the

proposed system to GeoTrellis, GDAL, and Sedona, and show its performance gain and scalability.

In the sixth and final chapter, we discuss three real-world applications that require to process the combination of raster and vector data and were solved using the proposed system *Raptor*.

In the future, we intend to make Raptor inter-operable with R and Python. R and Python are popular tools used for analysis by the geospatial community. Incorporating Raptor with them would allow more users to take advantage of the system. We also intend to extend Raptor to work with SparkSQL and implement the query optimizer for it. Another future work includes extending Raptor to work with machine learning algorithms that use both raster and vector data.

Bibliography

[1] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. Hadoop-GIS: A high performance spatial data warehousing system over mapreduce. *PVLDB*, 6(11):1009–1020, 2013.

[2] Zonal Statistics in ArcGIS. `http://desktop.arcgis.com/en/arcmap/10.3/tools/spatial-analyst-toolbox/h-how-zonal-statistics-works.htm`, 2017.

[3] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. The multidimensional database system RasDaMan. In *SIGMOD*, pages 575–577, Seattle, WA, June 1998.

[4] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. *ACM SIGMOD Record*, 22(2):237–246, 1993.

[5] Nieves R Brisaboa, Guillermo de Bernardo, Gilberto Gutiérrez, Miguel R Luaces, and José R Paramá. Efficiently querying vector and raster data. *The Computer Journal*, 60(9):1395–1413, 2017.

[6] Daniel Brown, Stephen Platt, John Bevington, Keiko Saito, Beverley Adams, Torwong Chenvidyakarn, Robin Spence, Ratana Chuenpagdee, and Amir Khan. Monitoring and evaluating post-disaster recovery using high-resolution satellite imagery–towards standardised indicators for post-disaster recovery. *Martin Centre: Cambridge, UK*, 2015.

[7] Tina Diao et al. Uncertainty aware wildfire management. In *AI for Social Good Workshop, AAAI Fall Symposium Series*, 2020.

[8] Chris Dickens, Vladimir Smakhtin, Matthew McCartney, Gordon O'Brien, and Lula Dahir. Defining and quantifying national-level targets, indicators and benchmarks for management of natural resources to achieve the sustainable development goals. *Sustainability*, 11(2):462, 2019.

[9] Jennie Duggan, Olga Papaemmanouil, Leilani Battle, and Michael Stonebraker. Skew-aware join optimization for array databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 123–135, 2015.

[10] Ahmed Eldawy et al. CG_Hadoop: computational geometry in MapReduce. In *SIGSPATIAL*, pages 284–293, Orlando, FL, November 2013.

[11] Ahmed Eldawy et al. SHAHED: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. In *31st IEEE International Conference on Data Engineering, ICDE 2015*, pages 1585–1596, 2015.

[12] Ahmed Eldawy, Vagelis Hristidis, Saheli Ghosh, Majid Saeedan, Akil Sevim, AB Siddique, Samriddhi Singla, Ganesh Sivaram, Tin Vu, and Yaming Zhang. Beast: Scalable exploratory analytics on spatio-temporal data. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 3796–3807, 2021.

[13] Ahmed Eldawy and Mohamed F. Mokbel. Pigeon: A Spatial MapReduce Language. In *ICDE*, pages 1242–1245, Chicago, IL, March 2014.

[14] Ahmed Eldawy and Mohamed F. Mokbel. SpatialHadoop: A MapReduce Framework for Spatial Data. In *ICDE*, pages 1352–1363, April 2015.

[15] 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.

[16] Ahmed Eldawy, Lyuye Niu, David Haynes, and Zhiba Su. Large scale analytics of vector+raster big spatial data. In *SIGSPATIAL*, pages 62:1–62:4, 2017.

[17] Eosdis annual metrics reports, 2020. `https://earthdata.nasa.gov/eosdis/system-performance/eosdis-annual-metrics-reports`.

[18] The ESA Earth Observation Payload Data Long Term Storage Activities, 2017. `https://www.cosmos.esa.int/documents/946106/991257/13_Pinna-Ferrante_ESALongTermStorageActivities.pdf`.

[19] Y. Fan, M. Clark, D. M. Lawrence, S. Swenson, L. E. Band, S. L. Brantley, P. D. Brooks, W. E. Dietrich, A. Flores, G. Grant, J. W. Kirchner, D. S. Mackay, J. J. McDonnell, P. C. D. Milly, P. L. Sullivan, C. Tague, H. Ajami, N. Chaney, A. Hartmann, P. Hazenberg, J. McNamara, J. Pelletier, J. Perket, E. Rouholahnejad-Freund, T. Wagener, X. Zeng, E. Beighley, J. Buzan, M Huang, B. Livneh, B. P. Mohanty, B. Nijssen, M. Safeeq, C. Shen, W. van Verseveld, J. Volk, and D Yamazaki. Hillslope Hydrology in Global Change Research and Earth System Modeling. 2019.

[20] GDAL/OGR contributors. *GDAL/OGR Geospatial Data Abstraction software Library*. Open Source Geospatial Foundation, 2022.

[21] GeoTrellis on Spark. `https://github.com/wri/geotrellis-zonal-stats/blob/master/src/main/scala/tutorial/ZonalStats.scala`, 2019.

[22] GeoTrellis on Spark. `https://github.com/wri/geotrellis-zonal-stats/blob/master/src/main/scala/tutorial/ZonalStats.scala`, 2019.

[23] Pierre Gernez, Stephanie CJ Palmer, Yoann Thomas, and Rodney Forster. remote sensing for aquaculture. *Frontiers in Marine Science*, 7:1258, 2021.

[24] Omid Ghorbanzadeh et al. Spatial prediction of wildfire susceptibility using field survey GPS data and machine learning approaches. *Fire*, 2(3):43, 2019.

[25] 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.

[26] Thomas W Gillespie, Jasmine Chu, Elizabeth Frankenberg, and Duncan Thomas. Assessment and prediction of natural hazards from satellite imagery. *Progress in Physical Geography*, 31(5):459–470, 2007.

[27] Noel Gorelick et al. Google earth engine: Planetary-scale geospatial analysis for everyone. *Remote sensing of Environment*, 202:18–27, 2017.

[28] Noel Gorelick, Matt Hancher, Mike Dixon, Simon Ilyushchenko, David Thau, and Rebecca Moore. Google earth engine: Planetary-scale geospatial analysis for everyone. *Remote sensing of Environment*, 202, 2017.

[29] David Haynes, Steven Manson, and Eric Shook. Terra Populus' Architecture for Integrated Big Gepspatial Services. *Transactions on GIS*, 2017.

[30] David Haynes, Suprio Ray, Steven M. Manson, and Ankit Soni. High Performance Analysis of Big Spatial Data. In *Big Data*, pages 1953–1957, Santa Clara, CA, November 2015.

[31] ISO 19111:2019: Geographic information - Referencing by coordinates. `https://www.iso.org/obp/ui/#iso:std:iso:19111:ed-3:v1:en`, 2019.

[32] G Darrel Jenerette, Sharon L Harlan, Anthony Brazel, Nancy Jones, Larissa Larsen, and William L Stefanov. Regional Relationships Between Surface Temperature, Vegetation, and Human Settlement in a Rapidly Urbanizing Ecosystem. *Landscape Ecology*, 22:353–365, 2007.

[33] G. Darrel Jenerette, Sharon L. Harlan, William L. Stefanov, and Chris A. Martin. Ecosystem Services and Urban Heat Riskscape Moderation: Water, Green Spaces, and Social Inequality in Phoenix, USA. *Ecological Applications*, 21:2637–2651, 2011.

[34] Maxwell B Joseph et al. Spatiotemporal prediction of wildfire size extremes with bayesian finite sample maxima. *Ecological Applications*, 29(6), 2019.

[35] Daniel Kachelriess, Martin Wegmann, Matthew Gollock, and Nathalie Pettorelli. The application of remote sensing for marine protected area management. *Ecological Indicators*, 36:169–177, 2014.

[36] Ameet Kini and Rob Emanuele. Geotrellis: Adding Geospatial Capabilities to Spark, 2014.

[37] Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Adaptive main-memory indexing for high-performance point-polygon joins. In *EDBT 2020*, pages 347–358. OpenProceedings.org, 2020.

[38] Nick Koudas and Kenneth C Sevcik. Size separation spatial join. In *SIGMOD*, pages 324–335, 1997.

[39] Ming-Ling Lo and Chinya V Ravishankar. Spatial joins using seeded trees. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 209–220, 1994.

[40] Ming-Ling Lo and Chinya V Ravishankar. Spatial hash-joins. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 247–258, 1996.

[41] Bernardo Maestrini and Bruno Basso. Predicting spatial patterns of within-field crop yield variability. *Field Crops Research*, 219, 2018.

[42] Map Algebra, 2022. `https://gisgeography.com/map-algebra-global-zonal-focal-local/`.

[43] Jeremy Mennis, Roland Viger, and C Dana Tomlin. Cubic map algebra functions for spatio-temporal analysis. *Cartography and Geographic Information Science*, 32(1):17–32, 2005.

[44] Priti Mishra and Margaret H Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.

[45] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. MD-HBase: Design and Implementation of an Elastic Data Infrastructure for Cloud-scale Location Services. *DAPD*, 31(2):289–319, 2013.

[46] NLCD dataset. https://www.mrlc.gov/data/type/land-cover, 2020.

[47] Jignesh M Patel and David J DeWitt. Partition based spatial-merge join. *ACM Sigmod Record*, 25(2):259–270, 1996.

[48] Nathalie Pettorelli. *Satellite remote sensing and the management of natural resources*. Oxford University Press, 2019.

[49] Donna J Peuquet. A hybrid structure for the storage and manipulation of very large spatial data sets. *Computer Vision, Graphics, and Image Processing*, 24(1):14–27, 1983.

[50] Kamleshan Pillay, Naeem Hoosen Agjee, and Srinivasan Pillay. Modelling changes in land cover patterns in mtunzini, south africa using satellite imagery. *Journal of the Indian Society of Remote Sensing*, 42(1):51–60, 2014.

[51] Gary Planthaber, Michael Stonebraker, and James Frew. Earthdb: scalable analysis of modis data using scidb. In *BIGSPATIAL*, pages 11–19, 2012.

[52] Postgis: Spatial and geographic objects for postgresql, 2020. `https://postgis.net`.

[53] PostGIS, 2022. `https://postgis.net/`.

[54] QGIS. `http://www.qgis.org/`, 2015.

[55] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw-Hill, 2000.

[56] Michael Reibel and Aditya Agrawal. Areal interpolation of population counts using pre-classified land cover data. *Population Research and Policy Review*, 26(5-6):619–633, 2007.

[57] Hossein Saadat, Jan Adamowski, Robert Bonnell, Forood Sharifi, Mohammad Namdar, and Sasan Ale-Ebrahim. Land use and land cover classification over a large area in iran based on single date analysis of satellite imagery. *ISPRS Journal of Photogrammetry and Remote Sensing*, 66(5):608–619, 2011.

[58] Elia Scudiero et al. Regional scale soil salinity evaluation using landsat 7, western san joaquin valley, california, usa. *Geoderma Regional*, 2-3:82 – 90, 2014.

[59] Elia Scudiero, Todd H Skaggs, and Dennis L Corwin. Regional scale soil salinity evaluation using landsat 7, western san joaquin valley, california, usa. *Geoderma Regional*, 2:82–90, 2014.

[60] Shashi Shekhar and Sanjay Chawla. *Spatial Databases: A Tour*. Prentice Hall Upper Saddle River, NJ, 2003.

[61] Andrii Shelestov, Mykola Lavreniuk, Nataliia Kussul, Alexei Novikov, and Sergii Skakun. Exploring google earth engine platform for big data processing: Classification of multi-temporal satellite imagery for crop mapping. *frontiers in Earth Science*, 5:17, 2017.

[62] Fernando Silva-Coira et al. Efficient processing of raster and vector data. *Plos one*, 15(1):e0226943, 2020.

[63] Bogdan Simion, Angela Demke Brown, and Ryan Johnson. Skew-resistant Parallel In-memory Spatial Join. In *SSDBM*, pages 6:1–6:12, Aalborg, Denmark, July 2014.

[64] Samriddhi Singla, Tina Diao, Ayan Mukhopadhyay, Ahmed Eldawy, Ross Shachter, and Mykel Kochenderfer. WildfireDB: A Spatio-Temporal Dataset Combining Wildfire Occurrence with Relevant Covariates. 2020.

[65] Samriddhi Singla and Ahmed Eldawy. Distributed Zonal Statistics of Big Raster and Vector Data. In *SIGSPATIAL*, 2018.

[66] Samriddhi Singla and Ahmed Eldawy. Raptor Zonal Statistics : Fully Distributed Zonal Statistics of Big Raster + Vector Data. In *Proceedings of the 2020 IEEE International Conference on Big Data (IEEE BigData 2020)*. IEEE, December 2020.

[67] Samriddhi Singla, Ahmed Eldawy, Rami Alghamdi, and Mohamed F. Mokbel. Raptor: Large Scale Analysis of Big Raster and Vector Data. *PVLDB*, 12(12):1950 – 1953, 2019.

[68] Samriddhi Singla, Ahmed Eldawy, Tina Diao, Ayan Mukhopadhyay, and Elia Scudiero. Experimental Study of Big Raster and Vector Database Systems. In *ICDE*, page To Appear, April 2021.

[69] Samriddhi Singla, Ahmed Eldawy, Tina Diao, Ayan Mukhopadhyay, and Elia Scudiero. The raptor join operator for processing big raster+ vector data. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*, pages 324–335, 2021.

[70] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.

[71] Planet Team. Planet application program interface: In space for life on earth, 2018–.

[72] UCR-Star: The UCR Spatio-temporal Active Repository. `https://star.cs.ucr.edu/`.

[73] UCS Satellite Database, 2022. `https://www.ucsusa.org/resources/satellite-database`.

[74] Tin Vu and Ahmed Eldawy. R*-grove: Balanced spatial partitioning for large-scale datasets. *Frontiers in Big Data*, 3:28, 2020.

[75] Yafei Wang et al. Parallel scanline algorithm for rapid rasterization of vector geographic data. *Computers & geosciences*, 59:31–40, 2013.

[76] Randall T. Whitman, Michael B. Park, Sarah A. Ambrose, and Erik G. Hoel. Spatial Indexing and Analytics on Hadoop. In *SIGSPATIAL*, pages 73–82, Dallas, TX, November 2014.

[77] European XFEL: Data Handling, 2017. `http://www.xfel.eu/research/data_handling/`.

[78] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient In-Memory Spatial Analytics. In *SIGMOD*, June 2016.

[79] Chenghai Yang, James H Everitt, Qian Du, Bin Luo, and Jocelyn Chanussot. Using high-resolution airborne and satellite imagery to assess crop growth and yield variability for precision agriculture. *Proceedings of the IEEE*, 101(3):582–592, 2012.

[80] Jia Yu, Mohamed Sarwat, and Jinxuan Wu. GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data. In *SIGSPATIAL*, Seattle, WA, November 2015.

[81] 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*. USENIX Association, 2012.

[82] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

[83] Ramon Antonio Rodriges Zalipynis. Chronosdb: distributed, file based, geospatial array dbms. *PVLDB*, 11(10):1247–1261, 2018.

[84] Ramon Antonio Rodriges Zalipynis. Chronosdb: distributed, file based, geospatial array dbms. *Proceedings of the VLDB Endowment*, 11(10):1247–1261, 2018.

[85] Jianting Zhang and Dali Wang. High-Performance Zonal Histogramming on Large-Scale Geospatial Rasters Using GPUs and GPU-Accelerated Clusters. In *IPDPS Workshops*, pages 993–1000, Phoenix, AZ, May 2014.

[86] Jianting Zhang, Simin You, and Le Gruenwald. Efficient Parallel Zonal Statistics on Large-Scale Global Biodiversity Data on GPUs. In *BIGSPATIAL*, pages 35–44, Bellevue, WA, November 2015.

[87] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *CLUSTER*, pages 1–8, 2009.

[88] Yaming Zhang and Ahmed Eldawy. Evaluating computational geometry libraries for big spatial data exploration. In *ACM SIGMOD*, pages 1–6, 2020.

[89] Gang Zhao, Brett A Bryan, Darran King, Xiaodong Song, and Qiang Yu. Parallelization and optimization of spatial analysis for large scale environmental model data assembly. *Computers and electronics in agriculture*, 89:94–99, 2012.

[90] Weijie Zhao, Florin Rusu, Bin Dong, and Kesheng Wu. Similarity join over array data. In *Proceedings of the 2016 International Conference on Management of Data*, 2016.