RDPro : Distributed Processing of Big Raster Data

Anonymous authors
Affiliation
City, State, Country
email

ABSTRACT

Advancements in remote sensing technology have led to a tremendous increase in the amount of spatial data. Satellite and aerial imagery are examples of spatial data which are now available at pixel resolutions as high as 1 cm. This data is available in the raster format, which is an important component for research in various fields such as disaster response, agricultural monitoring, and marine biology. However, there are challenges in processing big raster data due to the intertwined dependency between pieces of the data that are distributed among multiple files and machines. This dependency is challenging to untangle when the pieces do not align in location, resolution, and coordinate reference system. This paper proposes a novel distributed system, RDPro, implemented in Spark that can efficiently process and analyze big raster data. RDPro adds a new logical and physical data model for raster data that is friendly with distributed query processing engines and efficiently captures the data dependency. Then, it adds a distributed raster data loader and writer that work with distributed file systems. RDPro also provides efficient implementations for the core raster operations based on Spark. Users can build a complex pipeline along with traditional Spark operations to build scalable data science applications. We compare RDPro to existing systems for big raster data and show that it achieves up to two orders of magnitude performance gain while being perfectly able to scale to big raster data.

ACM Reference Format:
Anonymous authors. 2025. RDPro : Distributed Processing of Big Raster Data. In *SIGMOD '25: ACM International Conference on Management of Data, June, 2025, Berlin, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

There is an ever-increasing amount of geospatial data, which has been made available due to the advancements in remote sensing technology. There are currently more than 1000 active satellites [42] 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 taken from drones, air balloons, or airplanes which can reach a resolution of up to 1 cm per pixel. Both public organizations such as NASA, USGS, and European Space Agency (ESA) and private organizations 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. Moreover, there are numerous computer-generated raster products, such as Global Land Cover (GLC), Normalized Difference Vegetation Index (NDVI), and Cropland Data Layer (CDL), which are derived from these raw satellite images based on defined functions over pixel values.

Satellite and aerial imagery is an example of raster data which is represented using multi-dimensional arrays of values. Raster data is an important component of research in fields such as disaster response and monitoring [8, 20], management of energy and natural resources [13, 31, 32], agricultural monitoring [19, 36, 37, 45], and marine biology [18, 25]. The earthquake that occurred in Turkey in February 2023 and the wars that happen in several parts also aroused consideration of using satellite images to help assess the damage caused by such disasters. Satellite images further have a high potential to advance machine learning datasets, such as providing more high resolution terrain images in machine-learning-acceptable format that can benefit object detection and segmentation models [1, 11]. 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.

While there have been many systems [2, 14, 43, 44, 47] for big spatial *vector* data management, big *raster* data has unique challenges [3, 5, 15, 26, 49]. The data is typically collected by different satellites in a set of *scenes*. These scenes are not always perfectly aligned. For example, they can have different coordinate reference systems (CRS) depending on their geographical locations, they might partially overlap, and their pixels are usually not perfectly aligned. In addition, when combining datasets from different satellites, the resolutions do not necessarily match. However, real data science applications need to combine all these heterogeneous raster datasets in one query pipeline which is extremely challenging.

There exist systems for processing raster data such as GeoTrellis [17], Rasdaman [6], Google Earth Engine [22] and others [16, 33, 38, 46, 49]. However, these systems suffer from one or more limitations as further detailed below:

(1) **Single Machine:** There are various raster-based systems [16, 33] that run on a single machine. These systems are not efficient when working with large raster datasets. They either run out of memory or they may take days to perform a query. In comparison, distributed systems are able to scale to larger datasets and can perform the same query in significantly less amount of time.
(2) **Limited Functionality:** The data model used by some raster-based systems limits the type of operations that can be performed on the raster data. Most importantly, some systems [10, 12, 29], cannot perform operations such as reprojection or reshaping which

are necessary when working with datasets from multiple sources and in different coordinate reference systems.
(3) **Heavy Ingestion:** Many raster-based systems require an expensive data ingestion step to load the data in its own internal model to partition and index the data [6, 22, 38]. These systems are optimized for repetitive queries on the loaded datasets while many modern applications focus on one-time ad-hoc queries.
(4) **Constrained Query Runtime:** Some systems [16, 49] can only perform basic queries that consist of one primitive operation. To run a complex query pipeline, the user has to run one operation at a time. Each operation has to read the data from disk, perform the computation, and write the results back to disk. A more efficient system should be able to optimize complex query pipelines.
(5) **Expensive Memory Usage:** Some systems [17, 46] require reading the whole data in memory before processing, making them unable to scale to large datasets. This limitation arises because the size of large datasets often exceeds the available memory capacity.

This paper proposes a novel distributed system, *RDPro*, implemented in Spark that can efficiently perform analysis on big raster data. It overcomes the limitations of existing systems as follows: (1) RDPro is a distributed system that introduces a novel distributed data processing model based on a new concept of *Maplets*. As further detailed in the paper, a Maplet is a small unit of the raster that has location information to ensure correct and efficient query processing. (2) RDPro implements map algebra operations which include local, focal, zonal, and global operations. These operations provide an exhaustive list of operations that users may need to analyze raster data. (3) RDPro avoids data ingestion step and can directly process raster files in common formats including GeoTIFF. *RDPro* only reads raster metadata (a few KBs in size) to prepare and submit the job. (4) RDPro is implemented on Spark and extends the RDD model to support raster data which gives the users the advantage to combine multiple operations and run a complex spatial query pipeline on their datasets. RDPro utilizes and extends the optimization techniques in Spark to run raster operations efficiently. (5) RDPro has built-in components to partition data loading, computation, and writing into small units that can fit in memory which keeps its memory usage under control. We run an extensive experimental evaluation of RDPro compared to the state-of-the-art Spark-based raster processing systems, GeoTrellis [17] and Sedona [46]. RDPro has up-to two orders of magnitude performance gain over baselines while being perfectly able to scale to big raster data.

The rest of this paper is organized as follows: Section 2 covers the related work. Section 3 describes the logical raster data model and the operations required to analyze raster data. Section 4 provides an overview of the proposed system. Section 5 runs experimental evaluations of the proposed system. Section 6 concludes the paper.

2 RELATED WORK

This section covers the relevant work in the area of raster data processing. First, we cover the raster data processing tools that are generally used by the GIS community. After that, we give an overview of the distributed systems that can work with raster data.

2.1 Single Machine

The GIS community generally makes use of tools such as QGIS [34], ArcGIS [35], PostGIS [33], GDAL [16] and raster analysis packages in R [23] and Python [21] for their application needs. However, all of these systems are limited to a single machine which makes them inefficient when working with large raster datasets. When compared to distributed systems, these single-machine systems either fail to process large datasets or take significantly more time to perform queries on large raster datasets. The proposed system *RDPro* is implemented in Spark, which gives it an opportunity to be more efficient and scalable for processing large raster datasets than single-machine systems.

2.2 Distributed Raster Systems

Distributed systems that can process raster data include SciDB [9, 38, 39]„ Rasdaman [5, 6], GeoTrellis [17], Apache Sedona [46, 48], Google Earth Engine [22], and ChronosDB [49]. Some systems use MapReduce such as SciSpark [29], SciHadoop [10], MrGeo [26], and ClimateSpark [12, 24]. The next part gives an overview of these distributed systems and framework, especially the fundamental operations, such as reprojection and file loading process.

Some distributed work [10, 12, 29, 30, 38, 41] use the array data model to support big scientific data and ignore the geographical component, a.k.a., coordinate reference system (CRS), of the raster data. They lack the reprojection operation which is particularly critical when dealing with datasets with mixed projections. It is common to store a big raster dataset as a set of files, called scenes, with mixed projections, e.g., Landsat8. Thus, ignoring the geographical component renders these system unusable for big raster data.

Other works [6, 22] support geographical data, but are limited by an expensive data ingestion step. These systems implement their own data model and require an ingestion phase where they read the data and re-structure it according to their data model. This process is suitable for repetitive queries on the same dataset where the cost of the ingestion step is amortized over the number of queries. However, modern data science applications often require one-time ad-hoc queries which makes the data ingestion step a bottleneck. ChoronosDB works directly on files but it requires the user to manually and carefully place these files in a specific way for operations to work correctly.

Some systems [46, 49] support limited raster operations and are not suitable for most data science applications. Furthermore, they suffer from memory issues [17, 46] as they require loading large data blocks into memory before processing. This limits their scalability as shown in this paper.

The proposed system, *RDPro*, falls in the category of distributed raster systems and it fully supports geographical operations. It works directly on raster files as they are obtained from satellite data repositories with no need for any data conversion or ingestion. RDPro provides a rich processing runtime that includes the major raster operations including reprojection which is necessary to combine multiple datasets together. RDPro allows users to express an entire query pipeline which enables further optimization while breaking the data down into small units that keep its memory usage under control and allows it to support terabytes of satellite data.

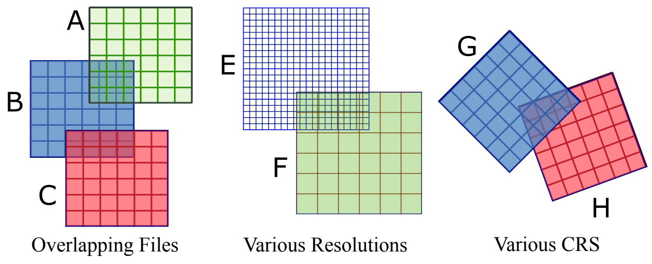


Figure 1: Challenges with Real-world Raster Data

3 PROBLEM FORMULATION

This section formulates the problem of raster data processing solved by the proposed system *RDPro*. First, we describe the problem setting and challenges of handling real-word raster data. Second, we define the logical data model for raster data. Finally, we define the operations for raster analysis using the defined data model.

3.1 Problem Setting: Real Raster Data

Real-world raster data generally comes from various sources including raw satellite data, rectified data, aerial imagery, and machine-generated raster products such as land cover maps. This data has high variety due to the way it is collected and maintained. For example, the two biggest sources of public satellite data are NASA and the European Space Agency (ESA) who use different technology and sensors to collect their data which results in varying resolutions of the data. Raster data store values in a two-dimensional array of *pixels* each storing an array of values, named *bands*. For example, pixels in visible images have three bands, red, green, and blue, but scientific data can have a dozen of bands representing several wave lengths, e.g., infra-red. Another essential feature of raster data is that each array of pixels has a geographical location that associates pixels to locations on the earth surface. This location information includes a coordinate reference system (CRS) that sometimes change depending on the location of earth that it covers, e.g., UTM zones.

These raster datasets are frequently used in data science to study land surface phenology, such as monitoring land cover or climate change. One research demand requires the combination of multiple data sources, e.g., Sentinel-2 and Landsat-8, for better performance and coverage [7, 27]. However, the harmonization methodology becomes increasingly complex as more raster data sources are used. We summarize these challenges in the following few points which could all occur at the same time.

(1) **Multiple overlapping files:**. In general, a single raster dataset can be made available as a large set of files, sometimes thousands of files. For example, Figure 1 shows files A, B, and C which could belong to one dataset. Sometimes, these files slightly overlap which results in conflicting data that need to be resolved. Thus, one cannot simply treat the dataset as a single large array. Yet, these small arrays need to be aligned correctly to process overlapping or boundary regions.
(2) **Various resolutions:** Raster products can have a wide range of resolutions depending on the sensor used to collect the data, e.g., E & F in Figure 1. For example, MODIS data has 1 km resolution while some Sentinel products have 10 m resolution. This means that a single pixel of MODIS can overlap 10,000 Sentinel pixels. Users might want to harmonize these datasets.
(3) **Various CRS:** A CRS defines a geographical projection that maps a region of the earth surface to a two-dimensional space that corresponds to a raster dataset. Various datasets could have different CRSs such as G&H in Figure 1. Even one raster product could have multiple CRS depending on the geographical location, e.g., UTM zones in Landsat data. To combine these datasets, users need to easily reproject datasets to a common reference space before processing them.
(4) **Large scales size:** Satellite images can be collected daily, with up-to several terabytes of data on a single day with high-resolution.

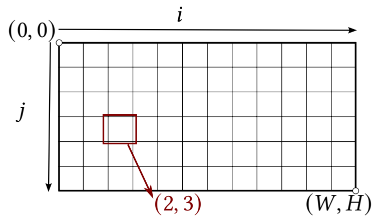


Figure 2: Raster Data Model

3.2 Raster Data Model

This part describes a logical data model that represents a single raster file as illustrated in Figure 2.

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 2. 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 $p = (i, j)$ 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). Some pixels might be empty in which the measure value is null.

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's surface.

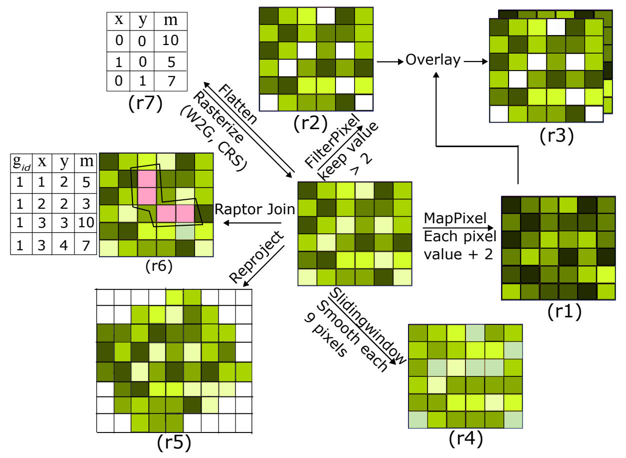


Figure 3: Raster Operations

Definition 6 (World Space). *The world space represents a rectangular space on the Earth's 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 maps world coordinates map to earth's surface.*

Definition 7 (Grid-to-World, $\mathcal{G2W}$). *$\mathcal{G2W}$ is a 2D affine transformation that transforms a point from the grid space to the world space. The inverse of this matrix is called world-to-grid, $\mathcal{W2G} = \mathcal{G2W}^{-1}$ and can be used to map locations from world space back to grid space.*

Definition 8 (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 $\mathcal{G2W}$ transformation, and a CRS defined by a unique spatial reference identifier (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 occupied grid space using the associated $\mathcal{G2W}$. 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*.

3.3 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 [28]: (1) local, (2) focal, (3) zonal, and (4) global. Below, we describe the main set of operations that we support based on the above raster data model.

(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 $\mathcal{G2W}$. They can only differ in the number of bands b_1 and b_2. The user-defined function f maps a measure value from R_1 to R_2, i.e., $f : \mathbb{R}^{b_1} \to \mathbb{R}^{b_2}$. The MapPixels operation applies the user-defined function f to each pixel measurement in R_1 and the output of the function defines the pixel value in the output raster R_2. This is a local operation and can be used , for example, to add a constant value to each pixel in the raster, as illustrated in Figure 3 (r1). It is defined as:

$$MapPixels(R_1, f) \to R_2$$
$$M_2(i, j) = f(M_1(i, j)) \forall 0 \le i < W, 0 \le j < H$$

(2) *FilterPixels*: This operation takes as input a raster dataset R_1 and a function f, and outputs the raster dataset R_2 with pixel values after filtering. R_2 retains only the pixels that pass the user-defined filter function $f : \mathbb{R}^{b_1} \to \{true, false\}$ and keeps other pixels as empty values. For example, in Figure 3 (r2) filters out the pixels with a value smaller than 2 and keeps these pixels as white representing empty values. It is defined as:

$$FilterPixels(R_1, f) \to R_2$$
$$M_2(i, j) = \begin{cases} M_1(i, j), & f(M_1(i, j)) = true \\ null, & \text{otherwise} \end{cases} \forall 0 \le i < W, 0 \le j < H$$

(3) *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 $\mathcal{G2W}$. The inputs might have different number of bands, b_1 and b_2 and the output has $b_3 = b_1 + b_2$ bands. 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. As shown in the Figure 3 (r3), this operation takes two separate rasters as input, and the output is one rater with multiple bands displayed as two layers. This operation is defined as follows:

$$Overlay(R_1, R_2) \to R_3$$
$$M_3(i, j) = M_1(i, j) || M_2(i, j) \forall 0 \le i < W \land 0 \le j < H$$

and $||$ is the concatenation operator for two arrays. Overlay can be called repetitively to stack more than two raster datasets.

(4) *Reproject*: This operation takes as input a raster dataset R_1, target CRS CRS_2, and target raster size W_2 and H_2. It reprojects R_1 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 calculated from the *set of nearby pixels in R_1 in world space*. The definition of nearby pixels and how their values are combined can be defined in various ways. The simplest is nearest neighbor which picks the value of the nearest neighbor pixel in the source. Other forms of interpolation, e.g., bilinear and bicubic, can also be used depending on the application. This is an example of a focal operation and is crucial to integrate multiple datasets of mismatching CRS. Figure 3 (r5) provides a reprojected raster with the nearest neighbor method. Notice that pixel size is also affected, and pixels outside of the raster area are marked as empty values. Considering nearest neighbor method, reproject is defined as:

$$Reproject(R_1, CRS_2, W_2, H_2) \to R_2$$
$$M_2(i_2, j_2) = M_1(i_1, j_1) : dist(i_1, j_2, i_2, j_2)$$
$$(i_1, j_1) = argmin_{i \in [0, W_1[, j \in [0, H_1[} dist(i, j, i_2, j_2)$$

where $dist(i_1, j_1, i_2, j_2)$ is the geodetic distance between the locations of pixels (i_1, j_1) and (i_2, j_2) on R_1 and R_2, respectively. When $CRS_1 = CRS_2$ this method is called regrid or resample. This special case is used to change the resolution of a raster dataset without affecting its CRS.

(5) *SlidingWindow*: This operation takes as input a raster dataset R_1, a window size w, and a function f. It outputs 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 $-w \le d_i, d_j \le w$ using the function f. A special case of this operation is called *convolution* which is a focal operation that applies a linear combination over pixels in the windows and is used for smoothing, sharpening, and edge detection. Figure 3 (r4) gives a smoothing function example that each pixel in the output rater is calculated from averaging pixel values surrounding it in the input raster. It is defined as:

$$f : \mathbb{R}^{b_1 \cdot (2w+1)^2} \to \mathbb{R}^{b_2} \;\&\; CalcWindow(R_1, f, w) \to R_2$$

where

$$M_2(i, j) = f\left(\bigcup_{-w \le \delta i, \delta j \le +w} M_1(i + \delta i, j + \delta j)\right)$$

(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 where represents the pixel value m at position (i, j) in grid coordinates for each unique geometry. Figure 3 (r6) gives an example of such calculation. It is defined as:

$$RaptorJoin(R, V) \to \{(g_{id}, i, j, m) : \theta((i, j), g_{id})\}$$

The predicate θ is formally defined in [4]. A special case when only one polygon is provided is called *clipping*.

(7) *FlattenValues*: This operation extracts all the measure values from the given input raster and produces a set that contains all these values. This method is a global operation that can be used, in combination with an aggregate function, to compute aggregates such as min, max, average, or a histogram. Figure 3 (r7) illustrates an example where each row represents a pixel location and its measure value.

$$FlattenValues(R) \to \{M(i, j) : \forall 0 \le i < W \wedge 0 \le j < H\}$$

(8) *Rasterize*: This operation takes as input as a set of pixels in the form $P = \{(i, j, m)\}$, the $\mathcal{G}2\mathcal{W}$ transformation, and CRS, and it outputs a raster dataset that represents all these pixels, as introduced in Figure 3 (r7).

$$Rasterize(P = \{(i, j, m)\}, \mathcal{G}2\mathcal{W}, CRS) \to R$$

Where

$$W = Max_{p \in P}(p.i) + 1, H = Max_{p \in P}(p.j) + 1$$

$$M(i, j) = \begin{cases} p.m, & \exists p \in P, p.i = i, p.j = j \\ null, & \text{otherwise} \end{cases}$$

4 RDPRO DESIGN

This section describes the proposed system **RDPro** (Raster Distributed Processor). RDPro is designed with three primary objectives: (1) **Distributed Processing:** *RDPro* enables distributed reading, writing, and processing of raster data, optimizing for parallel computing environments. (2) **Efficiency:** *RDPro* is capable of handling high-resolution and large-scale raster data with enhanced processing speed and reduced memory usage. (3) **Comprehensiveness:** *RDPro* supports a broad spectrum of raster analysis operations, facilitating the construction of complex spatial query pipelines.

Most existing distributed raster processing systems are limited by two assumptions. First, that a raster dataset is represented as a collection of files that are small enough to fit in memory. Second, that individual files can be processed independently. Such systems typically load each file into memory on a separate machine for embarrassingly parallel processing. However, this approach has two significant drawbacks. (1) If a big raster product is stored as a single file, it will be challenging to load into memory or parallelize. (2) If a large raster product is divided into smaller files, these systems struggle to treat them as a cohesive raster dataset, especially for focal or zonal operations involving pixels from separate files.

To overcome these challenges, *RDPro* introduces a fully distributed raster engine based on Apache Spark. RDPro's innovative approach introduces the concepts of *MapLocator* and *Maplet* to load, process, and write massive raster datasets efficiently. A *MapLocator* stores curcial geographic information about a raster dataset's location on Earth, including, location, resolution, orientation, and projection. A *Maplet* pairs a small raster piece (tile) with its corresponding MapLocator, providing comprehensive information for data processing. This lightweight and self-contained nature of Maplets enables their free movement between machins during distributed processing, ensuring accurate and efficient output.

RDPro utilizes Maplets in three main components: 1) A parallel loader that can handle both large raster files and collections of smaller files. It divides files into Maplets, preparing them for distributed processing. 2) A parallel query processor that understands the relationships between different data segments. It efficiently processes local, focal, zonal, and global operations. The MapLocator in each Maplet allows RDPro to handle files with differing resolutions or projections. 3) A parallel raster writer that can write output large raster files in a distributed file system.

Figure 4 illustrates RDPro's architecture, featuring four main parts: (1) **RDD[Maplet]** : RDPro extends Spark's RDD programming interface of Spark to implement a *physical* data model based on *Maplets*. (2) **Data Loading** : RDPro introduces a specialized data loader for distributed raster files, overcoming the limitations of Spark's default text file loader. (3) **Data Writing**: RDPro provides a parallel writer for large raster files, enhancing Spark's predominant writing capabilities. (4) **Raster Query Processing**: RDPro enriches Spark with comprehensive RDD operations for entire raster datasets, supporting complex query pipelines based on the operations defined in section 3. RDPro also introduces low-level optimizations for partitioning Maplets and reducing memory usage, which are elaborated throughout the paper.

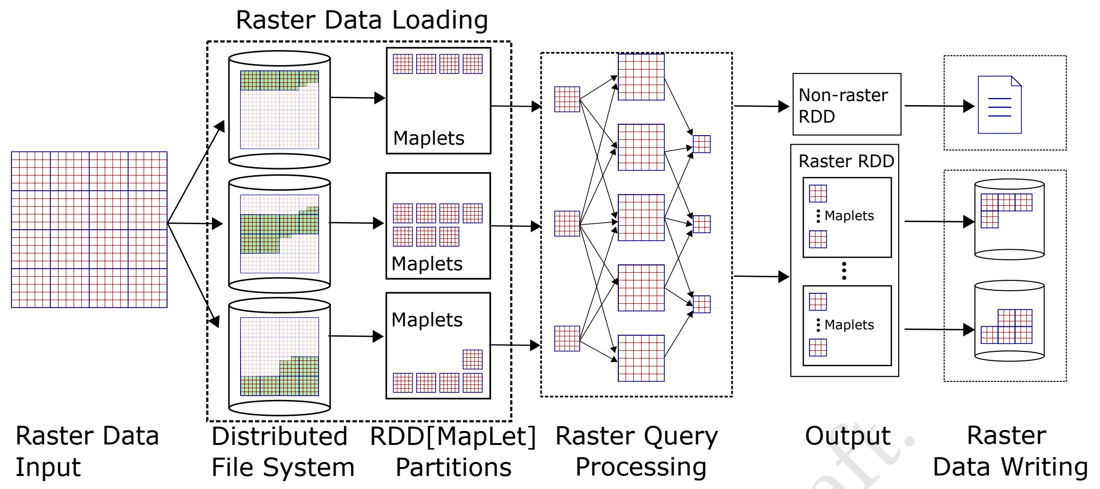


Figure 4: RDPro Architecture

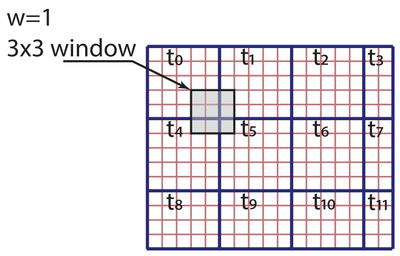


Figure 5: A raster dataset with width $W = 17$ and height $H = 14$ partitioned into a grid of 5×5 pixels. A window operation will require accessing up-to four tiles at a time.

4.1 RDPro Physical Data Model (RDD[Maplet])

RDPro's primary goal is to efficiently process raster data in a distributed environment. This involves breaking down a large raster dataset into smaller parts that can be distributed across machines as resilient distributed datasets (RDDs). section 3 showed that a raster dataset is logically a vast array of values, which becomes impractical to load entirely into memory for large rasters with trillions of pixels. Moreover, dividing it into subarrays risks losing spatial continuity among neighboring pixels. Addtionally, when a raster dataset is split among files, the system must be able to reassemble these arrays into a coherent dataset. RDPro's *Maplet* concept addresses these challenges by representing a subset of a raster dataset, complete with all the necessary location information. This enables RDPro to partition large datasets into smaller, manageable units without compromising their spatial coherence. Since Maplets are self-contained, they can be seamlessly combined, even if originating from files with different coordinate reference systems (CRS). This capability is a notable improvement over many existing distributed raster processing systems. Below, we elaborate on the earlier raster definitions to introduce the concept of Maplets.

Definition 9 (Raster Tile Grid, G_T). *Given a raster dataset R, the raster tile grid splits the raster grid space $[0, W[\times[0, H[$ into disjoint cells, called tiles, each with (integer) width tw and height th. The number of columns and rows in G_T is $c = \lceil W/tw \rceil$ and $r = \lceil H/th \rceil$, respectively. The total number of tiles is $|G_T| = c \cdot r$.*

Definition 10 (Raster Tile). *Given a raster dataset R and tile grid G_T, a tile consists of all pixels in the raster dataset R that overlap with one cell in the tile grid. This means that each tile contains $tw \times th$ pixels, except for the last tile in each row or column which might contain fewer pixels. Typically, each tile contains about tens of thousands of pixels. Each tile is identified by a sequence identifier $0 \le t_{id} < c \cdot r$.*

Figure 5 gives an example of a raster dataset with width $W = 17$ and height $H = 14$. This layer is split using a tile grid with 5×5 pixels for each tile, i.e., $tw = th = 5$. In this example, the number of columns $c = \lceil W/tw \rceil = \lceil 17/5 \rceil = 4$. Similarly, the number of tile rows $r = 3$. Finally, there is a total of $c \cdot r = 12$ tiles numbered from 0 to 11 as shown in the figure.

The notion of tiles is widely adopted in raster file formats and by existing systems. However, a tile is always tied to a parent raster which limits its use in distributed systems and across multiple rasters. Thus, we introduce the *MapLocator* which adds location information that allows a Maplet to be independent.

Definition 11 (Raster MapLocator (RM)). *Each raster is associated with auxiliary information called MapLocator, which is a lightweight record that consists of the following information: (1) Raster grid size, W and H. (8 bytes) (2) The raster grid-to-world, $\mathcal{G2W}$, transformation. (48 bytes) (3) The SRID of the raster CRS. (4 bytes) (4) Tile width and height, tw and th. (8 bytes)*

Definition 12 (Raster Maplet). *A Maplet consists of a MapLocator, a tile ID (t_{id}), and measure values for this tile.*

Each Maplet is stored as one block in memory and is treated by Spark as one record in the RDD. Notice that the MapLocator is the same for the entire raster but it becomes specific to each tile when tied to a specific tile ID. At the same time, this information is enough to independently process each Maplet without losing any information. As shown later in the paper, a set of Maplets can be processed completely in parallel as one coherent dataset even if they belong to different rasters which makes it a very powerful tool for distributed raster processing.

The measure values in a Maplet are stored in a three-dimensional array of size $tw \times th \times b$, where b is the number of bands in the raster. To further optimize memory usage and reduce network cost when moving Maplets between machines, the values array automatically switches between compressed and decompressed forms to balance memory overhead and computation efficiency.

A very important feature in RDPro is that one raster dataset can contain a set of Maplets with different MapLocators since each Maplet is an independent unit. This addresses a major limitation of many existing systems as it allows users to treat a datasets with tens of thousands of files as one coherent raster product.

To summarize, RDD[Maplet] (or RasterRDD) is a distributed collection of fault-tolerant tiles where each tile consists of MapLocator (RM), tile id (t_{id}), and the array of measure values. For the rest of this paper, we use the terms *tile* and *Maplet* interchangeably.

4.2 Raster Data Loading

The first step of a Spark RDD program is to create the first RDD. In RDPro, the first RasterRDD is normally created by loading raster files from disk. RDPro follows an ad-hoc approach that can directly read and process raster files without any pre-loading to overcome the limitation of other systems [6, 22, 38]. The main challenge in this step is to partition large datasets into bite-size pieces that can be easily parallelized over hundreds of machines. Some systems [46] load each file as a single record that cannot be parallelized and might cause an out of memory (OOM) error. RDPro is able to break down big files into fixed-size records that fit in memory and can be processed independently as shown below.

RDPro introduces the raster data loading component as a new RDD, called RasterFileRDD that implements the RDD[Maplet] interface. The entire raster data loading process encompasses two key steps: *splitting* and *reading* as detailed below.

Step 1: Splitting: This step runs on the main node before the Spark job starts. It produces a set of splits that define the file ranges that are assigned to each machine. Given an input path for the raster data, this step first lists all raster files in the source directory, e.g., all GeoTIFF files. Then, it splits each file according to the existing physical partitioning of the file. In other words, it creates one split for each 128 MB block of the distributed file system. In HDFS, the contents of directories and the sizes of the files are all stored in the main memory of the name node. This allows this step to run very efficiently since it only accesses main-memory data that is already stored in the master node. In summary, the initial RDD contains one partition for each HDFS block of the input raster data. As shown in Figure 4, this data-agnostic partitioning will end up splitting tiles in the raster file but this is handled in the next step.

Step 2: Reading: In this step, a worker node takes one split and reads all Maplets in that split. First, regardless of which part of the file is being read, the worker node reads the header of the raster file that the assigned split points to. The header is only a few kilobytes in size and its location in the file depends on the file format but it is typically at the beginning. Notice that this means that reading the header might entail remote access of that part on another machine. However, given its small size, the overhead is minimal. The worker node extracts two pieces of information from the header, the MapLocator-related information and the position of each tile in the file. Next, the worker node reads all tiles that have starting positions in the range assigned to it by the split. Notice that this decision can be done independently by each node which makes it more efficient and compliant with Spark methodology that keeps each worker independent. Furthermore, it ensures that each tile is read exactly once since the starting position can only fall in one split. Finally, as shown in Figure 4 since the last tile in a split might span to the next split, reading that last tile might also entail remote reading but given the small tile size, this overhead is minimal. To create each Maplet, the reader combines the raster MapLocator (RM), the tile ID, and the array of measure values.

Notice that to keep in line with Spark's lazy evaluation model, the reading step is only executed when raster data needs to be processed. This gives *RDPro* the advantage of ad-hoc processing. Thus, as tiles are loaded, they immediately get processed through the query pipeline, and Spark will not have to keep all tiles in memory at any point in time. The data loading component of *RDPro* also saves it from having an expensive ingestion and restructuring phase. Furthermore, to reduce the memory overhead, the measure values are initially loaded in a compressed form and are only decompressed when their measure values are accessed.

4.3 Raster Data Output

This section discusses the advanced raster data writing component of *RDPro*. Figure 4 shows that the output of a raster operation can be a raster RDD. *RDPro* implements its own distributed raster data writer which allows it to write big raster datasets to the distributed file system. This innovative writer takes a raster RDD and writes it to the distributed file system in standard raster file formats, e.g., GeoTiff, so that users can use the output in another system.

There are two main challenges in writing raster files to a distributed file system. First, a distributed file system writes a file only in 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. RDPro addresses these challenges by offering two modes for writing, a fully distributed mode, and a compatibility mode, each detailed below.

In the **Fully distributed mode** each worker node independently writes a separate raster file to the output containing only the tiles available on that machine. This is the common approach used by all Spark-based systems because it enhances efficiency by eliminating the need for inter-node coordination during writing. A notable challenge in common raster file formats is that the header, which must reference the storage locations of each tile, cannot be written until the file's contents are finalized. An easy solution is to create a new file format that does not have this limitation, e.g., has a footer rather than a header. However, we want users to be able to open the files that RDPro generates in existing geospatial systems such as QGIS. To address this challenge, *RDPro* works in two rounds, *data writing* followed by *header writing*. MapLocator significantly benefits both rounds as RDpro utilizes it to ensure that each unique

MapLocator generates a distinct file. This allows it to combine Maplets into a few files with shared header.

The *data writing* round goes over all the Maplets and writes the data array of measures to disk in a temporary data file. Compression, like LZW, is applies if configured by the user. This round also tracks the tile sizes on disk. Since this mode runs independently on each worker node, each one will only have a subset of tiles in one raster dataset which results a sparse file. Most raster file formats, e.g., GeoTIFF, do not support sparse files. To resolve this issue, this round finishes by writing an additional empty tile at the end of the data file. Then, it updates the position of all missing tiles to point to this empty tile. Another non-standard approach that RDPro and GDAL support is to write empty tiles with zero position and length.

The *header writing* round uses tile sizes to create the header of the file, including tile offsets and lengths in the final file. MapLocator is conveniently used to create the GeoTiff header by storing all the necessary information, including the extent, CRS, raster width, and length. RDPro writes this header to a separate file and then the header and data files are merged into one. In HDFS, this concatenation runs efficiently in constant time since it runs entirely in the name node's main memory by rearranging HDFS blocks. This approach avoids the need to physically read and write file contents. For file systems lacking this efficient concatenation, RDPro defaults to a traditional method, rewriting both files into one.

Compatibility mode: While the previously described distributed mode is efficient, it is not ideal for conventional GIS software like QGIS, which treats the output files as separate entities. To bridge this gap, RDPro offers a compatibility mode, ensuring seamless integration with traditional GIS software. This mode produces a single raster file, even when data spans multiple machines.

Like the distributed mode, compatibility mode operates in two rounds. The first round, data writing, runs completely in parallel and is similar to that of fully distributed mode. Once all worker nodes are done, the data file names, the tile IDs, and tile sizes are all collected into a single machine. This machine runs the second round, header writing. It first prepares all the information needed to write the header and calculates the header size before writing it. Then, it sorts the data files by name and calculates the final position of each tile based on header and tile sizes. Finally, it writes the header to a separate file and all header + data tiles are concatenated as described earlier. Notably, even if the efficient concatenation method is unavailable, this mode remains faster than single-machine writing due to parallel tile compression and encoding.

4.4 Raster Query Processing

This part explains how RDPro runs raster operations on the innovative RDD[Maplet] construct in RDPro. In particular, we show examples of the four main categories of raster operations, namely, local, focal, zonal, and global, that were introduced in section 3. Each operation is crafted as an RDD transformation. This design enables developers to integrate these operations into a query pipeline alongside existing RDD operations. Spark then optimizes and executes this pipeline as a single application. Such an approach significantly enhances the potential of *RDPro*, providing users with the flexibility to blend proposed operations with standard Spark operations, thereby allowing the construction of complex analytical processes.

(1) *MapPixels(R, f)*: This operation applies a user-defined function f on measure values of the input RasterRDD to produce the output. MapPixels is implemented as a Spark `map` operation that takes one tile at a time and produces a tile with the same MapLocator and t_{id} as the input. However, it modifies the measure values based on the user-provided function. To optimize this operation and reduce memory usage, we do not create a new array of measures for the output tile. Rather, we create a wrapper tile that bypasses all methods to the wrapped tile *except* for retrieving the pixel values in which the user-provided function is applied on the fly.

(2) *FilterPixels(R, f)*: This operation retains pixels that match the user-defined filter and sets other pixels to null. This is considered a special case of MapPixels and is implemented in a similar way except that it returns `null` for pixels that do not match the filter.

(3) *Overlay(R_1, R_2)*: This local operation takes two RasterRDDs, R_1 and R_2, and returns a single RasterRDD where the measure values from the two inputs are stacked together. Overlay requires all Maplets in the two inputs to have the same MapLocator to ensure one-to-one correspondence between pixels. If not, we need to run the Reshape operation (described next) before running this operation. The Overlay operation first checks if the two rasters are co-partitioned by tile ID. If not, it will co-partition them, i.e., use the same partitioner on the tile ID attribute. After that, it uses the `zip` method to bring each pair of tiles with the same t_{id} together. Next, it uses the `map` operation to create one output tile that wraps the two input tiles. The output tile has same MapLocator and t_{id} of the inputs but it override the pixel measure values by concatenating the measure values of the two input tiles.

(4) *Reshape(R_1, RM_2)*: The reshape operation converts the input raster dataset to a new MapLocator RM_2 with possibly different CRS, raster size, and tile size. This operation is crucial in aligning one or more raster datasets to apply one of the other operations that expects input rasters to have Maplets with the same MapLocator, e.g., Overlay, SlidingWindow, and Convolution. Reshape is a focal operation since there is no one-to-one map between input and output pixels. There are different approaches to determine the value of the output pixel such as linear interpolation, spline interpolation, majority, and nearest neighbor. These functions determine how to combine multiple pixels of the source into one target pixel to produce a single value for the output pixel. RDPro supports both linear interpolation and nearest neighbor. We describe the nearest neighbor approach since it works for both continuous values, e.g., temperature, and categorical values, e.g., land use.

One challenge with the Reshape function is that there is no easy mapping between input and output pixel locations. This mapping depends not only on the source and target CRS, but also on the location of each tile on Earth. To address this challenge, the Reshape operation needs to calculate a grid-to-grid $\mathcal{G}2\mathcal{G}$ transformation function that maps a location from the source to the target grid space. This function could differ from one Maplet to another since the input Maplets might have different MapLocators, e.g., if they are loaded from different files.

To calculate $\mathcal{G}2\mathcal{G}$ for a specific source tile r_{id}, RDPro begins by retrieving a transformation function $\mathcal{W}2\mathcal{W}$ that transforms between the world coordinates of the two CRS. Some existing libraries, e.g., GeoTools and Proj4J, provide this function for all standard CRS definitions. After that, the $\mathcal{G}2\mathcal{W}$ from the two MapLocators, $\mathcal{G}2\mathcal{W}_1$

and $\mathcal{G}2\mathcal{W}_2$, are used to compute $\mathcal{G}2\mathcal{G}$ as:

$$\mathcal{G}2\mathcal{G} = \mathcal{G}2\mathcal{W}_2^{-1} \circ \mathcal{W}2\mathcal{W} \circ \mathcal{G}2\mathcal{W}_1$$

, where $\circ$ denotes function composition. Since $\mathcal{G}2\mathcal{W}$ is an affine transformation, RDPro will detect if the transformation function $\mathcal{W}2\mathcal{W}$ is also an affine transformation in which case RDPro will calculate the composite function $\mathcal{G}2\mathcal{G}$ as an affine transformation by multiplying the three affine matrices. This speeds up the computation for this common special case. The inverse function $\mathcal{G}2\mathcal{G}^{-1}$ is also calculated similarly.

The Reshape operation runs in two steps. The first step applies a flatMap operation on the input tiles to map each one to a set of output tiles. To do that, it begins by computing $\mathcal{G}2\mathcal{G}$ of the source tile r_{id} and uses it to convert the four corners of the r_{id} to the target raster. Then, the target MapLocator is used to determine the range of target tiles that overlap that transformed range. After that, the center of each pixel location in each target tile s_{id} is mapped back to the source raster using $\mathcal{G}2\mathcal{G}^{-1}$ to find the nearest pixel in the source. If the location of the source pixel falls in the source tile r_{id}, its measure value is copied to the target tile; otherwise, the target pixel is left empty. This flatMap step results in a set of partial intermediate tiles. Since one target tile might overlap with multiple source tiles, potentially on different machines, these intermediate tiles need further processing to produce the final result.

The second step merges the intermediate tiles using the *reduceByKey* operation to calculate output tiles. To merge two intermediate tiles with the same ID, it copies all non-empty pixels from one tile to the other one. Notice that one pixel could have different values in two intermediate tiles, e.g., if input tiles partially overlap. In this case, we can keep any of them since both could be considered nearest neighbors. If an interpolation method is used, then both values can be merged into one but we omit this part for brevity.

Special cases: The Reshape operation is used to implement three other common raster transformations, Reproject, Regrid, and Retile. Reproject converts a raster to another CRS while keeping the resolution and tile size. Regrid keeps the same CRS and tile size and modifies the resolution (raster size). Retile keeps the same CRS and resolution and only modifies the tile size. All the three operations first compute a new MapLocator and then call the Reshape operation defined above.

(5) *SlidingWindow(R, w, f)*: The SlidingWindow operation performs a window calculation function on an input raster dataset and produces an output raster dataset. Unless $w = 0$, this operation is a focal operation that needs all pixels with a window of size $2w+1 \times 2w+1$ around each pixel. Figure 5 shows an example when $w = 1$, there are $3 \times 3 = 9$ pixels in the window. As shown in the figure, some pixels in the output that are near tile boundaries depend on pixels that come from two or more Maplets. Furthermore, those Maplets might be on different machines which adds to the complexity of this operation.

RDPro designs the SlidingWindow method in two steps illustrated in Figure 6. In this example, the input is a raster with nine Maplets and is partitioned into three parts, indicated by yellow lines, on three different machines. We use r_{id}, w_{id}, and s_{id}, to refer to input tiles, intermediate window tiles, and output tiles, respectively. Notice that while some tiles, e.g., s_0, can be computed locally in partition 1, all other output tiles need to access data from two or

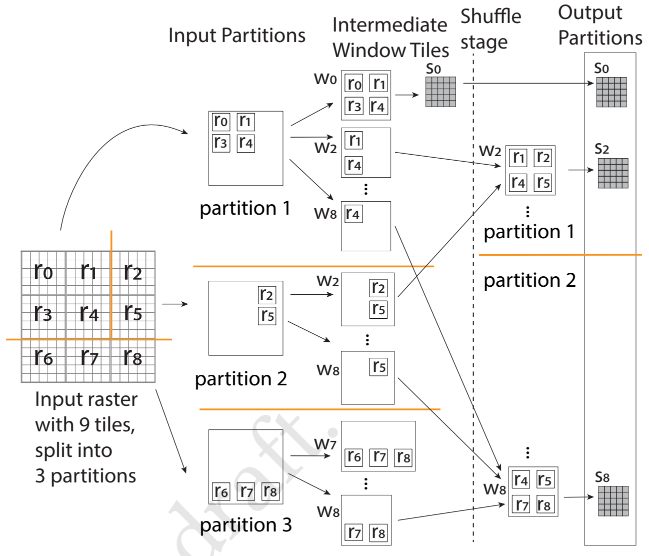


Figure 6: Overview processing of SlidingWindow function

more input partitions which requires a shuffle stage. RDPro optimizes this operation to reduce the amount of data that gets shuffled as described below.

To implement the SlidingWindow efficiently, RDPro introduces intermediate WindowTiles. Each WindowTile holds all input tiles that are needed to compute one of the output tiles. This operation runs in two steps. The first step applies a flatMap operation that processes each input tile t_{id} and produces a set of intermediate tiles W for each output tile that is affected by the input tile. For example, when we process r_0, four window tiles are produced, W_0, W_1, W_3, and W_4, since they are all neighboring tiles. Each of these tiles is initialized with the input tile, r_0, in it. For input tile r_4, nine intermediate tiles $W_0 \ldots W_8$ are created. Notice that the figure shows only some of the intermediate and output tiles for simplicity.

The second step uses *reduceByKey* to merge partial window tiles and compute the final output tiles. If needed, this step can change the number of partitions for better load balance. To combine two window tiles, we first compute the union of the input tiles in them. Once the set of input tiles are complete, i.e., all required input tiles are in the window tile, the window computation is done with the user-defined function to compute the output tile. Notice that window tiles are combined locally first within each machine and then they are shuffled to complete this step. In that sense, as shown in Figure 6, tile s_0 depends on four tiles, r_0, r_1, r_3, and r_4, which all reside in partition 1. This allows tile s_0 to be computed and finalized locally within partition 1. On the other hand, tile s_2 depend on tiles r_1, r_2, r_4, and r_5, which are spread across two partitions, hence, it will be computed after the shuffle stage. Similarly, s_8 depends on four tiles in three partitions so it will be computed after the shuffle.

Special case - Convolution(R, w, k): This is a special version of SlidingWindow where the user-defined function is a weighted average of all pixels in the window. The list of weights is called the *kernel*. RDPro provides a more optimized version of SlidingWindow for convolution function by eagerly applying the kernel to partially compute the output tile. In other words, we eliminate the intermediate WindowTiles and directly compute the output tiles. We still need the shuffle stage to merge output tiles together and this is only possible in this case because the linear function can be easily broken down and computed partially.

(6) *RaptorJoin(R, V)*: The input to this zonal operation is a RasterRDD and a vector dataset. The output is a set that contains every pixel that matches any of the geometries in the vector data. This method is implemented by first creating a Flash index on the vector data that maps between vector and raster and allows it to efficiently retrieve matching pixels from all tiles. For brevity, we refer interested readers to [4] that studies this problem in detail.

(7) *FlattenValues(R)*: This global operation takes as input one raster dataset and produces a set of all measure values for all pixels in the raster data. The output of this operation is a Spark RDD that contains all the values, which can be processed using any aggregation operation. There are two important points to mention about this operation. First, when this method is followed by an aggregate operation, the values are pipelined by Spark into the next operation, hence, the values never need to be materialized in memory or on disk. Second, we chose to implement this method instead of introducing new global aggregate operations so that users can build on the existing and optimized Spark functionality.

(8) *Rasterize*: This operation is the opposite of FlattenValues. It takes as input a set of pixel values, their corresponding locations, and a MapLocator, and produces a set of Maplets that represents all pixel values. Along with raster loading, this is the second operation that can create a first RasterRDD, i.e., not from another RasterRDD. This operation starts by mapping each pixel location to a tile (t_{id}) using the given MapLocator. Then it applies a groupByKey operation to group pixel values by their t_{id}. After that, it runs on each group to initialize a Maplet from t_{id} and the MapLocator. Finally, it iterates over all pixel values in the group and sets the corresponding measure value in the Maplet.

5 EXPERIMENTS

This section provides an experimental evaluation that compares RDPro to the distributed systems, Apache Sedona [46] and GeoTrellis [17]. We show that RDPro is more scalable and robust than the baselines for big raster datasets. The following sections describe the experimental setup and provide a comparison of all raster operations described in previous content subsection 3.3 for these systems. To keep the experiments concise, we do not provide experiments on Raptor Join since it was well-studied in existing work [4].

5.1 Setup

We run *RDPro*, GeoTrellis, and Sedona on a cluster with one head node and 12 worker nodes running Spark 3.1.2. The header node has 128 GB of RAM, 2×8 core processors, and Intel(R) Xeon(R) CPU E5 - 2609 v4 1.70GHz processor. Each worker node has 64 GB of RAM and 2×6 core Xeon processors on CentOS Linux. We compare to the GeoTrellis-Spark package 3.6.3 and Apache Sedona 1.5.0. We run each experiment three times and report the average end-to-end running time.

Table 1: Raster Datasets

Dataset	# pixels	Resolution	Size
CDL_City	52.71 M	30 m	201.09 MB
CDL_State	1.57 B	30 m	5.33 GB
CDL_Country	16.88 B	30 m	62.88 GB
Landsat8_City	237.97 M	30 m	689.86 MB
Landsat8_State	1.57 B	30 m	4.40 GB
Landsat8_Country	52.53 B	30 m	146.78 GB
Landsat8_World	798.47 B	30 m	2.18 TB
Planet_Country	2.46 T	3 m	6.7 TB

Note that we did not compare to Google Earth Engine since it is not suitable for a systematic and reproducible experimental evaluation due to three reasons. (1) It is not open source and Google did not publish its internal design. (2) We cannot choose the hardware that it runs on for a fair comparison. (3) The running time is highly unstable as it depends on the jobs scheduled by other users in the world. Furthermore, we did not include a comparison with Rasdaman since the public version of Rasdaman runs a single machine and it will be unfair to compare it to distributed systems.

Table 1 lists the datasets used in the experiments and their attributes. All raster datasets except Planet Data are publicly available. The CDL and Landsat8 dataset are made available by the US Department of Agriculture (USDA) and US Geoegogical Survey (USGS), respectively. To test scalability, we created three subsets for the regions of city, state, and country[1]. Planet_Country is sourced from Planet Labs [40]. The size in bytes of each dataset reflects the number of bands and data type of measurement values.

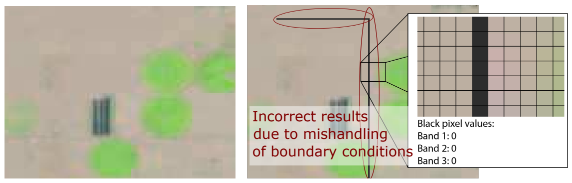
5.2 Robustness and Correctness

Before heading into the performance experiments, this part highlights some results related to correctness and robustness. In particular, we highlight why two important operations, reprojection and convolution, should handle tile boundaries correctly.

In this part, we run the two operations on RDPro and GeoTrellis and visualize the results. Sedona does not support any of these operations. Figure 7 shows the results of the reproject operation. RDPro shows the correct result while GeoTrellis shows several empty pixels due to mishandling of partitioned datasets. The figure highlights the pixels that are incorrectly set to zero in GeoTrellis. Figure 8 shows the result of the convolution function on both RDPro and GeoTrellis where RDPro provides the correct result while GeoTrellis does not. Recall that pixels that are at tile boundaries requires additional communication between machines to ensure correct result. This figure shows that favoring performance at the cost of correctness would produce clearly incorrect results. The extent of severity of these incorrect results depend on the application being processed. However, such inaccuracies should only be justified by clear application requirements and should not be driven by mishandling of data.

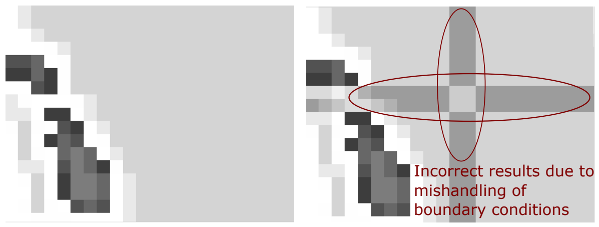
In addition to the above correctness examples, we noticed that baseline systems often fail with either out-of-memory (OOM) or some other exceptions, especially when loading and writing large datasets. These failing cases will be highlighted later in this section.

[1]City, state, and country names removed for anonymity.



(a) RDPro (Correct) (b) GeoTrellis (Wrong)

Figure 7: Correctness of reproject as opposed to GeoTrellis



(a) RDPro (Correct) (b) GeoTrellis (Wrong)

Figure 8: Correctness of convolution as opposed to GeoTrellis

Table 2: Load+write time on RDPro and baselines in seconds

Data Size	689.86MB	4.4GB	146.78GB	2.18TB	6.7TB
RDPro Distributed	58.39	62.51	352.59	5957	17819
RDPro Compatibility	61.94	63.74	357.37	5851	17434
GeoTrellis	105.09	156.08	OOM	OOM	OOM
Sedona	142.61	OOM	OOM	OOM	OOM

5.3 Data Loading and Writing

This experiment covers data loading and writing. It simply loads each dataset and writes it back as a new GeoTIFF file (or a set of files). Table 2 shows the running time of both the loading and writing steps combined for datasets that range from 700MB to 6.7TB. The table includes both distributed writing and compatibility writing modes for RDPro. As explained earlier, both modes perform similarly since the concatenation step runs almost instantly. The slight variation in numbers is due to cluster variability but is not significant enough to distinguish the two. Comparing baselines, first, RDPro is consistently faster than baselines with up-to 3x speedup than GeoTrellis even in compatibility mode. Second, most existing systems fail to load and/or write large datasets as they tend to throw an out-of-memory (OOM) exception. In particular, Sedona is designed to load each file as one piece and cannot partition large files. This fits its proposed design to load small files that represent individual features rather than processing large raster products [11]. On the other hand, GeoTrellis raster writer needs to stitch all the data into a single machine which causes failure even for moderately sized datsets, e.g., 150 GB.

Table 3: MapPixels running time in seconds

Data Size	689.86MB	4.4GB	146.78GB	2.18TB	6.7TB
RDPro	16.91	16.55	84.90	1206	3170
GeoTrellis	28.48	31.81	112.6	1212	OOM
Sedona	255.47	OOM	OOM	OOM	OOM

Table 4: FilterPixels running time in seconds

Data Size	689.86MB	4.4GB	146.78GB	2.18TB
RDPro	18.38	85.30	1207.02	3167
GeoTrellis	31.09	114.60	1152	OOM
Sedona	222.74	OOM	OOM	OOM

Table 5: Overlay dataset sizes and running time in seconds on Landsat8 and Planet

Data Sizes	689.86MB 201.09MB	4.40GB 5.33GB	146.78GB 5.33GB	2.18TB 5.33GB	6.7TB 5.33GB
RDPro	64.07	132.96	203.3	858.68	3171.0
GeoTrellis	114.0	433.4	727.9	Error	OOM

5.4 MapPixels and FilterPixels Operation

This experiment evaluates the performance of two simple local operations, MapPixels and FilterPixels. In MapPixels, we read all the bands and sum them. In FilterPixels, we filter pixels with a value less than 100. Due to the lazy execution nature of RDPro, we iterate over all pixel values to ensure that the job runs to completion. Note that since these operations are fairly light-weight, the running time is dominated by data loading.

Table 3 shows the MapPixels running time in seconds. Both RDPro and GeoTrellis handle small and medium sized datasets efficiently. However, GeoTrellis fails with the largest dataset with an out-of-memory exception due to its poor loading component. Sedona not only fails to load big datasets, it suffers from poor performance due to the lack of parallelization when processing a few large files. Table 4 shows similar behavior for FilterPixels since they are both dominated by data loading.

5.5 Overlay

This experiment evaluates the performance of the Overlay operation as it combines two datasets, namely CDL and Landsat8. Recall that the Overlap operation requires the two inputs to share the same MapLocator. However, due to the different CRS and extents between the two datasets, we reshape the Landsat8 to match the MapLocator of the CDL. Effectively, this adjusts the resolution, the CRS, and the tile size to perfectly align the two datasets. Sedona is excluded from this experiment due to the lack of a reproject function. To run this operation in GeoTrellis, it needs to collect all metadata centrally on a single machine to clip and align the two datasets which results in its poor performance shown in Table 5. RDPro is about three times faster than GeoTrellis for small and medium datasets. As the data size grows, RDPRo continues to scale while GeoTrellis starts to throw several errors which shows its poor performance and lack of robustness.

Table 6: Reproject running time on Landsat8

Data Size	689.86MB	4.4GB	146.78GB	2.18TB
RDPro	190.49	244.52	2499.94	48606.41
GeoTrellis	217.71	445.06	OOM	OOM

Table 7: Rescale running time in seconds on Landsat8

Data Size	689.86MB	4.4GB	146.78GB	2.18TB
RDPro	56.74	48.79	158.37	2781.68
GeoTrellis	81.85	94.11	negative array	OOM
Sedona	97.06	OOM	OOM	OOM

Table 8: SlidingWindow running time in seconds on CDL

Data Size	201.09MB	5.33GB	62.88GB
RDPro SlidingWindow	27.48	52.26	252.97
RDPro Convolution	25.45	34.09	94.93
GeoTrellis	35.45	121.05	591.22

5.6 Reshape Operation

This experiment evaluates the performance of the Reshape operation on RDPro and GeoTrellis. We evaluate Reshape in two scenario, Reproject and Rescale. Sedona can only support the Rescale operation so it is excluded from Reproject. In Reproject, we convert the CRS of Landsat8 datasets to EPSG:4326. In Rescale, we reduce the resolution of the Landsat8 dataset to produce a lower-resolution version by reducing both width and height by a factor of 10, which will return a 100 times smaller raster. In both cases, we write the output to GeoTIFF to ensure that the computation of the final raster is complete. Table 6 shows the performance of the Reproject operation. RDPro is almost twice as fast as GeoTrellis and can scale to the biggest dataset which GeoTrellis runs out of memory and fails. Similarly, for the Rescale operation in Table 7, RDPro scales seamlessly and provides a better performance than GeoTrellis. This better performance is despite the fact that GeoTrellis ignores boundary conditions and produces incorrect results as shown in Figure 7.

5.7 SlidingWindow (Convolution)

This experiment evaluates the performance of the SlidingWindow operation in RDPro and GeoTrellis. Sedona does not have SlidingWindow or kernel function on raster data. Since GeoTrellis only supports the convolution function, we use the optimized convolution implementation in RDPro for fair comparison. We use a window of size one ($w = 1$) and calculate the average of all the nine pixels in the window. This experiment requires all tiles in the input dataset to have the same MapLocator. To focus on the performance of the SlidingWindow function, we only apply it on the CDL dataset because it ships as a single file. As shown in Table 8, both RDPro and GeoTrellis can process all the datasets but RDPro is up-to 6 times faster. This is an impressive result, especially that GeoTrellis has an unfair advantage of ignoring the difficult boundary cases as shown in Figure 8. Table 8 also shows that the optimized implementation for the convolution method is up-to 2.5x faster than the generic SlidingWindow implementation. Notice that GeoTrellis only implements the specialized convolution method.

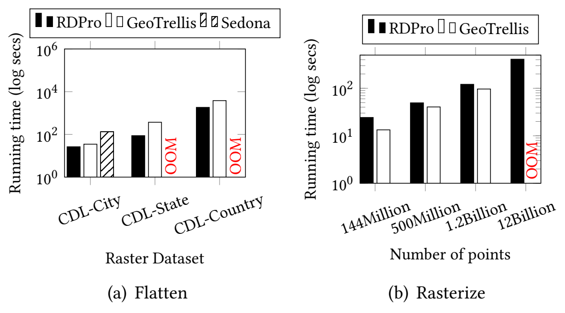


Figure 9: Flatten and Rasterize operations

5.8 Flatten Value and Rasterize Operation

This part covers two global operations, Flatten and Rasterize. We run Flatten on CDL datasets followed by a countByKey operation to compute a histogram. We measure the end-to-end running time and since countByKey is a light-weight operation, the time is dominated by the Flatten operation. 9(a) reports the results of the Flatten operation. Sedona does not support the histogram function so we compute the mean which is also a light-weight function. Both RDPro and GeoTrellis provide decent results since they can partition the input with RDPro being slightly faster. Sedona is slower for the smaller data and fails for bigger data due to the lack of input partitioning and parallelization.

To evaluate the performance of Rasterize, we generate random points of increasing scale, from 144 million to 12 billion, within the extents of the world. We then rasterize them and write the output raster to a GeoTiff file. Sedona is excluded due to the lack of support of this function. Both RDPro and GeoTrellis scale well for this operation. GeoTrellis is slightly faster since it avoids the overhead of parallelization by writing the file on a single machine. However, GeoTrellis requires input as an array of points at a given time and then operates the rasterization. This implies it cannot exceed the array size limitation in resulting out of memory issue.

6 CONCLUSION

This paper introduced the distributed system *RDPro* to efficiently process big raster data. RDPro is implemented in Spark and uses a custom raster data model, RDD[Maplet] to represent and process raster data in a distributed environment. It also implements its own data loading and data output components which facilitate distributed reading and writing of raster data. RDPro uses RDD[Maplet] to implement various operations required for raster analysis. Since *RDPro* 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. An extensive experimental evaluation compared RDPro to GeoTrellis and Apache Sedona, both run on Spark. The experiments confirmed that RDPro is the most scalable system among all of them. Furthermore, the experiments revealed severe limitations of existing systems that always fail with very large datasets.

REFERENCES

[1] Shantanu Aggarwal. 2022. *SATLAB-an End to End Framework for Labelling Satellite Images*. Ph.D. Dissertation. Arizona State University.

[2] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop-GIS: A high performance spatial data warehousing system over MapReduce. In *Proceedings of the VLDB endowment international conference on very large data bases*, Vol. 6. NIH Public Access.

[3] Anonymous authors. [n.d.]. Anonymous title. ([n. d.]).

[4] Anonymous authors. [n.d.]. Anonymous title.

[5] Peter Baumann. 2022. The Rasdaman Array DBMS: Concepts, Architecture, and What People Do With It. In *SSDBM*. Association for Computing Machinery, Article 27.

[6] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. 1998. The Multidimensional Database System RasDaMan. In *SIGMOD*. Seattle, WA, 575–577.

[7] Douglas K. Bolton, Josh M. Gray, Eli K. Melaas, Minkyu Moon, Lars Eklundh, and Mark A. Friedl. 2020. Continental-scale land surface phenology from harmonized Landsat 8 and Sentinel-2 imagery. *Remote Sensing of Environment* 240 (2020), 111685.

[8] Daniel Brown et al. 2015. Monitoring and evaluating post-disaster recovery using high-resolution satellite imagery–towards standardised indicators for post-disaster recovery. *Martin Centre: Cambridge, UK* (2015).

[9] Paul G Brown. 2010. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 963–968.

[10] Joe B Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. 2011. Scihadoop: Array-based query processing in hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[11] Kanchan Chowdhury and Mohamed Sarwat. 2023. A Demonstration of GeoTorchAI: A Spatiotemporal Deep Learning Framework. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*. ACM, 195–198.

[12] ClimateSpark 2018. ClimateSpark. https://github.com/feihugis/ClimateSpark

[13] Chris Dickens et al. 2019. Defining and quantifying national-level targets, indicators and benchmarks for management of natural resources to achieve the sustainable development goals. *Sustainability* 11, 2 (2019), 462.

[14] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A MapReduce Framework for Spatial Data. In *2015 IEEE 31st international conference on Data Engineering*. IEEE, 1352–1363.

[15] Ahmed Eldawy, Mohamed F Mokbel, Saif Alharthi, Abdulhadi Alzaidy, Kareem Tarek, and Sohaib Ghani. 2015. Shahed: A MapReduce-based System for Querying and Visualizing Spatio-temporal Satellite Data. In *2015 IEEE 31st international conference on data engineering*. IEEE, 1585–1596.

[16] GDAL/OGR contributors. 2022. *GDAL/OGR Geospatial Data Abstraction software Library*. Open Source Geospatial Foundation. https://doi.org/10.5281/zenodo.5884351

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

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

[19] Rahul Ghosh, Praveen Ravirathinam, Xiaowei Jia, Ankush Khandelwal, David J. Mulla, and Vipin Kumar. 2021. CalCROP21: A Georeferenced multi-spectral dataset of Satellite Imagery and Crop Labels. In *2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021*. IEEE, 1625–1632. https://doi.org/10.1109/BIGDATA52589.2021.9671569

[20] Thomas W Gillespie et al. 2007. Assessment and prediction of natural hazards from satellite imagery. *Progress in Physical Geography* 31, 5 (2007), 459–470.

[21] Sean Gillies et al. 2013–. *Rasterio: geospatial raster I/O for Python programmers*. Mapbox. https://github.com/rasterio/rasterio

[22] Noel Gorelick et al. 2017. Google Earth Engine: Planetary-scale geospatial analysis for everyone. *Remote sensing of Environment* 202 (2017), 18–27.

[23] Robert J. Hijmans and Jacob van Etten. 2012. *raster: Geographic analysis and modeling with raster data*. http://CRAN.R-project.org/package=raster R package version 2.0-12.

[24] Fei Hu, Chaowei Yang, John L Schnase, Daniel Q Duffy, Mengchao Xu, Michael K Bowen, Tsengdar Lee, and Weiwei Song. 2018. ClimateSpark: An in-memory Distributed Computing Framework for Big Climate Data Analytics. *Computers & geosciences* 115 (2018), 154–166.

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

[26] Yi Liu, Luo Chen, Wei Xiong, Lu Liu, and Dianhua Yang. 2012. A MapReduce Approach for Processing Large-scale Remote Sensing Images. In *The 20th International Conference on Geoinformatics, Geoinformatics 2012, Hong Kong, China, June 15-17, 2012*. IEEE, 1–7. https://doi.org/10.1109/GEOINFORMATICS.2012.6270312

[27] Emanuele Mandanici and Gabriele Bitelli. 2016. Preliminary Comparison of Sentinel-2 and Landsat 8 Imagery for a Combined Use. *Remote Sensing* 8, 12 (2016). https://doi.org/10.3390/rs8121014

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

[29] Rahul Palamuttam, Renato Marroquín Mogrovejo, Chris Mattmann, Brian Wilson, Kim Whitehall, Rishi Verma, Lewis McGibbney, and Paul Ramirez. 2015. SciSpark: Applying in-memory Distributed Computing to Weather Event Detection and Tracking. In *2015 IEEE International conference on big data (big data)*. IEEE, 2020–2026.

[30] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. 2016. The TileDB Array Data Storage Manager. *Proceedings of the VLDB Endowment* 10, 4 (2016), 349–360.

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

[32] Kamleshan Pillay et al. 2014. Modelling changes in land cover patterns in Mtunzini, South Africa using satellite imagery. *Journal of the Indian Society of Remote Sensing* 42, 1 (2014), 51–60.

[33] PostGIS 2022. PostGIS. https://postgis.net/.

[34] QGIS Development Team. 2022. *QGIS Geographic Information System*. QGIS Association. https://www.qgis.org

[35] CA: Environmental Systems Research Institute Redlands. 2022. ArcGIS. https://www.esri.com/en-us/arcgis/about-arcgis/overview

[36] Elia Scudiero, Todd H Skaggs, and Dennis L Corwin. 2014. Regional scale soil salinity evaluation using Landsat 7, western San Joaquin Valley, California, USA. *Geoderma Regional* 2 (2014), 82–90.

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

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

[39] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. 2013. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science Engineering* 15 (05 2013), 54–62.

[40] Planet Team. 2018–. Planet Application Program Interface: In Space for Life on Earth. https://api.planet.com

[41] TileDB, Inc. 2023. *tiledb: Universal Storage Engine for Sparse and Dense Multidimensional Arrays*. https://github.com/TileDB-Inc/TileDB-R R package version 0.20.1.

[42] UCSUSA 2022. UCS Satellite Database. https://www.ucsusa.org/resources/satellite-database.

[43] Randall T Whitman, Michael B Park, Sarah M Ambrose, and Erik G Hoel. 2014. Spatial indexing and analytics on Hadoop. In *Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems*. 73–82.

[44] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 international conference on management of data*. 1071–1085.

[45] Chenghai Yang et al. 2012. Using high-resolution airborne and satellite imagery to assess crop growth and yield variability for precision agriculture. *Proc. IEEE* 101, 3 (2012), 582–592.

[46] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*.

[47] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2016. A Demonstration of GeoSpark: A Cluster Computing Framework for Processing Big Spatial Data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 1410–1413.

[48] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. 2019. Spatial Data Management in Apache Spark: The GeoSpark Perspective and Beyond. *GeoInformatica* 23 (2019).

[49] Ramon Antonio Rodriges Zalipynis. 2018. ChronosDB: Distributed, File based, Geospatial Array DBMS. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1247–1261.