Raptor Join: In-situ Processing of Big Raster + Vector Data

Samriddhi Singla *, Ahmed Eldawy *, Tina Diao †, Ayan Mukhopadhyay ‡, Elia Scudiero §
* Computer Science and Engineering, § Environmental Sciences
University of California, Riverside
{ssing068, eldawy, elias}@ucr.edu
† Management Science & Engineering ‡ Aeronautics and Astronautics
Stanford University
{tdiao, ayanmukh}@stanford.edu

***Abstract*—There has been a rapid increase in the amount of spatial data due to the advancements in remote sensing and the increasing use of smart devices. This has allowed for greater research opportunities and efforts have been made to develop systems that can process big amounts of data to facilitate this research. Spatial data can be represented using two data models, raster and vector. Raster data refers to satellite imagery while vector data includes GPS data, Tweets, regional boundaries, etc. Traditional systems implement algorithms that work with only one of these data models. However, there exist problems that require combining both forms of data. We propose a formal raster-vector join algorithm, *Raptor Join* that can process the combination of raster and vector data. It is modeled as a relational join operator in Spark that can easily be combined with other operators, while also offering the advantage of in-situ processing. This makes it ideal for ad-hoc query processing. We run an extensive experimental evaluation on large scale satellite data with up-to a trillion pixels, and big vector data with up-to hundreds of millions of edges and billions of points, and show that the proposed method can scale to big data with up-to three orders of magnitude performance gain over GeoTrellis and Google Earth Engine.**

I. Introduction

The recent decade has seen an explosive increase in the amount of spatial data. The advancement in remote sensing technology has led to the availability of petabytes of satellite imagery. 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. Spatial data can generally be modeled in two representations: raster and vector. 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 growth of geospatial data helped in new scientific discoveries in a wide range of applications such as areal interpolation [1], wildfire risk assessment [2], [3], the effect of vegetation and temperature on human settlement [4], [5], analysis of terabytes of socio-economic and environmental data [6], [7], and land use classification [8]. All these applications process the two spatial data representations simultaneously, e.g., compute average temperature per city or maximum vegetation per agricultural field. Unfortunately, big spatial data systems still focus on either big vector data, e.g., SpatialHadoop [9], GeoSpark [10], and Simba [11], or big raster data, e.g., SciDB [12], RasDaMan [13], GeoTrellis [14], ChronosDB [15], and Google Earth Engine [16]. The main limitation that kept these two types of systems isolated is the different data and query processing models that cannot be easily combined together. The only way to combine these two data representations is to either vectorize the raster data or rasterize the vector data but we show in this paper that none of these approaches scale to high-resolution data.

First, consider raster-based systems. These systems adopt the array data model which represents all the data as multidimensional arrays which is suitable for scientific applications and querying raster data [12], [13], [14], [15], [16]. Raster based systems use a query processing model based on linear algebra which is efficient for convolutional operations such as field smoothing and edge detection. These systems suffer from *three main limitations* that limit their scalability. First, all these systems require an expensive data loading phase that structures the input data into the array model. With petabytes of remote sensing satellite data, the data loading phase usually takes longer than the query processing itself. This makes it unsuitable for ad-hoc exploratory queries which are widely used in data science applications. The second limitation is the inability to process geospatial vector data as-is. Vector data needs to be rasterized and loaded into the raster-based system. The size of the converted data increases quadratically with the raster data resolution and becomes a bottleneck with high-resolution raster data. The third limitation is the complexity of the data processing models that use linear algebra, especially for simple queries that can be easily expressed using relational algebra. For example, in EarthDB [17], 40 lines of code were used in SciDB to compute the Normalized Difference Vegetation Index (NDVI) which is just a simple formula $NDVI = (nir - red)/(nir + red)$. The same logic can be expressed as one simple SQL query.

Second, consider vector-based systems. Most of these systems use the relational data model and relational algebra which makes it relatively easier to use them for most users who are familiar with SQL [18], [9], [10], [11]. Moreover, some of these recent systems use the in-situ approach which does not require a separate data loading phase. Unfortunately, these systems fail to process high-resolution *raster data* due to the huge overhead of converting raster data to vector. For example, a small GeoTIFF compressed raster file of 34 mb would explode to nearly 3 gb of data in vector format. The reason is that a single-byte pixel value m in a raster form needs to be associated with at least three values in vector form (x, y, m) with at least five bytes, where (x, y) is the pixel location. Overall, this single limitation makes vector-based systems much slower than raster-based systems [19], [7], [6]. In contrast, the proposed approach in this paper can directly process compressed GeoTIFF files which makes it orders of magnitude faster than baselines.

This paper proposes a novel approach for concurrent processing of raster and vector data. The key contribution is a new raster-vector join algorithm, termed *Raptor Join* (RJ$_{\bowtie}$). Raptor Join overcomes the limitations of existing systems as follows. First, RJ$_{\bowtie}$ is implemented using an in-situ approach in Spark [20] 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, RJ$_{\bowtie}$ is modeled as a relational operator which makes it easier to use in SQL queries in combination with other relational operators such as selection, join, and aggregation. In this paper, we show the effectiveness of the Raptor Join operator by applying it to three real scientific applications that allows them to scale to big data.

The proposed RJ$_{\bowtie}$ operator models both raster and vector data as relational data. Vector data is modeled as (g_{id}, g) tuples where the geometry g can be a point, line, or polygon and is associated with a unique identifier g_{id}. On the other hand, raster data is modeled as (x, y, m, bb) where (x, y) is the pixel location in the raster array, m is the measure value of that pixel, and bb is a bounding box that represents the pixel's geometric location. While it seems that this model expands the size of the raster data, this is only a logical model and we do not physically convert the raster data into this form. RJ$_{\bowtie}$ is modeled as a join operator between geometries and pixels. Based on the proposed model, we define three predicates for points, lines, and polygons. Furthermore, we define how RJ$_{\bowtie}$ supports both inner and outer joins. The key idea that allows RJ$_{\bowtie}$ to work very efficiently is a novel data structure, termed *intersection iterator*, which acts as a mapping between the raster and vector files. The intersection iterator is constructed and processed in parallel which allows it to scale to extremely large datasets where others fail. In addition, it minimizes the disk access by reading only the required pixels once whereas existing raster- and vector-based systems typically read the entire raster data and sometimes more than once. RJ$_{\bowtie}$ is implemented as a relational operator in Spark which makes it easy to integrate with other operators such as selection, projection, and join to build a complex query tree that is optimized for distributed processing. We compare the proposed system to GeoTrellis [21] and Google Earth Engine [16] and show that it has over two 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 paper:

1) Defines a relational data model that represents both raster and vector data.
2) Proposes a new operator Raptor Join (RJ$_{\bowtie}$) that joins raster and vector data.
3) Formulates three predicates for joining points, lines, and polygons, with raster data.
4) Provides three real applications that use RJ$_{\bowtie}$ in complex analytical queries.
5) Proposes a new data structure, intersection iterator, and uses it to implement RJ$_{\bowtie}$ efficiently in Spark.
6) Runs a comprehensive experimental evaluation on real datasets to show the efficiency of RJ$_{\bowtie}$.

The rest of this paper is organized as follows: Section II covers the related work in literature. Section III describes the data and query model of the proposed system. Section IV details the algorithm used to implement the proposed operator. Section V compares the proposed operator to baseline methods such as GeoTrellis and Google Earth Engine. Section VI concludes the paper and discusses future work.

II. Related Work

A. Non-spatial Joins

The join operation [22] is a fundamental relational database query operation that can bring together two 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 [23] 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 paper, we make an analogy between traditional join algorithms and raster-vector-join algorithms to explain their limitations.

B. Spatial Join on Raster Data

Raster data is usually represented as multidimensional arrays and it is analysed using operations [24] such as map algebra and map overlay. Map algebra can be described as a raster-raster join operation that combines two or more multidimensional arrays to produce another multidimensional array possibly of a different size. It may join a pixel in one raster layer to many pixels in other raster layers. On the other hand, a map overlay operation often joins a pixel in one raster layer to a pixel in another raster layer based on their locations. If the two raster layers do not have matching resolutions, a regridding operation is applied on one dataset to match the other dataset. Systems such as SciDB [12], RasDaMan [13], GeoTrellis [14], ChronosDB [15], and Google Earth Engine [16] implement algorithms for raster operations that can process large amounts of raster data. However, none of these systems provide a sepcialized join operation for raster and vector data. They usually rasterize the vector data with a matching resolution and apply the map overlay operation.

C. 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, and contains. Spatial join algorithms for vector data include R-Tree join [25], Spatial Hash join [26], Partition Based Spatial Merge join (PBSM) [27], and many more [28], [29]. Efforts [30] have also been made to implement these spatial join algorithms in a distributed environment in order to process big data. SpatialHadoop [9], Hadoop-GIS [31], GeoSpark [10], and Simba [11] are a few systems that implement algorithms for vector data. 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.

D. Raster-Vector Joins

Raster and vector data are implemented using very different data and query processing models. This makes it difficult to perform a raster-vector join. There have been efforts that join them by using a new data structure to store or query raster and vector data. An early work on combining raster and vector data [32] proposes a hybrid data structure to store both raster and vector data. It requires an offline preprocessing step that converts both datasets to an intermediate form before it performs any processing. The additional preprocessing step makes it unattractive so our proposed approach is designed to work on the raster and vector data without any reformatting or preprocessing. Another work on querying raster and vector data [33], [34] focuses on compact representation of raster in memory using a new tree-like data structure and performing range queries with vector data represented by an R-tree. On the other hand, the proposed approach does not require any index construction while achieving a better query performance. The proposed system can also be used to process other queries in addition to range queries.

Other efforts to join raster and vector data often include converting them to the same format and using join algorithms for that format. Systems like PostGIS and QGIS [35] support vector and raster data on the user interface but they internally rely on two isolated libraries, one for each type of data. A few of these efforts like ArcGIS [36], Zhang *et al* [37], [38] and Zhao *et al* [39] focus on joining raster and vector data to only solve the *zonal statistics* problem. Scanline algorithm [19], [40] 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 [41] (a poster), we tried a straightforward parallelization of the Scanline approach that showed some performance improvement but it was limited to zonal statistics query for polygons. The proposed system can work with points, lines and polygons and is modeled as a relational operator which allows it to process ad-hoc queries that involve a complex relational query.

III. Problem Formulation

In this section, we formally define the problem that this paper addresses. We begin by defining the data model, for both *raster* and *vector* data. We then define the new Raptor Join operator for points, lines, and polygons. After that, we continue the formalization by defining inner and outer Raptor joins. Finally, we provide three real applications that use Raptor Join in complex queries.

A. Data Model

We use a relational data model to represent both raster and vector data. Note that though we define them in a relational format for formalization purposes, the proposed system neither requires data to be input in a tabular form nor does it it internally convert it into a relational form.

Raster dataset $\mathcal{R}$: is a collection of images that are input to the Raptor Join algorithm. Each image $R \in \mathcal{R}$, identified by a unique ID R_{id}, is a matrix of cells organized into rows and columns where each cell is called a pixel and contains a value representing some information. Each pixel in the image can be represented by the tuple *(x, y, m, bb)* where y is the row identifier and x is the column identifier of the pixel in the image, m is its measure value, and *bb* is the bounding box of the pixel which represents the geographical region that this pixel covers. It can be calculated from the row and column identifier of the pixel as and when needed. An image is also divided into equi-sized sub-arrays called *tiles*, and each tile is assigned an identifier, T_{id}. A pixel may belong to only one of the tiles whose identifier can be calculated using the row and column identifiers of the pixel. The raster data also has some additional information called metadata M associated with it. Metadata contains information such as the spatial extent, resolution and the coordinate reference system of the raster. It also contains two mappings, 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 while the $\mathcal{G}2\mathcal{W}$ mapping does the opposite. The latter can also be used to calculate the bounding box *bb* from the pixel location (x, y). These mappings are used to map data between the vector and raster datasets.

We can thus represent the raster dataset, $\mathcal{R}$ as a set of *(R_{id}, T_{id}, x, y, m, bb, M)* tuples by simply flattening all the pixels and rasters in it.

$$\mathcal{R} = \{(R_{id}, T_{id}, x, y, m, bb, M)\} \tag{1}$$

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 ($\mathcal{G}2\mathcal{W}$) and world-to-grid ($\mathcal{W}2\mathcal{G}$) transformations which are two affine transformations that convert a pixel location in the grid to a point location in the world, and vice-versa.
- Coordinate Reference System (CRS) which describes the projection that maps the Earth surface to the world coordinates as defined by the ISO-19111 standard [42]

The above information can be used to calculate other metadata for the raster layer such as the resolution p which is the pixel size in real world, and the number of tiles in the file ($numTiles$). We reiterate that the metadata is not replicated for each pixel, but is stored only once in memory and is associated to pixels through an accessor function. Furthermore, the metadata along with the pixel location (x, y) are used to compute the bounding box bb and tile ID T_{id} when needed.

Vector Dataset V: is defined as a collection of geometries that are input to the Raptor Join algorithm. The geometries in V can represent either points, lines, or polygons. Points are used to represent discrete data values. Each point is represented by the combination of a latitude and longitude value. Lines or linestrings are used to represent linear features, such as rivers, roads, and trails. Each line is represented by an ordered list of at least two points. Polygons are used to represent areas such as the boundary of a city, lake, or forest. Polygon features 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 paper, we represent 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 which can be a point, line, or a polygon.

$$V = \{(g_{id}, g)\} \tag{2}$$

B. Query Model

In this section, we formally define the *Raptor Join* operator and show how it can bring together raster and vector data.

Raptor Join RJ$_{\bowtie}$: is defined as a spatial join algorithm that takes as input a vector dataset V, a raster dataset $\mathcal{R}$, and a predicate θ. In this paper, we define three predicates θ_{point}, θ_{line}, and $\theta_{polygon}$ for the three types of geometries as follows. Each predicate θ takes two inputs, a geometry and a pixel, and returns true if the geometry and pixel match.

Point Predicate (θ_{point}): The point predicate takes as input a point geometry, defined by a location (x, y) and a pixel. It returns true if the point location lies inside the bounding box (bb) of the pixel. Figure 1a, shows an example of θ_{point} where the two shaded pixels match the two point locations.

Line Predicate θ_{line}: θ_{line} takes as input a line string and a pixel. It returns true if the line intersects the *crosshair* of the pixel. We define the crosshair of a pixel as the two lines splitting the bounding box in half, horizontally and vertically, as depicted by dotted lines in Figure 1b. That figure gives an

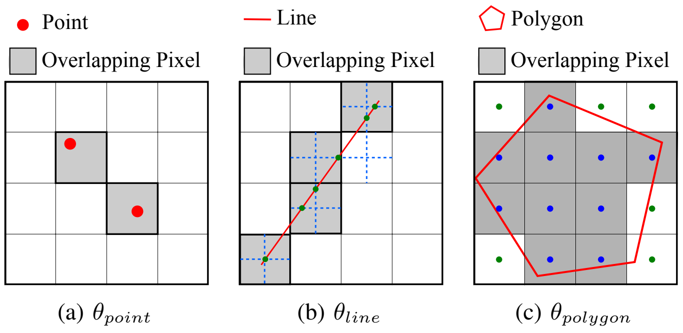


Fig. 1: Three predicates θ for Raptor Join

example of the θ_{line} predicate where the shaded pixels denote the pixels that match the line according to this definition.

Polygon Predicate $\theta_{polygon}$: The $\theta_{polygon}$ predicate takes a polygon and a pixel and returns true if the center of the pixel bounding box is inside the polygon boundary. Figure 1c 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.

For any of the three predicates, RJ$_{\bowtie}$ outputs a set of $(g_{id}, g, R_{id}, x, y, m)$ 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.

$$\mathcal{R} \bowtie_{\theta} V = (g_{id}, g, R_{id}, x, y, m) \tag{3}$$

A slight variation of the RJ$_{\bowtie}$ output includes only the geometry ID (g_{id}) but not the geometry. While it can be simply obtained by dropping the geometry field, we show later that it can be obtained more efficiently in the case when only g_{id} is needed.

$$\mathcal{R} \bowtie'_{\theta} V = (g_{id}, R_{id}, x, y, m) \tag{4}$$

C. Types of Join

Similar to traditional database join operators, Raptor Join is defined for inner and outer joins as detailed below.

Inner Join: is a Raptor join where the resulting tuples include only those pixels and geometries that satisfy the predicate θ. Therefore, any pixel that does not match with a geometry in the vector layer and any geometry that does not match with the raster layer are not included in the output.

Left Outer Join: (Also called Raster Outer Join) is a Raptor join where the resulting tuples include all pixels but only those geometries that satisfy the predicate θ. Therefore, any geometry that does not match with the raster layer is not included in the output. However, if a pixel does not match with any geometries, it will still be included in the answer with `null` values for g and g_{id}.

Right Outer Join: (Also called Vector Outer Join) is a Raptor join where the resulting tuples include only those pixels that satisfy the predicate θ but all geometries. In other words, a pixel that does not match with any geometries is not included, but a geometry that does not match any pixels is included in the answer with `null` values for (R_{id}, x, y, m).

Full Outer Join: is a Raptor join where the resulting tuples include all the pixels and geometries. `null` values are added to the non-matching tuples as described in the left and right outer joins.

D. Applications

Raptor Join can be used in a wide range of applications. We mention three real applications below and discuss how they can be implemented using $\mathrm{RJ}_{\bowtie}$.

1) Combating Wildfires: Wildfire is a natural disaster which cause 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. An initial step to study the spread of wildfires is to divide the target geographic region into numerous polygons called $zones$, and then calculate various statistics such as mean, median, standard deviation, min, and max, of contributing factors for each zone. These statistics may or may not have associative and commutative properties, e.g., median is not associative or commutative. The contributing factors such as vegetation and fuel level are often available in the form of rasters. Once the statistics are calculated for all wildfire zones, this information is spatially joined with the $fire$ dataset. The $fire$ dataset is a collection of points where each point represents the geographic location of a wildfire along with its various attributes such timestamp of fire and intensity of fire. The resulting dataset is then used to train machine learning algorithms to predict the spread of wildfires, and in turn devise methods to deploy resources to suppress them. This analytic query can be implemented as a SQL query as follows:

```
SELECT * FROM Fire JOIN
(SELECT gid, g, Statistics(m)
 FROM Zones RJ⋈ Vegetation
 GROUP BY gid) AS ZoneStats
WHERE ST_Contains(ZoneStats.g, Fire.Location)
```

Where, the function `Statistics` computes all the statistics needed for the analysis, and `ST_Contains` tests if the polygon $ZoneStats.g$ contains the point $Fire.Location$.

2) Crop Yield Mapping: The problem of crop yield mapping in agriculture [43] studies the crop yield of various agriculture fields using NDVI (Normalized Difference Vegetation Index), 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. NDVI needs to be calculated for all pixels that overlap the agricultural fields under study for a period of time ranging over multiple years. Once NDVI is calculated for the all the agricultural fields, it results in a time series of NDVI values per pixel. The n^{th} percentile of the NDVI values per pixel per year is calculated. The standard deviation of these n^{th} percentile NDVI values per field over multiple years is then used to classify crop yield into various classes. This query can be expressed using the following SQL query:

```
SELECT gid, STDEV(p) FROM (
 SELECT gid, g, x, y, Percentile(NDVI, n) AS p
 FROM (SELECT gid, g, x, y,
     (NIR.m−Red.m) / (NIR.m+Red.m) AS NDVI
    FROM Fields RJ⋈ (NIR UNION Red))
 GROUP BY gid, g, x, y)
GROUP BY gid, g
```

The inner most `SELECT` query computes the NDVI values near-infrared (NIR) and red measurements. Notice that `NIR UNION Red` is a shorthand for computing the union of two raster datasets. The second level computes the n^{th} percentile per pixel. Finally, the top-level query computes the standard deviation per field. Notice that just the NDVI calculation in raster-based systems, e.g., SciDB [17], requires nearly 40 lines of code which shows the simplicity of using relational algebra as compared to linear algebra for simple queries like this one.

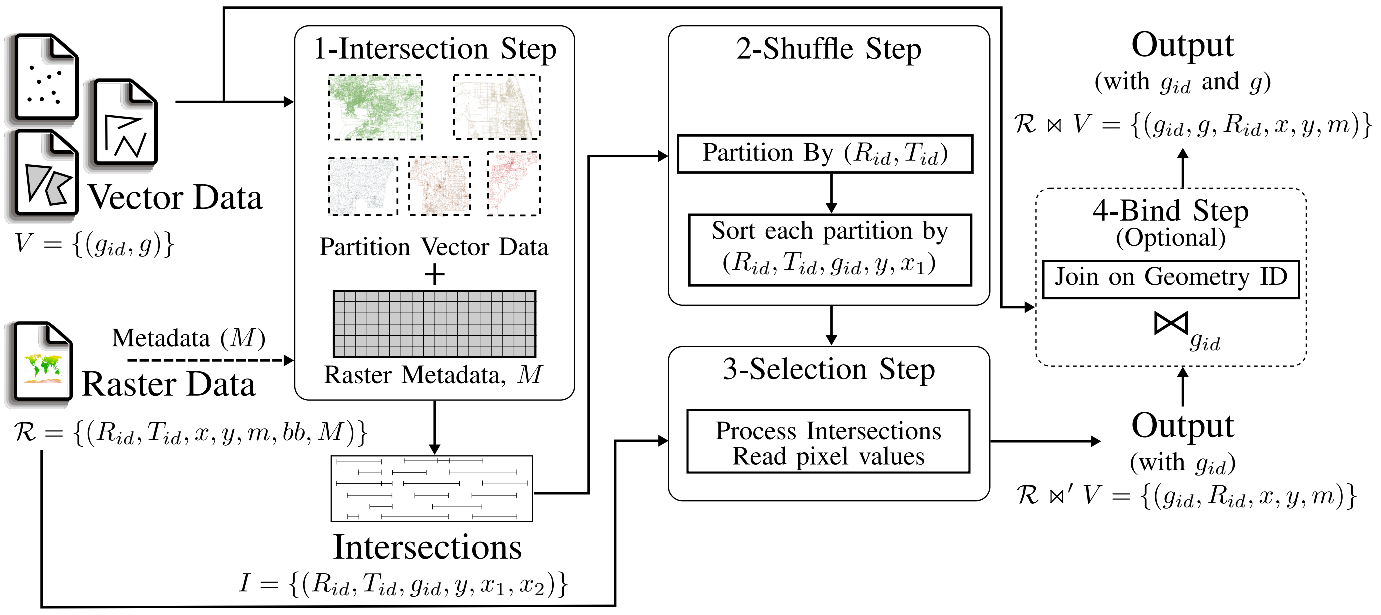
3) Areal Interpolation: Areal Interpolation is the problem of estimating a function in arbitrary areas, e.g., city boundaries, based on values in other non-aligned areas, e.g., census tracts. One application of this problem is to estimate the population of arbitrary regions using landcover data [1]. 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 [1] uses the National Land Cover Database (NLCD) [44] raster dataset as a reference to disaggregate the population counts into pixels and then aggregate them back into target polygons. This process requires to compute the histogram for each polygon in the TRACT and the ZIP codes datasets on the NLCD dataset. This can be done using the following SQL statement:

```
SELECT TRACT.gid, NLCD.m, COUNT(*)
FROM TRACT RJ⋈ NLCD
GROUP BY TRACT.gid, NLCD.m
```

IV. Raptor Join Design

This section describes the algorithm used to implement the proposed *Raptor Join* operator. The key design objectives for $\mathrm{RJ}_{\bowtie}$ are: 1) **Efficiency:** $\mathrm{RJ}_{\bowtie}$ handles big raster and vector data. 2) **In-situ:** $\mathrm{RJ}_{\bowtie}$ does not require a preprocessing phase for loading or converting the input data. 3) **Fully distributed:** $\mathrm{RJ}_{\bowtie}$ runs in a fully distributed mode for scalability.

The key idea of $\mathrm{RJ}_{\bowtie}$ is to resemble a sort-merge join algorithm which can scan 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



Vector Data
$V = \{(g_{id}, g)\}$
Raster Data
$\mathcal{R} = \{(R_{id}, T_{id}, x, y, m, bb, M)\}$
Metadata (M)
1-Intersection Step
Partition Vector Data
+
Raster Metadata, M
Intersections
$I = \{(R_{id}, T_{id}, g_{id}, y, x_1, x_2)\}$
2-Shuffle Step
Partition By (R_{id}, T_{id})
Sort each partition by ($R_{id}, T_{id}, g_{id}, y, x_1$)
3-Selection Step
Process Intersections Read pixel values
4-Bind Step (Optional)
Join on Geometry ID
$\bowtie_{g_{id}}$
Output (with g_{id} and g)
$\mathcal{R} \bowtie V = \{(g_{id}, g, R_{id}, x, y, m)\}$
Output (with g_{id})
$\mathcal{R} \bowtie' V = \{(g_{id}, R_{id}, x, y, m)\}$

Fig. 2: Implementation Overview of Raptor Join

$O(|V| \cdot |\mathcal{R}|)$, 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 intersection iterator, described shortly) and the second term resembles the linear merge step. The details of the analysis are omitted due to limited space but interested reader can refer to this technical report [45] for details.

To accomplish this idea, $\text{RJ}_{\bowtie}$ exploits the inherent structure of the raster data and produces an intermediate representation of vector data, called the *Intersections Iterator*. This representation is computed to match the structure of the raster data in terms of tiles, rows, and columns. Furthermore, to produce the intersections iterator, we only need to process the vector data and the *metadata* of the raster data which means that the intersection iterator computation time depends only on the vector data size while the raster dataset needs to be scanned at most once.

The $\text{RJ}_{\bowtie}$ algorithm runs in four steps as shown in Figure 2: *Intersection*, *Shuffle*, *Selection*, and *Bind*. The first step *intersection*, takes as input the vector dataset, the metadata of the raster dataset and a predicate to produce a collection of intersections between the raster and vector data called the *Intersections Iterator*. In the next step *shuffle*, the intersections produced in the previous step are grouped and partitioned among machines to ensure that the raster data is read only once. The third step *selection*, uses these grouped and partitioned intersections to read actual pixel values from the raster dataset. It produces a collection of tuples which constitute of identifiers to a geometry in the vector dataset and a raster file in the raster dataset, along with the value and location of the pixel that satisfies the predicate. The fourth *bind* step joins the output of the previous step with the vector data (V) on g_{id} to bring back the geometry definition. This optional step is only executed if the geometry is needed in the rest of the query tree or if the join type is right outer join or full outer join. The four steps are further detailed below, followed by a discussion on how to support inner and outer joins.

A. *Intersection Step*

The input to this step is the vector dataset (V) and the metadata (M) of the raster dataset and the output is a set of intersections between the raster and vector data. Each intersection represents a range of pixels that overlap a geometry, which is called the *intersection iterator*.

First, the input vector dataset is assumed to be stored on the Hadoop distributed file system (HDFS) which splits the file into blocks of 128 MB each, by default. The default Spark behavior is to create one partition per block. However, due to the computational overhead of this step, we change this behavior to create one block per 16 MB. If the number of input partitions is still less than the number of executors in the cluster, we further repartition the data to ensure that the executors are fully utilized and no workers stay idle.

After the vector data is partitioned, the geometries are transformed on-the-fly to the coordinate reference system (CRS) of the raster layer to ensure that they are ready to process. The actual transformation depends on the CRS of both the vector and raster layers and is defined by the ISO 19111:2019 standard [42].

Once the file is partitioned and transformed, each partition is processed with the metadata to produce the intersection iterator. The intersections are computed in a lazy manner as they are needed and do not need to be kept in memory unlike existing models that transform the entire dataset before the join step. The intersections are computed based on the predicate θ which is defined based on the geometry type as described in Section III. The computation for the three predicates is detailed below.

1) Point Intersections, θ_{point}: The intersection of a point with raster layer is defined as the pixel whose bounding box BB contains the point. To compute point intersections, each point is mapped to the raster layer using the $\mathcal{W}2\mathcal{G}$ transformation. The resultant point coordinates are then used to determine the pixel that contains the point. This results in a set of *(R_{id}, g_{id}, y, x)* tuples, one for each point, where R_{id} identifies the raster file, y and x are the row and column identifiers of the intersecting pixel in the raster layer and g_{id} is the unique identifier of the point. These tuples are further processed to add T_{id} which identifies the subsequent tile in the raster layer to which the pixel belongs. T_{id} can be calculated using the y and x values of the pixel along with the tile width and height ($M.tw$ and $M.th$). Finally, they are output as collection of *(R_{id}, T_{id}, g_{id}, y, x_1, x_2)* tuples, where $x_1 = x_2$ refer to the column of the intersecting pixel. Even though only x_1 is sufficient to identify the column, we add x_2 to maintain uniformity among the intersections of different geometry types.

2) Line Intersections, θ_{line}: 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 has been used in the field of computer graphics. The limitation of the mid-point algorithm is that it assumes that the start and end points of the line are at integer coordinates, which is generally not true for lines in spatial datasets where the points can be anywhere in the pixel bounding box. However, the main logic of the algorithm stays the same and it is simple to extend the algorithm in that direction as needed.

To compute line intersections, the start and end points of each line are mapped to the raster layer using the $\mathcal{W}2\mathcal{G}$ transformation. These coordinates are then used as input to the mid-point line drawing algorithm which outputs a list of pixel coordinates whose centers lie on/closest to the line. This results in a collection of *(R_{id}, g_{id}, y, x)* tuples, where R_{id} identifies the raster file, y and x are the row and column identifiers of the intersecting pixel and g_{id} is a unique identifier for the line.

Similar to points, the intersections generated above for lines identify individual pixels. We convert them to intersection ranges to maintain output uniformity for the next step. Furthermore, this usually reduces the overall size since we can merge consecutive intersections in the same row in one range. To accomplish that, we first sort all the intersections lexicographically by (y, g_{id}, x). Then, we scan the list from the end and perform the following tests. If we find two or more consecutive intersections that have the same (T_{id}, g_{id}, y) and their x coordinates are consecutive, we merge them into one range by marking all the intersections for removal except the first and last ones. To mark an intersection for removal, we set its T_{id} to $R.M.numTiles$ which is an invalid tile ID. This approach allows us to mark a record for removal in constant time. On the other hand, if we find a single intersection that cannot be merged with adjacent intersections, we convert it to a range by by appending a new intersection with the same exact coordinates. Similar to marking for deletion, inserting an intersection is a constant time operation. After the entire list is scanned, we sort the list again by (T_{id}, y, g_{id}, x). This will move all the intersections marked for removal to the end of the list which can be removed by just reducing the list size. After this step, each pair of consecutive intersections are converted to a range in the form $(R_{id}, T_{id}, g_{id}, y, x_1, x_2)$.

3) Polygon Intersections, $\theta_{polygon}$: The intersection of a polygon with the raster layer is defined as the pixels whose bounding box center is inside the polygon. To find these pixels, we iterate over each line segment in a polygon and calculate all the intersections between the line segment and the horizontal scan lines that go through the center of pixels. First, given the start and end points, of the line segment, we use the $\mathcal{W}2\mathcal{G}$conversion to identify the first and last scan lines that intersect the line segments. Then, we iterate over these scan lines and for each one we use the $\mathcal{G}2\mathcal{W}$conversion to calculate the $latitude$ coordinate of the scan line. Given the $latitude$ and the line segment, we calculate the $longitude$ intersection in constant time. Finally, the intersection $longitude$ along with the scan line $latitude$ are processed with the $\mathcal{W}2\mathcal{G}$conversion to produce the (x, y) grid intersection coordinates which are appended to the list of intersections. After processing all the line segments, the output will be the list of all intersections between the boundaries of the polygons and the scan lines. At this stage, if we sort these intersections by (y, g_{id}, x), each pair of consecutive intersections will define a range inside the polygon g_{id} as detailed in [19]. However, we do not stop here for two reasons. First, one intersection range might span two tiles which would be inefficient to process since we will need to read two tiles instead of one. Second, we can further reduce the number of intersection ranges by merging consecutive ranges for the same polygon and removing empty ranges.

To optimize the list of intersections, we first sort all the intersection points by (y, g_{id}, x) which brings end points of ranges next to each other in the sort order. After that, the ranges are scanned from the end while performing the following three tests.

1) If a range is empty, i.e., its start and end are the same, both end points are marked for removal.
2) If a range spans more than one tile, i.e., its start and end have different T_{id}, the range is split across tile boundaries. This is done by inserting $2.(t-1)$ new intersections where t is the number of different tiles that the intersection goes through. Each pair of inserted intersections end at the end of one tile and start at the beginning of the next tile.

3) If two consecutive ranges have the same (T_{id}, g_{id}) and their x ranges are contiguous, they are merged together into one. To merge them, we mark the end of the first range and the start of the second range for removal.

To mark an intersection for removal, we change its tile ID to $R.M.numTiles$ which is an invalid tile ID after all valid tile IDs. All inserted intersections are appended at the end of the list. After the intersection ranges are optimized, we sort them by (T_{id}, y, g_{id}, x) which moves all removed intersections at the end and brings together the two end points of each intersection range next to each other.

The final output of this step is an intersection iterator that iterates over tuples of the form $(R_{id}, T_{id}, g_{id}, y, x_1, x_2)$ and the intersections are ordered by (T_{id}, y, g_{id}, x_1).

B. Shuffle Step:

It takes as input the *Intersections Iterator* computed in the previous step, and partitions it across machines. The goal is to bring together all intersections in the same tile in one machine to ensure that each tile in the raster file is read at most once. Further more, it orders the intersections in each partition to match the order of the pixels in the raster file which ensures that each tile is processed sequentially.

The intersections are first partitioned by (R_{id}, T_{id}) which means that all the intersections belonging to the tile identified by T_{id} in raster file R_{id} are sent to the same machine. This ensures that when intersections are processed, any tile in the raster files will be read only by a single machine. Notice that since we use hash partitioning, multiple tiles might be assigned to the same partition. For example, if the input raster data contains one million tiles while the cluster has only 100 machines, we will have 100 partitions in the output with approximately 10,000 tiles per partition. Inside each partition, we sort the intersections by $(R_{id}, T_{id}, g_{id}, y, x_1)$. Sorting by (R_{id}, T_{id}) means that all intersections belonging to a particular tile will be processed consecutively. This allows the algorithm to read the tile in memory and process all corresponding intersections before reading another tile in memory. Thus the partitioning and sorting by (R_{id}, T_{id}) guarantees that the intersections can be processed with only one scan over the raster file and while loading one tile in memory at a time.

The sorting based on (y, x_1) allows the algorithm to take advantage of cache locality when processing the intersections. Given that the pixels in each tile are stored in row-major order, data would be read in cache row by row. Thus, it is more efficient to process all pixel values for one row in tile before moving to the next row. Since, the size of cache is unknown, sorting by x_1 helps minimizing cache misses.

To summarize, this step ensures that the raster file is scanned only once, while also maintaining a load balance among the machines.

C. Selection Step:

It takes as input the partitioned *Intersections Iterator* from the last step. For each partition, it processes all intersection tuples to read the pixel values from the raster. It outputs a set of $(g_{id}, R_{id}, x, y, m)$ tuples which can be further processed based on the query.

Each tuple in the partitioned *Intersections Iterator* is of the form (R_{id}, T_{id}, g_{id},y,x_1,x_2). This step starts by loading the tile T_{id} of the first tuple in memory. It then proceeds to read all pixel values in the row y and column range $x_1 - x_2$ from the tile. For each pixel read, it outputs a corresponding tuple (g_{id}, R_{id},x,y,m), where m is the pixel value. This step can also be easily adapted to output a value computed using the values of the same pixel from multiple rasters or raster with multiple bands. A common example for such a case is NDVI calculation, which is computed using pixel values from the red and near-infrared bands of the spectral reflectance rasters.

Since the last step sorted all tuples based by (R_{id}, T_{id}), consecutive tuples belong to the same tile. Once all the tuples corresponding to the current tile are processed, only then will the next tile be read into memory. This ensures that the raster dataset is only scanned over once, hence making this approach very efficient. In fact, this approach is optimal in terms of disk access since it only reads the necessary tiles from the raster file and each tile is loaded exactly once.

D. Bind Step (Optional):

The goal of this step is to bind each g_{id} to its corresponding geometry. This step can be skipped if the geometry (g) is not needed by subsequent steps. It runs as an equi-join operation on g_{id} between the vector dataset V and the output of the previous step.

E. Implementing Inner and Outer Joins

Inner Join: The algorithm described above inherently performs an inner join. In the intersection step, if a geometry does not overlap any pixels, no corresponding intersections will be generated and it will not appear in the output. Similarly, if a pixel is not included in any intersection, it will not be in the output.

Left (Raster) Outer Join: should include all raster pixels in $\mathcal{R}$ even if they do not overlap any geometries. In order to support it, we modify the *selection step* so that it keeps a bitmap with size equal to the tile width tw that is initially set to zeros. As the intersections are processed, the bit that corresponds to each read pixel is set to one. Before moving from one row to the next, the bit mask is checked for unset bits. For each unset bit, an output record is written to the output with $g_{id} = null$. After that, the bitmap is reset to zeros. If an entire row does not contain any intersections, all its pixels are written to the output in the same way.

Right (Vector) Outer Join: should include all geometries in V even if they do not overlap any pixels. To support it, we apply the *bind step* and use an outer join to ensure that all geometries are in the output. The resulting tuples will then contain information for all geometries with *null* values for pixel and raster information.

Full Outer Join: should be the union of left and right outer joins. To implement it, we simply apply both techniques described above for left and right outer joins. That is, we use

TABLE I: Vector and Raster Datasets

Vector datasets

Dataset	$\lvert V\rvert$	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
Counties	3k	52k	978 KB	Polygons	US
States	49	165k	2.6 MB	Polygons	US
Boundaries	284	3.8m	60 MB	Polygons	World
TRACT	74k	38m	632 MB	Polygons	US
ZCTA5	33k	53m	851 MB	Polygons	US
Parks	10m	336m	8.5 GB	Polygons	World

Raster datasets

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

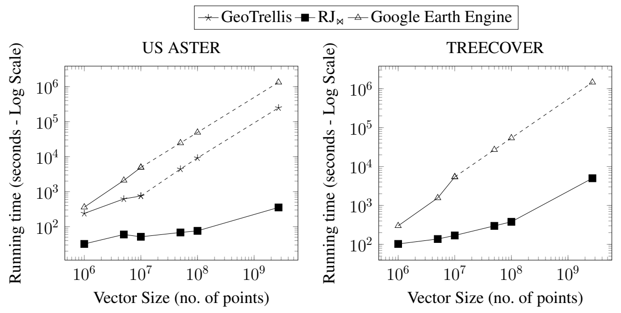


Fig. 3: Comparison of total running time of RJ$_\bowtie$, GeoTrellis and Google Earth Engine for point features

the bitmaps to include all pixels and the outer join in the bind step to include all geometries.

V. Experiments

This section provides an extensive experimental evaluation of the proposed algorithm for the Raptor Join (RJ$_\bowtie$) operator. We compare the proposed RJ$_\bowtie$ algorithm to Google Earth Engine (GEE) [16] a free public (but closed source) system that is backed by the Google Cloud Platform, GeoTrellis [14] an open source raster GIS system with an extension to run on Spark, and RZS, a fully distributed Hadoop-based algorithm for zonal statistics based on the idea in [41]. RZS is limited to the zonal statistics operation on polygons so it is tested only with that operation. We show that the proposed RJ$_\bowtie$ provides up-to two orders of magnitudes performance gain over Google Earth Engine, GeoTrellis, and RZS. 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 Google Earth Engine and GeoTrellis we need to manually split big files into smaller ones, process each one separately, and combine the results, to work around system limitations.

Section V-A describes the setup of the experiments, the system and the datasets used. Section V-B compares the proposed RJ$_\bowtie$, GEE, GeoTrellis, and RZS based on the total running time. Section V-C discusses the vector and raster dataset ingestion time for each of these methods. Section V-D discusses how RJ$_\bowtie$ was used to implement the applications discussed earlier. Section V-E shows the effect of partitioning vector data on the total running time of RJ$_\bowtie$, while Section V-F shows the scalability of RJ$_\bowtie$ for different statistical functions.

A. Setup

We run RJ$_\bowtie$, GeoTrellis, and RZS 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 [16] but it does not make available the actual resources used by each query.

For the zonal statistics problem, we perform a raster-vector join and 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 I lists the datasets that are used in the experiments along with their attributes. All these datasets are publicly available. All vector datasets are available on UCR-Star [46], [47]. All raster datasets are also publicly available as described in [41]. Some of these datasets cover the entire world and some cover only the US as listed in the table.

For GeoTrellis, we used the *geotrellis-spark* package version 1.2.1, as described in its documentation. 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. 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.

B. Overall Execution Time

This parts compares RJ$_\bowtie$, RZS, GeoTrellis, and GEE based on the end-to-end execution time for the zonal statistics problem. This experiment is run for all the combinations of vector and raster datasets shown in Table I for polygons, while for points and lines we only show results for larger raster datasets, US Aster and Treecover. The join is implemented in GEE by using the *reduceRegions* function and for GeoTrellis by using the *ClipToGrid* function in Spark [48]. Since GeoTrellis does not support points and lines, we use the *buffer* operation on them which results in a polygon. The size of the buffer set to

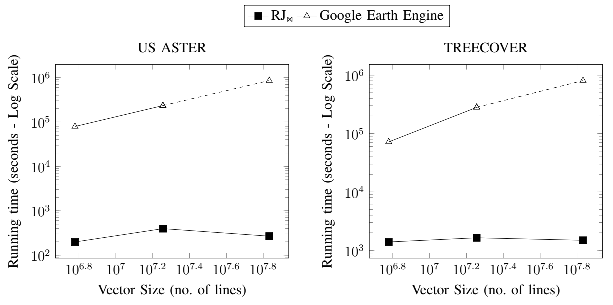


Fig. 4: Comparison of total running time of RJ$_\bowtie$, GeoTrellis and Google Earth Engine for line features

be equal to the resolution of the raster, i.e., the size of one pixel.

Points θ_{point}: The experiment results for the combination of raster datasets, US Aster and Treecover, and the points dataset, all_nodes can be seen in Figure 3. Since all_nodes is a dataset with 2.7 billion points, we show the scalability of the systems by conducting experiments on subsets of it as well. As can be seen from the figure, the proposed RJ$_\bowtie$ is atleast $7\times$ faster than GeoTrellis and $11\times$ faster than GEE. For large datasets, RJ$_\bowtie$ is up-to four orders of magnitude faster. The dotted lines indicate an extrapolation that we did to estimate the running time for large datasets since both GEE and GeoTrellis could not process the entire dataset in one run. Thus, we had to split the file into smaller split with one million points each, for a total of 2,700 splits, and process each one separately. We process only the first few splits and extrapolate based on the average time per split.

GeoTrellis does not scale to large raster datasets, such as Treecover and large vector datasets due to memory errors. It can only process up-to 10 million points at once. This happens because it uses the raster-based approach that rasterizes geometries on the fly and hence fails to scale for big vector data. It may also require to read the same raster tile multiple times for different polygons. RJ$_\bowtie$ overcomes this by partitioning the intersections iterator across machines in such a way that raster data needs to be read only once. Even though **GEE** can scale to large raster datasets, it can only process 1 million points at once. It gives an error *Computed value is too large* for more points than that. Since, the knowledge of how it implements this join is not public, it is difficult to ascertain the exact reason behind this error but it makes it clear that it is not designed to handle large vector data.

Lines θ_{line}: Figure 4 shows the results for for combination of all raster datasets with all line datasets. As can be seen in the figure, we only show results for RJ$_\bowtie$ and GEE. **GeoTrellis** does not support the raster-vector join operation for lines even after we tried to use the buffer as a workaround. For RJ$_\bowtie$ and GEE, the figure shows that RJ$_\bowtie$ achieves at least three orders of magnitude performance gain over GEE, even though GEE may run on up-to 1000s of machines [16]. We also had to divide the datasets into smaller files, since GEE can run on only up-to two million geometries at once.

Polygons $\theta_{polygon}$: Figure 5 shows the experiments results for the zonal statistics problem with polygons on all systems. This is the only operation that RZS supports so we include it here. As can be seen, RJ$_\bowtie$ is consistently faster with several orders of magnitude speedup for large vector and raster datasets. Furthermore, GeoTrellis failed for the largest vector datasets (Parks) and the biggest raster dataset (TreeCover). We also omit its result for the combination of US Aster and boundaries, since it ran for more than 24 hours. While **GEE** is able to scale to large raster and vector datasets, it only shows comparable performance for smaller vector datasets, counties and states. For larger vector datasets, RJ$_\bowtie$ is up-to three orders of magnitude faster than GEE. Notice that there are a few points where the running time decreases as the vector dataset increases. This is due to the different coverage of raster and vector datasets. Therefore, we might increase the dataset in size, but only a small portion of it actually gets processes due to the partial overlap with the other dataset.

When compared to **RZS**, RJ$_\bowtie$ provides a comparable performance for small and medium datasets and up-to two orders of magnitude faster for large datasets. This is to be expected since they both use the concept of intersections to reduce the number of scans on raster data. However, performance gain in RJ$_\bowtie$ is attributed to the shuffle step which ensures that raster data is scanned only once. Furthermore, RZS can perform only the zonal statistics query on polygons while RJ$_\bowtie$ is a general purpose operator that can support zonal statistics as one of endless number of queries that can be built on it.

C. *Ingestion Time*

Figure 6 shows the ingestion time of the raster and vector datasets for the proposed RJ$_\bowtie$ algorithm and GEE. RJ$_\bowtie$ and GeoTrellis both require raster and vector datasets to be loaded 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. RJ$_\bowtie$ is two to three orders of magnitude faster than GEE for ingesting raster data as shown in Figure 6a. For vector data, it is one to three orders of magnitude faster than GEE as can be observed from Figure 6b. The reason for that is that RJ$_\bowtie$ implements an in-situ approach, hence, making it a perfect choice for ad-hoc queries.

D. *Applications*

This section shows how we implemented the applications discussed in Section III using RJ$_\bowtie$. In these examples, RJ$_\bowtie$ is combined with other relational operators in a complex query tree.

The first application for *combating wildfires* as implemented in [49] requires to join over 3 million polygons with 23 rasters from *landfire.gov* each containing over a billion pixels, using the predicate $\theta_{polygon}$. It computes statistics such as max, min, sum, count, mean, mode and median to be calculated for combination of each raster and each polygon. It then requires to be spatially joined with *VIIRS* fire dataset containing 4 million points representing actual fire locations. The total

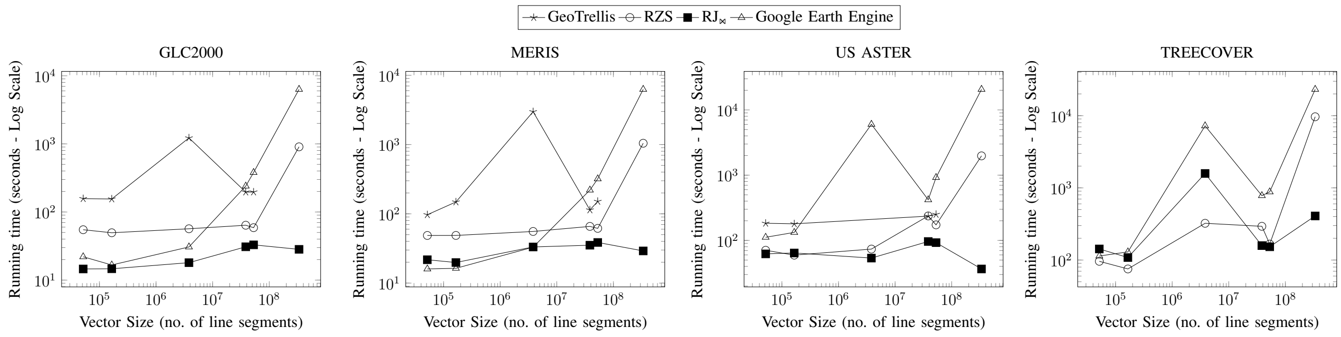
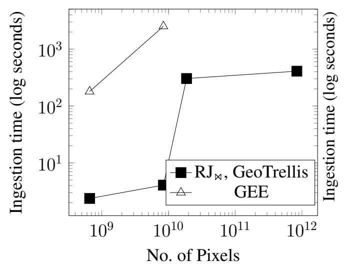
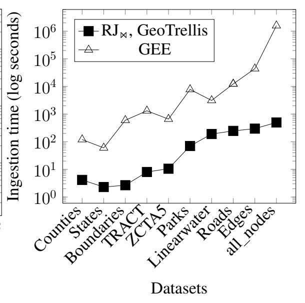


Fig. 5: Comparison of total running time of RJ$_{\bowtie}$, GeoTrellis, RZS and Google Earth Engine for polygon features



(a) Raster Dataset



(b) Vector Dataset

Fig. 6: Ingestion Time

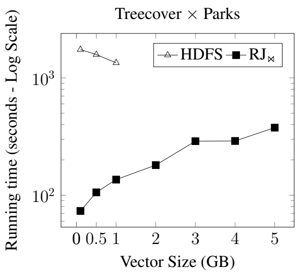


Fig. 7: Vector Partitioning

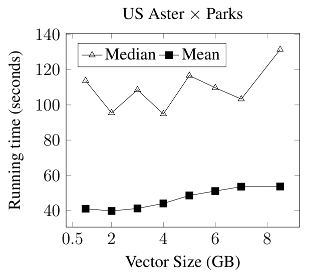


Fig. 8: Median vs Mean

time taken to calculate statistics and perform the spatial join using RJ$_{\bowtie}$ was about 2 hours.

For the second application of *crop yield mapping*, we calculated the NDVI for 360k agricultural fields in California using USGS Landsat 8 ARD surface reflectance images. There were 20,000 images amounting to about 1.8 TB, each for the red and NIR bands (required to compute NDVI) ranging over the temporal period of 6 years from 2014 - 2019, each having a resolution of 5000 $\times$ 5000 pixels per file. We computed NDVI for each pixel in each agricultural field and computed the standard deviation for the resulting time series of NDVI values per pixel. RJ$_{\bowtie}$ algorithm was able to complete this in 3 hours while it took GEE over 30 hours to do the same.

The third application of *areal interpolation* required to join the National Land Cover (NLCD) raster dataset, of resolution 161,190 $\times$ 104,424, with 74k TRACT polygons to estimate their population. Since, the baseline used by the developers of this application was a vector-based Python implementation, we tested RJ$_{\bowtie}$ on single machine as well. 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 of that work were able to scale their work to the entire US which took under 2 hours on a single machine.

E. Partitioning Vector Data

This experiment studies the effect of partitioning vector data on the total running time of the proposed RJ$_{\bowtie}$ algorithm. We compare using the default Spark partitioning, that creates a partition for each 128 MB block, against the one proposed in RJ$_{\bowtie}$ 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, Treecover and subsets of vector dataset, Parks. As can be seen in Figure 7, using the RJ$_{\bowtie}$-based partitioning allows the algorithm to run 100 times faster for small datasets due to the additional repartitioning 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.

F. Statistical Functions

This experiment compares the time taken to compute a commutative and associative statistical function such as mean to a non-commutative and non-associative one such as the median. MapReduce-based systems often take advantage of the associative and commutative properties of a function and compute a part of the function locally on each worker node and then globally combine these results at the master node. This cannot be done for non-associative and non-commutative functions, making them difficult to compute. In case of raster-vector joins, this requires all the required pixels to be kept in memory until the function is performed. The lazy processing of *Intersections Iterator* during the *selection* step allows RJ$_{\bowtie}$ to compute both kinds of functions for large datasets without running out of memory, as shown in figure 8.

VI. Conclusion and Future Work

The paper 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 data structure *Intersections Iterator*. 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 four steps, namely, *intersections*, *shuffle selection*, and *bind*. The *intersections* step computes *Intersections Iterator* that serves as a mapping between raster and vector. The *shuffle* step partitions this data structure across machines in such a way that each tile in the raster dataset is scanned by only one machine. The *selection* step processes the partitioned *Intersections Iterator* to read the required pixel values from the raster dataset. The *bind* step is optional and is used to add the geometry field in the result if needed. We run extensive experiments for the system against GeoTrellis and Google Earth Engine on large raster and vector datasets to show its scalability and performance gain. In the future, we plan on extending the system to work with machine learning algorithms and answering approximate queries.

References

[1] M. Reibel and A. Agrawal, "Areal interpolation of population counts using pre-classified land cover data," *Population Research and Policy Review*, vol. 26, no. 5-6, pp. 619–633, 2007.

[2] M. B. Joseph *et al.*, "Spatiotemporal prediction of wildfire size extremes with bayesian finite sample maxima," *Ecological Applications*, vol. 29, no. 6, 2019.

[3] O. Ghorbanzadeh *et al.*, "Spatial prediction of wildfire susceptibility using field survey gps data and machine learning approaches," *Fire*, vol. 2, no. 3, p. 43, 2019.

[4] G. D. Jenerette *et al.*, "Regional Relationships Between Surface Temperature, Vegetation, and Human Settlement in a Rapidly Urbanizing Ecosystem," *Landscape Ecology*, vol. 22, pp. 353–365, 2007.

[5] ——, "Ecosystem Services and Urban Heat Riskscape Moderation: Water, Green Spaces, and Social Inequality in Phoenix, USA," *Ecological Applications*, vol. 21, pp. 2637–2651, 2011.

[6] D. Haynes, S. Manson, and E. Shook, "Terra Populus' Architecture for Integrated Big Gepspatial Services," *Transactions on GIS*, 2017.

[7] D. Haynes *et al.*, "High Performance Analysis of Big Spatial Data," in *Big Data*, Santa Clara, CA, Nov. 2015, pp. 1953–1957.

[8] H. Saadat *et al.*, "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*, vol. 66, no. 5, pp. 608–619, 2011.

[9] A. Eldawy and M. F. Mokbel, "SpatialHadoop: A MapReduce Framework for Spatial Data," in *ICDE*, Apr. 2015, pp. 1352–1363.

[10] J. Yu, M. Sarwat, and J. Wu, "GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data," in *SIGSPATIAL*, Seattle, WA, Nov. 2015, pp. 70:1–70:4.

[11] D. Xie *et al.*, "Simba: Efficient In-Memory Spatial Analytics," in *SIGMOD*, Jun. 2016.

[12] M. Stonebraker *et al.*, "SciDB: A Database Management System for Applications with Complex Analytics," *Computing in Science and Engineering*, vol. 15, no. 3, pp. 54–62, 2013.

[13] P. Baumann *et al.*, "The multidimensional database system rasdaman," in *SIGMOD*, Seattle, WA, Jun. 1998, pp. 575–577.

[14] A. Kini and R. Emanuele, "Geotrellis: Adding Geospatial Capabilities to Spark," 2014.

[15] R. A. R. Zalipynis, "Chronosdb: distributed, file based, geospatial array dbms," *PVLDB*, vol. 11, no. 10, pp. 1247–1261, 2018.

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

[17] G. Planthaber, M. Stonebraker, and J. Frew, "Earthdb: scalable analysis of modis data using scidb," in *BIGSPATIAL*, 2012, pp. 11–19.

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

[19] A. Eldawy, L. Niu, D. Haynes, and Z. Su, "Large scale analytics of vector+raster big spatial data," in *SIGSPATIAL*, 2017, pp. 62:1–62:4.

[20] M. Zaharia *et al.*, "Apache spark: a unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.

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

[22] R. Ramakrishnan and J. Gehrke, *Database management systems*. McGraw-Hill, 2000.

[23] P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Computing Surveys (CSUR)*, vol. 24, no. 1, pp. 63–113, 1992.

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

[25] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 237–246, 1993.

[26] M.-L. Lo and C. V. Ravishankar, "Spatial hash-joins," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, 1996, pp. 247–258.

[27] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," *ACM Sigmod Record*, vol. 25, no. 2, pp. 259–270, 1996.

[28] M.-L. Lo and C. V. Ravishankar, "Spatial joins using seeded trees," in *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, 1994, pp. 209–220.

[29] N. Koudas and K. C. Sevcik, "Size separation spatial join," in *SIGMOD*, 1997, pp. 324–335.

[30] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu, "Sjmr: Parallelizing spatial join with mapreduce on clusters," in *CLUSTER*, 2009, pp. 1–8.

[31] A. Aji *et al.*, "Hadoop-gis: A high performance spatial data warehousing system over mapreduce," *PVLDB*, vol. 6, no. 11, pp. 1009–1020, 2013.

[32] D. J. Peuquet, "A hybrid structure for the storage and manipulation of very large spatial data sets," *Computer Vision, Graphics, and Image Processing*, vol. 24, no. 1, pp. 14–27, 1983.

[33] N. R. Brisaboa *et al.*, "Efficiently querying vector and raster data," *The Computer Journal*, vol. 60, no. 9, pp. 1395–1413, 2017.

[34] F. Silva-Coira *et al.*, "Efficient processing of raster and vector data," *Plos one*, vol. 15, no. 1, p. e0226943, 2020.

[35] "QGIS," http://www.qgis.org/, 2015.

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

[37] J. Zhang, S. You, and L. Gruenwald, "Efficient Parallel Zonal Statistics on Large-Scale Global Biodiversity Data on GPUs," in *BIGSPATIAL*, Bellevue, WA, Nov. 2015, pp. 35–44.

[38] J. Zhang and D. Wang, "High-Performance Zonal Histogramming on Large-Scale Geospatial Rasters Using GPUs and GPU-Accelerated Clusters," in *IPDPS Workshops*, Phoenix, AZ, May 2014, pp. 993–1000.

[39] G. Zhao *et al.*, "Parallelization and optimization of spatial analysis for large scale environmental model data assembly," *Computers and electronics in agriculture*, vol. 89, pp. 94–99, 2012.

[40] S. Singla *et al.*, "Raptor: Large Scale Analysis of Big Raster and Vector Data," *PVLDB*, vol. 12, no. 12, pp. 1950 – 1953, 2019.

[41] S. Singla and A. Eldawy, "(Poster) Distributed Zonal Statistics of Big Raster and Vector Data," in *SIGSPATIAL*, 2018.

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

[43] B. Maestrini and B. Basso, "Predicting spatial patterns of within-field crop yield variability," *Field Crops Research*, vol. 219, 2018.

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

[45] S. Singla and A. Eldawy, "Raptor zonal statistics: Fully distributed zonal statistics of big raster + vector data [pre-print]," 2020.

[46] S. Ghosh, T. Vu, M. A. Eskandari, and A. Eldawy, "UCR-STAR: The UCR Spatio-Temporal Active Repository," *SIGSPATIAL Special*, vol. 11, no. 2, p. 34–40, Dec. 2019.

[47] "UCR-Star: The UCR Spatio-temporal Active Repository." [Online]. Available: https://star.cs.ucr.edu/

[48] "Geotrellis cliptogrid example," https://geotrellis.readthedocs.io/en/latest/guide/vectors.html#cliptogrid.

[49] T. Diao *et al.*, "Uncertainty aware wildfire management," in *AI for Social Good Workshop, AAAI Fall Symposium Series (to appear)*, 2020.