The Raptor Join Operator for Processing Big Raster + Vector Data*

Samriddhi Singla
Computer Science & Engineering
University of California, Riverside
ssing068@ucr.edu

Ahmed Eldawy
Computer Science & Engineering
University of California, Riverside
eldawy@ucr.edu

Tina Diao
Management Science & Engineering
Stanford University
tdiao@stanford.edu

Ayan Mukhopadhyay
Electrical Engineering & Computer Science
Vanderbilt University
ayan.mukhopadhyay@vanderbilt.edu

Elia Scudiero
Environmental Sciences
University of California, Riverside
elias@ucr.edu

ABSTRACT

Pre-processing spatial data for machine learning applications often includes combining different datasets into a form usable by the machine learning algorithms. Spatial data is generally available in two representations, raster and vector. The best data science and machine learning applications need to combine multiple datasets of both representations which is a data and compute intensive problem. This paper proposes a formal raster-vector join operator, *Raptor Join*, that can bridge the gap between raster and vector data. It is modeled as a relational join operator in Spark that can be easily combined with other operators, while also offering the advantage of in-situ processing. To implement the Raptor join operator efficiently, we propose a novel Flash index that has a low memory requirement and can process the entire operation with one data scan. 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 segments 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 baselines.

CCS CONCEPTS

• **Information systems** → **Query optimization**.

KEYWORDS

Vector Data, Satellite Imagery, Raster Data, Big Data, Spatial Data

ACM Reference Format:
Samriddhi Singla, Ahmed Eldawy, Tina Diao, Ayan Mukhopadhyay, and Elia Scudiero. 2021. The Raptor Join Operator for Processing Big Raster + Vector Data. In *29th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '21), November 2–5, 2021, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3474717.3483971

*This work is partly supported by the NIFA AFRI Competitive Grant no. 2019-67022-29696, ISIS (Vanderbilt University), and Department of MS & E (Stanford University).

SIGSPATIAL '21, November 2–5, 2021, Beijing, China

ACM ISBN 978-1-4503-8664-7/21/11.
https://doi.org/10.1145/3474717.3483971

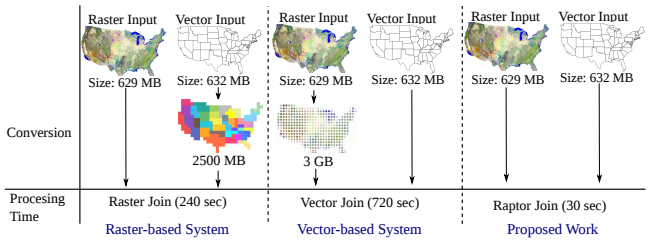


Figure 1: Comparison of raster, vector and raster+vector based systems

1 INTRODUCTION

Machine Learning has become a popular tool to analyze and utilize spatial data for research applications such as areal interpolation [32], wildfire risk assessment [11, 17], crop yield mapping [23], studying the effect of vegetation and temperature on human settlement [15, 16], and land use classification [33]. These applications often use spatial data from various sources and in different data representations. This makes it necessary to pre-process data and combine it into a single data representation before it can be used by the algorithm. Spatial data can be classified into two data representations: vector and raster. Vector data includes points, lines, and polygons, while raster data, such as satellite images, is represented as multidimensional arrays. The major differences between these two representations make combining them difficult. This is why existing systems are designed to either process vector data [7, 19, 26, 30, 48] or raster data [2, 10, 13, 29, 43]. These systems are efficient for vector-vector join or raster-raster join but do not implement a raster-vector join. In this paper, we propose a new type of join, *Raptor Join* that can efficiently combine raster and vector data to implement complex spatial data analysis pipelines.

Existing systems support raster-vector join operation by converting one of the datasets so that both datasets become of the same representation. Figure 1 illustrates how raster- and vector-based systems support raster-vector join. First raster-based systems rasterize the vector data to the same resolution of the raster data and then process them using raster-raster join. Similarly, vector based systems vectorize the raster data by converting each pixel to a point and then join both datasets using a vector-vector join. These two approaches run fine for small and medium resolution data since the conversion process does not dramatically increase the data size.

However, with the recent availability of high-resolution satellite data, these approaches no longer scale. In fact, the converted data size increases *quadratically* with the resolution [39].

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

On the other hand, vector-based systems convert raster pixels to points and process them with the vector data as illustrated in Figure 1. Methods can be similarly categorized as *on-the-fly* and *materialized* methods. The on-the-fly method first indexes the vector data and then scans the pixels, convert each pixel to a point, and process it with the index. This method suffers from the large index size and the overhead of accessing the index for each pixel. The materialized method first converts and materializes all pixels to points, and then runs an efficient distributed spatial join algorithm on the result. This method suffers from the huge size of the materialized data. For example, a 34MB compressed GeoTIFF file with 600 million pixels will be converted to nearly 3 GB of decompressed vector representation.

The proposed *Raptor Join* can concurrently process raster and vector data. Raptor Join ($RJ_{\bowtie}$) overcomes the limitations of existing systems as follows. First, $RJ_{\bowtie}$ is implemented using an in-situ approach in Spark [50] 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 combine with other relational operators in Spark such as selection, join, grouping, and aggregation. Previous work showed that a similar approach is efficient for the zonal statistics problem [38–40] but that work is still limited and cannot support complex analytical queries [42]. The $RJ_{\bowtie}$ operator is the first general-purpose operator that can build complex distributed processing pipelines for raster and vector data.

The proposed $RJ_{\bowtie}$ operator models both raster and vector data as relational data. Vector data is represented as a set of geometries while raster data is *virtually* represented as a set of pixels. To join them, we first define three predicates that define the logic of matching a pixel with either a point, a line, or a polygon. Based on that, we define the Raptor join output and show how it can be combined with standard operators to perform arbitrarily complex query pipelines for real scientific applications. To implement the Raptor join operator efficiently, we propose a novel distributed index structure termed *Flash* index which is stored as a set of integer arrays that represent ranges in the raster data that join with the vector data. *Flash* index has a very small memory footprint which allows it to efficiently process terabytes of data. We compare the proposed system to GeoTrellis [10], Google Earth Engine [13], Rasdaman [2], Sedona [48], Adaptive Cell Trie (ACT) [19], and Beast [53], and show that it has up-to three orders of magnitude performance gain over them while being perfectly able to scale to big data and use fewer resources.

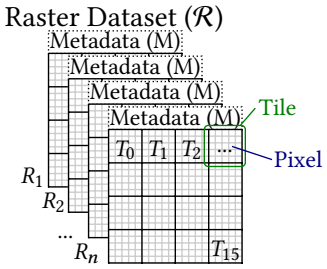


Figure 2: Raster file structure

To summarize the contributions, this paper: (1) Defines a logical 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) Proposes a new data structure, *Flash* Index, and uses it to implement $RJ_{\bowtie}$ efficiently in Spark. (5) 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 2 describes the data and query model of the proposed system. Section 3 details the algorithm used to implement the proposed operator and the *Flash*-index construction and processing. Section 4 runs an extensive experimental evaluation of the proposed system. Section 5 covers the related work in literature. Section 6 concludes the paper and discusses future work.

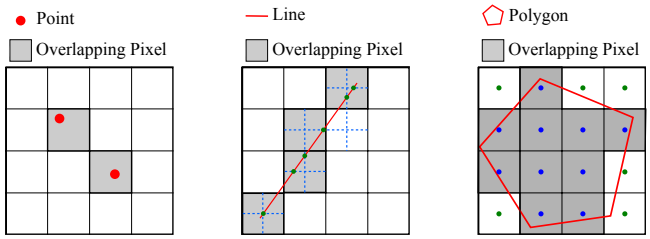
2 PROBLEM FORMULATION

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

2.1 Input Data Model

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

Raster dataset $\mathcal{R}$: Figure 2 illustrates the physical structure of the raster data. The input consists of a collection of raster files. Each file $R \in \mathcal{R}$, identified by a unique ID R_{id}, is a matrix of pixels organized into rows and columns. Each pixel $px = (R_{id}, x, y, m)$



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

Figure 3: Three predicates θ for Raptor Join

in file R_{id} is located at column x, row y, and has a numeric value m, e.g., temperature or vegetation. For efficient storage and access, raster file formats group pixels into equi-sized non-overlapping sub-arrays called tiles, and each tile is assigned an identifier, T_{id}. For example, in Figure 2, each file contains 16 tiles and each tile contains 25 pixels.

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

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

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

(1) Number of columns (c) and rows (r) of pixels in it. (2) Tile width (tw) and tile height (th) in pixels. (3) The grid-to-world ($\mathcal{G}2\mathcal{W}$) affine transformation matrix that converts a pixel location on the grid to a point location in the world, and the inverse, world–to-grid ($\mathcal{W}2\mathcal{G}$) transformation. (4) Coordinate Reference System (CRS) which describes the projection that maps the Earth surface to the world coordinates as defined by the ISO-19111 standard [14]. The metadata is *not* replicated for each pixel, but is stored only once in memory and is associated with pixels through the raster ID that contains the pixel. It can be used to calculate the following attributes, as needed, for each raster layer R or pixel px: (1) Pixel bounding box, $bb(px)$; (2) Pixel resolution, p_x, p_y; (3) Tile ID, $T_{id}(px)$; (4) Number of tiles in the file, $numTiles$. For information on how the attributes are calculated, please refer to Appendix A.

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

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

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

2.2 RJ$_{\bowtie}$ Output Definition

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

Raptor Join RJ$_{\bowtie}$: is a spatial join operator that takes as input a vector dataset V, a raster dataset $\mathcal{R}$, and a predicate θ. It produces the set of (geometry, pixel) pairs which satisfy the predicate. Based on our collaboration with domain scientists in various fields [42], we define three predicates θ_{point}, θ_{line}, and $\theta_{polygon}$ for the three types of geometries defined shortly. Each predicate θ takes two input records, a geometry g and a pixel $px = (R_{id}, x, y, m)$, and returns *true* if the geometry and pixel match. Unlike vector-based systems, we show later in the paper that we do not need to test this predicate for individual pixels but we can still find the correct result using the proposed *Flash* index.

Point Predicate (θ_{point}) returns true if the point location lies inside the bounding box ($bb(px)$) of the pixel as illustrated by the two shaded pixels in Figure 3(a).

Line Predicate θ_{line} returns true if the line intersects the *crosshair* of the pixel, which is defined as the two lines splitting the pixel bounding box in half, horizontally and vertically, as depicted by dotted lines in Figure 3(b). In this figure, shaded pixels are the ones that match the line according to this definition. This predicate can be used in hydrology applications to measure the altitude profile along hillslopes to detect watersheds [9].

Polygon Predicate $\theta_{polygon}$ returns true if the center of the pixel bounding box is inside the polygon boundary. Figure 3(c) depicts an example where the pixel centers are marked as points. Blue (green) points mark the centers of the pixels that are inside (outside) the polygon. The shaded pixels are the ones whose centers lie inside the polygon. This predicate can be used in agriculture to calculate the average vegetation per farmland [34].

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

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

$$\mathcal{R} \bowtie_{\theta} V = \{(g_{id}, R_{id}, x, y, m)\} \quad (1)$$

This allows us to define RJ$_{\bowtie}$ as a new Spark RDD (Resilient distributed dataset) [49] operator as shown in Appendix B. Keep in mind that the RJ$_{\bowtie}$ operator can be combined with other operators as needed by the application to satisfy the desired query logic. For example, it can be followed by an equi-join with the vector dataset V on the attribute g_{id} if the geometric feature is needed. Also, it can be followed by a *group by* operator on g_{id} to group all pixels that match a single geometry. Similarly, the result can be grouped by the pixel value m if it represents a raster object, e.g., contour or land type. We can extend our RJ$_{\bowtie}$ definition for outer joins but we omit these definitions for brevity. Appendix C gives three examples that integrate RJ$_{\bowtie}$ into real data science applications.

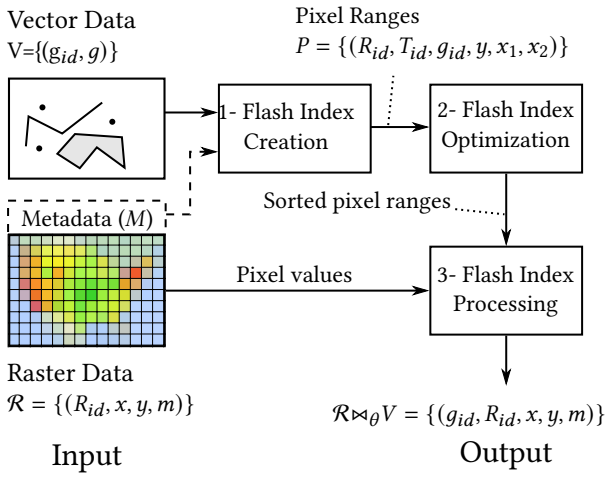


Figure 4: Implementation Overview of Raptor Join

3 RAPTOR JOIN SPARK IMPLEMENTATION

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

The key idea of RJ$_{\bowtie}$ is to resemble a sort-merge join algorithm which scans the input datasets only once. In contrast, raster-based systems resemble a hash-join where each geometry is hashed to pixels (i.e., buckets) which are then joined with input raster data. This will have a cost of $O(|V| \cdot |\mathcal{R}|)$, since each geometry in V can be rasterized into a layer with as many pixels as the raster layer, $|\mathcal{R}|$. On the other hand, vector-based systems resemble a nested-loop join where each pixel is compared to geometries to find the matching ones. If an index is built on geometries, it will resemble an *index* nested loop join with a running time of $O(|V| \log |V| + |\mathcal{R}| \cdot \log |V|)$, where the first term is for building the index and the second term is for searching the index for each pixel in $\mathcal{R}$. The proposed algorithm has a running time of $O(|V| \log |V| + |\mathcal{R}|)$, where the first term resembles a sort step for the vector dataset (in our case, building the *Flash* index, described shortly) and the second term resembles the linear merge step. The details of the analysis are omitted due to limited space but the analysis follows the approach used in [39].

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

Figure 4 gives an overview of the three phases of the RJ$_{\bowtie}$ algorithm, namely, *Flash* index *creation*, *optimization*, and *processing*. The *creation* phase takes the input vector data and only the *metadata* of the raster data to produce an initial *Flash* index that consists of a set of unordered pixel ranges. The second phase, *optimization*, repartitions, groups, and orders the pixel ranges to match the structure of the raster data. The goal is to ensure that the third phase can process the entire join query in a single scan over the raster data. The final *processing* phase uses the *Flash* index to scan the raster dataset and produce the final output. The output is produced in parallel and is streamed into the next operator depending on how the Spark job is structured.

3.1 Flash Index Creation

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

Preparation: Before the vector data can be processed, it might need some of these preparation steps. (1) If the geographical coordinate reference system (CRS) of the vector and raster data do not match, we convert the vector data to match the raster one using a map transformation as defined by the ISO 19111:2019 standard [14]. (2) If the geometric data does not already have a unique ID, we use the Spark operation `zipWithUniqueID` to generate a unique ID for each geometry. (3) If the input vector RDD has fewer partitions than the number of cores in the Spark cluster, we randomly repartition the records to have at least one partition per core. This ensures that we fully utilize the cluster during index creation.

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

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

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

$$(x, y) = (\lfloor x' \rfloor, \lfloor y' \rfloor) \tag{3}$$

First, we apply the $\mathcal{W}2\mathcal{G}$ transformation to find the grid coordinates and then use the floor function to find the pixel coordinate. Since each point covers a single pixel, the range is formed

as $(R_{id}, T_{id}, y, x, x)$, where R_{id} is the ID of the raster file and T_{id} is calculated using Equation 4. Even though there is some redundancy in repeating x, we use this format to maintain uniformity in the output for all geometry types.

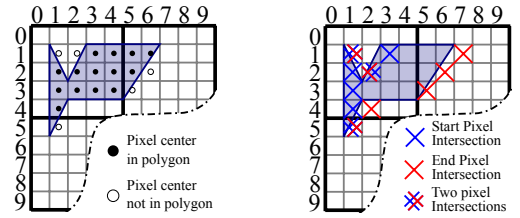
3.1.2 Pixel ranges for lines. Our definition for the θ_{line} predicate matches a line segment with those pixels whose centers are closest to the line, either horizontally or vertically. This definition is analogous to the traditional mid-point line drawing algorithm that is used in the field of computer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

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

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

The first step is to compute the pixel intersections between the boundary of the polygon and the center lines of raster rows. Therefore, the two end points of each line segment in the polygon, are converted from the model to grid space. After that, the range of raster rows that intersect with the line segment is computed, without going out of raster boundaries. Then, for each row, we compute the intersection between the horizontal line at $row + 0.5$ and the polygon segment. The computed intersection is then added to the list of intersections. Figure 5(b) depicts the computed intersections with a cross. Pixels with double-crosses indicate two intersections at the same pixel. A blue cross depicts the start of an intersection range while a red cross depicts its end.

The next step is to convert the pixel intersections into ranges in the same structure of points and lines. By inspecting Figure 5, one can realize that each row must have an even number of intersections for closed polygons. Each pair of consecutive intersections represent a range of pixels. However, these ranges have three issues that are fixed using a simple algorithm. First, range are open-ended, i.e., last pixel in the range is excluded which can be easily fixed by decreasing the position of the range end by one pixel. Second, some range are empty, e.g., the first range at rows 1 and 5, and



(a) Polygon raster join (b) Pixel intersections

Figure 5: Pixel intersection computation for polygons

some consecutive ranges can be merged, e.g., the two ranges in line 2. Both of these can be fixed by removing every pair of pixel intersections with the same exact coordinates. Third, some ranges span two tiles which would result in an efficient disk access pattern since each tile is stored in a different disk location. These are fixed by breaking each range that spans multiple tiles at tile boundaries by creating two new intersections one at the end of the first tile and one at the beginning of the next tile. All these adjustments are done in one scan over the list of pixel intersections as further detailed in Appendix D, along with the pseudo-code for this step.

RangeIterator: We store the intersections in memory in a column format, i.e., arrays of integer values. This reduces the memory overhead and improves the cache performance. It also gives an opportunity for efficient column compression to reduce memory overhead which we leave for a future work. However, Spark is a row-oriented system and cannot directly process column-oriented data in memory. To solve this mismatch issue, we create a component, termed RangeIterator, that iterates over the ranges and streams them into the next phase.

3.2 Flash Index Optimization

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

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

tiles k. The goal is to adjust the processing time for each task to be a few seconds which balances the trade-off between parallelization overhead and load skewness.

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

3.3 Flash-Index Processing

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

4 EXPERIMENTS

This section provides an extensive experimental evaluation of the proposed algorithm for the Raptor Join (RJ$_{\bowtie}$) operator. We compare RJ$_{\bowtie}$ to three vector-based approaches, Adaptive Cell Trie (ACT) [19], Sedona [48] (formerly GeoSpark), and Beast [53]; and three raster-based approaches, Rasdaman [2], Geotrellis [18], and Google Earth Engine (GEE) [13]. When applicable, we also compare to RZS [39] which supports only the zonal statistics problem on polygons using Hadoop. Zonal statistics aggregates pixel values within polygonal regions.

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

4.1 Setup

We run RJ$_{\bowtie}$, GeoTrellis, Sedona, and Beast on a cluster with one head node and 12 worker nodes. The head node has Intel(R) Xeon(R) CPU $E5-2609$ v4 @ 1.70GHz processor, 128 of GB RAM, 2 TB of HDD, and 2×8-core processors running CentOS and Oracle Java 1.8.0_131. The worker nodes have Intel(R) Xeon(R) CPU E5-2603 v4 @ 1.70GHz processor, 64 GB of RAM, 10 TB of HDD, and 2×6-core processors running CentOS and Oracle Java 1.8.0_31-b04. The methods are implemented using the open source GeoTools library 17.0. Google Earth Engine runs on the Google Cloud Platform on up-to 1,000 nodes [13] but it does not reveal the actual resources used by each query. Rasdaman and ACT are run on a single machine with Intel(R) Core $i5-6500$ CPU @ 3.20GHz × 4, 32 of GB RAM, 1 TB of HDD running Ubuntu 16.04.

Table 1: Vector and Raster Datasets

Vector datasets

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

Raster datasets

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

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

For each experiment, we perform the zonal statistics query that performs a raster-vector join and then compute the four aggregate values, minimum, maximum, sum, and count for the resulting tuples. We measure the end-to-end running time as well as the performance metrics which include reading both datasets from disk and producing the final answer. Table 1 lists the datasets that are used in the experiments along with their attributes. All vector datasets are available on UCR-Star [12, 46]. All raster datasets except Planet Data are also publicly available as described in [38]. Planet data is sourced from Planet Labs [44] and has a temporal range of a month. The coverage of these datasets is either California, the contiguous 48 states (US48), the entire US, or the world.

4.2 Vector-based Systems

In this section, we compare to three vector-based baselines, Adaptive Cell Trie (ACT) [19], Sedona [48], and Beast. Since ACT is a single-machine algorithm, we compare it separately to a single-machine version of RJ$_{\bowtie}$. Then, we compare Sedona and Beast to the Spark-based RJ$_{\bowtie}$ implementation.

On-the-fly method: ACT is a highly-efficient in-memory index for point-in-polygon queries. We use it as a representative for the

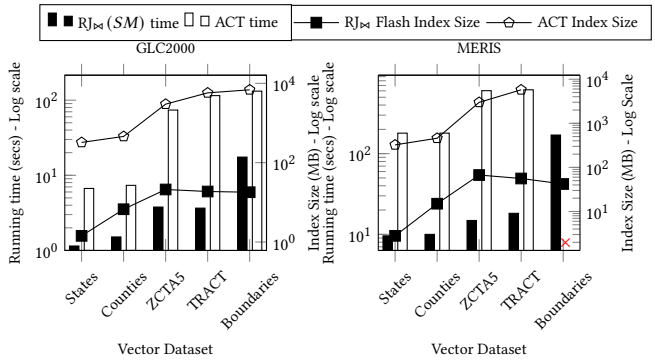


Figure 6: Running time (bars) and index size (lines) of ACT and RJ⋈ for small raster data on a single machine

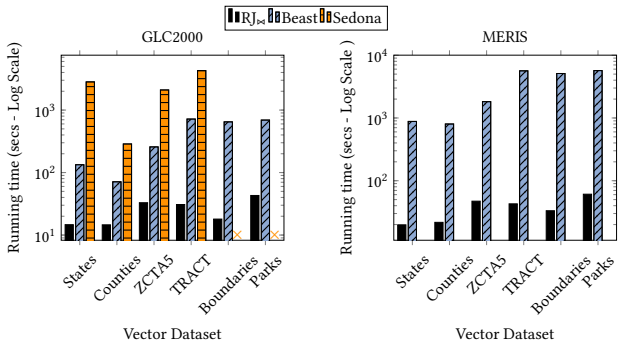


Figure 7: Running time of vector-based systems

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

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

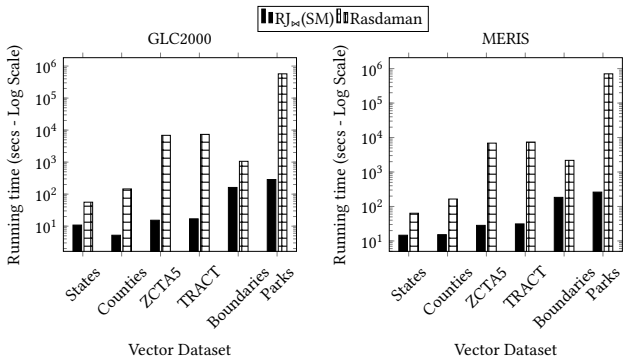


Figure 8: Single machine performance of Rasdaman and RJ⋈

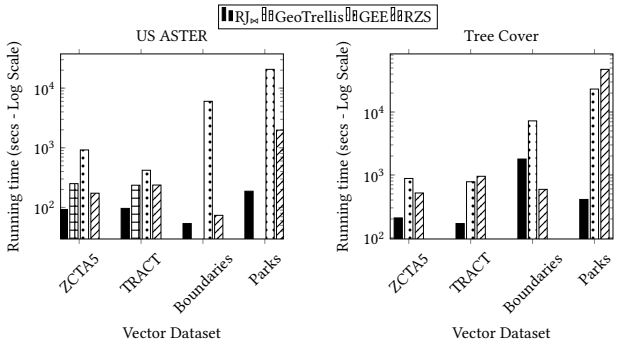


Figure 9: Running time of raster-based systems

Materialized method: Sedona and Beast are used to test the performance of the materialized method. To use them, we first convert the raster dataset to points in the format (lon, lat, m), which encodes the pixel location and value. We do not consider the overhead of the conversion process. We show the results on only the smallest raster datasets since none of the baselines was able to finish for the larger datasets. Figure 7 shows that RJ⋈ outperforms all baseline systems hands down. Furthermore, as the raster resolution increases, from GLC2000 to MERIS, the gap grows to more than three orders of magnitude. The reason is that the number of pixels increases quadratically with the resolution which incurs a huge overhead on partitioning and processing this data.

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

4.3 Raster-based Systems

In this part, we compare RJ⋈ to four baselines, Rasdaman, GeoTrellis, Google Earth Engine (GEE), and Raptor Zonal Statistics (RZS). The latter is our previous work which is designed only for the zonal statistics problem between polygons and raster data. Since the free version of Rasdaman runs on a single-machine, we compare it to a single-thread implementation of RJ⋈. We compare all other baselines to the Spark version of RJ⋈. The experiments in [42] show that

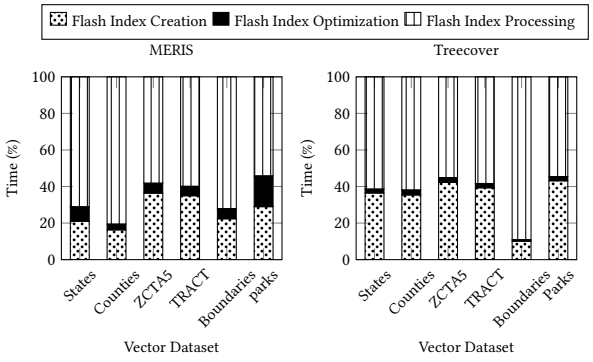


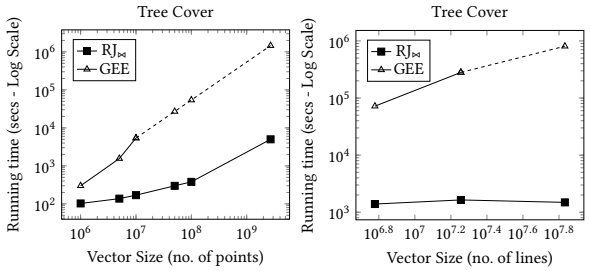
Figure 10: Breakdown of RJ⋈ running time

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

Single-machine Systems: Figure 8 shows the performance of Rasdaman as compared to the single-machine RJ⋈ algorithm. Rasdaman iterates over polygons, clips, and aggregates the raster data for each one. This would result in some redundant access to the raster data for nearby or overlapping polygons. On the other hand, RJ⋈, even when running on a single machine, ensures that the raster tiles are accessed only once. Rasdaman would still be helpful for processing a single polygon or a very few polygons but it does not scale for large vector data.

Distributed Systems: Figure 9 shows the overall running time for RJ⋈ as compared to RZS, GeoTrellis, and GEE. Since all these systems are more scalable than the previous ones, we only try on the two bigger raster datasets, US-Aster and TreeCover. RJ⋈ is still the fastest algorithm in almost all cases. The only case where RZS is faster is when joining Boundaries with TreeCover. Since the Boundaries dataset is small, RZS would broadcast it to all machines where it then processes each file locally. RJ⋈ would still partition the *Flash* index which might result in some overhead as multiple machines might process different parts of the same file. However, for big vector and raster data, RJ⋈ is more than 50 times faster than RZS. Additionally, RJ⋈ is more flexible since the RZS algorithm solves only the zonal statistics problem and runs only with polygons.

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



(a) Point Features (b) Line Features

Figure 11: Performance on non-polygon joins with big raster data. Dotted lines represent extrapolated values.

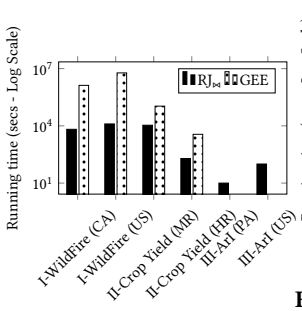


Figure 12: Applications

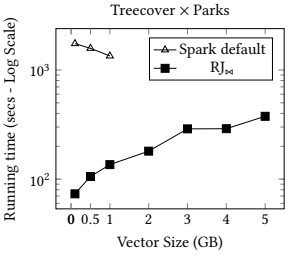


Figure 13: Vector Partitioning

4.4 Flexibility of RJ⋈

Non-polygon joins: To show the flexibility of RJ⋈, we test its performance on non-polygonal joins, i.e., point and linestring joins. An application of point joins is agricultural applications that combine ground sensors at fixed points with remote sensors and build a regression model between them [34]. Linestring joins can be used in hydrology applications to compute the elevation model along water paths to measure the water flow and delineate watersheds [9]. Figure 11 shows the performance of point and line joins for the vector datasets in Table 1 and the largest raster dataset Treecover for RJ⋈ and GEE. We chose GEE as a baseline as we observed it to be the most scalable during the experiments for polygon joins. As can be observed, RJ⋈ outperforms GEE for both linestring and point joins with over two orders of magnitude performance gain. The dotted lines for GEE indicate an extrapolation that we did to estimate the running time for larger vector datasets, since it was not able to process the entire dataset in one run.

Applications: Figure 12 shows the results for applications discussed in [42]. As can be observed, RJ⋈ is at least 10x faster than GEE for the first two applications. For the first application, wildfire combating, RJ⋈ is used to calculate statistics for both California (3 million polygons) and the entire US (55 million polygons). It takes as input 23 rasters from *landfire.gov* each containing over a billion pixels. The second application, crop yield mapping, takes as input 360,000 agricultural fields and over 1.8 TB of medium resolution(MR) raster data and 31 GB of high resolution(HR) Planet Data. For the third application of areal interpolation(ArI), the results are for the single machine implementation of RJ⋈. It was used to join the National Land Cover (NLCD) raster dataset, with 16 billion pixels, with 74k TRACT polygons to estimate their population.

4.5 Optimizing RJ⋈

***Flash* Index Construction:** This experiment studies the effect of partitioning vector data during the *Flash* index construction step. We compare the default Spark partitioning, that creates a partition for each 128 MB block, against the one proposed in RJ⋈ that partitions the file into blocks of 16 MB each, followed by another partitioning if the number of blocks is less than the number of workers in the cluster. The experiment is conducted using the raster dataset, Treecover and subsets of vector dataset, Parks. As can be seen in Figure 13, the proposed partitioning runs 100 times faster for small datasets due to the additional re-partitioning step that ensures that all executors participate in the processing. For large datasets, the default partitioning method fails with out of memory exception due to the large number of intersections per 128 MB partition. Using a 16 MB partition reduces the overall memory overhead per executor.

***Flash* Index Optimization:** This experiment studies the effect of global and local index optimizations on the running time of RJ⋈ algorithm. As can be observed in Figure 14, using only the global optimization helps with small vector datasets, i.e., boundaries, but it does not help much with medium-scale datasets, i.e., ZCTA5 and TRACT, and reduces the performance for large datasets, parks. The reason is that for a small dataset, the *Flash* index is small enough that only partitioning by tile ID (global optimization) is enough while sorting within partition might not help much. For large vector data, local optimization is critical due to the large number of pixel ranges. This can be observed with the parks dataset where global+local optimizations achieve an order of magnitude speedup.

Efficient Aggregation: There are two types of aggregate functions, *algebraic* and *holistic*. Algebraic functions can be computed efficiently using `reduce` or `aggregate` operation. local and then global aggregation, e.g., min, max, and average. These can be computed in Spark using the `reduce` or `aggregate` operations. Holistic functions, on the other hand, may require collecting all values in one machine and are thus less efficient to compute, e.g., median and percentile. They can be implemented in Spark using the less efficient `groupBy` operation. Since RJ⋈ is integrated in Spark, it can compute both types of functions by simply following the RJ⋈ operation with the appropriate Spark operation. This design breaks from the limitation of RZS which can only support a limited number of algebraic aggregate functions. Figure 15 compares the computation of mean and median as an example of algebraic and holistic functions, respectively. As shown, RJ⋈ is three times faster when computing the mean due to the more efficient calculation method. This experiment also shows the flexibility of RJ⋈ with any function that the users want to compute.

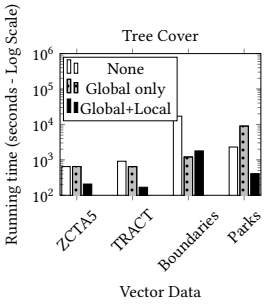


Figure 14: Optimization

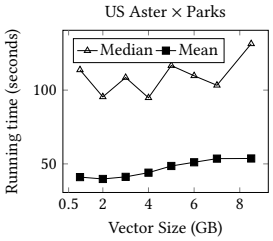


Figure 15: Aggregation

5 RELATED WORK

Non-spatial Joins The join operation [31] is a fundamental relational database query operation that brings together two ore more relations. Logically, it can be modeled as a Cartesian product followed by a filter on the join predicate. Non-matching attributes from the two relations can be included in the output depending on the join type, inner, left outer, right outer, or full outer join. The most common join operation in relational databases is equi-join which uses the equality join predicate. Traditional join algorithms [25] 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.

Spatial Join on Raster Data Raster data is usually represented as multidimensional arrays and it is analysed using map algebra [24, 35]. To join two raster layers, they must have the same dimensions. If the dimensions mismatch, a regridding operation is applied on one dataset to match the other dataset. Systems such as SciDB [43], RasDaMan [2], GeoTrellis [18], ChronosDB [51], and Google Earth Engine [13] implement algorithms for raster operations that can process large amounts of raster data. However, none of these systems provide an efficient join operation for raster and vector data. They usually rasterize the vector data with a matching resolution and apply the raster operation. [6, 54] optimize raster joins by utilizing its tiled structure. However, they focus on either similarity joins or skewed data, which generally do not apply to raster-vector joins.

Spatial Join on Vector Data Vector data is represented as a set of points, lines, and polygons, which are all represented as a set of coordinates. A spatial join on vector data can be defined as a join that finds pairs of geometries that satisfy a spatial predicate, such as intersection, overlap, or contains. Spatial join algorithms for vector data include R-Tree join [3], Spatial Hash join [22], Partition Based Spatial Merge join (PBSM) [27], index-based in-memory joins [19, 37], and many more [20, 21]. Efforts have also been made to implement these spatial join algorithms in a distributed environment in order to process big data [1, 7, 47, 48, 52]. 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.

Raster-Vector Joins There has been some efforts to combine raster and vector data efficiently at the data representation, indexing, and query processing levels. At the data representation levels, a new *vaster* model was proposed [28] which converts both vector and raster data to a common representation. However, it was not practical as it needs to convert both datasets prior to doing any processing. At the indexing levels, the k^2-raster index is proposed [4, 36] to index raster data that can be combined with an R-tree vector index. However, this work is limited to top-k and range queries and is limited to small data as it is a main-memory index. At the query processing level, the single-machine Scanline algorithm [8, 40] was proposed to solve the zonal statistics problem on raster and vector data. The algorithm was also ported to Hadoop

for scalability [38, 39]. However, this work was limited to the zonal statistics computation on polygons and had a limited scalability when it comes to big vector data.

This paper is the first to define and implement a general-purpose join operator for raster and vector data that can be used to build any complex relational query plan that runs on the distributed Spark framework. It does not require a conversion process like [28]. The proposed Flash index is lighter than [4, 19, 36] and can scale to terabytes of data and support any relational operation. Finally, the proposed work is more scalable and flexible than [8, 38–40] since it can support any relational query plan and not only zonal statistics.

6 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 index structure *Flash Index*. This algorithm is modeled as a relational join operator and uses an in-situ approach, hence, making it attractive for ad-hoc queries. It runs in three steps, namely, *Flash Index creation*, *Flash Index optimization* and *Flash Index processing*. The *Flash Index creation* step computes a mapping between raster and vector in the form of pixel ranges. The *Flash Index optimization* step partitions and reorganizes this data structure across machines in such a way that each tile in the raster dataset is scanned by only one machine. The *Flash Index processing* step processes the partitioned pixel ranges to read the required pixel values from the raster dataset. We run extensive experiments for the system against Rasdaman, GeoTrellis, Google Earth Engine, Adaptive Cell Trie, GeoSpark, and Beast on large raster and vector datasets to show its scalability and performance gain. In the future, we plan on extending the system to work with machine learning algorithms and answering approximate queries.

REFERENCES

[1] A. Aji et al. Hadoop-GIS: A high performance spatial data warehousing system over mapreduce. *PVLDB*, 6(11), 2013.
[2] P. Baumann et al. The multidimensional database system RasDaMan. In *SIGMOD*, pages 575–577, Seattle, WA, June 1998.
[3] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. *ACM SIGMOD Record*, 22(2):237–246, 1993.
[4] N. R. Brisaboa et al. Efficiently querying vector and raster data. *The Computer Journal*, 60(9):1395–1413, 2017.
[5] T. Diao et al. Uncertainty Aware Wildfire Management. In *AI for Social Good Workshop, AAAI Fall Symposium Series*, 2020.
[6] J. Duggan et al. Skew-aware join optimization for array databases.
[7] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce Framework for Spatial Data. In *ICDE*, pages 1352–1363, Apr. 2015.
[8] A. Eldawy, L. Niu, D. Haynes, and Z. Su. Large scale analytics of vector+raster big spatial data. In *SIGSPATIAL*, pages 62:1–62:4, 2017.
[9] Y. Fan et al. Hillslope Hydrology in Global Change Research and Earth System Modeling. 2019.
[10] GeoTrellis on Spark. https://github.com/wri/geotrellis-zonal-stats/blob/master/src/main/scala/tutorial/ZonalStats.scala, 2019.
[11] O. Ghorbanzadeh et al. Spatial prediction of wildfire susceptibility using field survey GPS data and machine learning approaches. *Fire*, 2(3):43, 2019.
[12] S. Ghosh, T. Vu, M. A. Eskandari, and A. Eldawy. UCR-STAR: The UCR Spatio-Temporal Active Repository. *SIGSPATIAL Special*, 11(2):34–40, Dec. 2019.
[13] N. Gorelick et al. Google earth engine: Planetary-scale geospatial analysis for everyone. *Remote sensing of Environment*, 202:18–27, 2017.
[14] ISO 19111:2019: Geographic information - Referencing by coordinates. https://www.iso.org/obp/ui/#iso:std:iso:19111:ed-3:v1:en, 2019.
[15] G. D. Jenerette et al. Regional Relationships Between Surface Temperature, Vegetation, and Human Settlement in a Rapidly Urbanizing Ecosystem. *Landscape Ecology*, 22:353–365, 2007.
[16] G. D. Jenerette et al. Ecosystem Services and Urban Heat Riskscape Moderation: Water, Green Spaces, and Social Inequality in Phoenix, USA. *Ecological Applications*, 21:2637–2651, 2011.
[17] M. B. Joseph et al. Spatiotemporal prediction of wildfire size extremes with bayesian finite sample maxima. *Ecological Applications*, 29(6), 2019.
[18] A. Kini and R. Emanuele. Geotrellis: Adding Geospatial Capabilities to Spark, 2014.
[19] A. Kipf et al. Adaptive main-memory indexing for high-performance point-polygon joins. In *EDBT 2020*, pages 347–358. OpenProceedings.org, 2020.
[20] N. Koudas and K. C. Sevcik. Size separation spatial join. In *SIGMOD*, pages 324–335, 1997.
[21] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *ACM SIGMOD*, pages 209–220, 1994.
[22] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *ACM SIGMOD*, pages 247–258, 1996.
[23] B. Maestrini and B. Basso. Predicting spatial patterns of within-field crop yield variability. *Field Crops Research*, 219, 2018.
[24] J. Mennis, R. Viger, and C. D. Tomlin. Cubic map algebra functions for spatio-temporal analysis. *Cartography and Geographic Information Science*, 32(1), 2005.
[25] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.
[26] S. Nishimura et al. MD-HBase: Design and Implementation of an Elastic Data Infrastructure for Cloud-scale Location Services. *DAPD*, 31(2):289–319, 2013.
[27] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. *ACM Sigmod Record*, 25(2):259–270, 1996.
[28] D. J. Peuquet. A hybrid structure for the storage and manipulation of very large spatial data sets. *Computer Vision, Graphics, and Image Processing*, 24(1), 1983.
[29] G. Planthaber, M. Stonebraker, and J. Frew. Earthdb: scalable analysis of modis data using scidb. In *BIGSPATIAL*, pages 11–19, 2012.
[30] Postgis: Spatial and geographic objects for postgresql, 2020. https://postgis.net.
[31] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw-Hill, 2000.
[32] M. Reibel and A. Agrawal. Areal interpolation of population counts using pre-classified land cover data. *Population Research and Policy Review*, 26(5-6), 2007.
[33] 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*, 66(5):608–619, 2011.
[34] E. Scudiero et al. Regional scale soil salinity evaluation using landsat 7, western san joaquin valley, california, usa. *Geoderma Regional*, 2-3:82 – 90, 2014.
[35] S. Shekhar and S. Chawla. *Spatial Databases: A Tour*. Prentice Hall Upper Saddle River, NJ, 2003.
[36] F. Silva-Coira et al. Efficient processing of raster and vector data. *Plos one*, 15(1):e0226943, 2020.
[37] B. Simion, A. D. Brown, and R. Johnson. Skew-resistant Parallel In-memory Spatial Join. In *SSDBM*, pages 6:1–6:12, Aalborg, Denmark, July 2014.
[38] S. Singla and A. Eldawy. Distributed Zonal Statistics of Big Raster and Vector Data. In *SIGSPATIAL*, 2018.
[39] S. Singla and A. Eldawy. Raptor Zonal Statistics : Fully Distributed Zonal Statistics of Big Raster + Vector Data. In *IEEE BigData 2020*. IEEE, Dec. 2020.
[40] S. Singla et al. Raptor: Large Scale Analysis of Big Raster and Vector Data. *PVLDB*, 12(12):1950 – 1953, 2019.
[41] S. Singla et al. WildfireDB: A Spatio-Temporal Dataset Combining Wildfire Occurrence with Relevant Covariates. 2020.
[42] S. Singla et al. Experimental Study of Big Raster and Vector Database Systems. In *ICDE*, page To Appear, Apr. 2021.
[43] M. Stonebraker et al. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3), 2013.
[44] P. Team. Planet application program interface: In space for life on earth, 2018–.
[45] M. P. Thompson et al. Integrating pixel-and polygon-based approaches to wildfire risk assessment: Application to a high-value watershed on the pike and san isabel national forests, colorado, usa. *Environmental Modeling & Assessment*, 21(1), 2016.
[46] UCR-Star: The UCR Spatio-temporal Active Repository. https://star.cs.ucr.edu/.
[47] D. Xie et al. Simba: Efficient In-Memory Spatial Analytics. In *SIGMOD*, June 2016.
[48] J. Yu, M. Sarwat, and J. Wu. GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data. In *SIGSPATIAL*, Seattle, WA, Nov. 2015.
[49] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX Association, 2012.
[50] M. Zaharia et al. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.
[51] R. A. R. Zalipynis. Chronosdb: distributed, file based, geospatial array dbms. *PVLDB*, 11(10):1247–1261, 2018.
[52] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *CLUSTER*, pages 1–8, 2009.
[53] Y. Zhang and A. Eldawy. Evaluating computational geometry libraries for big spatial data exploration. In *ACM SIGMOD*, pages 1–6, 2020.
[54] W. Zhao, F. Rusu, B. Dong, and K. Wu. Similarity join over array data. In *Proceedings of the 2016 International Conference on Management of Data*, 2016.

A RASTER METADATA ATTRIBUTES

This part describes how information from raster metadata $R.M$ can be used to calculate the attributes required for each raster layer R or pixel px:

- Pixel bounding box $bb(px)$: is the bounding box of a pixel in world coordinates. The two corner points of $bb(px)$ are computed by applying the $\mathcal{G}2\mathcal{W}$ transformation for the pixel locations (x, y) and $(x + 1, y + 1)$.
- Pixel resolution p_x, p_y: is the width and height of a pixel bounding box. This value needs to be calculated only once since all pixels have the same resolution. For square pixels, we use $p = p_x = p_y$ to refer to either of them.
- Tile ID $T_{id}(px)$: is the tile that contains any given pixel.

$$T_{id} = \left\lfloor \frac{y}{th} \right\rfloor \cdot \left\lceil \frac{c}{tw} \right\rceil + \left\lfloor \frac{x}{tw} \right\rfloor \tag{4}$$

- Number of tiles in the file:

$$numTiles = \left\lceil \frac{c}{tw} \right\rceil \cdot \left\lceil \frac{r}{th} \right\rceil \tag{5}$$

B INTEGRATION WITH SPARK

This part describes how the RJ$_{\bowtie}$ operation is integrated with the Spark RDD API [49]. We use the Scala *implicit classes*[1] feature to extend the SparkContext and RDD classes without touching the internal code of Spark. In SparkContext, we add two sets of functions for loading *vector* and *raster* data in various formats, e.g., `geoTiff`, `shapefile`, and `geojson`. Vector data is loaded as `RDD[IFeature]` where IFeature represents a geometric feature that contains a geometry and any additional non-spatial attributes in the file. Raster data is loaded as `RDD[ITile]` where each ITile is an interface for accessing pixel values and raster metadata.

We also extend `RDD[ITile]` with the `raptorJoin` method that takes `RDD[IFeature]` and returns `RDD[(Long, Int, Int, Int, Float)` which represents a set of tuples $(g_{id}, R_{id}, x, y, m)$. Finally, `raptorJoin` is implemented as a transformation so it can be preceded or followed by other transformations and they will be compiled into one Spark job. For example, it can be preceded by filter on the vector data and followed by a grouped aggregation on the geometry ID or the pixel value. In addition to the Scala API, we also provide Java and Python wrappers but we omit their details for brevity.

C APPLICATION EXAMPLES

RJ$_{\bowtie}$ can be used in a wide range of applications. This section describes three real applications we collaborated on for combating wildfires, crop yield estimation in agriculture, and population estimation in political science. Interested readers can refer to [42] for more details about these applications. The following examples use the Spark Scala RDD API assuming that Spark context is initialized as `sc`. We assume that the reader is familiar with the Spark RDD API[2] since explaining it is beyond the scope of this work.

Combating Wildfires In this application [5, 41, 45], the input consists of a set of fire zones in the US, defined as polygons, and a set of satellite layers that contain vegetation, fuel and temperature as contributing factors for the spread of fire. It calculates a set of statistics including mean, median, and standard deviation for the contributing factors in each fire zone. This data can then be used to train a machine learning model on how wild fires spread. The following code snippet shows how to compute the statistics for temperature (vegetation and fuel is done similarly).

```
val zones: RDD[IFeature] = sc.shapefile("zones.zip")
val temperature: RDD[ITile] = sc.geoTiff("temperature.tif")
val zone_temp: RDD[(IFeature, Array[Float])] =
 temperature.raptorJoin(zones)
  .map(x => (x.feature_id, x.m)) // Keep geometry and pixel value
  .groupByKey() // Group by vector feature
  .map(x => (x._1, statistics(x._2))) // Compute statistics
```

The result is grouped by feature-id to be able to compute holistic aggregate functions, e.g., median, which can only be computed when all values are present.

Crop Yield Mapping In this application [23], the input consists of a vector dataset that contains farmlands in California and several raster layers that represent the vegetation, i.e., how green each pixel is, at different dates. It computes the 90^{th} percentile of vegetation per pixel over time and then computes the average per farmland.

```
val farmland: RDD[IFeature] = sc.shapefile("farmland.shp")
  .filter(_.getAs[String]("State") == "CA"))
val vegetation: RDD[ITile] = sc.geoTiff("vegetation")
val percentiles: RDD[(IFeature, Float)] =
 vegetation.raptorJoin(farmland)
  .groupBy(v => (v.x, v.y)) // Group by pixel location
  .map(x => percentile(x, 90)) // Compute 90th percentile
val averages = percentiles.aggregateByKey((0.0f, 0L))(
  // Aggregate sum, count
  (sumcount, x) => (sumcount._1 + x, sumcount._2 + 1),
  (a, b) => (a._1 + b._1, a._2 + b._2)) // Combine (sum,count)
 .map(x => (x._1, x._2._1 / x._2._2)) // average=sum/count
```

The above example filters the farmland dataset before the join to keep only the ones in California. We join and group the results by pixel to include all pixels values across time. After that, we compute the 90^{th} percentile per pixel while still keeping the farmland feature in the result. We use the Spark *aggregate* operation which can efficiently compute algebraic aggregate functions, e.g., average.

Areal Interpolation (AI) This application [32] estimates the population in arbitrary areas using areal interpolation. At its core, this method takes a set of polygonal regions, e.g., census tracts or ZIP codes, and a raster layer that contain land use type. It computes the histogram of use types for each region as explained below.

```
val regions: RDD[IFeature] = sc.shapefile("tracts.shp")
val landuse: RDD[ITile] = sc.geoTiff("landuse")
val histograms: Map[((IFeature, Float), Long)] =
 landuse.raptorJoin(regions)
  .map(x => ((x.feature, x.m), null)) // Map to histogram bins
  .countByKey() // Count per region and land type
```

This example directly joins the tracts vector data with the landuse raster data. To compute the histogram, it uses a map transformation to define a histogram bin for each result entry as the pair (tract,land type). It finishes by simply counting the number of entries per bin.

[1] https://docs.scala-lang.org/overviews/core/implicit-classes.html
[2] https://spark.apache.org/docs/latest/rdd-programming-guide.html

Algorithm 1 LinePixelIntersectionsToRanges

1: **Input:** A list of pixel intersections represented by three arrays $g_{id}[], x[], y[]$ with equal length n
2: **Output:** A list of pixel ranges represented by five arrays $t[], g_{id}[], y[], x_1[], x_2[]$ with equal length $n_{out} \leq n$
3: Sort (g_{id}, x, y) lexicographically by (y, g_{id}, x)
4: Let $t[]$ be an array of tile IDs with length n initialized to zeros
5: numDeletions ← 0
6: **for** i=n-1 **downto** 0 **do**
7: $t[i] \leftarrow T_{id}(x[i], y[i])$ ▷ Use Equation 4
8: **if** $i \neq n-1$ **and**
9: $(t[i+1], g_{id}[i+1], y[i+1]) = (t[i], g_{id}[i], y[i])$ **and**
10: $(x[i+1] = x[i]$ **or** $x[i+1] = x[i]+1)$ **then**
11: $t[i+1] \leftarrow$ numTiles ▷ Mark for deletion
12: numDeletions++
13: **else** ▷ Form a range by appending a similar pixel
14: $(x[n], y[n], g_{id}[n], t[n]) \leftarrow (x[i], y[i], g_{id}[i], t[i])$
15: $n++$
16: Sort (t, g_{id}, x, y) lexicographically by (t, y, g_{id}, x)
17: $n_{out} \leftarrow n - numDeletions$
18: $x_1 = x[i : \text{i is even} \wedge i < n_{out}]$ ▷ Arrays are zero-based
19: $x_2 = x[i : \text{i is odd} \wedge i < n_{out}]$
20: $t = t[i : \text{i is odd} \wedge i < n_{out}]$
21: $g_{id} = g_{id}[i : \text{i is odd} \wedge i < n_{out}]$
22: $y = y[i : \text{i is odd} \wedge i < n_{out}]$
23: **return** $(t, g_{id}, y, x_1, x_2, n_{out})$

D FLASH INDEX CREATION ALGORITHMS

This section describes in detail with help of pseudo-code the computation of intersection pixel ranges for lines and polygons.

Pixel ranges for lines: Algorithm 1 computes pixel ranges from line-pixel intersections. It takes as input a list of pixel intersections represented in three arrays g_{id}, x, and y. The output is five arrays that together represent the pixel ranges. Line 3 sorts the intersections to bring the ones that can be merged next to each other in the sort order. It then scans the list of pixel intersections from the end to allow new intersections to be appended without affecting the processing of the loop. Inside the loop, the tile ID is computed using Equation 4 and then the two adjacent pixel intersections at i and $i+1$ are compared to check if they are adjacent or overlapping. If they are, we remove the latter one at $i+1$. For efficiency, we only mark it for deletion by setting its tile ID to $numTiles$ which is larger than all valid tile IDs. Otherwise, if the two pixel intersections cannot be merged, we insert a new intersection at the tail of the list to form a range. Notice that the newly inserted intersection is not in the sort order but this will be fixed after the loop finishes.

After the for loop, Lines 16-23 sort the list of intersections again by (t, y, g_{id}, x) which has two goals. First, it will push all the pixels marked for deletion to the end of the list since they all have a maximum tile ID. Second, it will form pixel ranges from every two consecutive pixel intersections. The final output is formed by combining every two consecutive intersections into a single range as depicted in the algorithm. The running time of this algorithm is $O(n \log n)$ for the sort steps. It can insert and delete at most n intersections with a constant time for each insert or delete.

Pixel ranges for polygons: Algorithm 2 describes how to compute all intersections between a single polygon segment and centers of raster rows. The algorithm starts by converting the two end points of each segment to the grid space while keeping it as a real number. Then, it calculates the range of rows that intersect the line segment while keeping in mind that the valid range of rows is $[0, r]$. For each row, the segment is intersected with the horizontal line at the center of the row, i.e., $row + 0.5$ while keeping in mind that the valid range of intersections is $[0, c]$.

Algorithm 2 ComputePolygonSegmentIntersections

1: **Input:** One polygon segment $(lon_1, lat_1) \rightarrow (lon_2, lat_2)$ and raster metadata M
2: **Output:** A list of pixel intersections
3: $(x_1, y_1) \leftarrow M.\mathcal{W}2\mathcal{G}(lon_1, lat_1)$
4: $(x_2, y_2) \leftarrow M.\mathcal{W}2\mathcal{G}(lon_2, lat_2)$ ▷ xs and ys are real numbers
5: row1 ← max(0, round(min(y_1, y_2))) ▷ row1 and
6: row2 ← min(M.r, round(max(y_1, y_2))) ▷ row2 are integers
7: **for** row ← row1 **to** row2-1 **do**
8: xIntersection ← round$(x_2 - (y_2 - (row + 0.5))\frac{x_2 - x_1}{y_2 - y_1})$
9: Output ≪ (max(0, min(xIntersection, M.c)), row)

Algorithm 3 PolylgonPixelRanges

1: **Input:** A list of pixels intersections represented by three arrays $g_{id}[], x[], y[]$ with equal length n
2: **Output:** A list of pixel ranges represented by five arrays $t[], g_{id}[], y[], x_1[], x_2[]$ with equal length n_{out}
3: Sort (g_{id}, x, y) lexicographically by (y, g_{id}, x)
4: Let $t[]$ be an array of tile IDs with length n initialized to zeros
5: numDeletions ← 0
6: **for** i=n-2 **downto** 0 **step** -2 **do**
7: **if** x[i] ≥ x[i+1] **then**
8: $t[i] \leftarrow t[i+1] \leftarrow numTiles$ ▷ Mark points for removal
9: numDeletions ← numDeletions + 2
10: **else**
11: $t[i] \leftarrow T_{id}(x[i], y[i])$ ▷ Use Equation 4
12: $x[i+1] \leftarrow x[i+1] - 1$ ▷ Make the range inclusive
13: $t[i+1] \leftarrow T_{id}(x[i+1], y[i+1])$
14: **if** $t[i] \neq t[i+1]$ **then**
15: *// Break the range into two at tile boundary*
16: $(x[n], y[n], t[n]) \leftarrow (tw \cdot t[i+1] - 1, ys[i], t[i])$
17: $(x[n+1], y[n+1], t[n+1]) \leftarrow (tw \cdot t[i+1], ys[i], t[i+1])$
18: $n \leftarrow n+2$
19: **if** $(g_{id}[i+1], y[i+1], t[i+1])$=$(g_{id}[i+2], y[i+2], t[i+2])$
20: **and** $x[i+1] = x[i+2] - 1$ **then**
21: $t[i+1] \leftarrow t[i+2] \leftarrow$ numTiles
22: numDeletions ← numDeletions + 2
23: Similar to lines 16-23 of Algorithm 1

The next step is to convert the pixel intersections into ranges in the same structure of points and lines. Each row must have an even number of intersections for closed polygons. Therefore, each pair of consecutive intersections represent a range of pixels. Algorithm 3 describes how to compute pixel ranges from pixel intersections for polygons. Similar to the case of linestrings, it starts by sorting all pixel intersections lexicographically by (y, g_{id}, x). After that, it makes one scan over those ranges from end to start. Line 7 checks if the range is empty and marks the both ends for removal by setting the tile ID to a value bigger than all valid tile IDs. If not empty, it computes the tile ID for both pixel intersections and decrements the end to convert it from an open-ended to closed interval. Line 14 checks if the range spans two tiles. If so, it breaks it into two ranges at the tile boundary. This is done by inserting two new pixel intersections at the end of the first tile and the beginning of the second tile. While this part assumes that a range can intersect at most two tiles, it can be easily extended to ranges that span several tiles but we omit this part for brevity. Line 19 tests if two consecutive ranges can be merged into one. Merging two ranges is done by removing the end of the first one and the start of the second one. After all pixel intersections are processed, the final pixel ranges are computed in the same way as lines 16-23 in Algorithm 1. This algorithm has a running time of $O(n \log n)$ for the two sorting steps.