Viper: Interactive Exploration of Large Satellite Data*†

Zhuocheng Shang
University of California, Riverside
Riverside, California, USA
zshan011@ucr.edu

Ahmed Eldawy
University of California, Riverside
Riverside, California, USA
eldawy@ucr.edu

ABSTRACT

Significant increase in high-resolution satellite data requires more productive analysis methods to benefit data scientists. Interactive exploration is essential to productivity since it keeps the user engaged by providing quick responses. This paper addresses the progressive zonal statistics problem that given big satellite data, an aggregate function, and a set of query polygons, zonal statistics computes the aggregate function for each query polygon over raster data. Efficiently querying complex polygons, reading high resolution pixels and process multiple polygons simultaneously are three main challenges. This work introduces Viper, an interactive exploration pipeline to overcome these challenges and achieve requirements. Viper uses a raster-vector index to bootstrap the answer with an accurate result in a short time. Then, it progressively refines the answer using a priority processing algorithm to produce the final answer. Experiments on large-scale real data show that Viper can reach 90% accuracy or higher up-to two orders of magnitude faster than baseline algorithms.

CCS CONCEPTS

• **Information systems** → **Query languages**.

KEYWORDS

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

ACM Reference Format:
Zhuocheng Shang and Ahmed Eldawy. 2023. Viper: Interactive Exploration of Large Satellite Data. In *Symposium on Spatial and Temporal Data (SSTD '23), August 23–25, 2023, Calgary, AB, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3609956.3609966

1 INTRODUCTION

There is an ever growing amount of high-resolution satellite data available for analysis by data scientists. NASA EOSDIS archive is expected to reach nearly 250 petabytes by 2025. Planet Labs gathers more than 15TB of daily satellite data. Users explore these datasets by running queries over regions of interest. One fundamental query is *zonal statistics* which takes satellite data, a set of query polygons, and an aggregate function, and computes the aggregate function over pixel values within each query polygon. For example, compute the average vegetation cover index over each farmland. This problem has many applications in satellite data exploration and analysis [30], computation of soil moisture in agricultural fields [7], areal interpolation [23], wildfire combating [33], and others [31].

In recent studies, efforts have been made to optimize zonal statistics for large-scale satellite data. However, through our interactions with data scientists exploring such data, we discover their need for rapid results that closely approximate the final answer, enabling effective guidance of data exploration. This renders batch processing techniques inadequate [11, 30]. Additionally, they emphasize the importance of accurate results to avoid unbounded errors, rendering approximate approaches impractical [2, 3]. Consequently, this paper focuses on a progressive query processing approach that fulfills both requirements. It initially generates an approximate answer and subsequently refines it until an exact answer is attained.

Unfortunately, running zonal statistics progressively while exploring large satellite data encounters three challenges. First, there is a computational overhead when dealing with complex query polygons that have up-to tens of thousands of segments per query polygon, e.g., country borders. Second, there is a disk IO overhead to read pixel values from high-resolution satellite data with billions of pixels, e.g., 30-meter and 3-meter resolution data. Third, there is a scalability challenge when processing thousands of query polygons simultaneously. These challenges result in extended query processing time that impedes user productivity.

Efforts have been attempted to address this problem but have yet to tackle all challenges. Some methods only focus on the first challenge by utilizing approximation and GPU acceleration [38, 41]. These methods are limited in scale to high-resolution rasters due to memory and disk overhead. Other techniques only address the second challenge using raster indexes [5, 8, 10, 17, 18, 21, 25, 29, 34] that comprise pre-aggregated results but are limited to rectangular queries or simple polygons. In addition, all the above methods do not address the third challenge as they answer one query polygon at a time and do not scale to thousands of query polygons.

To overcome these limitations and challenges, this study introduces Viper, an interactive exploration framework with progressive query answering over arbitrarily complex query polygons and large satellite data. Figure 1 demonstrates the key concept. The bottom part shows a baseline approach that achieves interactivity by processing one query polygon at a time, resulting in a long wait time before fetching an answer with reasonable quality. On the top, the proposed method considers all queries collectively to produce a quality answer quickly and refine it until obtaining an exact answer.

Viper operates in two phases that overcome challenges and limitations. **Phase I** assembles a light-weight on-the-fly index for query polygons to enable rapid initial results with large satellite data and

*This work is supported in part by the National Science Foundation (NSF) Grants no. IIS-2046236.
†This work is supported in part by USDA National Institute of Food and Agriculture Grants no. 2020-69012-31914.

SSTD '23, August 23–25, 2023, Calgary, AB, Canada

ACM ISBN 979-8-4007-0899-2/23/08.
https://doi.org/10.1145/3609956.3609966

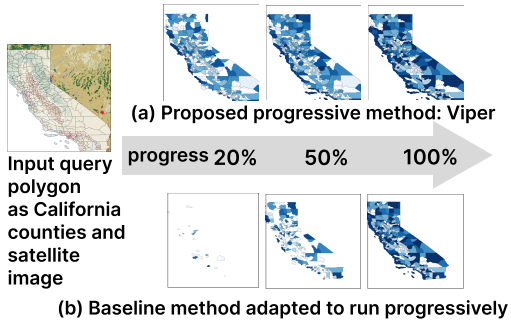
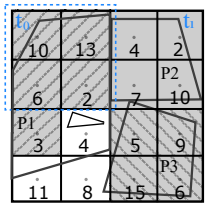


Figure 1: Progressive output of CA counties water cover



p_{id}	$Value$
2	17
1	31
3	21
2	6
3	5
1	3
3	9

Figure 2: Sequence of output tuples for polygons

complex polygons. Viper bootstraps answers within seconds using an offline one-time process, which constructs a *query-independent* raster index. This index maximizes computation as processing thousands of query polygons simultaneously and minimizes disk IO by reading pre-aggregated blocks of data. **Phase II** employs a novel priority-based query processing approach. It emphasizes on processing thousands of query polygons at the same time while fetching higher accuracy results early on. It prioritizes significant portions of the data, enhancing accuracy within a short timeframe and continuously improving until the exact answer is achieved.

Experiments on real data with hundreds of billions of pixels and tens of thousands of complex query polygons show that Viper achieves up-to 95% accuracy 40 times faster than the baseline. It also completes the processing up-to 16 times faster than baselines on high-resolution data. Furthermore, Viper only adds less than 1% disk and memory overhead to achieve these results. Contributions of Viper are summarized into three points. (1) Defines interactive exploration of zonal statistics. (2) Develops an on-the-fly index based on raster index, which helps to fetch an answer quickly. (3) Establishes priority order of query complex polygons and rasters that provides high accuracy early on.

This paper is organized as follows. Section 2 describes preliminary and problem definitions. Sections 3-4 detail implementations of index and interactive exploration approaches. Section 5 presents experimental analysis. Section 6 discusses related work. Section 7 concludes the work.

2 OVERVIEW

This section defines the problem and provides an overview.

2.1 Preliminary Definitions

Definition 2.1 (Raster Layer). A two-dimensional grid of pixels with w columns and h rows that is associated with a rectangular region on earth, e.g., Figure 2. A transformation function, grid-to-world (G2W), maps pixel locations on the grid to geographic locations, e.g., longitude and latitude. Each pixel has a numeric value for the area of earth it covers, e.g., temperature.

Definition 2.2 (Tile Grid). A tile grid groups raster pixels into equal-size and non-overlapping ranges of pixels called tiles. For example, the grid in Figure 2 has four tiles, each containing four pixels.

Definition 2.3 (Query Polygons (P)). A set of polygons that define geographical regions of interest. Each polygon $p \in P$ is represented as a pair $\langle p_{id}, geometry \rangle$, where p_{id} is a unique ID and *geometry* is a sequence of points that define one outer ring and zero or more inner holes. Figure 2 shows three polygons; the polygon with $p_{id} = 1$ has one hole while others have no holes.

Definition 2.4 (Aggregate function (f)). is a user-provided algebraic function that is applicable on pixel values, e.g., a query might calculate the **sum** of water areas or **average** temperature.

2.2 Problem Definition

This paper addresses the *progressive zonal statistics* problem. Zonal statistics problem is defined with a given raster layer r_o, a set of query polygons P, and an aggregate function f, the goal is to compute aggregate functions for all pixels in each query polygon. Similar to previous studies [11, 30], a pixel is considered inside a polygon if its center lies inside. Figure 2 shows three polygons where the matching pixels are highlighted correspondingly.

This paper focuses on *progressive answer* which delivers zonal aggregation results as a sequence of $\langle p_{id}, value \rangle$, where p_{id} is the query polygon ID and *value* is a *partial* aggregation, i.e., an aggregation of some overlapping pixels of p_{id}. These partial answers need to be further aggregated using the provided aggregate function to get the final answer. Figure 2 provides answers in a table of $\langle p_{id}, value \rangle$ pairs. Notice how the same p_{id} is repeated multiple times. One tuple in the answer might represent multiple pixels, e.g., $\langle 1, 31 \rangle$ represents a partial aggregate of four pixels with values 10, 13, 6, & 2. The sum of all values in the answer per p_{id} is the same as the sum of all overlapping pixels in the input. The order of the answer tuples does not affect its correctness. Query polygons could intersect with each other, this paper uses non-overlapping examples to provide a clearer view.

2.3 Proposed Approach Overview

The key idea to achieve interactive exploration in this work, is delivering progressive answers in a short time. This study uses a hybrid raster-vector index to speed up the calculation. It processes this index with a novel raster-vector method that provides an accurate result quickly and refines it until an exact answer is reached. Further, it adds a prioritization method so that the refinement step achieves high accuracy as soon as possible.

Figure 3 provides a high-level overview of the proposed method. Before any query process, the original input raster layer r_o is aggregated into a low-resolution layer r_l by applying the user-provided

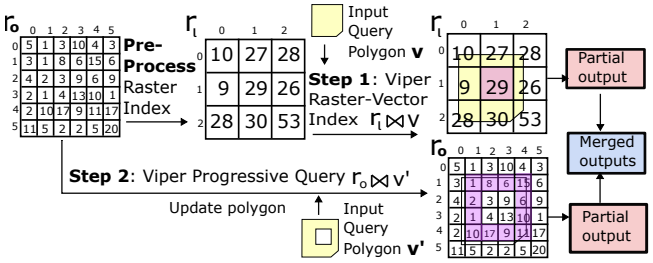


Figure 3: System flow pipeline

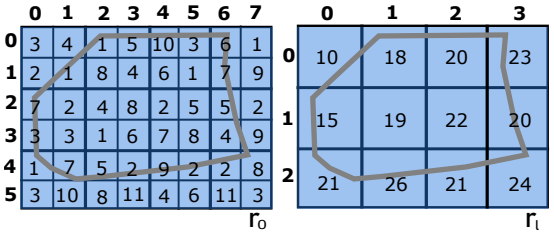


Figure 4: Raster index: aggregate pixels

aggregate function f on a fixed-size window. Figure 4 illustrates an example where the original raster r_o is of size 8×6, f is the summation function, and $s = 2$. Each 2×2 block is aggregated into a single pixel, termed *Superpixel*. To ensure an exact and efficient answer during the refining phase, each original pixel in r_o must overlap with a single superpixel in r_l, and the tile grid of r_l to perfectly align with the tile grid of r_o. To achieve both, we set s to a common factor of the tile size of r_o. For example, if the tile size is 128, we can use $s = 32$ or $s = 64$.

When the user provides a query, Viper first processes the low-res layer r_l with all query polygons to find all low-res pixels are *fully inside* query polygons to bootstrap the answer. Further, it updates the query polygons to avoid double-aggregation of the regions in low-res pixels as detailed in Section 3. The second step processes the remaining regions of the query polygons using the original raster r_o. We introduce a prioritization method that processes r_o in an order that helps in reaching high-accuracy quickly while still achieving an exact answer at the end as described in Section 4.

3 VIPER RASTER-VECTOR INDEX

This section describes the first part of the query processing. Given a set of query polygons, it computes an initial partial answer by processing the low-res raster layer r_l. This paper focuses on simultaneous processing of a large number of query polygons efficiently. To address that, we provide a novel raster-vector index that is constructed and processed on-the-fly. In this stage, it is crucial to identify the superpixels that are entirely inside query polygons. When working on a low-resolution layer, it is vital to exclude pixels beyond the bounds of polygons to avoid inaccurate results. Unlike existing work that produces an approximate answer [11, 38, 41], the partial answer this step provides can be further refined to produce the exact answer shown in the next section.

A straight-forward approach is to treat each superpixel as a rectangle and use ST_Contains from PostGIS for polygon-in-polygon

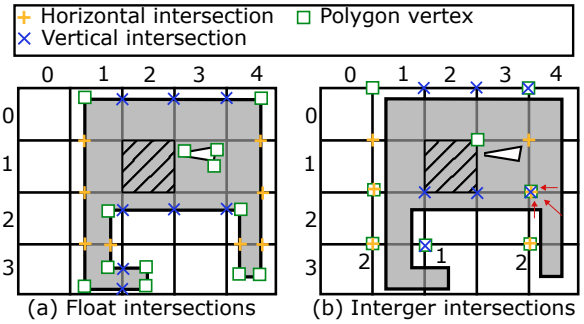


Figure 5: Polygon with one hole inside. Representation of intersections, vertices as both float and integer format. The hatched pixel is fully inside.

test with each query polygon, but it is very inefficient. Existing scan-line-based polygon filling algorithms [11, 41] find pixels whose centers are inside query polygons, but we need to find the pixels are completely contained in query polygons. Traditional scanline-based algorithms used in graphics fail to detect any parts of the geometries that fall between the horizontal scan lines, e.g., the small hole in pixel $(3, 1)$ in Figure 5. This section offers a different algorithm to find pixels that are fully inside query polygons while optimizing for multiple query polygons. In the rest of this section, we first formalize the conditions that make a pixel fully inside a polygon, then describe how to compute them efficiently for a large number of query polygons.

Pixel-in-polygon Conditions. For a rectangular pixel to be fully inside a polygon, it must satisfy three conditions:

(1) A pixel corner is inside the polygon, e.g., top-left corner.
(2) Pixel boundaries and polygon boundaries do not intersect.
(3) None of the polygon vertices is inside the pixel.

Figure 5(a) gives an example of a polygon with one hole on a raster layer. Each pixel is identified by a $(column, row)$ pair. The pixels $(0, 0)$, $(0, 1)$, $(0, 2)$ are not contained since their top-left corner is not inside the polygon (condition 1). Pixels $(2, 2)$, $(3, 2)$, $(4, 2)$ have a corner inside the polygon but pixel boundaries and polygons boundaries intersect (condition 2). Pixel $(3, 1)$ satisfies the first two conditions but fails with condition 3 since vertices of the hole (depicted by green squares) are inside the pixel. Only the hatched pixel $(2, 1)$ satisfies all three conditions. In contrast, systems that use scanline algorithm [11, 41] only tests the first condition to find all pixels have their center inside the polygon. Some approximate techniques [35] will consider the cell $(3, 1)$ as interior if they do not consider small holes.

Efficient Computation with Sorted Lists. This part describes a preliminary yet efficient method for computing all pixels inside query polygons. This work utilizes the grid structure of pixels to speed up tests, and is simple to implement as it relies mainly on binary search in sorted lists.

First, we clarify definitions used as illustrated in Figure 5. *Horizontal intersections*, depicted by yellow + marks, is a set of $\langle p_{id}, x, y \rangle$ tuples where each tuple indicates an intersection between the boundary (or a hole) of the polygon with id p_{id} and the horizontal grid lines of the pixel. *Vertical intersection*, depicted by blue × marks,

are the intersections between polygon boundaries and the vertical grid lines. Finally, *corners*, depicted by green squares □, is the set of polygon vertices and is part of the input. These three intersections, all formatted as a set of $\langle p_{id}, x, y\rangle$, can be computed efficiently in almost linear time in terms of the number of query polygon vertices. To do that, we scan each query polygon, one segment at a time, and compute the intersection between this line segment and both horizontal and vertical grid lines. Each intersection consists of two line segments and is computed in constant time.

These three sets help find all pixels completely inside query polygons as detailed next. First, sort the horizontal and vertical intersection lexicographically by (y, p_{id}, x) and (x, p_{id}, y), respectively. Such sort orders allow efficient search for intersections using binary search for a specific polygon. We build a kd-tree, or any spatial index, on pixel corners. Next, to test condition 1, we scan horizontal intersections in order and find horizontal segments inside the polygon as depicted by the yellow crosses in Figure 5. To test condition 2, we use the sorted horizontal and vertical intersections to search along the four edges of each pixel for intersections that belong to the same polygon. Finally, to test condition 3, we run a range search on the set of polygon vertices to find any vertex being tested inside the pixel. We omit the implementation details for brevity since we propose a more optimized algorithm.

This algorithm is simple to implement as it does not require complex geometric computations. Constructing the three sets is linear time if one polygon segment intersects with a few grid lines, which generally holds for real data. Sorting is $O(n \log n)$ where n is the number of polygon vertices which is proportional to the size of each set. Each pixel requires $O(\log n)$ for the binary searches and the tree search. Thus, this algorithm requires a running time of $O(w \cdot h \cdot \log n + n \log n)$, where w and h are the width and height of the raster layer in terms of the number of pixels.

Optimized Computation with On-the-fly Index. The algorithm depicted above has two shortcomings. First, it has a significant memory overhead for keeping lists of intersections where each coordinate is represented as a floating-point number. Second, frequent binary and tree search for each pixel slows executions. This part proposes an optimized in-memory structure for the intersections to address these drawbacks. The key idea is to convert all intersections from floating-point representation (x, y) to integer representation $(i = \lfloor x \rfloor, j = \lfloor y \rfloor)$ as shown in Figure 5(b). That is, horizontal intersections move to the left, vertical intersections move up, and polygon vertices move to the top-left. The cell at $(4, 2)$ gives an example of the three types of adjustments. Several intersections might be squished into one location, which the figure indicates by a subscript, e.g., the pixel at $(4, 3)$ has two coincident intersections of each type. Further, we compact horizontal intersections by combining each pair of intersections into a single tuple $\langle p_{id}, j, i_1, i_2\rangle$ to indicate that the segment $(y, i_1) - (y, i_2)$ is inside the polygon p_{id}. Such representation helps to store each intersection location as a single integer in the range $[0, w \times h]$ where w and h are the raster width and height in pixels, respectively. We redefine the three sets of intersections to form raster-vector index as follows.

The *horizontal intersections* is a list of intersection ranges in the form $\langle p_{id}, j, i_1, i_2\rangle$ sorted lexicographically by (j, p_{id}, i_1). The *vertical intersections* and *polygon vertices* are each represented as a set of intersections in the form $\langle p_{id}, i, j\rangle$ stored as a hash set. Notice that *vertical intersections* and *vertices* store each intersection only once, even if multiple intersections get snapped to the same integer location. It will be clear shortly how we still guarantee an exact answer with this new representation.

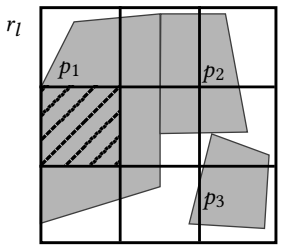
Algorithm 1: Check Pixel Fully Inside Polygons

```
Input  : H Sorted horizontal ranges [⟨p_id, j, i1, i2⟩],
         V Vertical intersections set {⟨p_id, i, j⟩},
         C Polygon vertices set {⟨p_id, i, j⟩}
Output : Pixels: list ⟨p_id, i, j⟩ of fully inside pixels
1  k2 ← 0
2  for k1 = 0 to |H| do
3  |  (p_id, j, i1, i2) ← H[k1]
4  |  while k2 < |H| and H[k2] ≺ H[k1] do
5  |  |  k2 + +
6  |  while k2 < |H| and H[k2].p_id == p_id and H[k2].j == j + 1
   |   and H[k2].i1 < i2 do
7  |  |  for i = max(i1, H[k2].i1) + 1 to min(i2, H[k2].i2) do
8  |  |  |  if ⟨p_id, i, j⟩ ∉ V and ⟨p_id, i + 1, j⟩ ∉ V then
9  |  |  |  |  if ⟨p_id, i, j⟩ ∉ C then
10 |  |  |  |  |  Pixels ≪ ⟨pID, x, y⟩
11 |  |  if H(k2).i2 < i2 then
12 |  |  |  k2 + +
13 |  |  else
14 |  |  |  break
15 return Pixels
```

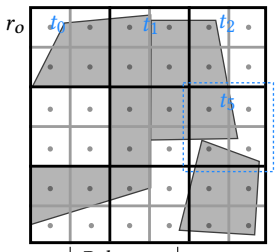
Algorithm 1 gives the pseudo-code of the proposed algorithm. The algorithm starts in line 1 by initializing two pointers k_1 and k_2 on the horizontal intersections list. k_1 is used to find ranges that satisfy condition 1 and k_2 is used to test the bottom edge of each pixel for condition 2 while avoiding a range search. The loop in line 2 scans all horizontal ranges satisfying condition 1. Then, the loop in line 4 advances the second pointer k_2 to the first range on the next row that overlaps with the current range horizontally and belongs to the same polygon. This step utilizes the existing sort order and defines the ordering operator $\prec$ as follows:

$$a \prec b \iff a.j < b.j + 1 \vee (a.j = b.j + 1 \wedge (a.p_{id} < b.p_{id} \vee (a.p_{id} = b.p_{id} \wedge a.i_2 < b.i_1)))$$

With this definition, the loop in line 4 advances k_2 until the first range in the next row $(j + 1)$ that horizontally overlaps with the current range pointed by k_1. For example in Figure 5, if k_1 points to the range $\langle j = 1, i_1 = 1, i_2 = 4\rangle$, then k_2 will point to the range $\langle j = 2, i_1 = 1, i_2 = 4\rangle$. The pixels in the horizontal intersection of both ranges satisfy condition 1 and half of condition 2, i.e., the top and bottom edges are clear of intersections. Remember that multiple ranges could be in the next row (pointed by k_2). Thus, the loop in line 6 iterates over all those overlapping ranges. To complete the test of condition 2, we need to ensure no intersections along the left and right edges of the pixel. The for loop in line 7 iterates over all pixels that overlap the two ranges and check the vertical intersections set. Notice that we do not care where the exact intersection is or whether there is more than one intersection which makes the hash set perfect for this test in constant time as shown in line 8. Next, line 9 tests condition 3 in constant time using the corners hash set. Finally, the pixel is added to the output *Pixels* list



p_{id}	OA	p_δ	p_π
1	14	0.29	0.71
2	6	0.0	1.0
3	5	0.0	1.0



t_{id}	Polygons	t_π
0	p_1	0.71
1	p_1, p_2	2.71
2	p_2	1.0
5	p_2, p_3	2.0

Figure 6: Polygon progress and priority, and tile priority

if all three conditions pass. The only remaining step is to advance k_2 if the range it points to will not overlap with future horizontal ranges by this polygon, i.e., $H(k_2).i_2 > H(k_1).i_2$. Otherwise, we do not advance that range since it could be used in the next iteration when k_1 is advanced by one.

Polygon Adjustment: We use the above algorithm to find all superpixels in the low-res raster r_l that are completely inside the query polygons. The partial aggregates for the matching superpixels are added to the output. To ensure that the regions covered by the superpixels will be excluded in the next-phase, we adjust the polygons by subtracting the regions covered by these superpixels. Rather than running a complex polygon difference operation, we utilize the fact that the superpixels are completely contained in the polygon, making it easy to add holes to existing polygons without doing any complex geometric tests. Thus, we run the object delineation algorithm [28] to create rings around matching superpixels and just add them as holes to corresponding query polygon.

4 VIPER PROGRESSIVE QUERY

This section describes the second step. Given updated query polygons and the high-res (original) raster r_o, it uses the scanline method [11] to progressively refine the answer until an exact answer is reached. The main contribution of this part is to prioritize the processing order to reach higher accuracy early on.

Notice that the scanline zonal statistics method was designed for batch processing and not progressive queries. However, since it processes one tile at a time, we can easily adapt it by producing the output immediately after processing each tile to ensure progressive answer. Thus, whenever it finds a pixel (x, y, m) to be inside a polygon p_{id}, we emit the value (p_{id}, m). The drawback of this straight-forward adaptation is that it might start by processing tiles that do not benefit the overall progressive answer. It would be better for the query answer to prioritize a tile that contains polygons that were never processed over another tile that contains polygons that were partially processed. Therefore, this work provides a *priority processing method* that tries to reach higher accuracy early on by prioritizing tiles. First, we start by defining priority and then we show how it improves the scanline method.

Definition 4.1 (Polygon Progress). Polygon progress $p_\delta \in [0, 1]$ is defined as the ratio of processed polygon area to the overall area of the polygon.

$$\text{Polygon progress: } p_\delta = \frac{\text{processed area}}{\text{polygon area (OA)}}$$

We avoid using actual polygon areas, since we process query polygons in pixels, we might never reach exactly 100% progress. Thus, we represent original area (OA) of the polygon in terms of the number of original pixels (r_o) in the polygon. This number can be easily calculated as we compute the intersections of the scanline process so it does not add a significant overhead. The processed area is incremented by one for each matching pixel of r_o and by s^2 for each matching superpixel of r_l. Figure 6 illustrates three query polygons with the low-res raster on the left and the original raster on the right. The query areas are 14, 6, and 5, respectively. The hatched cell indicates one superpixel that is fully inside the polygon p_1 on lower-res raster. The right image shows updated polygons over the high-res raster r_o, where each tile groups four pixels of r_o. After processing the superpixel inside p_1, the progress is $4/14 = 0.29$. The polygons p_2 and p_3 are not processed yet so they have a progress of zero.

Definition 4.2 (Polygon Priority). Priority of a polygon p_π is $1 - p_\delta$.

Polygon priority is also in the range $[0, 1]$. The higher the value is, the more important it is to process this polygon because only a few pixels were processes at that point. In Figure 6, the priority of the polygons 2 and 3 is 1.0 since none of their pixels were processed while the priority of polygon 1 is $1 - 0.29 = 0.71$

Definition 4.3 (Tile Priority). The priority of a tile t_π is the sum of all polygon priorities that have at least one pixel in this tile. Tile priority: $t_\pi = \sum_{p \cap t \neq \emptyset} p_\pi$

Figure 6 shows the priority of four of the nine tiles. t_0 contains only one polygon so its priority is equal to that of p_1. t_2 overlaps both p_1 and p_2 so its priority is $0.71 + 1.0 = 1.71$. t_5 overlaps both p_2 and p_3 which makes its priority equal to 2.0. This means that t_5 is the tile with the highest priority which makes sense because

Algorithm 2: Priority Tile Processing

```
Data: Intersection Array: I = [⟨t_id, p_id, x, y⟩]
      Polygon array P = [⟨OA, tiles⟩]
      Tile array T = [⟨t_π, t_i⟩]
while true do
    t_id = argmin{T[t_π]}
    return if T[t_id].t_π = 0
    T[t_id].t_π = 0
    Load tile t with ID t_id from r_o
    Initialize a hashtable PP for new polygon pixels as
     num-new-pixels
    for i ∈ I[T[t_id].t_i..T[t_id + 1].t_i] do
        output ≪ (i.p_id, t.get-pixel(i.x, i.y))
        num-new-pixels + +
        PP[i.p_id] = num-new-pixels
    for (p_id, num-new-pixels) ∈ PP do
        Remove the processed tile t_id from P[p_id].tiles
        diff-priority=num-new-pixels/P[p_id].OA
        for t_id ∈ P[p_id].tiles do
            T[t_id].t_π − = diff-priority
```

Table 1: Raster Data

Dataset	file size	resolution (pixels)
MERIS	7.8GB	129,600×64,800
Treecover	560 GB	1,296,001×493,200

Table 2: Query Polygons

Dataset	# query polygons	# segments	$\frac{\#segments}{query}$
ZIP code	33,144	52,894,188	1,560
US Counties	3,108	51,638	17
US States	49	165,186	3,371
World	284	3,817,412	13,442

processing it will produce some result for both p_2 and p_3 which have zero progress. The proposed priority algorithm processes tiles from highest to lowest priority instead of the default order by tile ID. After processing each tile, the algorithm efficiently updates tile progress to reflect the new polygon progress.

Algorithm 2 gives the pseudo-code of the priority processing algorithm which takes three inputs, I, P, and T. I is the list of intersections (I) calculated by the scanline algorithm and ordered by tile ID. P is the list of polygons that record the overall area (OA) for each polygon and the list of overlapping tiles. T is a list that records the priority of each tile (t_π) and the index of the first intersection in I for the tile t_i. Arrays P and T are indexed by the polygon ID p_{id} and tile ID t_{id}, respectively. These structures can be easily initialized during the processing of superpixels and the calculation of the intersections array I but we omit this part for brevity. This algorithm runs in a loop that processes one tile at a time as follows. First, line 2 finds the tile with the highest priority. While a max-heap can find that tile efficiently, we found that using a simple array is more efficient since updating the heap is very costly. A max-heap reduces the cost of finding the tile with highest priority from $O(n)$ to $O(\log n)$ but increases the update cost from $O(k)$ to $O(k \log n)$, where k is the number of updated tiles. We found that the number of updated tiles is generally large enough to make the max-heap ineffective. If the highest priority is zero, the algorithm terminates as there are no more tiles to process. Otherwise, the tile is loaded from disk to start processing (line 5).

In line 6, we initialize a hashmap to keep track of the number of pixels processed in each polygon which we use to update the priorities. The for loop in line 7 loops over all the intersections for this tile, progressively outputs all matching pixels, and increments the number of pixels for each polygon. After processing the tile, we set its priority to zero to mark it as done. Then, the for loop in line 11 loops over the map of processed polygons (PP) to update the priority. Line 12 removes the processed tile from the list of tiles in this polygon to reduce the size since this tile will not be considered anymore. Line 13 computes the reduction in the priority of the polygon given the number of new pixels processed for it. Then, the for loop in line 14 updates the priority of all matching tiles. After that, the algorithm repeats to find the next tile to process.

5 EXPERIMENTS

This section provides a comprehensive experimental evaluation on the performance of this work. Section 5.1 describes the experimental setup with the system environment, datasets, and baseline model. Section 5.2 describes the overall comparison of accuracy and performance. Sections 5.3 and 5.4 study the effect of each proposed component, superpixel aggregation and prioritization, individually. Section 5.5 details tuning analysis for different cases.

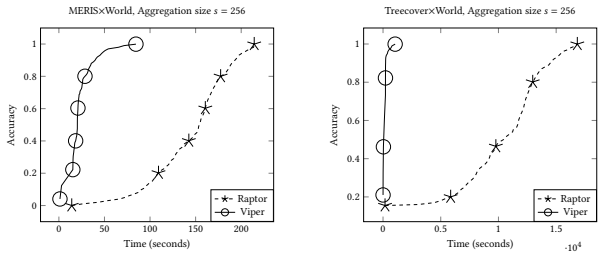


Figure 7: Progress accuracy for large query polygons

5.1 Setup

We run all experiments on a single machine with Intel Xeon E3-1220 v5 3.00GHz quadcore processor, 64GB RAM, and 2TB HDD on Ubuntu 16.04.2 applied Java 1.8.0_102.

The proposed technique is compared to Raptor [30] zonal statistic on a single machine. Raptor was not designed to produce progressive results so we modified it to produce partial results. This work focuses on single machine experiments to best study the effect of computation saving and prioritization. Extending this implementation to parallel environments is left for future work. Prior work showed that Raptor is the most efficient method for zonal aggregation over arbitrary polygons [30]. Other techniques are limited to rectangular queries or solve a different problem. ACT is the closest we found since it speeds up point-in polygon queries, it lacks of scalability according to [32].

The main performance metric we use is running time and accuracy over time. We compute four aggregate functions as results: minimum, maximum, sum, and count. At each timestamp, we calculate the current accumulated results by adding new found values. This experiment uses only the *sum* for accuracy analysis for consistency. Mean percentage error (MPE) is measured and reported along running time. In the formula below, sum_{pID} represents current accumulated sum for each polygon and the *sum* is the ground truth final sum for corresponding polygon.

$$\text{MPE} = \frac{1}{n} \sum_{p_{id}=1}^{n} \frac{|sum_{p_{id}} - sum|}{sum}$$

$$\text{Accuracy} = 1 - \text{MPE}$$

Tables 1 and 2 list the datasets and query polygons. This section focuses on large satellite data such as MERIS and Treecover with high resolution indicating the massive number of pixels, and each pixel stores one value as integer. The query polygons are ordered from the smallest geographical area (ZIP Code) to the highest (World countries).

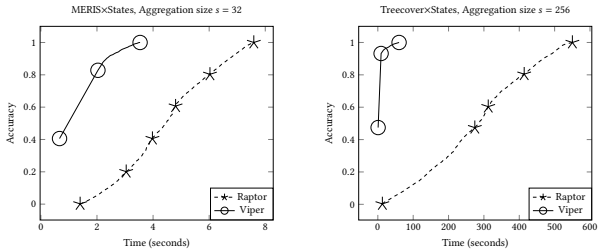


Figure 8: Progress accuracy for medium query polygons

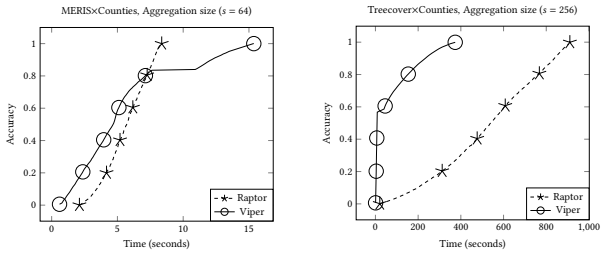


Figure 9: Progress accuracy for medium query polygons

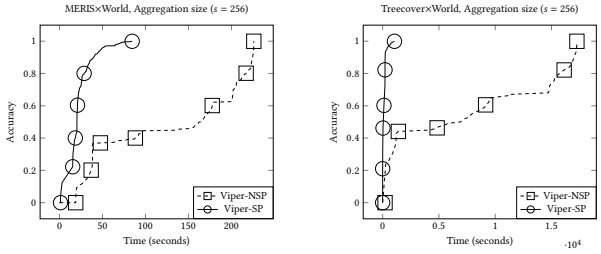


Figure 10: Effect of superpixels with large query polygons

5.2 Baseline Comparison with Viper

This experiment shows the performance and accuracy comparison between *Viper* and Raptor. Figures 7- 9 plot the accuracy as a function of time. Since all methods are exact, they start with an accuracy of zero and end with an accuracy of one. These figures provide two ways to compare methods. First, the curve that reaches an accuracy of 1.0 earlier requires less overall processing time. Second, a steep curve that grows faster provides better progressive result since it provides higher accuracy quickly.

Figure 7 shows the performance with large query polygons. Viper yields faster running time and better progress. The reason is that the big query polygons match with many super pixels that help Viper to lower the running time and get a big jump in accuracy briefly. As the raster resolution increases, the gain is even higher. With high-resolution data, Viper reaches 95% accuracy in 378 seconds and 100% accuracy in a little over 1,000 seconds. In comparison, Raptor reaches 95% in 15,426 seconds and 100% in 16,883. Also the effect of the super pixels is evident as it gives Viper a big boost at the beginning to reach 80% accuracy is less than 30 seconds.

Figure 8 shows the performance with medium query polygons, US states. We can observe a similar behavior but the gain starts to decrease since query polygons now match with fewer super pixels. In Figure 9, the US county dataset still yields a better performance

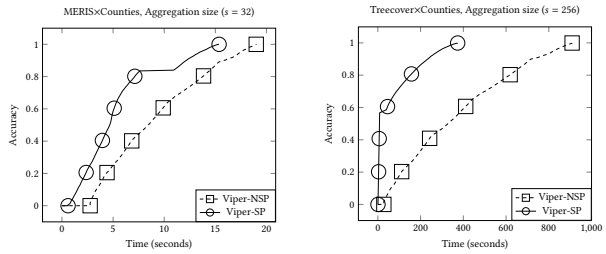


Figure 11: Effect of superpixels for medium query polygons

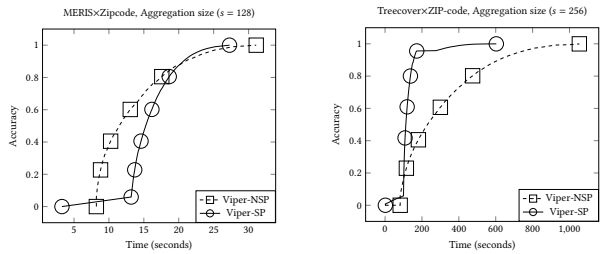


Figure 12: Effect of superpixels on small query polygons

with the high resolution raster but it becomes slower with the medium resolution raster due to matching fewer super pixels. Notice that in this case the baseline takes only seven second anyway so there is actually very little room for improvement. In conclusion, as the query polygons get bigger and the raster resolution get higher, Viper can provide significantly better performance.

5.3 The effect of superpixel aggregation

This section focuses on the effect of superpixel aggregation on the accuracy. We compare two variations of Viper, *Viper-SP* applies super pixel aggregation and prioritization, and *Viper-NSP* which applies only the prioritization. Viper-SP uses the same experimental aggregation size and vector data as in the previous experiments. Figures 10-12 show the accuracy improvement over time.

It is evident that Viper-SP gains an initial advantage from processing superpixels, e.g., Figures 10 and 11. This behavior is amplified for high-resolution raster and big query polygons, e.g., Figure 10 with Treecover and World datasets where Viper-SP reaches 90% in 200 seconds while Viper-NSP takes 17,000 seconds to reach 90%. On the other extreme, if we have a lower-resolution raster and many small polygons, e.g., MERIS and ZIP code in Figure 12, the superpixel aggregation adds some overhead with no significant benefit. In this case, Viper-NSP can be more efficient.

5.4 Effect of prioritization

This experiment focuses on the effect of prioritization in progressive query results. We compare two variations of Viper, *Viper-pt* applies super pixel aggregation and prioritization, and *Viper-np* which applies only the super pixel aggregation but no prioritization. Figures 13-15 show the accuracy improvement over time. There are two goals of this experiment. The first is to study the effect of prioritization on improving the accuracy. The second is to evaluate the overhead processing time of prioritization.

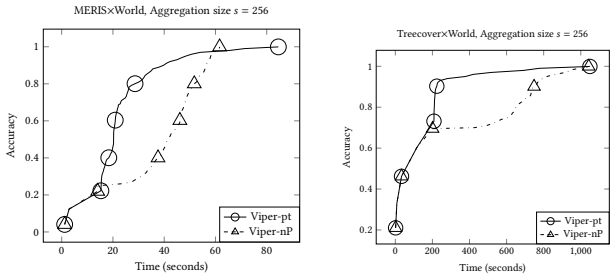


Figure 13: Effect of prioritization for large queries

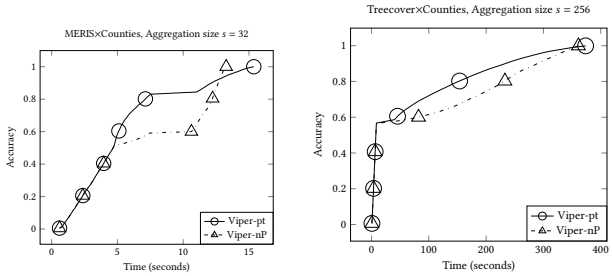


Figure 14: Effect of prioritization over medium queries

Figures show Viper-pt ramps up the accuracy much faster due to the effect of prioritization. In Figure 13, it reaches 90% accuracy in about 223 seconds as compared to 748 seconds with no priority when querying high-resolution data, Treecover. Although the gap between these two methods narrows while working on smaller polygons such as US county in Figure 14, the priority still benefits and it reaches 80% accuracy in about 150 seconds as compared to 250 seconds with no priority. Both techniques get the same initial boost from super pixels but the prioritization gives Viper a second boost showing its importance for progressive query answering.

From the performance perspective, we notice that the processing overhead depends heavily on the number of query polygons. For world and county datasets where we have up-to a few thousand query polygons, the overhead is minimal. Although in Figure 13 Viper with priority technique on MERIS and world polygons still can observe overhead, the priority already provides 97% accuracy when Viper-np reaches 100% accuracy. With nearly 33K query polygons in the ZIP code dataset in Figure 15, the overhead starts to be significant. In addition, due to the relatively small size of ZIP code polygons, they match with few super pixels which leaves more data to process in the original dataset which adds to the overhead of prioritization. In summary, these experiments show clear advantage of prioritization on accuracy. The overhead starts to be significant only if have tens of thousands of small query polygons.

5.5 Breakdown

Table 3 shows the storage overhead of the raster index, i.e., the pre-aggregated raster. For each raster, we show the percentage overhead as the aggregation size varies. The smaller the aggregation size, the more the superpixels will be created which means a higher overhead. Also, notice that each superpixel contains four floating-point values, min, max, sum, and count, which add up to 16 bytes.

Table 3: Aggregated Raster Index Overhead

Dataset	Aggregation size (s)	File size	Overhead %
MERIS	32×32	125MB	1.565%
MERIS	64×64	31MB	0.391%
MERIS	128×128	7.8MB	0.0978%
MERIS	256×256	1.97MB	0.0245%
Treecover	64×64	2.3GB	0.392%
Treecover	128×128	0.58GB	0.0979%
Treecover	256×256	145MB	0.0245%

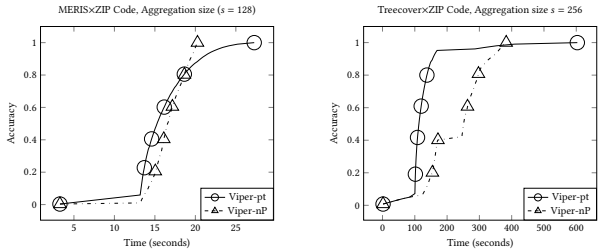


Figure 15: Effect of prioritization over small queries

As the table shows, the storage overhead ranges from 0.02% to 1.6% when the aggregation size ranges from 256 to 32. This shows that the storage overhead is minimal.

Figure 16 shows the memory overhead of the raster-vector index that Viper uses. This includes both the new raster-vector index that find superpixels that are completely contained in query polygons as well as the Raptor index that find pixels with centers in query polygons. Since these two indexes are used in separate phases, we report the maximum of the two. The x-axis shows varying spatial index sizes and query polygons. Also, to put the numbers in perspective, we normalize all numbers by dividing by the size of the baseline Raptor index. Any numbers that are below the red line indicate a lower memory overhead of Viper. In the figure, for almost all the cases, Viper significantly reduces the memory overhead as compared to Raptor. The reason is that the matching superpixels require only a small memory overhead and they also reduce the query polygon complexity due to the added holes which reduce the size of the Raptor index used with the original raster. Further, by comparing Viper-pt to Viper-np, we see that the overhead of pioritization is minimal. Similar to previous experiments, ZIP codes query polygons add an additional overhead due to the small query polygon size and the large number of queries.

Figures 17-20 show the time breakdown into two steps, *superpixel* and *priority tiles*. For clarity of the chart, we limit the range of the y-axis and we include the overall time at the top for reference. As the aggregation size increases, the superpixel time decreases while the priority tile time increases. First, as the aggregation size increases, we have fewer superpixels which reduces the superpixel processing time. In addition, a larger aggregation size means bigger superpixels which results in fewer matches of completely contained superpixels. This means that the query polygons will match more original pixels which increases the tile prioritization time. Together, these two observations result in a trade-off that allows us to tune the aggregation size to reduce the overall running time.

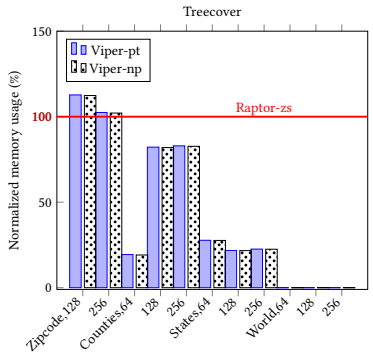


Figure 16: Normalized memory usage in different resolutions Treecover and vectors, comparison with baseline

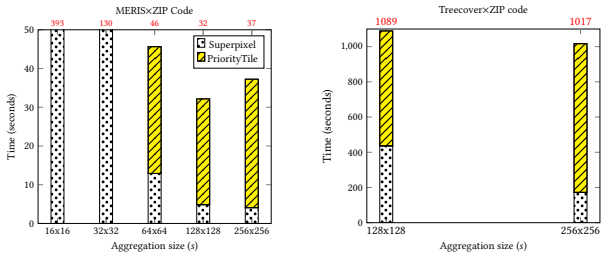


Figure 17: Effect of aggregation size with small queries

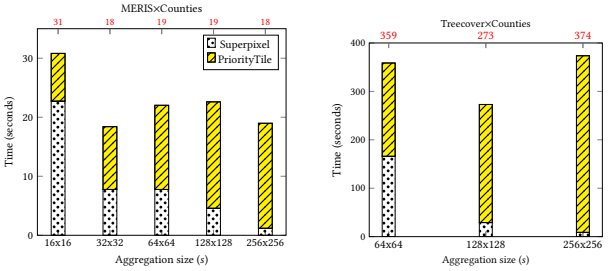


Figure 18: Effect of aggregation size with medium queries

6 RELATED WORK

This section summarizes the related to zonal statistics problem. We summarize indexes that assists with zonal statistics and interactive approaches for spatial data exploration.

6.1 Spatial Indexing

Zonal statistics algorithms have three types: vector, raster, and raster-vector [30]. Vector methods use point-in-polygon queries with pixel-to-point conversion. Raster methods process polygons by converting them to raster. Raster-vector methods combine raster and vector data without conversion. This section covers spatial indexes for optimizing these algorithms.

Vector indexes can be built on point or polygon datasets. Spatial indexes like R-tree [15], Quad-tree [13], and other specialized indexes [9, 16] suffer from high memory usage [32]. Techniques like Data cube [14] and works [18, 21, 34] modify traditional indexes for partial aggregates and rectangular range queries. Methods that

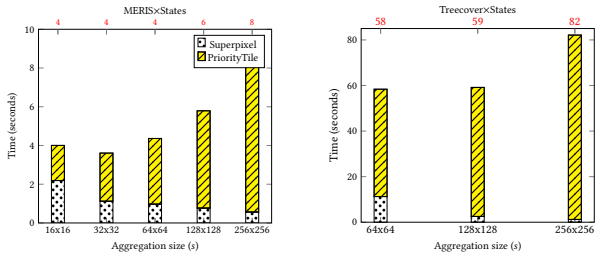


Figure 19: Effect of aggregation size with large queries

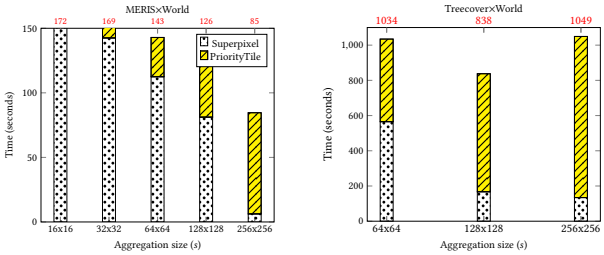


Figure 20: Effect of aggregation size with very large queries

support arbitrary polygons [19, 38] are limited to one polygon at a time and suffer from the overhead of pixel-to-point conversion.

Raster indexes build a hierarchical structure such as a quad-tree or grid structure and precomputes partial aggregates to speed up zonal aggregation [5, 10, 29]. CCQ-Tree [44] utilizes GPU to speed up the query processing. All these indexes are limited to processing rectangular queries and can only process one query range at a time.

Raster-vector indexes are built based on both raster and vector datasets. Raptor [30] builds an on-the-fly index to speed up the raster-vector join but does not provide any partial aggregates. IDEAL [35] rasterizes vectors and keeps track of spatial relationship between raster and vector but suffers from excessive storage for raster layers [11] and processes one query polygon at a time which limits its performance.

Our approach belongs to hybrid raster-index structure. It distinguishes itself from existing methods by speeding both the pixel selection and aggregation steps. Our method has two unique advantages. First, it is the only one optimized to aggregate a large number of queries rather than a single query polygon. Second, it is the only *progressive* method continuously updating the results to support interactive exploration of big satellite data.

6.2 Interactive Approaches

Interactive spatial data exploration systems require to achieve high accuracy answers in a short time. These methods can be categorized into approximation, query result caching, hardware acceleration, and progressive methods.

Approximate query processing (AQP) trades off the accuracy for lower latency. Standard methods include sampling and region filtering. Sampling methods [1–3, 40] provide sampled data or estimated results interactively. Region filtering [4] approximates results with different geometric shape filters. Works [18, 25, 38] comprises multi-resolution aggregation layers, continuously filtering cells by MBR with pixel values. Other works [42, 45] provide an approximated result after polygon rasterization. Approximate results are

not always acceptable, especially when the error is not bounded. Further, all these methods process one query polygon at a time.

Query result caching techniques cache the results of frequent queries as optimization. SpaceOdyssey[22] use an adaptive index to detect hot query spot, while this method drawback with redefining the index structure each query time. HQfilter[26] contributes to reducing data processing time by filtering out empty query results but restricted to rectangular MBR.

Hardware acceleration, such as GPU, is one method. Works [9, 41, 43] use GPUs to optimize point-in-polygon query but are limited by time and memory usage in converting high-resolution rasters into vectors. GPU also is unsuitable for dealing with extensive satellite data that can only be stored on disk.

Progressive query methods provide early results as the query runs and refine the results until execution finishes. We call a method that returns an exact answer at the end an *exact progressive* method, otherwise, it is an *approximate progressive* method. Progressive merge join [8] is an exact method but is limited to rectangular ranges. Approximated progressive work as pCube [24] also only have rectangular filter. Progressive query implemented with visualization as [39] does not provide exact answer. Other works [12] either solve progressive chunks with approximation or [37] focus on image progressive refine. Works [6, 20, 27, 36] are continuous query algorithms and take moving data stream, which is a different problem. These works have two differences from our method. First, they focus on streaming systems with a limited time window while we consider all the data. Second, they are main-memory techniques which assume that most of the data is in-memory while our method is disk-based and can scale beyond the memory size.

Our approach aims to realize an interactive exploration through progressive outputs. We contribute by supporting arbitrary polygons, outputting aggregation values as progressive feedback, and providing an exact final answer.

7 CONCLUSION

This paper presents an interactive exploration method on large satellite data, through progressive query processes separated into two phases. First step queries low-res rasters layer and query polygons with an on-the-fly raster-vector index. The second step queries the original layer with another on-the-fly raster-vector index, and refines results with a priority order unlit obtaining exact answer. The experimental analysis of real datasets shows the proposed algorithm provides high accuracy results quickly, especially for the combination of high resolution rasters and big polygons.

REFERENCES

[1] Sameer Agarwal et al. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys (EuroSys '13)*. 14 pages.
[2] Wan D. Bae et al. 2007. An Interactive Framework for Raster Data Spatial Joins. In *GIS*. Article 4, 8 pages.
[3] Wan D. Bae et al. 2010. IRSJ: Incremental Refining Spatial Joins for Interactive Queries in GIS. *Geoinformatica* 14, 4 (oct 2010), 507–543.
[4] Thomas Brinkhoff et al. 1994. Multi-Step Processing of Spatial Joins. In *SIGMOD*.
[5] Nieves R. Brisaboa et al. 2017. Efficiently Querying Vector and Raster Data. *Comput. J.* 60, 9 (02 2017), 1395–1413.
[6] Sanket Chintapalli et al. 2016. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *IPDPSW*. 1789–1792.
[7] Ramesh Dhungel et al. 2020. Restricted water allocations: Landscape-scale energy balance simulations and adjustments in agricultural water applications. *Agricultural Water Management* 227 (2020), 105854.
[8] Jens-Peter Dittrich et al. 2002. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *PVLDB*. Elsevier, 299–310.
[9] Harish Doraiswamy et al. 2016. A GPU-based index to support interactive spatio-temporal queries over historical data. In *ICDE*. 1086–1097.
[10] Ahmed Eldawy et al. 2015. SHAHED: A MapReduce-based system for querying and visualizing spatio-temporal satellite data. In *ICDE*. 1585–1596.
[11] Ahmed Eldawy et al. 2017. Large Scale Analytics of Vector+Raster Big Spatial Data. In *SIGSPATIAL* (Redondo Beach, CA, USA). Article 62, 4 pages.
[12] Jean-Daniel Fekete. 2015. Progressivis: A toolkit for steerable progressive analytics and visualization. In *1st Workshop on Data Systems for Interactive Analysis*.
[13] Raphael A Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4, 1 (1974), 1–9.
[14] Jim Gray et al. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* 1, 1 (1997), 29–53.
[15] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*. 47–57.
[16] Andreas Kipf et al. 2020. Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins. In *EDBT*. 347–358.
[17] Susana Ladra et al. 2017. Scalable and Queryable Compressed Storage Structure for Raster Data. *Inf. Syst.* 72, C (dec 2017), 179–204.
[18] Iosif Lazaridis and Sharad Mehrotra. 2001. Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure. In *SIGMOD* (Santa Barbara, CA). 401–412.
[19] Lauro Lins et al. 2013. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *TVCG* 19, 12 (2013).
[20] Mohamed F Mokbel and Walid G Aref. 2005. GPAC: generic and progressive processing of mobile queries over mobile data. In *MDM*.
[21] Dimitris Papadias et al. 2001. Efficient OLAP operations in spatial data warehouses. In *SSTD*. Springer, 443–459.
[22] Mirjana Pavlovic et al. 2016. Space Odyssey: Efficient Exploration of Scientific Data. In *ExploreDB*. 12–18.
[23] Michael Reibel et al. 2007. Areal interpolation of population counts using pre-classified land cover data. *Population Research and Policy Review* 26, 5 (2007).
[24] M. others Riedewald. 2000. pCube: Update-efficient online aggregation with progressive feedback and error bounds. In *SSDBM*. 95–108.
[25] Mohamed Sarwat. 2015. Interactive and Scalable Exploration of Big Spatial Data – A Data Management Perspective. In *MDM*, Vol. 1.
[26] Akil Sevim and Ahmed Eldawy. 2021. HQ-Filter: Hierarchy-Aware Filter For Empty-Resulting Queries in Interactive Exploration. In *MDM*. 49–58.
[27] Salman Ahmed Shaikh et al. 2020. GeoFlink: A distributed and scalable framework for the real-time processing of spatial streams. In *CIKM*.
[28] Zhuocheng Shang and Ahmed Eldawy. 2022. Object Delineation in Satellite Images. *SpatialGems* (2022).
[29] Fernando Silva-Coira et al. 2020. Efficient processing of raster and vector data. *Plos one* 15, 1 (2020).
[30] Samriddhi Singla et al. 2019. Raptor: large scale analysis of big raster and vector data. *PVLDB* 12, 12 (2019).
[31] Samriddhi Singla et al. 2021. Experimental Study of Big Raster and Vector Database Systems. In *ICDE*.
[32] Samriddhi Singla et al. 2021. The Raptor Join Operator for Processing Big Raster + Vector Data. In *SIGSPATIAL*. ACM.
[33] Samriddhi Singla et al. 2021. WildfireDB: An Open-Source Dataset Connecting Wildfire Spread with Relevant Determinants. In *NeurIPS*.
[34] Yufei Tao et al. 2002. Aggregate processing of planar points. In *EDBT*. Springer, 682–700.
[35] Dejun Teng et al. 2021. IDEAL: a Vector-Raster Hybrid Model for Efficient Spatial Queries over Complex Polygons. In *MDM*.
[36] Wee Hyong Tok et al. 2006. Progressive Spatial Join. In *SSDBM'06*. 353–358.
[37] Matt Williams and Tamara Munzner. 2004. Steerable, progressive multidimensional scaling. In *IEEE Symposium on information Visualization*. IEEE, 57–64.
[38] Christian Winter et al. 2021. GeoBlocks: A Query-Cache Accelerated Data Structure for Spatial Aggregation over Polygons. In *EDBT*. 169–180.
[39] Wei Xiong et al. 2019. Geo-Gap Tree: A Progressive Query and Visualization Method for Massive Spatial Data. *IEEE Access* 7 (2019).
[40] Jia Yu et al. 2020. Tabula in Action: A Sampling Middleware for Interactive Geospatial Visualization Dashboards. *PVLDB* 13, 12 (aug 2020), 2925–2928.
[41] Eleni Tzirita Zacharatou et al. 2017. GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons. 11, 3 (nov 2017), 352–365.
[42] Eleni Tzirita Zacharatou et al. 2020. The case for distance-bounded spatial approximations. *arXiv preprint arXiv:2010.12548* (2020).
[43] Jianting Zhang. 2011. Speeding up Large-Scale Geospatial Polygon Rasterization on GPGPUs. In *HPDGIS*. 8 pages.
[44] Jianting Zhang et al. 2010. Indexing Large-Scale Raster Geospatial Data Using Massively Parallel GPGPU Computing. In *GIS*. 450–453.
[45] Geraldo Zimbrao and Jano Moreira De Souza. 1998. A raster approximation for processing of spatial joins. In *VLDB*, Vol. 98. 24–27.