Raptor: Large Scale Analysis of Big Raster and Vector Data

Manange Data for and with Machine Learning
Final Report Spring 2025

Simón González
u1528314
u1528314@umail.utah.edu

Devagopal Anil Sreenivas Maya
u1527544
u1527544@umail.utah.edu

Maedeh Haghbin Yousefi
u1500341
u1500341@umail.utah.edu

Abstract

Geospatial analysis is undergoing unprecedented growth due to the explosion of remote sensing data. Traditional systems are typically optimized for either raster or vector data, but not for efficiently handling both simultaneously. We implement and evaluate Raptor [1] , a novel processing model designed to efficiently handle queries that involve both raster and vector data without requiring format conversions. Focusing on the zonal statistics problem—calculating aggregate statistics from raster data within vector-defined polygons—we compare six methods across three approaches: vector-based, raster-based, and Raptor-based. Our evaluation shows that Raptor-based approaches outperform traditional methods, validating their theoretical advantages. We also explore opportunities for integrating Raptor with distributed computing frameworks to further enhance scalability for real-world applications. The code for this project is available at: `https://github.com/simonpedrogonzalez/raptor-cs6964.git`

1 Introduction

Modern geospatial analysis increasingly relies on massive and heterogeneous datasets, fueled by advances in remote sensing and data collection technologies. NASA's Earth Observing System Data and Information System (EOSDIS) alone stores over 17 petabytes of raster data and is projected to exceed 330 petabytes by 2025. This rapid growth has highlighted inefficiencies in current spatial processing systems, which are typically optimized for raster data - grid values representing continuous phenomena such as imagery or elevation [2, 3] - or vector data - geometric features such as points, lines and polygons representing discrete objects such as roads or administrative boundaries [4, 5].

Many geospatial tasks require joint analysis of raster and vector data. A common example is the zonal statistics problem, where statistical aggregates (e.g., sum, mean, count) are computed over raster values within vector-defined zones [6]. Traditional GIS systems handle this by converting data between formats: vector-based methods transform raster cells into points for geometric containment tests, while raster-based methods rasterize polygons into masks [7, 8]. These conversions introduce significant computational overhead, potential spatial inaccuracies, and poor scalability as datasets grow in resolution and complexity.

These limitations are addressed by Raptor, a processing model designed to perform raster-vector queries directly on native data structures without format conversion [1]. Raptor eliminates the need for intermediate representations, reducing processing time and memory usage while preserving

spatial precision. We focus on the zonal statistics problem as a representative case, comparing six implementations—Naive Point-in-Polygon, QSplit, Masking, Clipping, Scanline, and Aggregated QuadTree—across traditional and Raptor-based approaches. Our experiments demonstrate that Raptor consistently improves performance on large-scale datasets, validating the results obtained in [1]. We also explore integration opportunities with distributed computing frameworks such as Apache Spark and Kafka to support scalable deployments.

2 Dataset

We decided to get the data from Natural Earth [9] and the U.S. Census Bureau [10], which provide public domain raster and vector map data., which provide public domain raster and vector map data. The specific raster file we chose represents elevation data (relief) for the contiguous U.S. As part of preprocessing, we created several lower and higher resolution versions of the original raster to study how the methods scale with different raster sizes. The vector files contain polygon geometries for U.S. states and counties. We removed non-overlapping features (e.g. Hawaii, Puerto Rico, Alaska) from the vector files and reprojected all files in the same projection. The files are described in Tables 1 and 2. One limitation of our dataset choice is that they are smaller than the truly massive datasets (petabytes) discussed in the motivation. Still, the performance differences we observe should scale similarly to larger datasets.

File (Code)	Description	Resolution (m)	Pixels (H×W)	Size
R1	US Relief	726 × 726	9493 × 12578	12 MB
RDn2	Relief downsampled ×2	1452 × 1452	4746 × 6289	6.5 MB
RDn4	Relief downsampled ×4	2904 × 2904	2373 × 3144	2.1 MB
RUp2	Relief upsampled ×2	363 × 363	18986 × 25156	61 MB
RUp4	Relief upsampled ×4	182 × 182	37972 × 50312	192 MB

Table 1: Summary of raster datasets.

File (Code)	Description	Features	Vertices	Complexity
VStates	US States	49	11,150	0.512
VCounties	US Counties	3,186	52,073	0.361

Table 2: Summary of vector datasets. The complexity is calculated using the inverse Polsy-Popper Compactness [11].

3 Background on Zonal Statistics Methods

Vector-based methods treat the raster as a set of points. The Naive Point-in-Polygon method converts each raster cell into a point and checks for containment within each polygon using standard geometric tests. To reduce unnecessary computations, each polygon's minimum bounding rectangle (MBR) is projected into raster coordinates to restrict the region of interest. QSplit (Quadratic-Split) optimizes this approach by recursively subdividing the polygon's extent into quadrants. Quadrants fully contained within or outside the polygon are classified directly. Only boundary-intersecting quadrants require further subdivision or containment testing, reducing the number of point-in-polygon operations.

Raster-based methods convert vector geometries into raster form and operate entirely in raster space. In the Clipping method, each polygon is used to create a clipped raster in which pixels outside the polygon are set to a NoData value. Aggregation functions are applied only to the remaining valid pixels. A new clipped raster is produced for each polygon individually. The Masking method improves efficiency by rasterizing all polygons into a single mask layer, where each pixel holds the ID of the polygon it falls within (or zero if none). This mask is overlaid on the original raster to pair pixel values with polygon IDs. Aggregation is then performed by grouping pixels by ID, allowing batch computation across multiple polygons.

Raptor-based methods operate directly on both raster and vector data without format conversion. The Scanline method begins by mapping the MBR of each polygon to raster coordinates to identify the rows (scanlines) that intersect the polygon. For each scanline, it computes intersections with the polygon's edges in vector space, sorts them by x-coordinate, and maps them back to raster space. Pixels between each pair of intersections are assumed to be inside the polygon and are aggregated. This approach avoids geometric conversion, minimizes intermediate storage, and performs a single pass through the raster, reducing disk I/O.

Aggregate Quad-Tree extends the Scanline approach by organizing the raster into a hierarchical quad-tree structure containing precomputed aggregate values. When processing a polygon, the method starts at the root node and recursively tests each node for complete containment. Fully contained nodes contribute their stored aggregates directly. For partially overlapping nodes, the underlying pixels are processed using the Scanline method. The final result combines precomputed values and pixel-level computations, reducing disk I/O and improving scalability.

4 Implementation details and limitations

For implementing the algorithms, we chose Python due to its extensive ecosystem of GIS libraries and because it allows easy integration of the resulting methods into larger systems, such as PySpark pipelines. Although Python can introduce overhead, most heavy computations are handled by C-based libraries like NumPy, mitigating performance losses to some extent. We use GeoPandas for vector data handling, Rasterio for reading raster layers and performing operations such as masking and windowing, and Shapely for geometric manipulations like computing intersections. Additionally, we use Rtree to accelerate the Aggregate Quad-Tree method with C-based spatial indexing. The implementations are summarized as follows:

- **RasterStatsMasking**: Serves as the reference method. It uses the RasterStats masking approach to process all polygons in bulk. This is the method we aimed to outperform with the raptor methods.
- **Masking**: Based on Rasterio's rasterization function, this method processes one polygon at a time, reading only the pixels within the polygon's bounding window. We adopted this approach because it showed improved performance over Rasterstats on larger rasters, likely due to smaller raster reads.
- **Clipping**: Uses Rasterio's masked reading function to clip raster data.
- **Naive Point-in-Polygon (NaivePP)**: Implements a custom NumPy-based matrix vectorization to build the point layer. This approach speeds up computation but increases memory usage.
- **QSplit**: Recursively splits each polygon into quadrants down to a minimum window size, processes each quadrant individually, and combines the results. At the lowest level, it applies NaivePP.
- **Scanline**: Constructs all horizontal lines corresponding to the center of pixels in bulk, computes their intersections with all polygons, and builds a table in "reading plan" that contains a representation similar to run-length encoding of the pixel segments to read and the polygon they belong to. Raster lines are then processed sequentially, retrieving the relevant pixel values and computing per-polygon statistics. As a result, each relevant pixel is read only once. This "reading table" approach may consume more memory than strictly necessary, but the use of vectorized operations resulted in a significant execution time improvement in Python.
- **AggQuadTree**: The Aggregate Quad-Tree method precomputes a Z-ordered quadrant structure (QuadTree) using the Masking method to calculate per-quadrant statistics, and stores the results in an R-tree for fast polygon intersection queries. When computing zonal statistics, it first runs Scanline to build the reading table, then updates it by removing segments corresponding to quadrants fully contained within the query polygons. Figure 1 illustrates this process. After this adjustment, processing continues as in the standard Scanline method. The first time the Aggregate Quad-Tree is used on a raster file, the method builds and saves the QuadTree index; subsequent runs load the index from disk. In the experiments, the tree construction phase was excluded from runtime measurements and treated as a precomputation step.

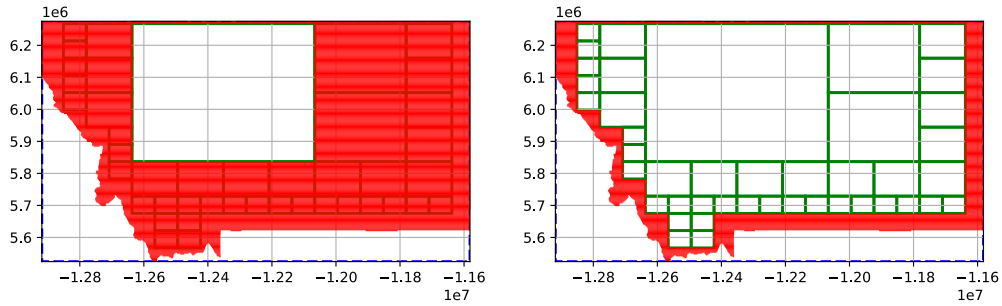

Figure 1: Scanlines in the "reading plan table" after the first (left) and last (right) fully contained QuadTree node has been processed.

Some limitations and caveats of the implementations include:

- QSplit was not as heavily optimized as other methods, since it is less theoretically promising. Our main comparison focuses on RasterStatsMasking vs. the raptor methods.
- The Clipping method, while functionally correct, underperforms. We chose not to optimize it further, as Masking already provides a strong baseline for raster-based performance.
- Raptor methods do not produce pixel-perfect identical results compared to the reference. However, the average difference is under 0.15% for the coarsest resolution raster, and even smaller at higher resolutions. These small discrepancies could be addressed with further testing and refinement.
- The implementations were tested specifically on the raster and vector datasets described. They do not account for variations such as different projections or multi-band rasters.

5 Evaluation

To evaluate the performance of the six implemented methods, we developed a benchmarking framework that:

- Executes each method on the dataset
- Verifies correctness by comparing results against a reference implementation, Rasterstats masking method.
- Measures the following performance metrics:
  - Total execution time
  - CPU time
  - Peak RAM usage
  - I/O read time
  - Total recorded I/O reads (in MB)
- Records the results in CSV and JSON files, and generates plots (depending on configuration).

Each experiment is repeated 10 times (configurable), clearing the system cache between runs to ensure consistent starting conditions.

6 Results

6.1 General performance

First, we compared all methods on the R1–VStates dataset. The results are shown in Fig.2 and Fig.3. We observe that the less optimized methods, QSplit and Clipping, do not achieve their

expected theoretical performance. More notably, even on this relatively small dataset, Scanline and AggQuadTree already show competitive performance.

In Fig.4 and Fig.5, Masking and the raptor methods are compared on the largest raster configuration, RUp4–VStates. We can see that the raptor methods scale better; this will be analyzed further in the following sections.

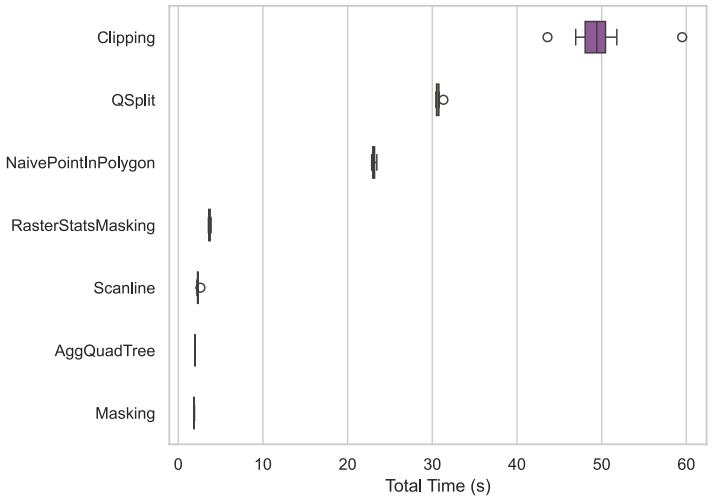

Figure 2: Comparison of all algorithms' execution times on the R1–VStates dataset.

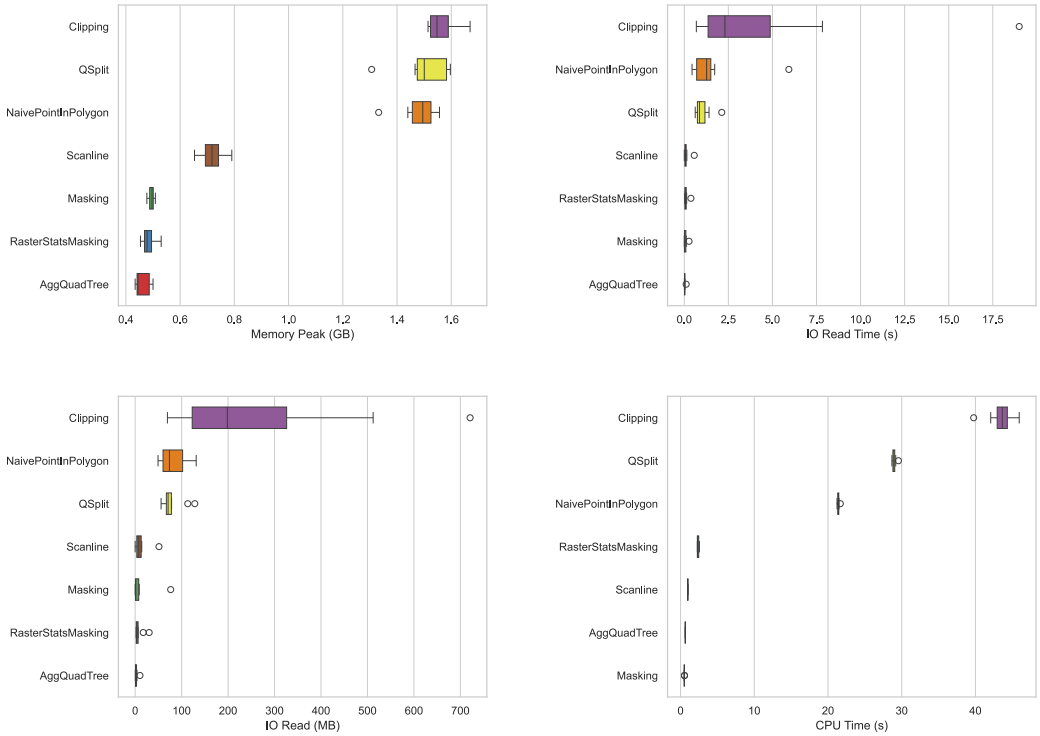

Figure 3: Comparison of all algorithm across all metrics with R1-VStates dataset.

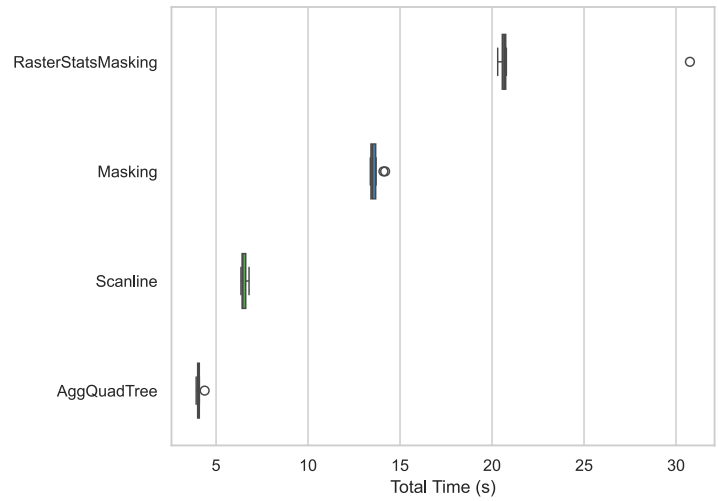

Figure 4: Comparison raster and masking algorithms execution times on the RUp4–VStates dataset.

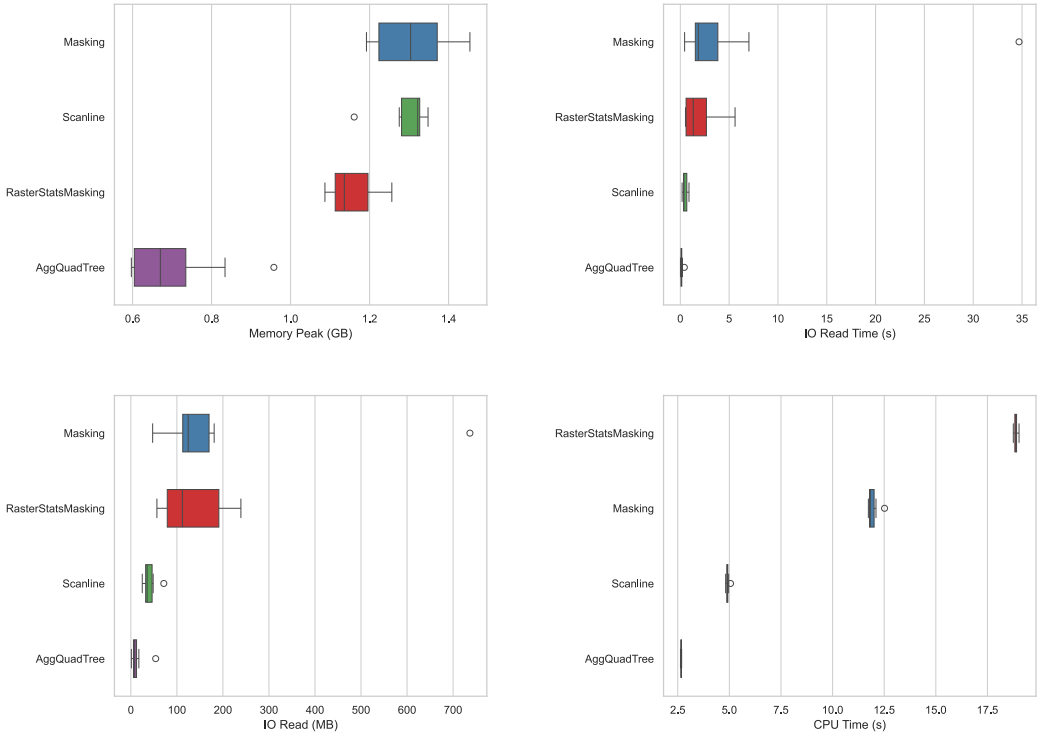

Figure 5: Comparison of masking and raptor algorithms across all metrics with RUp4-VStates dataset.

6.2 Scaling the raster size

In Fig. 6, all algorithms' execution times are compared across different raster sizes, from smaller to larger files. We observe that the RasterStats method performs best on smaller rasters. However, with R1, RUp2 and RUp4, it becomes advantageous to process one polygon at a time using the Masking method. Finally, Scanline and especially AggQuadTree demonstrate better scalability as raster size increases. Regarding memory usage (Fig. 7), the results are closer, with AggQuadTree maintaining a slight advantage.

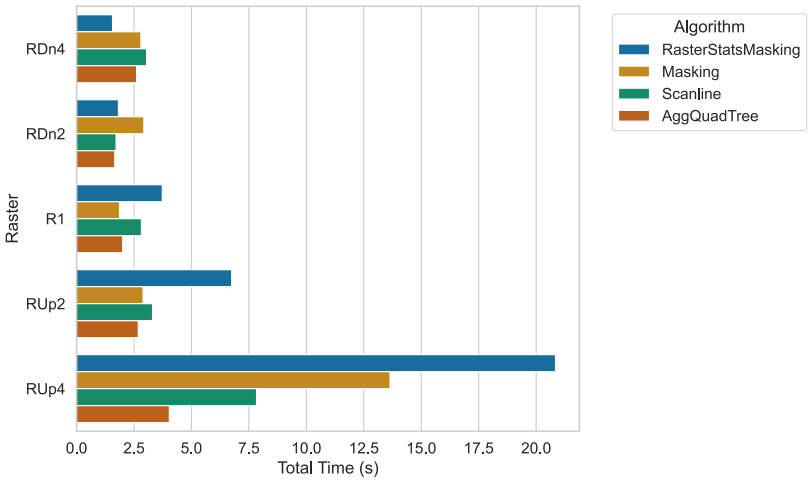

Figure 6: Mean algorithm execution time with increasing raster sizes.

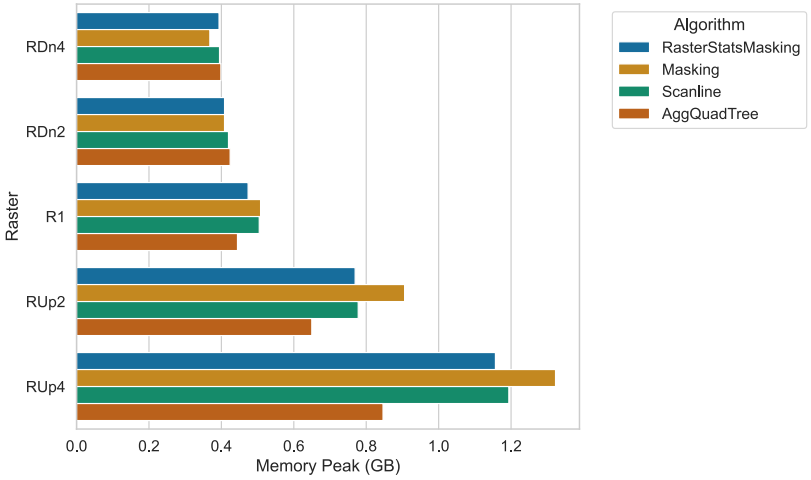

Figure 7: Mean algorithm memory usage with increasing raster sizes.

6.3 Scaling number of features

In Fig. 8, all algorithms are compared on the RUp4 raster using both VStates and VCounties. Once again, we observe that the raptor methods scale better. Notably, using a higher tree depth in the AggQuadTree method shows a small improvement when processing smaller features (VCounties).

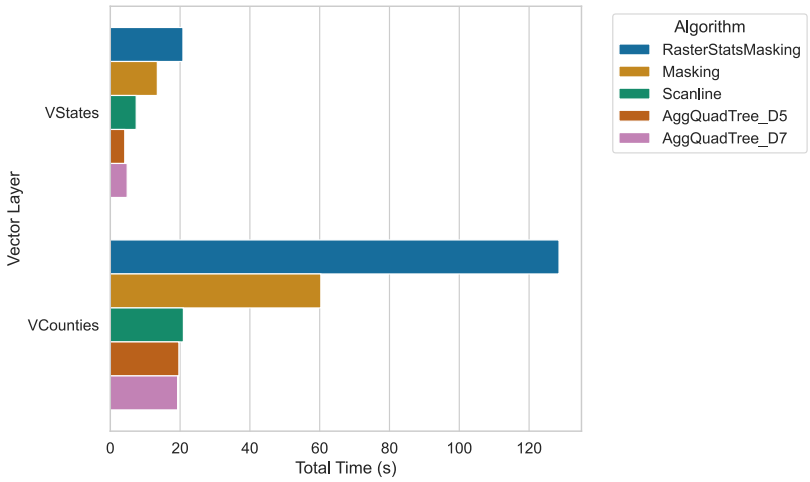

Figure 8: Mean algorithm execution time with increasing number of features. For the AggQuadTree method, two maximum tree depths were used: 5 and 7.

6.4 Effect of different QuadTree depths

We ran an experiment using different QuadTree depths on the RUp4–VStates dataset, with the results shown in Fig. 9 and 10. Increasing the depth leads to additional processing time due to the higher number of quad–polygon intersection checks. However, as shown in the memory usage figure, deeper trees might roughly correlate with reduced memory usage, although the results are dataset-dependent. Combined with the findings from the previous section, this suggests that an optimal QuadTree depth could potentially be derived from a metric based on polygon size and available memory.

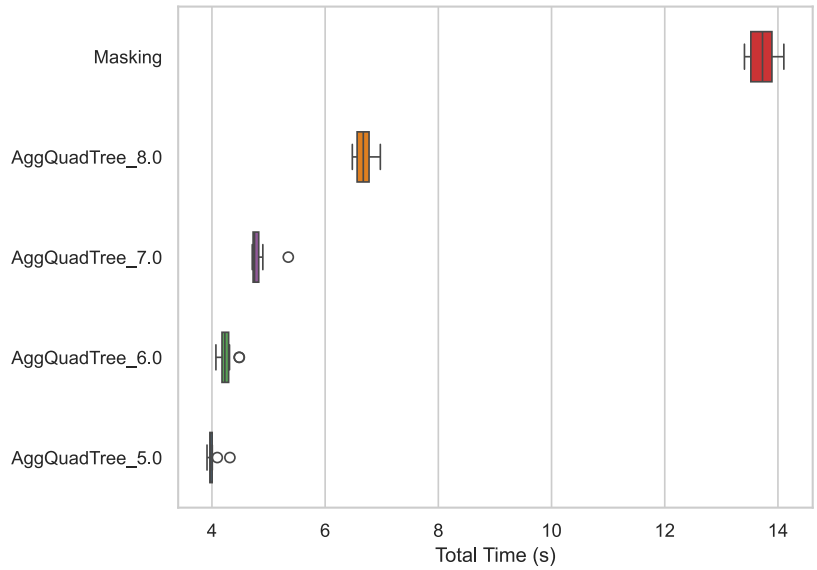

Figure 9: Execution time for the RUp4-Vstates dataset with different QuadTree depths.

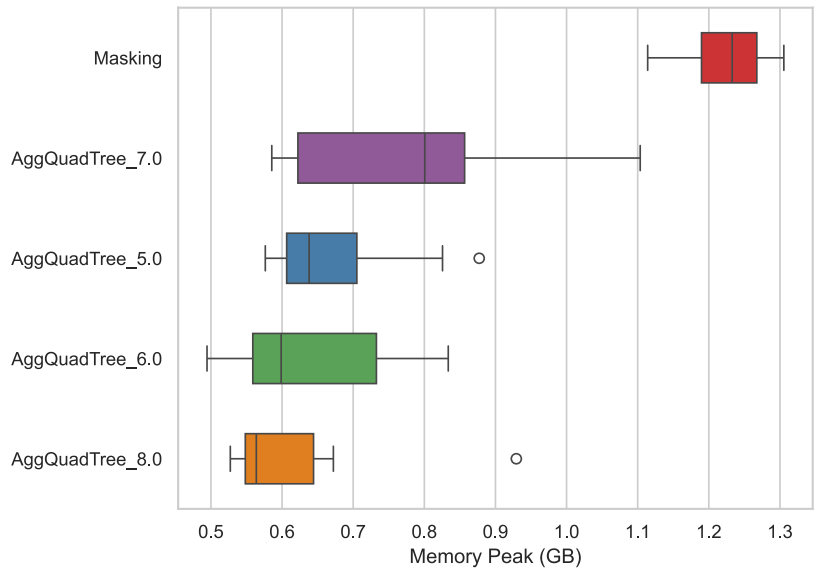

Figure 10: Memory usage for the RUp4-Vstates dataset with different QuadTree depths.

7 Kafka-Spark Integration

7.1 Kafka Integration

In this project, Kafka has been successfully implemented as a distributed messaging systems inorder to decouple the different stages of the analysis of geospatial data

7.1.1 Architecture

- Three distinct Kafka topics were established: zonal-stats,raw-polygons,raw-titles for processing, raster, vector data respectively
- Producer-consumer architecture allowed for independent processing rates for data ingestion and analysis
- Serialization/deserialization processes were put in place to manage complex geospatial data types

7.1.2 Components

- Tile Producer: Successfully ingested and serialized multiple TIFF raster files to Kafka
- Polygon Generator: Created synthetic vector geometries and published them to Kafka
- Consumer Framework: Configured to collect zonal statistics results

7.1.3 Realized benefits

- Data Stream Management: Successfully managed multiple geospatial data streams at the same time
- Data Type Handling: Successfully serialized and shipped complex spatial data types
- Decoupled Architecture: Decoupled data generation from consumption, leading to a modular setup

7.2 Future Integration with Spark

While we effectively utilized Kafka as a message broker in our geospatial pipeline, time did not permit complete integration with Apache Spark for distributed processing. A detailed integration plan would involve

- Stream Processing Architecture: Using Spark Structured Streaming to consume from Kafka topics in micro-batches and provide stateful geospatial data stream processing with windowed operations for supporting time-based analytics.
- Parallel Processing Pipeline: Configuring Spark to parallelize computing zonal statistics over a cluster, where the worker nodes compute subsets of intersections of polygons and rasters in order to significantly reduce computation time.
- Optimization Methods: Leverage spatial partitioning techniques in Spark to ensure data locality, minimizing data transfer across the network by assigning spatially proximal data to the same executors.
- Persistence Layer Integration: Integrating write-ahead logs and checkpointing for fault tolerance, with the processed output being persisted into a distributed file system or spatial database for downstream consumption.
- Performance Monitoring: Having measurements taken at strategic points in the pipeline to identify bottlenecks, with special emphasis on serialization overhead between Kafka and Spark when processing complex spatial data structures.

This merge would transform our existing implementation into a fully scalable, fault-tolerant system capable of handling petabyte-scale geospatial analytics and still maintain the benefits of the Raptor processing model.

8 Conclusion

Our implementation and evaluation of the Raptor processing model confirm its significant advantages for large-scale geospatial analysis involving both raster and vector data. Not only do our results validate the findings presented in [1], but they also shows Raptor's superiority across additional metrics not covered in the original paper, including memory usage, I/O time, read volume, and CPU time. Additional experiments highlight the scalability of Raptor methods with increasing data sizes, and ablation studies on the Aggregate Quad-Tree method show preliminary results on how performance might be influenced by the choice of maximum tree depth. Importantly, our implementation outperforms Rasterstats, a widely used Python library for zonal statistics, making our code a candidate for community contribution as a Python package.

9 Future Work

Looking ahead, future work should focus on several key directions. Integrating Raptor with distributed computing frameworks such as Apache Spark would allow it to scale to truly massive datasets. Real-time analytics capabilities could be enabled by incorporating streaming platforms like Kafka, either independently or in combination with Spark Streaming. Expanding the current implementation to support additional spatial operations beyond zonal statistics would further increase its applicability. Additionally, adaptive execution strategies that dynamically select optimal methods based on dataset characteristics could improve efficiency. Finally, exploring hardware acceleration—particularly for compute-intensive steps like polygon-scanline intersection—could yield significant speedups.

References

[1] Samriddhi Singla, Ahmed Eldawy, Rami Alghamdi, and Mohamed F. Mokbel. Raptor: large scale analysis of big raster and vector data. *Proc. VLDB Endow.*, 12(12):1950–1953, August 2019.

[2] M. J. Mineter. *Partitioning Raster Data*. CRC Press, 2020.

[3] *Geographic data processing—raster data*. Elsevier eBooks, 2023.

[4] R. Sun, M. Zhou, R. Chen, Q. Shi, and J. Xiong. Method and device for rendering vector data. 2019.

[5] D. Conzelmann. The geographic vector. pages 25–48. 2022.

[6] C. Xu, J. M. Chen, H. Wu, R. Li, and Y. J. Zhao. An improved pixel counting method for arbitrary zonal statistics on globeland30. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 4220:101–103, 2019.

[7] E. Giofandi, K. Munibah, K. Kraugusteeliana, A. Novalinda, and C. E. Sekarrini. The comparison of vector and raster data for the calculation of landscape environment using a geographic information system approach. *IT Journal Research and Development*, 7(2):209–219, 2023.

[8] V. Perupogu. Learning continuous mesh representation with spherical implicit surface. 2023.

[9] Natural earth - free vector and raster map data. `https://www.naturalearthdata.com/`. Accessed: April 2025.

[10] U.S. Census Bureau and U.S. Department of Housing and Urban Development. 2021 national public use file (puf). `https://www.census.gov/programs-surveys/ahs/data/2021/ahs-2021-public-use-file--puf-/ahs-2021-national-public-use-file--puf-.html`, 2021. Accessed: April , 2025.

[11] Daniel D. Polsby and Robert D. Popper. The third criterion: Compactness as a procedural safeguard against partisan gerrymandering. *Yale Law & Policy Review*, 9(2):301–353, 1991.