Towards Timely, Resource-Efficient Analyses Through Spatially-Aware Constructs within Spark

Daniel Rammer, Sangmi Lee Pallickara, Shrideep Pallickara
Department of Computer Science
Colorado State University
Fort Collins, CO USA
rammerd@rams.colostate.com, sangmi@cs.colostate.edu, shrideep@cs.colostate.edu

***Abstract*—Across several domains there has been a substantial growth in data volumes. A majority of the generated data are geotagged. This data includes a wealth of information that can inform insights, planning, and decision-making. The proliferation of open-source analytical engines has democratized access to tools and processing frameworks to analyze data. However, several of the analytical engines do not include streamlined support for spatial data wrangling and processing. Here, we present our language-agnostic methodology for effective analyses over voluminous spatiotemporal datasets using Spark. In particular, we introduce support for spatial data processing within the foundational constructs underpinning development of Spark programs DataFrames, Datasets, and RDDs. Our empirical benchmarks demonstrate the suitability of our methodology; in contrast to alternative distribution spatial analytics frameworks, we achieve over 2x speed-up for spatial range queries. Our methodology also makes effective utilization of resources by reducing disk I/O by a factor of 18, network I/O by 5 orders of magnitude, and peak memory utilization by 58% for the same set of analytic tasks.**

***Index Terms*—Spatial Analytics; Data Wrangling; Analytical Engines**

I. INTRODUCTION

Data volumes have grown exponentially over the last couple of decades. This growth can be attributed to several factors including falling hardware costs that have occurred alongside improvements in the quality and capacity of both networks and disks. The precisions, resolutions, and frequencies at which observations are being recorded have also increased. A majority of the data that are being generated is geospatial; with each observation also being geotagged to record the <lat, long>coordinates.

The growth in data volumes has also coincided with the development of several open-source libraries for processing the data. These include libraries for managing data encoding formats, data preprocessing including operations for cleaning, noise-removal, and imputations, model fitting libraries that implement statistical and machine learning algorithms, and frameworks for distributed orchestration of data processing.

Spark offers a rich ecosystem for data processing and analytic operations. The framework includes libraries for data pre-processing, model fitting, and a suite of statistical and machine learning algorithms. Spark incorporates support for different types of processing; in particular, Spark combines batch and stream processing operations in a single framework. Furthermore, Spark facilitates expressing computations in different languages such as Java, Python, and Scala.

The crux of this paper is effective support for spatiotemporal data in the Spark ecosystem. There are two key aspects to this:

1) Providing a sufficiently rich set of capabilities for working with geospatial data. These capabilities must benefit a broad class of analyses and applications.
2) Ensuring the efficiency of these operations. In particular, the operations must scale, preserve latency requirements, and ensure high throughput.

A. Challenges

There are several challenges in incorporating effective support for spatial analytics within Spark.

- Data Storage: If data storage and dispersion does not account for spatiotemporal characteristics it would result in excessive data movements. However, once staged, data movements are prohibitively expensive since they entail both disk I/O and network I/O.
- Data movements: Several data wrangling operations including polygon-based functionality can trigger data movements. A side effect of such data movement is I/O amplification. Since analytics is being performed in shared clusters, such I/O amplifications often induce interference for other operationally unrelated applications.
- Spatial data wrangling: Support for effective selection, filtering, etc. must account for spatial data characteristics. Naive operator implementations can result in every observation within the dataset being inspected.
- Voluminous data: Data volumes exacerbate the aforementioned challenges. Without support for effective indexing and filtering of data based on their spatiotemporal characteristics, the effects of data movement are considerably amplified by data volumes.

B. Research Questions

The overarching research question that guides our investigation is the following: *How can we effectively incorporate support for spatiotemporal data analysis in Spark?* Specific research questions within this broader context that we have formulated include:

RQ-1: *How can we align data retrieval with spatial access patterns?* This includes disk accesses and minimizing data movements during spatial data processing.
RQ-2: *How can we support effective data wrangling over spatiotemporal datasets?* These must account for the spatial characteristics and geometry that are inherent in such operations.
RQ-3: *How can we ensure performance characteristics of these operations?* Operations that are expressive, but inefficient lead to prolonged execution times and induce interference that adversely impacts other collocated applications.

C. Approach Summary

Spark supports two broad classes of operators: transformations and actions. Transformations are the primary mechanism to perform data wrangling operations prior to launching an action. A key feature in Spark is that transformations are performed in a distributed fashion as well. Depending on the operation being performed and the data dispersion over physical machines these transformations may result in excessive data movements (network I/O) and are referred to as wide transformations. Spark leverages lazy evaluations; transformations are not evaluated till such time that an action is initiated.

The Spark DataFrame forms the basis for the Structured API within the Spark ecosystem, and represents data in its most foundational form as a table comprising rows and columns. A typical DataFrame in Spark spans multiple machines. Code written for the DataFrame is converted by Spark into a logical plan. The conversion of the logical plan of execution into a physical plan involves exploring optimizations that ensure locality, minimizing data movements, and leveraging statement reorganization to ensure faster execution times. The Catalyst Optimizer within Spark is a key component that is responsible for generation of the physical plan. A key component of this is predicate pushdowns, which ensures that filtering operations are performed prior to any data movements and computing. Our methodology targets every aspect of this physical manifestation.

Our methodology includes support for: (1) reducing data movements at the storage level, (2) seamless interoperation with the Spark ecosystem, (3) support for a rich set of data wrangling operations, including transformation operators, that are aligned with spatiotemporal data processing requirements, and (4) incorporating optimizations into the Spark runtime, especially the Catalyst Optimizer, so that functionality expressed by users over Spark DataFrames (Datasets, or RDDs) are performant with fast completion times and reduced interference for collocated applications.

We leverage our file system, Atlas, for effective integration of spatial data with Spark. Atlas reduces data movements via effective collocation of proximate spatiotemporal data. Atlas leverages the geohash algorithm to convert geotagged (latitude, longitude) coordinates associated with individual observations into 2D bounding boxes. The Atlas file system is distributed and breaks up large files into contiguous chunks that are dispersed over multiple machines.

To plug-in to the Spark ecosystem, we leverage its coupling with HDFS. Spark has mature capabilities that allow it to use HDFS as the source (input) or destination (for results) of data processing operations. We have designed Atlas to be HDFS compliant. This involved making sure that Atlas is able to support all control plane traffic relating to discovery, replication, check summing, etc. that may be initiated, but implements them in ways that are optimized for spatiotemporal data.

We support a rich set of data wrangling operations for Spark that are aligned with the needs of spatial data processing applications. In particular, these include support for inspection operations involving spatial geometry such as `Contains`, `Covers`, `Crosses`, `Disjoint`, `Equals`, `EqualsTollerance`, `Intersects`, `IsEmpty`, `IsSimple`, `IsValid`, `Overlaps`, `Touches`, and `Within`. We also support calculation over complex polygon defined geometry such as distances, lengths, and area. Our transformation operators include set and join operations that result in a new DataFrame comprising the initial, pre-computed, or resulting geometry. Transformations that we support include: `Buffer`, `ConvexHull`, `Difference`, `Envelope`, `Intersection`, `Normalize`, `SymDifference`, `Union`, `Simplify`, and `Densify`. Once subsets of the data have been identified using our data wrangling operations, they are amenable to further filtering and sifting operations using SQL queries as available within Spark SQL.

We have incorporated support for spatiotemporal data wrangling within Spark's physical plan manifestation system. This includes support for filtering and predicate pushdown for spatiotemporal operations. Consider the following exemplifying scenario; we are working with a dataset with 100,000 observations, 10,000 of which are located in Fort Collins, CO. If we are interested in analysis of the Fort Collins observations, canonical HDFS will read the entire dataset into a DataFrame and evaluate each row individually, filtering out observations which are not in Fort Collins. Alternatively, since Atlas supports spatial queries over the underlying files, we can issue a query to retrieve only the requested data. Achieving the same 10,000 observation DataFrame of Fort Collins data, but bypassing the unnecessary I/O and filtering computations over the unwanted 90,000 observations.

D. Paper Contributions

This study addresses effective processing of voluminous, geotagged data using Spark. Both legacy and new Spark applications benefit from this study. In particular, our contributions include:

1) Support for feature-rich and performant spatiotemporal constructs *(DataFrames, Datasets, and RDDs)* within Spark.
2) A rich suite of spatial data wrangling operators that are performant. To ensure effective spatiotemporal data processing, we leverage space-efficient spatial indexes, controlled data

dispersion that preserves data collocation, leverage predicate pushdowns to reduce redundant operations, and reduce data movements.
3) Users can continue to use their preferred Spark libraries for data processing alongside Atlas.
4) Our extensions, besides being language-agnostic, could be used either in batch or stream processing modes.

II. Systems Overview

A. Geohash

The geohash algorithm [1] converts geotagged <*latitude, longitude* >coordinates into a one-dimensional string that represents a unique spatial bounding box. The length of the string controls the extent of the spatial bounding box. The algorithm computes a bit sequence in iterations of 4 bits (each 4 bit iteration corresponding to an output character) where alternating bits index the X and Y coordinates respectively. The bit sequence is computed by iteratively splitting the current minimum and maximum bounds for each value and appending a 0 or 1 to the bit string if the value resides in the lower or upper half respectively. For example, in the Cartesian coordinate system we begin with X minimum and maximum bounds [-180, 180]. For a value of -50 we append a 0 as it resides in the lower half, namely - 180 to 0. The next iteration will be computed under minimum and maximum bounds of -180 to 0. The algorithm performs iterations until the desired precision (i.e., the number of output characters) is reached.

The Atlas File System [2] is a distribution, spatiotemporal file system. A depiction of the framework's architecture is provided in Figure 1. Atlas provides efficient data retrieval for spatiotemporal access patterns. This is achieved by leveraging a combination of techniques. (1) Sequential disk access for spatially and temporally contiguous data. Excessive disk head movements degrades system performance. The Atlas File System dynamically rearranges data on disk to provide contiguous and more efficient read / write operations. (2) Distributed spatiotemporal indices enable targeted data retrieval. Given that Atlas data is stored spatially and temporally sequential, indices may be compiled at a low granularity. Meaning they index blocks of data instead of individual observations. This vastly reduces the overhead incurred in maintaining these commonly expensive indices. (3) Data is replicated between nodes based on their spatiotemporal properties. The goal is for each spatiotemporal extent to maintain replicas where one host contains one instance of each replica and other replicas are distributed among other cluster hosts. Therefore, operations may be scheduled with data locality with either collocated or distributed properties. To support dispersion, spatial indices, and queries Atlas leverages the geohash algorithm.

The framework provides an HDFS compliant interface, enabling seamless integration into existing workflows. Architecturally, as seen in Figure 1, the system is setup using namenodes and datanodes to ensure the separation of control and data planes. Namenodes maintain filesystem metadata including the directory hierarchy, file ownership, etc. Alternatively datanodes operate entirely on data blocks,

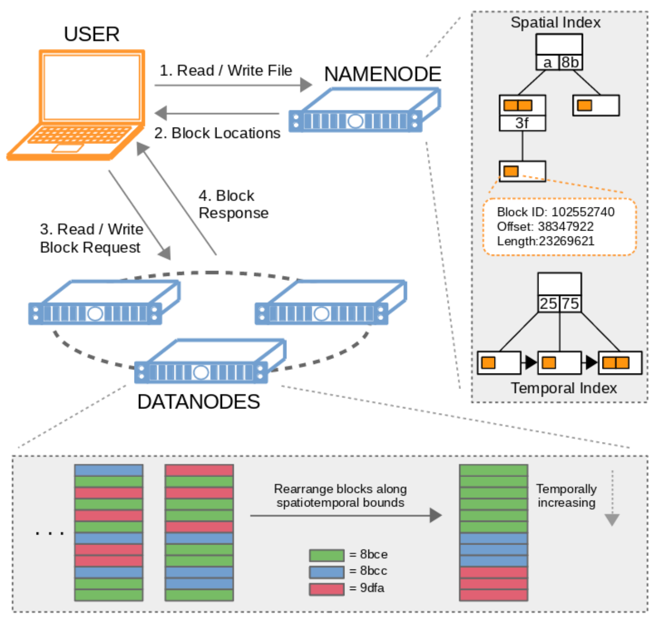


Fig. 1: Depiction of the Atlas File System architecture. Most importantly, the framework rearranges observations in the dataspace to be spatially and temporally contiguous, facilitating concurrent retrieval. Additionally, the system maintains spatial and temporal indices over the dataspace to enable efficient dataspace identification.

or 128MB (configurable) sequences of file data. The Atlas File System extends HDFS URL's to facilitate spatiotemporal queries using HTTP-like parameters. For example, the query for "hdfs://noaa/data0.csv+g=8bc&t>1564593384" returns data for the specified file where the geohash is equal to 8bc and timestamp is greater than 1564593384. In addition to individual file queries, the system supports queries over directories, returning blocks corresponding to all children files which satisfy the specified query. Finally, the Atlas File System supports a variety of data formats. It leverages HDFS' storage policy construct to inform the system of data formats, in particular spatial and temporal features within the dataspace.

B. Atlas File System

The Atlas File System datanodes perform indexing by rearranging block observations into a striped set, the algorithm is outlined at the bottom of Figure 1. This process begins by computing the geohashes for each observation, a process that handles points, lines, and polygons. Block data is then rearranged, on a per observation basis, so that data for each geohash is consecutively stored in the block, with observations for the same geohash temporally contiguous. These operations are performed in-memory to reduce I/O and computational costs. Data is then written to disk and replicated to secondary nodes. Block attributes, including offsets and lengths of each geohashes data segments and block temporal ranges are then reported to the namenode for indexing.

To facilitate efficient queries the namenode maintains spatiotemporal indices. The spatial index is constructed using a radix tree over geohashes where each node may contain information on block IDs, data offsets, and their lengths. This example contains data with geohashes a, a3f, 8bce, and 8b4.

The temporal index is maintained using a B+-Tree, which is designed for efficient range queries, with start and end timestamps for each data block. Using the radix tree based spatial index and B+-Tree based temporal index namenodes can efficiently evaluate myriad spatiotemporal queries. An example of the spatial and temporal index structures is depicted on the right of Figure 1.

III. Methodology

Our AtlasSpark framework is designed as a library built on top of the Apache Spark framework. We provide a suite of spatial extensions including spatial data types, functions, and an Atlas based DataFrame. AtlasSpark's location in the analytics stack is provided in Figure 2. By implementing functionality as a library, AtlasSpark may be seamlessly integrated into existing tools and frameworks including spark-shell, PySpark, and many others.

Our methodology for providing efficient spatiotemporal functionality encompasses the following.

- *Aligning data storage and retrieval with spatiotemporal access patterns:* Data reporting and storage in traditional storage systems tends to be misaligned with spatiotemporal data characteristics and access patterns. During data retrieval, non-sequential access patterns adversely impact performance. By integrating the Atlas File System with Spark we may leverage dynamically reorganized observations to better align with spatiotemporal access patterns, providing more efficient data retrieval. **[RQ-1]**
- *Preserving data locality during processing:* Efficient distributed analytics relies on reducing data movements during processing by collocating data based on their proximity in the spatiotemporal space. This allows operations to be performed while preserving locality; within the Spark context these are referred to as narrow instead of wide transformations. We focus on providing spatiotemporal collocality during Spark data processing by leveraging the spatiotemporally-aligned storage provided within Atlas. **[RQ-1]**
- *Support for spatial data wrangling operations:* Lacking functionality designed specifically for spatial operations, Spark suffers from inconsistent performance in myriad spatiotemporal analytics. We have integrated a collection of spatial operations including point, line, and polygon construction and functions to test relationships within or between these spatial objects. **[RQ-2, RQ-3]**
- *Building optimizations into Spark:* Ensuring performance during many analytical operations requires efficient identification of data subspaces. We have designed extensions to the Spark Catalyst execution optimizer to ensure performant data filtering operations. **[RQ-2]**

A Spark Scala example of the aforementioned spatiotemporal functionality is provided in Listing 1. When executed; data is read from the Atlas File System, filtered based on the specified polygonal bounds, and the resulting rows are counted. Various components of the code will be referenced in this Section as necessary to clarify and elaborate explanations.

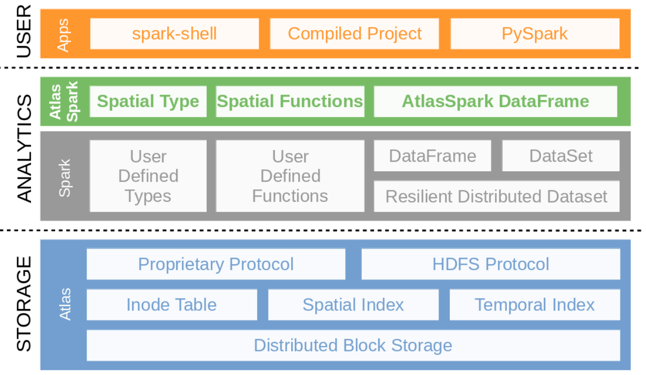


Fig. 2: AtlasSpark's integration into the analytical hierarchy using Atlas File System as a storage framework and Spark as the base analytics suite. Built on top of Spark, AtlasSpark integrates into existing workflows; intrfacing with spark-shell, PySpark, and myriad other tools.

A. Aligning Data Storage and Retrieval with Spatiotemporal Access Patterns **[RQ-1]**

Spatiotemporal analytic challenges are exacerbated by dataset reporting and storage, where observations reporting is seldom aligned spatially or temporally. Instead, collection points tend to intertwine observation streams from multiple locations within the same data file. Therefore, identification and retrieval of data from a particular spatial or temporal extent is difficult. This issue is compounded by the exponential increase in spatiotemporal dataset sizes, datasets may approach petascales.

Distributed spatial analytic frameworks rely on a number of techniques to retrieve spatiotemporal extents. (1) Iterating over the dataspace observationally and dynamically filtering the dataspace. Such schemes result in poor evaluation performance with unacceptable times where indexed, in-memory data retrieval speeds are the norm. Additionally, queries which focus on a small subset of the dataspace require an unnecessary amount of computational overhead including CPU, disk and network I/O. (2) Maintenance of expensive distributed indices.

Listing 1: A Scala example outlining our spatiotemporal Spark extensions. The listing depicts registration of Atlas components, initialization of a Spark DataFrame over Atlas, filtering within polygonal bounds, and evaluation of the DataFrames observation count.

```scala
import org.apache.spark.sql.atlas.AtlasRegister

// register atlas spark components
AtlasRegister.init(spark)

// read dataframe from atlas csv file
val df = spark.read.format("atlas")
    .load("hdfs://129.82.208.10/data0/noaa")

// evaluate spark sql statement
df.createOrReplaceTempView("noaa")
var spatialDf = spark.sql("""
    SELECT BuildPoint(_c0, _c1) as point, *
    FROM noaa
    WHERE Within(point,
        BuildPolygon((0.0, 0.0), (0.0, 10.0),
            (10.0, 10.0), (10.0, 0.0)))""")

spatialDf.count
```

TABLE I: Our spatial extensions to Spark have been implemented as a collection of UserDefinedFunctions. Functionality encompasses a variety of data return types and includes both binary and unary operations.

Data Type	Operand Count	Function Name	Description
Boolean	Unary	IsEmpty IsSimple IsValid	Check if the geometry contains data. Check if the geometry is simple. Check if the geometry is topologically valid.
	Binary	Contains, Covers, Crosses, Disjoint, Equals, EqualTollerance, Intersects, Overlaps, Touches, Within	Tests whether the geometry satisfies the relationship with the argument geometry.
Numeric	Unary	Dimension NumPoints Area Length	Returns the dimension of the geometry. Returns the number of data points in the geometry. Returns the area of the geometry. Returns the length of the geometry.
	Binary	Distance	Returns the distance between the two closest points in the provided geometries.
Geometry	Unary	Buffer ConvexHull Envelope Normalize	Computes a buffer area around the geometry with the specified width. Computes the smallest convex polygon which contains all points in the geometry. Creates a geometry representing the bounding box of the geometry. Creates a geometry with the normalized form of this geometry.
	Binary	Difference, Intersections, SymDifference, Union	Creates a geometry with the point-set of data as the geometrys relationship.

This technique is an anti-pattern for iterative, ad-hoc analytics given the unavoidable startup cost in iterating over existing data to compile indices. Additionally, datasets too large to fit in cluster memory cannot be processed using this technique.

The AtlasSpark framework leverages a combination of attributes to provide efficient spatiotemporal dataset accesses. Atlas File System rearranges block observations before writing to disk so that data is spatially and temporally contiguous. Therefore, analytics over data may be performed using sequential disk access, mitigating expensive overhead of excessive disk-head movements. Additionally, by consulting Atlas File System spatiotemporal indices AtlasSpark may quickly identify dataspace subset locations.

Integrating the Atlas File System framework with Spark requires careful consideration of two key aspects. (1) Spatiotemporal extents are often distributed between multiple blocks making custom data reading / parsing tools necessary. (2) Traditionally Spark / HDFS data transfer functionality relies heavily on Google's Protobuf framework. However, serialization and deserialization of messages may result in unavoidable overhead, especially in metadata heavy workloads such as spatiotemporal queries. Therefore, we have developed an efficient data transfer protocol deliberately avoiding message serialization / deserialization and specialized for the spatiotemporal dataspace.

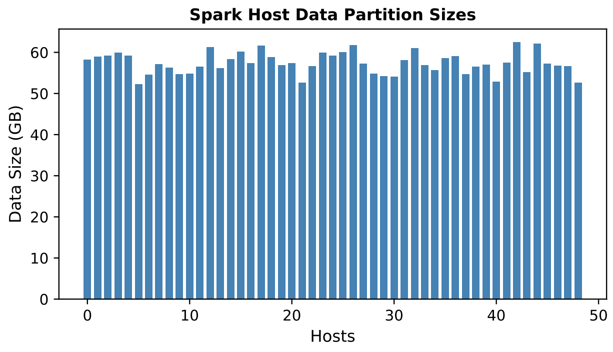


Fig. 3: Data partitions for Atlas analytics using Spark over 1TB of NOAA data preserve spatiotemporal proximity, but storage loads are not skewed. This results in load-balanced analytics.

B. *Preserving Data Locality During Processing* ***[RQ-1]***

The Spark framework relies on data partitioning algorithms to effectively distribute and orchestrate analytics workloads. These algorithms split source datasets into many smaller chunks that are then distributed. Data partitioning and distribution allows Spark to process the data in parallel. Data partitioning does not have a one-size-fits-all solution, because each scheme has its own inherent advantages and disadvantages for various analytics tasks – no scheme performs well in all scenarios.

Spark's default partitioning scheme for HDFS data is not suited for spatiotemporal data. By default, Spark partitions HDFS data along block boundaries. However, this scheme is inefficient for spatiotemporal access patterns that are triggered during data wrangling and analytics. The default scheme would entail large amounts of data transfers between nodes for many operations (i.e., wide transformations).

We have extended Spark's DataSourceV2 framework to partition datasets aligned with the spatiotemporal storage policies provided by Atlas. The DataSourceV2 implementation presents many abstraction improvements, including support for reads / writes with data streams and executing data filtering operations at the data source. Atlas storages policies are designed to address the competing pulls of data locality and dispersion. In Atlas, data within a particular spatiotemporal space is dispersed over a small subset of machines, facilitating parallel processing while minimizing data movements. By aligning our partitions with Atlas we ensure (1) there is little data movement during analytics, because processing is scheduled on data resident nodes and (2) many analytic operations may be performed with data locality; favoring narrow, rather than wide, Spark transformations during evaluation.

Micro-Benchmark: In this experiment we deployed Atlas and Spark on a 50 node cluster, where one machine houses the Atlas namenode and Spark master and the other 49 are Atlas datanodes and Spark workers. We dispersed over 1TB

TABLE II: Comparison of binary spatial operations over every combination of supported spatial object. Unsupported operations are represented with a -. Each operation was performed 1 million times over real data. Results show general variance between operation return types and spatial objects.

Operation	Line(Line)	Line(Point)	Line(Poly)	Point(Line)	Point(Point)	Point(Poly)	Poly(Line)	Poly(Point)	Poly(Poly)
Contains	1891.6	1548.8	-	-	1348.8	-	1835.4	1688.2	1876.4
Convers	1680.4	1525.0	-	-	1388.2	-	1886.8	1668.4	1935.0
Crosses	-	-	1727.0	-	-	-	1838.0	-	-
Disjoint	2300.2	1497.4	1809.0	1607.2	1412.2	1751.4	1938.4	1744.6	2003.2
Equals	1589.4	1460.2	1635.4	1508.0	1365.0	1571.2	1764.4	1598.8	1818.8
EqualsTollerance	1585.4	1465.2	1687.2	1631.8	1395.8	1661.6	1894.8	1691.4	1945.2
Intersects	3165.0	1530.0	1797.2	1640.4	1428.0	1697.2	1924.8	1728.0	2028.0
Overlaps	3090.0	-	-	-	-	-	-	-	2318.4
Touches	3664.4	1546.6	1793.6	1622.0	1426.6	1712.6	2044.4	1690.4	2042.4
Within	1848.2	-	-	-	-	1736.2	-	-	2019.0
Distance	2069.8	1728.4	2451.4	1825.8	1523.2	2016.2	2748.0	2015.4	2950.2
Difference	10857.0	6651.2	10539.6	7618.0	2759.0	6542.8	12451.8	7027.8	11717.6
Intersection	11647.4	6304.6	10336.6	5848.6	2793.8	6621.0	9098.4	6559.2	10809.8
SymDifference	12228.4	6765.2	11808.6	6218.0	3002.2	7322.6	10805.6	7336.8	11763.0
Union	13795.0	7317.8	12123.0	9347.2	3100.2	7444.4	13011.4	7259.8	11151.6

of data from our NOAA dataset over this cluster to profile its distribution effectiveness. Figure 3 plots the combined size of data partitions scheduled at each of the Spark worker hosts during analytics over the entire dataset. We see that even though Atlas preserves spatiotemporal proximity during data distribution, the data partitions are not skewed, but are rather evenly distributed over the cluster. This results in load-balanced processing, reducing hot-spots during analytics.

C. Support for Spatial Data Wrangling Operations ***[RQ-2, RQ-3]***

Apache Spark provides robust analytics operations, but is lacking in functionality relating to spatial data processing. This is becoming increasingly important as spatial data collection increases. Furthermore, applying non-performant operations results in significantly degraded analytics performance.

To support effective spatial data wrangling operations over voluminous datasets within Spark, we include three key features. First, we support a variety of spatial objects. We allow datasets to contain points, lines, polygons, or collections of objects. Second, we include extensive for support computing relationships within or between spatial objects. For example, computing the distance between a point and a polygon, or testing whether a line crosses a polygon. Third, we support datasets encoded in diverse formats. Spatial wrangling operations that we support and designed to be performant and operate over distributed, voluminous datasets. We have extended the JTS library [3] to provide the basis for spatial functionality introduced in this work.

Our AtlasGeometryUDT is an extension of Spark's UserDefinedType abstraction. Instances of the AtlasGeometryUDT are initialized by parsing Spark field(s). We currently support four different geometry initialization functions, namely BuildPoint, BuildLine, BuildPolygon, and ParseWkt. We have also incorporated support for extensions by simplifying steps needed to process additional data sources.

In Table I we outline the diverse collection of spatial operations. We have implemented these as extensions of Spark's UserDefinedFunction construct, enabling use during data transformations, filtering, and more. Our suite of data wrangling operations contains both binary and unary operators with numerous return types including booleans, numeric values, and diverse spatial objects.

Micro-Benchmark: We have benchmarked our spatial functionality by executing each operation 1 million times over various subsets of our NOAA and EPA datasets. Each experiment has been performed in isolation; detached from the Apache Spark application, but maintaining all Apache Spark field serialization / deserialization operations. The results in a measurement over the operation, excluding as much application overhead as possible.

All reported values are in milliseconds. Tables II and III provide information on binary and unary operators respectively. Each column represents operations performed on a specific spatial object type(s). The binary information provided in Table II is labeled as "operand1(operand2)" where the operation is performed in that order. For example, performing the operation "Within" operation under "Point(Poly)" means that a point is within a polygon. The columns for unary operations in Table III are labeled as a single spatial object. In both tables, empty cells denote operations which do not apply to the corresponding spatial object(s).

We notice variance in operation execution durations. Generally, durations increase as the operation return type moves from boolean, to numeric, to geometric. Additionally, opera-

TABLE III: Comparison of unary spatial operations over each spatial object. Unsupported operations are represented with a -. Each operation 1 million times using real data. Results show performance differences between spatial objects and operation return types.

Operation	Line	Point	Polygon
IsEmpty	847.8	832.8	1123.6
IsSimple	1823.6	803.4	2613.6
IsValid	1232.0	807.2	5125.8
BuildLine	1998.6	-	-
BuildPoint	-	980.8	-
BuildPolygon	-	-	2761.4
Area	-	-	1427.0
Dimension	916.8	851.8	1391.8
Length	917.0	-	-
NumPoints	924.2	880.2	1569.2
Buffer	48318.0	9534.0	10856.0
ConvexHull	1584.4	974.4	1885.0
Envelope	1010.8	872.0	1367.8
Normalize	946.4	863.2	1448.8

tions performed on polygons tend to be more expensive than lines, with a similar relationship between lines and points. Intuitively, these relationships are reasonable. As testing equality between polygons is comparatively simpler than computing the distance between two polygons, or computing an intersection.

D. Building Optimizations into Spark **[RQ-2]**

To reduce duplicate operations, during DataFrame/RDD source reads and subsequent data manipulations Spark performs lazy evaluations. To support lazy evaluations, Spark creates and consults an abstract syntax tree (AST) of operations. These operations may include field projections, dataset filtering, and data reads from a variety of sources. Spark dynamically performs a sequence of AST optimizations before evaluation to facilitate more efficient analytics. Catalyst is the AST optimizer distributed with Spark. Catalyst maintains a registry of predefined rules, which may be applied iteratively when evaluating the AST. Catalyst rules include boolean simplification, constant folding, column pruning, operation propagation over data joins and field aggregations, and many more.

A key Catalyst optimization that we target is predicate pushdowns. The predicate pushdown in Spark attempts to push dataset filtering operations to data sources by propagating filters down to data sources in the AST. The pushdown feature provides two broad advantages. First, data source implementations are often optimized to perform filtering by employing various data indexing structures and techniques. In some cases, this entails Spark iterating over each row and evaluating the filter individually. Second, it reduces data movements and network I/O. By performing filtering earlier in the AST evaluation, unnecessary data (i.e., data that does not satisfy the filter predicated) movement between stages is reduced.

We addressed a number of challenges to incorporate support for spatiotemporal predicate pushdowns within the Catalyst Optimizer. Currently, Spark lacks support for "pushdown" of UserDefinedFunctions. Furthermore, the diversity of UserDefinedFunctions introduces additional complexities when converting to filters that are aligned with Atlas.

To enable spatiotemporal query definitions using our UserDefinedFunctions we register a series of Catalyst optimization rules. These rules inject filters into the AST allowing them to be propagated using Catalyst Optimizer's predicate pushdown rules. For example, in Listing 1 lines 11-16 we use a Spark SQL query for filtering the NOAA dataset on data "Within" the defined polygon, with vertices identified by <latitude, longitude >pairs of <0,0 >, <0,10 >, <10, 0 >and <10, 10 >. We created and registered a Catalyst optimization rule to identify the "Within" functions where the Atlas spatial index field(s) are bounded by another geometry, in this case the aforementioned coordinates. We then compute the smallest bounding geohash for these coordinates, for example dac, and inject a filter into the AST for data within that bound, namely "atlasGeohash = 'dac'". Temporal filtering and bracketing is evaluated similarity. This new equality filter satisfies the requirements for Catalyst's predicate pushdown rules, and can therefore be propagated to the Atlas data source and evaluated accordingly. A list containing the bounding polygon defining the geohash for each of our supported spatial operations is provided below.

1) Contains: The bounding polygon by which "Contains" is processed.
2) Covers: The polygon which "Covers" other spatial objects.
3) Distance LessThan / LessThanOrEqual: A buffer computed with the provided distance from the source spatial object.
4) Equals / EqualsTollerance: An outer bounding polygon for which equality is tested.
5) Within: The polygon which other spatial objects are tested "Within".

Additionally, we have implemented two classes of filter combination rules over our spatiotemporal bounds. The first is the aggregation of spatial or temporal bounds. For example, consider a data source that requires filtering for "atlasTimestamp >1564701343" and "atlasTimestamp >1064701343". Instead of executing two separate queries, we aggregate the filter to the less restrictive filter, namely "atlasTimestamp >1064701343". The second is pruning AST branches when filter aggregations will result in no data. For example, combin-

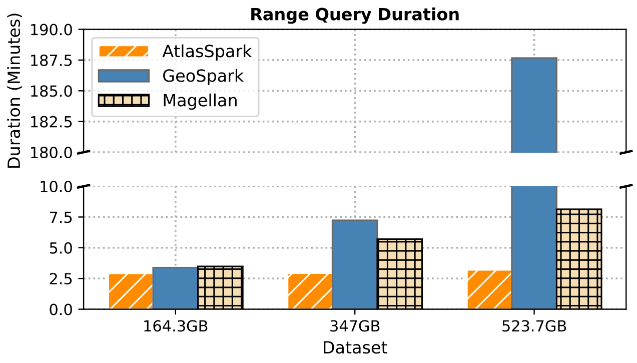


Fig. 4: Duration of spatially bounded range queries over 3 subsets of the NOAA dataset. AtlasSpark consistently out-performs, providing 1.2x, 2.6x, and 62.1x and 1.2x, 2x, and 2.7x reductions in duration when compared to GeoSpark and Magellan respectively for each of the datasets.

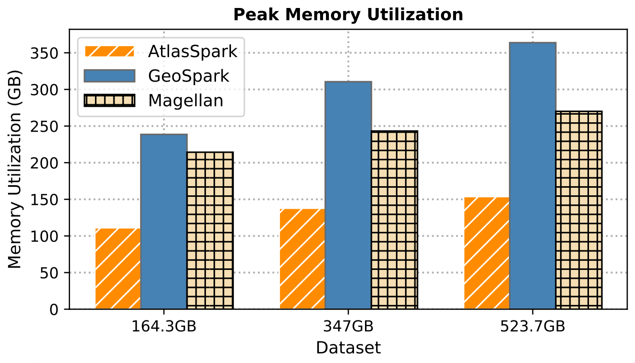
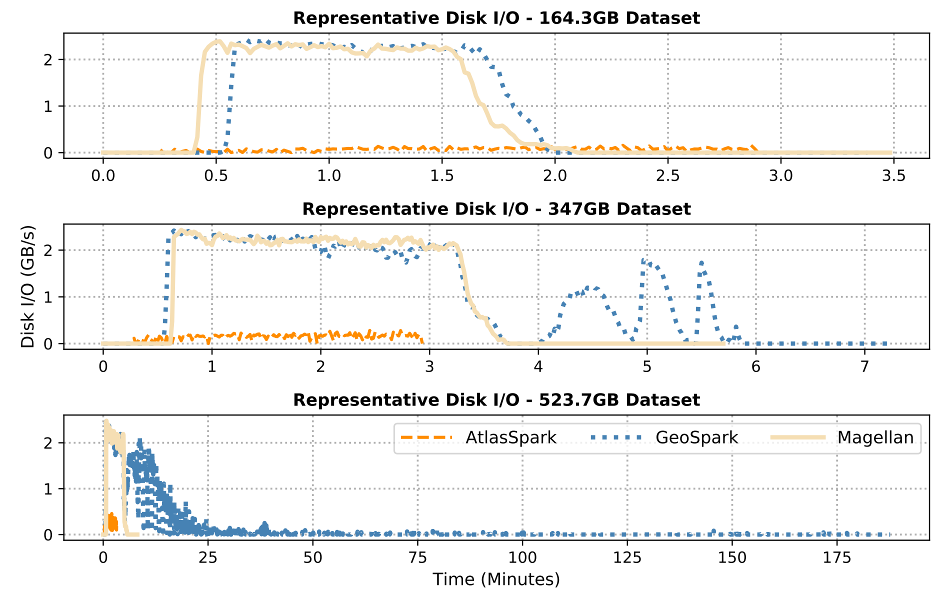
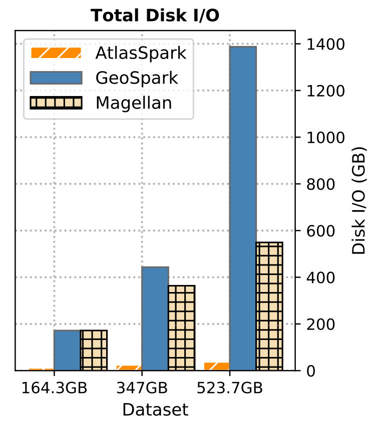


Fig. 5: Graph showing peak cluster memory utilization during range query experiment. AtlasSpark requires up to 57.5% and 48.1% less memory when compared to GeoSpark and Magellan. This is important as dataset sizes increase and may no longer fit into main memory.



(a) Representative disk I/O shows the clusters collective disk I/O at any given moment. Whereas GeoSpark and Magellan require iteration over the entire dataspace to dynamically compile distributed indices, AtlasSpark maintains a small disk I/O footprint by leveraging Atlas File System indices.



(b) Total disk I/O refers to the aggregated disk I/O over cluster nodes during the analytics duration. AtlasSpark reduces disk I/O by 18x and 14x over GeoSpark and Magellan respectively.

Fig. 6: Disk I/O metrics for a sequence of spatially bounded range queries comparing AtlasSpark, GeoSpark, and Magellan.

ing the two filters "atlasGeohash = 8bce" and "atlasGeohash = a97". There are no valid data that satisfy both filters, therefore evaluation is unnecessary.

IV. Empirical Benchmarks and Evaluation

A. Experimental Setup

For our experiments we deployed both the storage (i.e., Atlas and HDFS) and analytics (i.e., Spark) frameworks on a cluster of 25 HP-DL60-G9-E-2620v4 machines. Each machine runs Fedora 29 and is outfitted with an Intel Xeon E5-2620 (8 cores / 16 hyperthreads @ 2.10GHz) and 64GB RAM. Our empirical setup provisions one machine for the Atlas / HDFS namenode and Spark master, the other 24 machines run Atlas / HDFS datanodes and Spark workers. We have allocated 12GB of RAM for each Spark workers and driver, resulting in allocation of 300GB of system memory within the cluster.

We have contrasted the performance of AtlasSpark, our novel Spark spatial extension, with two popular spatial Spark frameworks. GeoSpark [4] and Magellan [5] facilitate efficient spatial retrievals by computing spatial indices over RDD's using R-Trees and Z-Order curves respectively. Table IV provides additional details of the software versions of the analytics stack for each framework.

The National Oceanic and Atmospheric Administration's (NOAA) climate dataset serves as the basis for our evaluation.

TABLE IV: Experimental evaluation storage and analytics framework names and versions.

Analytics		Storage
Spatial Extension	Spark	
AtlasSpark v0.2.1	v2.4.4	Atlas v0.1.1
GeoSpark v1.2.0	v2.3.4	Hadoop v2.9.5
Magellan v1.0.6	v2.2.3	

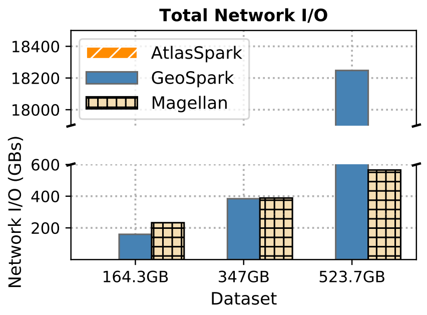
The NOAA NAM dataset records 56 features (represented as floating point values) including temperature, wind speed, pressure, and precipitation at 6 hour intervals from 1.3 million vantage points geographically dispersed within North America. Our analysis is performed on 1.67 TB of data from 2013.

B. Range Queries

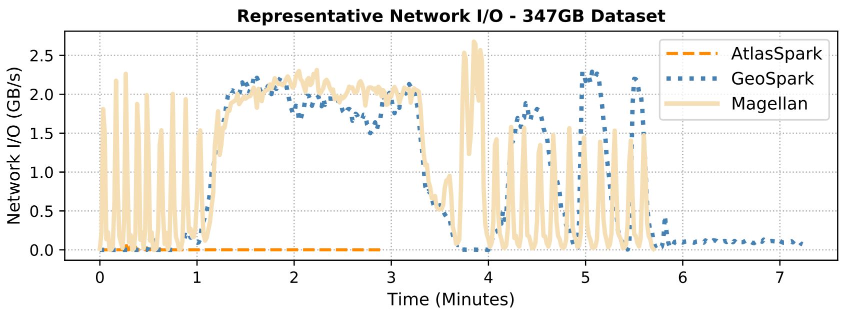
In this experiment we performed spatially constrained range queries over the NOAA dataset. Since the NOAA data comprises U.S.-based vantage points, we restricted our spatial queries within a latitude/longitude envelope governing the continental US. Query predicates were formulated using geohash precisions of length four, giving each query a +/- 0.087 and +/- 0.18 latitude and longitude error respectively. For each platform evaluation we performed 200 randomly defined spatial queries. Since the aforementioned continental US envelope contains 1722 unique geohashes, each platform evaluation provides 12% coverage of the dataspaces spatial scope.

Spatial range queries are a popular metric for evaluation and comparison of spatial analytics frameworks. Current surveys, such as [6], effectively employ them to contrast framework performance statistics. We evaluated spatial range queries under three distinct temporal ranges of the NOAA dataset; namely 1 month, 2 months, and 3 months – these queries effectively represent data subsets of sizes 164.3 GB, 347 GB, and 523.7 GB respectively. These larger dataset sizes seek to profile AtlasSpark's ability to maintain performant analytics for datasets that are too large to fit into memory.

In Figure 4 we display range query duration for the three platforms for each NOAA temporal range dataset. We see that AtlasSpark consistently outperforms both GeoSpark and Magellan. In particular, these benchmarks show that **Atlas**



(a) Total network I/O for 3 different datasets. AtlasSpark requires 500MB – a 4 and 5 order of magnitude reduction compared to Magellan and GeoSpark respectively.



(b) Representative network I/O for the 347GB dataset. GeoSpark and Magellan require significant data movements when constructing distributed spatial indices, whereas AtlasSpark does not.

Fig. 7: Network I/O incurred during 200 random spatial range queries using AtlasSpark, Magellan, and GeoSpark.

outperforms Geospark by a factor 1.2x, 2.6x, and 62.1x; AltasSpark also outperforms Magellan by a factor 1.2x, 2x, and 2.7x for these datasets. These performance improvements are attributable to AtlasSpark's targeted data retrieval, where the system leverages the underlying spatiotemporal indices to perform efficient data retrievals unlike GeoSpark and Magellan which rely on dynamically indexing underlying datasets for queries.

We also profiled peak memory utilization metrics for each experiment; these are depicted in Figure 5. As dataset sizes increase, peak memory utilization for the frameworks increases as well. At the largest dataset the GeoSpark framework *breaches* the cluster's available memory threshold at 300GB. The corresponding analytics durations increases steeply in GeoSpark as a result (see Figure 4) since expensive operations need to be performed to determine data memory residency. Furthermore, AtlasSpark consistently maintains a lower memory footprint – **up to a 57.5% and 48.1% reduction in peak memory utilization compared to GeoSpark and Magellan respectively**.

In Figures 6a and 6b we depict both the representative, and total, disk I/O incurred during analytics respectively. **AtlasSpark reduces disk I/O by a factor of 18x over GeoSpark and by a factor of 14x over Magellan respectively**. Computation of indexing structures in GeoSpark and Magellan entail reading the entire dataset. This spike in disk I/O is easily seen in the beginnings of evaluation in each tier for the representative graphs. Alternatively, AtlasSpark requires significantly less disk I/O by leveraging the underlying spatiotemporal indices. It is also important to note that both representative and total network I/O for each experiment is similar to the aforementioned disk I/O metrics.

Finally, we present representative and total network I/O in Figures 7b and 7a respectively. Figure 7a presents the total cluster network I/O for the experiment with each dataset, whereas Figure 7b only provides a representative look at a single dataset. The representative network I/O data shows strong similarity with representative disk I/O for the provided 347GB dataset in Figure 6a, this characteristic carries for all datasets. We see that **Atlas Spark incurs just 500MB of total network I/O, reducing network I/O by 4 and 5 orders of magnitude when compared with Magellan and Geospark respectively**. Again, this reduction is a construct of AtlasSpark's ability to leverage existing spatial indices instead of dynamically constructing the necessary data structures.

V. Related Work

Data structures specifically designed for spatial indices commonly employ minimum bounding rectangles to group nearby data. Quad Trees [7] employ a tree-based lookup structure where each level partitions the previous level into quadrant buckets. In this algorithm, buckets have a maximum capacity that when reached, adds another level to the tree by splitting the bucket. Alternatively, the R-Tree [8] and its many variants [9], [10], [11] allow for non-uniform bucket sizes and are designed as a self-balancing tree, facilitating more efficient queries. The spatial index in Atlas (and leveraged by our system) builds a radix tree over geohashes. The fine granularity of the former solutions is not suited for two reasons (1) it is much more computationally expensive to construct a distributed index and (2) the limitations imposed by HDFS compliance only support data filtering at a much higher granularity.

A number of efforts have focused on distributing spatial indices in a peer-to-peer application. Some efforts [12] and [13] have proposed algorithms for distributing quad-tree and r-tree implementations respectively. Vbi-tree [14] aims to build a multi-dimensional distributed index, which could be used for spatiotemporal filtering. Spatial P2P [15] introduces modern improvements for more efficient spatial indexing in p2p environments. These systems provide distributed spatial indices over spatiotemporal data, but are unable to interface with modern popular analytics tools. HDFS-compliance and our Spark extensions within Atlas provide the basis to seamlessly interface with myriad tools, including stream processing and machine learning libraries.

Distributed file systems have emerged as a viable data source in distributed analytics. Hadoop Distributed File System [16] is the open-sourced version of Google File System [17]. This system partitions files into blocks and distributes them over a cluster of machines, enabling parallel processing and fault tolerance. HBase [18] and Hive [19] are projects built over Hadoop's MapReduce paradigm to provide a relational DB interface. Lustre [20] and Gluster File System [21] aim to provide highly scalable solutions. These projects provide the basis for efficient, distributed analytics but lack spatial support.

Spatial functionality has been integrated with HDFS in individual projects targeting specifically distributed spatial indices or MapReduce spatial query support. R-Tree and HQ-Tree spatial indices have been added by [22] and [23] respectively. Additionally, range queries [24], K-nearest neighbor [25] and variants [26], [27], and spatial join [28], [29] algorithms have been implemented. A drawback in these efforts is that they are piecemeal efforts; each tool is useful individually, but integration between tools is exceedingly difficult. Additionally, systems such as Galileo incorporate support for approximate [30], analytic [31], and polygon-constrained [32] queries over spatiotemporal datasets; however, the systems lacks integration with analytical engines. In this work, we have provide a suite of spatiotemporal tools that may seamlessly integrate with additional libraries as well.

Hadoop has been extended by many projects to provide a suite of spatial operations. SpatialHadoop [33] modifies the HDFS source-code to support a two layer spatial index using Grid and R-Tree implementations. GISQF [34] exploits SpatialHadoop indices to provide efficient MapReduce queries. Alternatively, MD-HBase [35], Dart [36], and HadoopGIS [37] provide spatial functionality by extending HBase and Hive without modifying any source code. Support for spatiotemporal sketches within Hadoop is implemented in [38] and [39] These systems provide spatial integration with HDFS, but are limited to Hadoop's MapReduce paradigm but not for Spark. Additionally, they lack an in-memory analytical platform facilitating fast iterative, ad-hoc analytics.

The Apache Spark [40] framework provides efficient distributed, in-memory analytics [41]. It has been extended to provide support for relational data processing [42]. This simplifies the Spark interface, easing adoption into workflows. The Spark project has emerged as the standard for in-memory based analytics, but it lacks native spatiotemporal data support.

A variety of efforts provide spatial functionality to the Apache Spark framework. SparkGIS [43], GeoSpark [4], LocationSpark [44], and Simba [45] all provide a multitude of functionality. These systems all rely on building a spatial index over the datasets during data source reads. This results in two limitations (1) the source dataset needs to fit in-memory and (2) dynamic indexing of input data reduces the viability of ad-hoc queries on variable datasets. Atlas provides spatiotemporal indexing of data within the underlying file source. By leveraging the provided indices our system reduces I/O for analytics requiring a filtered dataset. Additionally, analytics tasks encounter fewer memory restrictions because only data of interest is memory resident.

VI. Conclusions and Future Work

This study describes our methodology for supporting effective spatiotemporal analyses within the Spark ecosystem. Our methodology facilitates support for expressive data wrangling operations, and ensures that these operations are performant. Data wrangling operations that we support can be incorporate into canonical HDFS (as we have done) and also any HDFS-compliant system. Our methodology allows these operations to be performant by preserving data locality, minimizing data movements and I/O amplifications. Our methodology is agnostic of the language used to express data processing operations.

RQ-1: Minimizing data movements is predicated on data collocation. By staging such that data from proximate geographical regions are collocated on the same machine, we ensure significantly reduced data movements during analytics. Our benchmarks in Section IV demonstrate this.

RQ-2: Supporting effective data wrangling over spatiotemporal datasets involves plugging into Spark APIs and ensuring that these capabilities are available within the core constructs underpinning Spark. Our set of inspection and transformation operations work with DataFrames, Datasets, and RDDs that are used to express analytics functionality in Spark. Data wrangling support include customizing behavior of certain popular and powerful operations, such as set operations and joins, so that they are better aligned with the spatial characteristics of the data.

RQ-3: To preserve performance, we manage the competing pulls of data dispersion and collocation. We leverage spatial indices to reduce search space during complex geometry and polygon-constrained operations. Interoperating with the Catalyst Optimizer in Spark allows us to leverage code inspection features that facilitate reorganization via predicate pushdowns to prioritize spatiotemporal filtering operations. This reduces not just the amount of data that needs to be loaded into memory, but also the amount of I/O that needs to be performed. Our benchmarks demonstrate the suitability of our methodology: not only do the operations we perform complete faster (1.2-62.1x faster than GeoSpark and 1.2-2.7x faster than Magellan), but we also utilized memory more frugally (57.5% and 48.1% less than GeoSpark and Magellan respectively) and with significantly less disk and network I/O (18x / 5 orders of magnitude improvement over GeoSpark and 14x / 4 orders of magnitude over Magellan). We posit that these benchmarks demonstrate that AtlasSpark is also less likely to induce interference (and degrade performance) of collocated applications.

As part of future work, we will explore building support for specialized spatiotemporal operations such as construction of digital elevation models, information fusion, and anomaly detection. Another avenue that this research leads to is exploring support for spatiotemporal wrangling and analytics in deep learning systems such as TensorFlow and PyTorch.

Acknowledgment

This research was supported by grants the US National Science Foundation [OAC-1931363, ACI-1553685], the Advanced Research Projects Agency-Energy(ARPA-E), and a Cochran Family Professorship.

References

[1] (2020) Geohash project. [Online]. Available: https://www.geohash.org/

[2] D. Rammer, S. Lee Pallickara, and S. Pallickara, "Atlas: A distributed file system for spatiotemporal data," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, pp. 11–20.

[3] (2020) Location tech jts project. [Online]. Available: https://locationtech.github.io/jts/

[4] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2015, p. 70.

[5] R. Sriharsha. Geospatial analytics using spark. [Online]. Available: https://github.com/harsha2010/magellan

[6] V. Pandey, A. Kipf, T. Neumann, and A. Kemper, "How good are modern spatial analytics systems?" *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1661–1673, 2018.

[7] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[8] A. Guttman, *R-trees: A dynamic index structure for spatial searching*. ACM, 1984, vol. 14, no. 2.

[9] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects." Tech. Rep., 1987.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: an efficient and robust access method for points and rectangles," in *Acm Sigmod Record*, vol. 19, no. 2. Acm, 1990, pp. 322–331.

[11] I. Kamel and C. Faloutsos, "Hilbert r-tree: An improved r-tree using fractals," Tech. Rep., 1993.

[12] E. Tanin, A. Harwood, and H. Samet, "Using a distributed quadtree index in peer-to-peer networks," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 16, no. 2, pp. 165–178, 2007.

[13] A. Mondal, Y. Lifu, and M. Kitsuregawa, "P2pr-tree: An r-tree-based spatial index for peer-to-peer environments," in *International Conference on Extending Database Technology*. Springer, 2004, pp. 516–525.

[14] H. V. Jagadish, B. C. Ooi, Q. H. Vu, R. Zhang, and A. Zhou, "Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes," in *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006, pp. 34–34.

[15] V. Kantere, S. Skiadopoulos, and T. Sellis, "Storing and indexing spatial data in p2p systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 2, pp. 287–300, 2008.

[16] K. Shvachko, H. Kuang, S. Radia, R. Chansler *et al.*, "The hadoop distributed file system." in *MSST*, vol. 10, 2010, pp. 1–10.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," 2003.

[18] M. N. Vora, "Hadoop-hbase for large-scale data," in *Proceedings of 2011 International Conference on Computer Science and Network Technology*, vol. 1. IEEE, 2011, pp. 601–605.

[19] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[20] P. Schwan *et al.*, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux symposium*, vol. 2003, 2003, pp. 380–386.

[21] (2020) Glustre file system project. [Online]. Available: https://www.gluster.org/

[22] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on processing spatial data with mapreduce," in *International Conference on Scientific and Statistical Database Management*. Springer, 2009, pp. 302–319.

[23] J. Feng, Z. Tang, M. Wei, and L. Xu, "Hq-tree: A distributed spatial index based on hadoop," *China communications*, vol. 11, no. 7, pp. 128–141, 2014.

[24] Q. Ma, B. Yang, W. Qian, and A. Zhou, "Query processing of massive trajectory data based on mapreduce," in *Proceedings of the first international workshop on Cloud data management*. ACM, 2009, pp. 9–16.

[25] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng, "Spatial queries evaluation with mapreduce," in *2009 Eighth International Conference on Grid and Cooperative Computing*. IEEE, 2009, pp. 287–292.

[26] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, "Voronoi-based geospatial query processing with mapreduce," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE, 2010, pp. 9–16.

[27] K. Wang, J. Han, B. Tu, J. Dai, W. Zhou, and X. Song, "Accelerating spatial data processing with mapreduce," in *2010 IEEE 16th International Conference on Parallel and Distributed Systems*. IEEE, 2010, pp. 229–236.

[28] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1016–1027, 2012.

[29] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *Proceedings of the 15th international conference on extending database technology*. ACM, 2012, pp. 38–49.

[30] M. Malensek, S. Pallickara, and S. Pallickara, "Fast, ad hoc query evaluations over multidimensional geospatial datasets," *IEEE Transactions on Cloud Computing*, vol. 5, no. 1, pp. 28–42, 2015.

[31] ——, "Analytic queries over geospatial time-series data using distributed hash tables," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 6, pp. 1408–1422, 2016.

[32] ——, "Polygon-based query evaluation over geospatial data using distributed hash tables," in *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. IEEE, 2013, pp. 219–226.

[33] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *2015 IEEE 31st international conference on Data Engineering*. IEEE, 2015, pp. 1352–1363.

[34] K. M. Al Naami, S. Seker, and L. Khan, "Gisqf: An efficient spatial query processing system," in *2014 IEEE 7th International Conference on Cloud Computing*. IEEE, 2014, pp. 681–688.

[35] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi, "Md-hbase: A scalable multi-dimensional data infrastructure for location aware services," in *2011 IEEE 12th International Conference on Mobile Data Management*, vol. 1. IEEE, 2011, pp. 7–16.

[36] H. Zhang, Z. Sun, Z. Liu, C. Xu, and L. Wang, "Dart: A geographic information system on hadoop," in *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 2015, pp. 90–97.

[37] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop gis: a high performance spatial data warehousing system over mapreduce," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1009–1020, 2013.

[38] D. Rammer, W. Budgaga, T. Buddhika, S. Pallickara, and S. L. Pallickara, "Alleviating i/o inefficiencies to enable effective model training over voluminous, high-dimensional datasets," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 468–477.

[39] D. Rammer, T. Buddhika, M. Malensek, S. Pallickara, and S. Pallickara, "Enabling fast exploratory analyses over voluminous spatiotemporal data using analytical engines," *IEEE Transactions on Big Data*, 2019.

[40] (2020) Apache spark project. [Online]. Available: http://spark.apache.org

[41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[42] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 2015, pp. 1383–1394.

[43] F. Baig, H. Vo, T. Kurc, J. Saltz, and F. Wang, "Sparkgis: Resource aware efficient in-memory spatial query processing," in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2017, p. 28.

[44] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: A distributed in-memory data management system for big spatial data," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1565–1568, 2016.

[45] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1071–1085.