Beast: Scalable Exploratory Analytics on Spatio-temporal Data

Ahmed Eldawy, Vagelis Hristidis, Saheli Ghosh†, Majid Saeedan, Akil Sevim, A.B. Siddique‡,
Samriddhi Singla, Ganesh Sivaram, Tin Vu, Yaming Zhang§ *
University of California, Riverside
Riverside, CA, USA
†PayPal ‡ University of Kentucky §Tencent Technology Co.
eldawy@ucr.edu,vagelis@cs.ucr.edu,{sghos006,msaee007,asevi006}@ucr.edu,siddique@cs.uky.edu
{ssing068,gsiva005,tvu032,yzhan737}@ucr.edu

ABSTRACT

This paper introduces the open-source Beast system for scalable exploratory data science on big spatio-temporal data. Beast is based on well-established research and has been released to assist the research community with analyzing big spatio-temporal data. Beast provides a set of extensible components that naturally integrate with Spark to build exploratory data science pipelines. Beast can install in less than a minute on an existing Spark cluster and provides a wide array of features including loading vector and raster data represented in standard file formats, synthetic data generation for benchmarking, load-balanced spatial partitioning, data summarization, interactive visualization, and more. Beast builds on several research projects; its goal is to make all this research widely available to researchers in one integrative and coherent system.

CCS CONCEPTS

• **Information systems** → **Data management systems**.

KEYWORDS

Data science,Exploration,Visualization,Spatio-temporal data,Geospatial Data

ACM Reference Format:

Ahmed Eldawy, Vagelis Hristidis, Saheli Ghosh†, Majid Saeedan, Akil Sevim, A.B. Siddique‡, Samriddhi Singla, Ganesh Sivaram, Tin Vu, Yaming Zhang§ . 2021. Beast: Scalable Exploratory Analytics on Spatio-temporal Data. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management (CIKM '21), November 1–5, 2021, Virtual Event, QLD, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3459637.3481897

1 INTRODUCTION

There has been an increasing interest in releasing public data by governments, non-governmental organizations, and industry. The goal is to assist students, researchers, and data scientists in various domains to advance their research in their corresponding domain. At least 60% of this data is geospatial and most of it has a time component [34]. Therefore, there is a pressing need to interactively ingest, explore and utilize this data in data science applications.

To illustrate how users typically use these open data repositories, suppose that a researcher decides to work on a data science project that studies the effect of demographics and homelessness on crime. The researcher will first navigate to 'Data.gov' and search for: 'crime', 'police', 'demographics', and 'homeless' which yield nearly 600, 750, 2000, and 130 datasets, respectively. Now, the next step is to *interactively explore* these thousands of datasets, which are available in various formats, to further narrow down the search to a few that serve the purpose of the research. This exploration process includes data loading, filtering, aggregation, join, and visualization. By looking into existing big spatial data systems, they either focus on big *volume* data [5, 20, 54, 83, 86] or big *velocity* data [26, 55, 73], while in the example above, the user has a problem with the big *variety* in the data.

Existing systems fall short in satisfying user requirements for spatio-temporal data exploration due to four limitations. First, due to the complexity of geospatial data file formats, these systems support only a limited number of them and users still need to run a sequential conversion process that becomes a bottleneck. Existing file formats were designed for traditional file system, thus, providing a distributed data loader and parser is a challenging task. Second, any distributed system needs an efficient load balancer that can partition the data among processing nodes while taking into account its multidimensional aspect. Third, existing systems focus on processing one dataset at a time while data exploration tasks need the concurrent processing and exploration of many datasets. Fourth, users need to explore the raw data and results through a visual-based interface that displays data on an interactive map.

This paper introduces Beast, an end-to-end big-data framework for exploratory analytics on spatio-temporal data. Beast provides a unique design that aim at providing exploratory queries on large scale spatio-temporal data. 1) Beast ships with a set of distributed loaders and writers for various binary and textual formats for both vector data, e.g., Shapefile and GeoJSON, and raster data, e.g., GeoTIFF and HDF. These loaders can also decompress the data on-the-fly to minimize the need of an external preprocessing step. Addtionally, Beast adds a parallel data generator for testing and benchmarking with reproducibility in mind. 2) It provides memory and disk-based partitioning framework for multidimensional data to speed up data loading and query processing [75, 76, 78]. 3) It encompasses an optimized set of operations, e.g., range selection and join framework for multidimensional data to support the efficient

*Work done while at UCR
Authors are ordered alphabetically

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CIKM '21, November 1–5, 2021, Virtual Event, QLD, Australia
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8446-9/21/11.
https://doi.org/10.1145/3459637.3481897

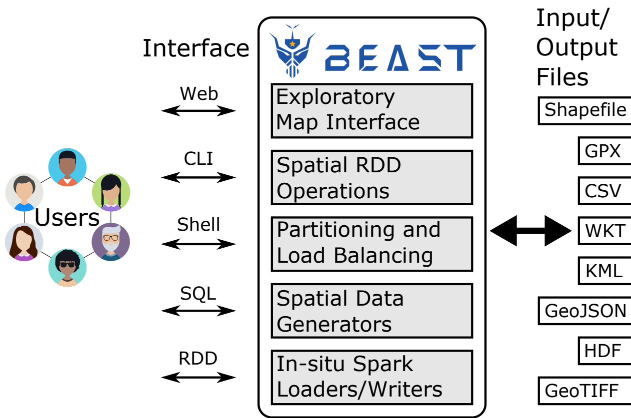


Figure 1: Architectural overview of Beast

exploration of multiple datasets [24, 69, 71, 72]. 4) Beast provides a scalable interactive visualization for exploring big data [27–30].

While some of the components of Beast have been published individually in earlier work [13, 24, 27–30, 42, 66–72, 75–79, 88], this paper focuses on three specific components that have not been explained in earlier work, distributed data loading and writing, spatial data generator, and scalable query and join framework. This paper also describes how Beast is integrated into Spark to build one cohesive system[1].

In the rest of this paper, Section 2 gives an overview of the system components. Section 3 describes the interfaces that Beast provides followed by details of system components in Sections 5-9. Section 10 gives an experimental evaluation. The related work is described in Section 11. Finally, Section 12 concludes the paper.

2 OVERVIEW

Figure 1 gives an architectural overview of Beast which consists of five main components. First, to support in-situ data processing, Beast provides a set of parallel *loaders and writers* for popular file formats such as Shapefile, CSV, GeoJSON, and GeoTIFF. This component also provides scalable spatial data generators for stress testing and benchmarking. Second, the *spatial partitioner and load balancer* component provides a set of spatial partitioning techniques which can group spatially relevant records into partitions while balancing the load across the executor nodes. The partitioned data can be written to disk in any of the standard file formats to be reused by Beast or any other system. Third, to support interactivity, the *interactive query processor* offers a set of data synopses to facilitate approximate query processing, e.g., sample, point histogram, Euler histogram, and Bloom filter. It also uses these synopses to build some approximate algorithms such as clustering and selectivity estimation. Fourth, the *scalable join framework* is crucial for big variety data since it allows users to integrate multiple datasets together. Beast provides a set of distributed join algorithm with various optimizations to handle big spatial data efficiently and uses a rule-based optimizer to choose the most appropriate algorithm. Finally, the *exploratory map interface* helps users in visually exploring the input data or the query results on an interactive map interface.

Users can interact with all components of Beast through various interfaces. The web interface provides a graphical interface for some features in Beast such as the map visualization, data retrieval, and conversion. UCR-Star [30] is a an example of a web application built using this web interface. The command-line interface (CLI) gives quick access to some common features in Beast such as data conversion, indexing, and visualization. The interactive shell extends the Spark Scala shell with all features of Beast. It allows developers to try out the features of Beast or write short code snippets. In this paper, all samples shown in yellow boxes are *complete code examples* that can run in the Beast interactive shell. Finally, the SQL and RDD APIs are built on the corresponding APIs in Spark and helps developers who want to integrate Beast component into their own Scala or Java programs.

3 INTERFACES

There are five main interfaces to work with Beast the RDD API for Scala and Java, SQL API, interactive shell, command-line interface (CLI), and a web interface. The lowest-level is the RDD interface which developers can use in their Java or Scala project. The SQL interface extends SparkSQL with geometric data types and functions for relational operations and it provides interoperability with the RDD API. The interactive shell interface extends the Spark shell for quick access of all Beast functions in Scala without having to worry about writing a complete program. All code examples given in this paper can directly run in the interactive shell. The command-line interface (CLI) provides quick access to some common functions such as data conversion, indexing, and visualization, without having to write a single line of code. Finally, the RESTful interface provides a limited access to certain functions in Beast that need to be accessed from the web such as web map visualization.

The only required prerequisite for Beast is a running Spark system. The RDD and SQL APIs are installed by adding Beast Maven coordinates in the project configuration file. Beast provides a Maven template that generates a sample project that is readily integrated with Beast. The interactive shell and CLI are installed by downloading and extracting the Beast binary package which takes only a few seconds to download and install and does not require a restart of Spark. Finally, the REST interface is started by running the '`server`' command from the CLI interface.

4 SPATIAL DATA TYPES

This section describes the new data types and classes that are introduced in Beast to enable geospatial data exploration. Our goal is to have a highly-compatible design that allows users to run exploratory queries that combine both spatial and non-spatial data and operations. Other systems, e.g., Apache Sedona, use a *wrapper* approach that creates new constructs that wrap around Spark classes. For example, the SpatialRDD in Sedona is a wrapper around *four separate* RDDs in addition to some statistics of the data. While this design allows queries to run more efficiently by caching multiple versions of the data, it makes SpatialRDD incompatible with existing RDD functions in Spark. Therefore, it does not well support exploratory queries that combine multiple operations and datasets.

[1] Available as open source at https://bitbucket.org/bdlabucr/beast

The design of Beast maximizes compatibility with Spark core by extending existing classes rather than introducing new ones. For example, SpatialRDD in Beast is just an alias for RDD[IFeature] which makes it compatible with all existing RDD transformations and actions. This part describe the data types defined in Beast.

IFeature: is a geometric feature that combines a geometry, e.g., point, line, or polygon, with zero or more non-geometry attributes, e.g., unique ID or name. For compatibility with the Spatial Dataframe API, IFeature extends the Row interface.

SpatialPartitioner: is an extension of Spark partitioner that assigns a geometric feature to a partition *based on its minimum bounding box (MBB).*

SpatialRDD=RDD[IFeature]: SpatialRDD is an alias for RDD[IFeature]. It defines an RDD that contains geometric features. A SpatialRDD can optionally include a SpatialPartitioner that defines the MBB of each partition.

ITile: is the smallest unit of a raster layer, e.g., satellite data. It represents a two-dimensional array of values and a mapping to associated geo-location to these values.

RasterRDD=RDD[ITile]: is an RDD that contains a set of tiles. All tiles in an RDD collectively define a raster layer.

5 SPARK LOADERS AND WRITERS

Beast ships with Spark-compliant loaders and writers for most common spatial formats for both vector and raster. The goal is to provide in-situ processing for a wide range of data formats that helps scientists in exploring publicly available data without preprocessing. All data loaders are accessible from the Spark context instance. The following example reads a GeoJSON file.[2]

```
val buildings: SpatialRDD = sc.geojsonFile("buildings.geojson")
```

Beast decompresses most formats on the fly to avoid a separate decompression step. Vector and raster files are loaded as `SpatialRDD` or `RasterDD`, respectively. The spatial loader is defined as a top-level RDD, i.e., one that does not depend on other RDDs, and produces the initial set of geometric features. Once the RDD is created, it can be used as a regular RDD in Spark.

GeoJSON Parser: In the following part, we explain the design of the *GeoJSON parser* which is one of the challenging data formats that is fully integrated in Beast. To the best of our knowledge, Beast is the *only* system that can process any valid GeoJSON file on Spark. Other systems, e.g., Sedona, can process a GeoJSON file only if the header is omitted and records are organized as one per line; these conditions do not generally hold on publicly available data.

The main challenge in GeoJSON files is when a big file is partitioned across executors. In this case, there are two main issues that are not yet solved by other systems. First, how to correctly parse records that cross partition boundaries. Second, how to correctly parse GeoJSON data when the parser does not start from the beginning. Figure 2 gives an example of these challenges. The given file is split into two partitions as indicated. In this case, a regular parser will fail to read the first partition since it does not contain a complete Feature object. It will also fail to parse the second partition since it starts with a string which is not a valid JSON syntax.

[2] Input/Output: https://bitbucket.org/bdlabucr/beast/src/master/doc/input-output.md

```
{
  "type" : "FeatureCollection",
  "features" : [ {
    "type" : "Feature",
    "geometry" : {
      "type" :
--- Partition boundary ----
      "Point",
      "coordinates" : [ -117.4, 34.0 ]
    }
  }, {
    "type" : "Feature",
    "geometry" : {
      "type" : "Point",
      "coordinates" : [ 120.0, 30.4 ]
    }
  } ]
}
```

Figure 2: GeoJSON parsing example

To solve the problem of records that span two partitions, we define an *anchor point* for each record as the position of the start object '{' character. Then, we make a rule that the partition that contains the anchor point is responsible of reading the record even if it spans the next partition. Since the anchor point can only fall in one partition, this ensures that each record is read exactly once.

To solve the problem of starting the JSON parser at any point, we create a new component termed *silent JSON parser*, which acts a stream parser that emits one token at a time. Unlike existing parsers, the silent parser absorbs any parsing errors and restarts the parser at the next token. For example, when the silent parser works on the second partition in the example above, it will encounter many errors until it reaches the first start object token. Then, it will correctly parse the second object. It will again find an error when it reaches the end array and end object tokens which close the array and the object that started in the first partition. However, this will not result in missing any GeoJSON features in the file.

We further extend our GeoJSON parser to work with block-based compressed files, i.e., the file consists of a sequence of independently compressed blocks. Since the block boundaries are not known, we will have the same issues of blocks that span multiple partitions and the decompressor not able to start at the middle. To solve these problems, we follow a similar approach to the one above. First, we define an anchor point for each block to be its first byte to ensure that we process each block exactly once. Additionally, when the decompressor starts from the middle of the file, it reads and ignores the invalid bytes until it reaches a special character that marks the block boundary at which it starts the decompression and parsing.

Data Writing: Similar to data loading, Beast can write any SpatialRDD in many standard file formats such as GeoJSON, KML, and Shapefile. This allows Beast to easily integrate with most existing spatial data systems. Spatial writing is implemented in Beast as a new Spark action that runs in parallel and writes each RDD partition to a separate file.

The following code reads a file in CSV format and writes it back as a Shapefile, which illustrates how Beast acts as a scalable converter of geospatial files.

```
sc.readCSVPoint("input.csv", "x", "y", ',').saveAsShapefile("output.shp")
```

To the best of our knowledge, the above single line of code is the most scalable data converter available out there for geospatial files which reads, decompresses, converts, and writes the output in parallel.

Integration with SparkSQL:. Any SpatialRDD can be converted to a dataframe to run with SparkSQL. Beast adds a set of standard functions similar to the ones used popular geospatial packages, e.g., PostGIS and Oracle Spatial. Similarly, a dataframe that contains a geometry can be converted to a SpatialRDD. For example, the following code loads a Shapefile, converts it to a DataFrame, calculates the area of each geometry using SQL, and finally, writes the result back as GeoJSON.

```
val counties: SpatialRDD = sc.shapefile("us_counties")
counties.toDataFrame(spark).createOrReplaceTempView("counties")
val counties_areas = spark.sql(
  "SELECT NAME, g, ST_Area(g) FROM counties")
counties_areas.toSpatialRDD.saveAsGeoJSON("us_counties_areas")
```

6 PARALLEL SPATIAL DATA GENERATOR

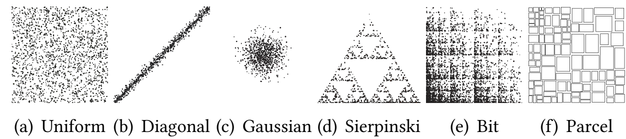
To assist with benchmarking and promote the reproducibility of results, Beast extends the award-winning spatial data generator Spider [42, 79] for parallel generation of multidimensional data. The data is generated in parallel as a SpatialRDD which makes it scalable and ready to integrate with other Spark RDDs. For example, the following code loads a set of polygons, and use their minimum bounding rectangle (MBR) to generate points and join them with the polygons. Users can easily change the size or distribution of the data to test the behavior of the program.[3]

```
val polygons: SpatialRDD = sc.shapefile("us_counties")
val randomPoints: SpatialRDD = sc.generateSpatialData.
  mbr(polygons.summary).uniform(1000000)
val sjResult = polygons.spatialJoin(randomPoints)
```

Figure 3 depicts the six data distributions currently supported by Beast [79]. The original generator that we base our work on is implemented as a single-machine script in Python and Ruby which limits its scalability.

To parallelize the data generation process, we have to overcome three limitations in Spark. First, Spark requires every task to be *deterministic* so that the failed tasks could be resubmitted to complete. Second, different tasks should generate different set of random data while still being deterministic. Third, and specifically for the parcel distribution, the generator is inherently sequential and cannot be directly parallelized.

To overcome the first two limitations, we define a new type of RDD in Beast called RandomSpatialRDD which consists of partitions of type RandomSpatialPartition. When the RandomSpatialRDD is constructed, a fixed seed s is generated (or provided by the user), and is attached to the RDD for the lifetime of the application. This seed defines the random number generator used to generate all data in this RDD. To make sure that each partition generates different data, each task that processes a partition initializes its own random number generator with seed $s + i$ where i is the partition index. This ensures that each task is still deterministic since both s and i are fixed for each task while being different.



(a) Uniform (b) Diagonal (c) Gaussian (d) Sierpinski (e) Bit (f) Parcel

Figure 3: Examples of supported spatial distributions [79]. Readers can explore them at https://spider.cs.ucr.edu

Accommodating the parcel generator is the third challenge that we need to address. The parcel generator works by recursively splitting the input space either vertically or horizontally until a desired number of records is reached. This process is inherently sequential. To untangle this problem, we remodel this generator as a tree traversal algorithm where the root is the original input space and the leaf nodes are the generated records. Then, we transform the parcel generator from a depth-first search (DFS) traversal to breadth-first search (BFS). When the RDD is initialized, we run the BFS generator to generate n boxes that defines the partition boundaries. Then, each partition will start with one of these boxes and continue the splitting process in parallel. This requires us to adjust the number of records generated in each partition to ensure that the correct number of records is generated.

7 PARTITIONING AND LOAD BALANCING

Any big data framework needs to partition the data over multiple machines. Beast provides a set of spatial partitioners that can take any SpatialRDD and produce a spatially partitioned RDD. The main difference between a regular SpaitalRDD and a spatially partitioned RDD is the existence of a SpatialPartitioner that is associated with the RDD. The SpatialPartitioner keeps track of the minimum bounding box (MBB) of each partition in the RDD. As shown later, this information is used to speed up spatial query processing. The following code shows how to partition a dataset using the R*-Grove partitioner[4].

```
import edu.ucr.cs.bdlab.beast.indexing.RSGrovePartitioner
val partitioned: RDD[IFeature] = sc.shapefile("points.shp").
  spatialPartition(classOf[RSGrovePartitioner])
```

Additionally, a spatially partitioned RDD can be saved to disk in any standard file format and loaded later along with its partitioning information.

```
// Save a partitioned RDD to disk
partitioned.saveAsShapefile("partitioned_data")
// To load the data back in another Spark application
val loadedPartitioned = sc.shapefile("partitioned_data")
```

Unlike other systems that store partitioned data as object files that are only readable by the same Spark application, Beast stores the data in any standard file format which makes it readable by all supported applications.

[3] https://bitbucket.org/bdlabucr/beast/src/master/doc/spatial-data-generator.md

[4] https://bitbucket.org/bdlabucr/beast/src/master/doc/partitioning-indexing.md

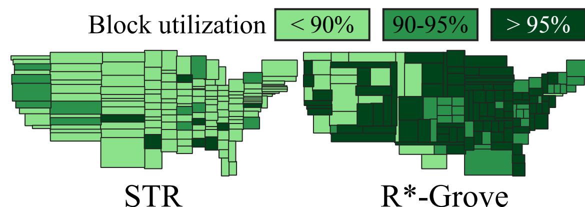


Figure 4: STR provides less than 90% block utilization while most R*-Grove partitions are more than 95% utilized

partitioning techniques in Beast can work with arbitrarily many dimensions which allows developers to use with spatio-temporal data. Existing big spatial data system use the sample-based partitioning method which was introduced in SpatialHadoop [20] which works in three phases. Phase 1 picks a random sample of the input dataset to infer its distribution. Phase 2 partitions the space into *n* partitions with roughly equal number of sample points. Phase 3 scans the input dataset in parallel and assigns records to partitions based on their location. The second phase is critical to the quality of final spatial partitions. Most existing techniques use a bulk loading mechanism to construct an index such as STR [44], Quad-tree [63], or Kd-Tree. Although these partitioning techniques have been integrated into many big spatial data systems, they are all holding one or more of the following three limitations. First, some spatial index structures (STR, Kd-tree) prefer load balance than spatial quality, e.g., total area or total margin of partitions [19]. Hence, they could produce very thin or wide partitions as shown in Figure 4, which is not good for spatial query processing. Second, some index structures (R-tree, Kd-tree, and Quad-tree) produce underutilized leaf nodes in order to occupy the future inserts. This is not optimal for distributed file systems such as HDFS, where it expects each block to be fully occupied to reduce the number of blocks. Third, all of existing index structures do not take the record's size into account. As a result, they can produce highly unbalanced partitions in term of storage size if the input datasets contains variable-size records, even though they might be balanced in terms of number of records.

Beast provides an efficient spatial partitioning technique, called R*-Grove [76, 78], that overcomes these limitations. First, it utilizes the R*-tree optimization methods [11] to produce high quality spatial partitions. Second, it produces full block and balanced partitions, by creating a new constraint of the node's size in the node splitting process. In short, it requires a lower bound on the ratio of the smallest and largest partition, e.g., 95%, so that the final partitions should occupy at least 95% of a full HDFS block (i.e., 128 MB). Third, if the input dataset contains variable-size records, R*-Grove combines the sample points with a histogram of data size so that each sample point is assigned a weight. These weights will be referred in the node splitting process to guarantee the storage size of every partition always falls into a desired range. To the best of our knowledge, R*-Grove is the first *static* partitioning method that supports load balancing for spatial datasets with variable-size records. It does not require the query workload's awareness in order to balance partitions as existing systems [1, 7, 8, 16]. Figure 4 shows the square-like partitions with high block utilization, which are produced by R*-Grove. This promises a better query processing performance than the STR-based partitions.

Deep learning-based spatial partitioning: Along with R*-Grove, Beast still supports other popular partitioning techniques such as Grid, STR, and Kd-tree. The best partitioner depends on the dataset characteristics, distribution, and query requirements. However, it is a challenging problem to anticipate the best partitioning technique given the high variability in spatio-temporal datasets. To overcome this problem, we propose a deep learning based system [75] that automatically selects a suitable partitioning technique that promises the best optimization metric. The system runs in two main phases: training and inference. The offline training phase builds a training dataset using our spatial data generator [42, 79]. Each dataset is partitioned using all partitioners and the best one is chosen to train a classification model. In the inference phase, the trained model is used to predict the best partitioning technique for the given input dataset. This system helps users to automatically choose the appropriate partitioning technique without any heuristic experience of spatial data partitioning.

8 SPATIAL RDD OPERATIONS

This section focuses on three operations that are very important for data exploration, *range query*, *spatial join*, and *zonal statistics*.

8.1 Range Query

The range query operation filters geometric features that intersect a multidimensional box and returns an RDD of matching filters as in the example below.

```
val partitionedData = sc.shapefile("partitioned_points")
val areaOfInterest = sc.shapefile("us_counties")
 .filter(_.getAs[String]("NAME") == "Los Angeles").first
val result = partitionedData.rangeQuery(areaOfInterest)
```

In standard Spark, this method is implemented using a filter operation which scans the entire file. Beast utilizes spatial partitioning of RDDs to run this query more efficiently by early pruning of partitions that are outside the query range. This is done through the use of `PartitionPruningRDD` which compares the bounding box of each partition to the query range before running the filter operation. If the data is not spatially partitioned, Beast falls back to the standard Spark implementation which scans the entire file.

8.2 Spatial Join

The spatial join operation takes two input datasets R and S that contain geometric features and a spatial predicate θ. It outputs a dataset of pairs of features $(r \in R, s \in S)$ that $\theta(r, s)$ is true. For example, it can take a GPS traces as points and ZIP code boundaries and associates each point with the containing ZIP code. The following code snippet shows an example that counts the number of GPS points in each ZIP code[5].

```
import scala.collection.Map
val zipCodes = sc.shapefile("zcta5")
val gpsPoints = sc.spatialFile("gps_tracks", "gpx")
val sjResult: RDD[(IFeature, IFeature)] =
    zipCodes.spatialJoin(gpsPoints)
val pointsPerZIP: Map[IFeature, Long] = sjResult.countByKey()
```

[5]https://bitbucket.org/bdlabucr/beast/src/master/doc/spatial-join.md

Spatial join optimization techniques can be broadly categorized under *filter optimization* and *refinement optimization*. The filter optimization aims at reducing the *number* of predicate tests while refinement optimization reduces the *cost* of each predicate test. The following part highlights the main optimization techniques employed by Beast.

Spatial Dependency: A Spark dependency defines how the partitions in one child RDD depend on the partitions of multiple parent RDDs. In Beast, we define a new dependency called *Spatial Dependency* which is created mainly for the spatial join operation. Spatial dependency relates one child RDD, the represents the spatial join result, to two spatially-partitioned parent RDDs, the represent the two inputs. It defines it in a way such that one partition in the child RDD depends on two partitions in the two parent RDDs that spatially intersect. This dependency is used to create a new RDD, called `SpatialIntersectionRDD`, which takes two spatially partitioned RDDs and produce a new RDD where each partition represents a pair of overlapping input partitions. The SpatialIntersectionRDD is used to implement the following three spatial join algorithms which are all categorized as *filter optimizations*.

Index-based join resembles the R-tree join algorithm [37] and is applicable only when both RDDs are spatially partitioned. It first uses the SpatialIntersectionRDD to find pairs of overlapping partitions. Then, it performs a plane-sweep join algorithm between every pair of partitions. Finally, it uses the reference point duplicate avoidance technique [17] only if both input datasets contain replicated geometries. The following code snippet gives a sketch of how this algorithm is impelemented in Beast.

```
def indexBasedJoin(r1: SpatialRDD, r2: SpatialRDD) = {
 require(r1.isSpatiallyPartitioned)
 require(r2.isSpatiallyPartitioned)
 val matchingPartitions = new SpatialIntersectionRDD(r1, r2)
 matchingPartitions.flatMap(joinedPartition =>
  planesweep(joinedPartition._1, joinedPartition._2)
 )
}
```

The first two lines verify that the two inputs are partitioned. Then, it uses the SpatialIndersectionRDD to produce a new RDD that contains pairs of partitions. Finally, it passes the contents of each partition, which is two sets of records, to the planesweep function which runs a local spatial join algorithm between the two sets of features.

Repartition Join (RJ) is another filter optimization algorithm that is based on the seeded tree join algorithm [48]. When only one dataset is partitioned, it repartitions the other dataset to match the partitioned dataset. Then, it performs the plane-sweep join algorithm for each pair of matching partitions. Finally, it employs the duplicate avoidance technique if needed. The following code snippet explains how it is implemented.

```
def repJoin(r1: SpatialRDD, r2: SpatialRDD) = {
 val r2Partitioned = r2.spatialPartition(r1.partitioner)
 new SpatialIntersectionRDD(r1, r2Partitioned)
  .flatMap(joinedPartition =>
   planesweep(joinedPartition._1, joinedPartition._2)
  )
}
```

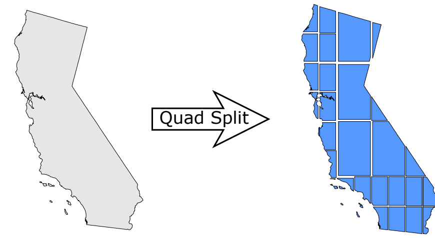


Figure 5: Quad split optimization for spatial join. Gaps in the broken geometry are added for illustration.

The algorithm simply partitions the second dataset to match the partitioner of the first dataset. Then, it uses the SpaitalIntersectionRDD to combine the matching partitions and join them.

Partition based spatial merge (PBSM) [59] join algorithm is used when non of the input datasets is partitioned. First, it partitions the input space into disjoint cells. Then, it replicates each record in both datasets to all overlapping cells. After that, it runs the plane-sweep algorithm on the records in each cell. Finally, it removes duplicates using the reference point technique. While the original PBSM algorithm uses a uniform grid to partition the space, Beast can utilize the R*-Grove partitioning technique which balances the load when the data is highly skewed. The following code snippet explains how PBSM is implemented in Beast.

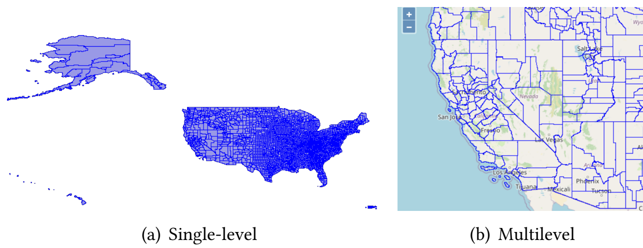
```
def pbsm(r1: SpatialRDD, r2: SpatialRDD) = {
 val intersectionMBR = r1.summary.intersection(r2.summary)
 val partitioner =
  new GridPartitioner(intersectionMBR, 1000, 1000)
 val r1Partitioned = r1.spatialPartition(partitioner)
 val r2Partitioned = r2.spatialPartition(partitioner)
 new SpatialIntersectionRDD(r1Partitioned, r2Partitioned)
  .flatMap(joinedPartition =>
   planesweep(joinedPartition._1, joinedPartition._2)
  )
}
```

The first three lines compute the intersection area between the two input datasets and use it to create a grid partitioner. Then, it partitions both datasets using the common partitioner. Finally, it uses the SpatialIntersectionRDD to combine matching partitions and join them.

For *refinement optimization*, Beast uses the quad-split approach as illustrated in Figure 5. It recursively breaks down a complex geometry into four quadrants until the number of points in each piece falls below some threshold, e.g., 100 points per geometry. This approach is applied when the average number of points per geometry is higher than a threshold of 100. The refinement optimization can be combined with any of the above three algorithms.

8.3 Zonal Statistics

Zonal statistics is a grouped aggregate operation that takes as input a polygon dataset and a raster dataset. It computes four aggregate functions over the values of the pixel that lie inside each polygon, namely, min, max, sum, and count. This operation becomes very expensive when operating on high-resolution raster and vector data. We recently proposed a Hadoop-based distributed implementation [71] that can partition both vector and



(a) Single-level (b) Multilevel

Figure 6: Visualization techniques in Beast. Explore hundreds of visualizations at https://star.cs.ucr.edu

raster data and process them efficiently. Porting this algorithm to Beast is a work-in-progress but initial experiments show up-to an order of magnitude speedup over the Hadoop implementation. For more details on how to run zonal statistics in Beast, check https://bitbucket.org/bdlabucr/beast/wiki/raptor.

9 EXPLORATORY MAP INTERFACE

Domain scientists heavily rely on visualization for exploring query results and for sharing their findings. Map visualization can be broadly categorized into two types, *client-side rendering* and *server-side rendering*. In **client-side rendering**, the database system processes the data regularly and produces a small-size result that the browser can render using JavaScript. Most systems support this type of visualization since it does not require any special features [15, 31, 38, 64, 65, 81, 84, 84] but it is not generally scalable.

In **server-side rendering**, the database server creates the image and the browser just displays it on the screen which gives an opportunity to handle big data efficiently [23, 45, 56, 60, 85, 87]. This can be further categorized as *single-level visualization* or *multilevel visualization*. Single-level visualization produces a single image with a fixed resolution, e.g., $2,000 \times 2,000$ pixels. The output can be included in a report or displayed on the screen as shown in Figure 6(a). The following code shows how to generate a single-level image in Beast.

```
sc.shapefile("us_counties").plotImage(2000, 2000, "counties.png")
```

The level-of-details of single-level images is limited by its resolution. To allow users to zoom in and see more details, multilevel images can be used as they consist of hundreds of millions of tiles that can be fetched and arranged on the screen depending on the region currently displayed on the map. Figure 6(b) shows the final displayed visualization where tile boundaries are completely transparent to the end user. The main challenge is how to manage those billions of tiles [57]. Existing systems either create and materialize all tiles [23, 87] or cache the data in memory and render tiles on the fly [85]. Both techniques consume too much resources that limit their scalability. Beast overcomes this problem by integrating a new adaptive visualization index, termed AID* [27, 28], which materializes a minimal number of tiles to ensure an interactive response and generates other tiles on-the-fly using a disk-based index. AID* has been successfully deployed in UCR-Star [https://star.cs.ucr.edu] [29, 30] which currently serves hundreds of datasets with terabytes in size. AID* is very conservative on resource consumption that UCR-Star is hosted on a single-machine and consumes less than a gigabyte of memory.

Name	Size	# records	format
All Nodes	97 GB	2.7 billion	CSV
All Objects	92 GB	264 million	WKT
MS Buildings	27 GB	125 million	GeoJSON
Roads	9 GB	18 million	Shapefile

Table 1: Datasets used in experiments

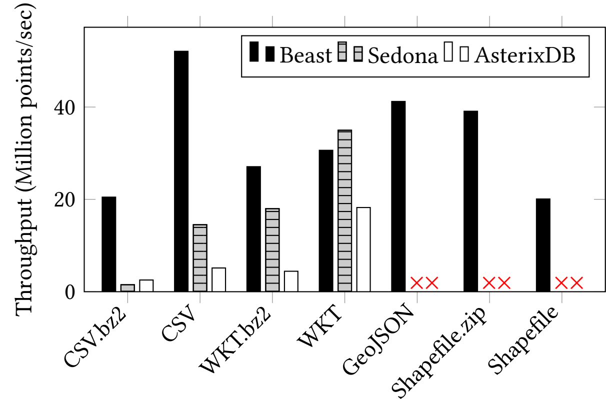


Figure 7: Data scanning throughput

The following code snippet generates a multilevel visualization for a spatial dataset with up-to 10 zoom levels. This will produce a directory that contains the generated tiles and an HTML file that displays them on an interactive map[6].

```
sc.shapefile("us_counties").
  plotPyramid("counties-multilevel", 10, opts = "mercator" -> true)
```

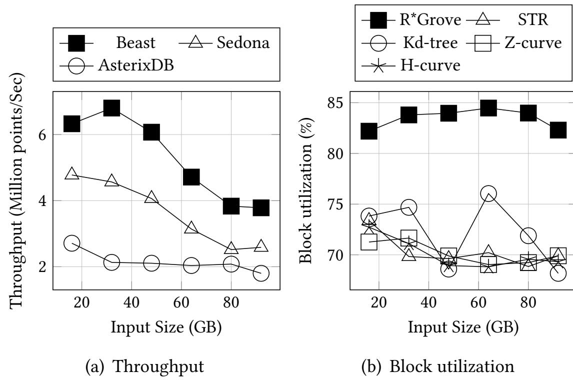
10 EXPERIMENTS

This section provides an experimental evaluation of Beast 0.9.1 in comparison to Sedona 1.0-incubating (formerly GeoSpark) and AsterixDB 0.9.5 [6]. The experiments run on a cluster with one head node and 12 worker nodes. The head node has Intel(R) Xeon(R) CPU $E5 - 2609$ v4 @ 1.70GHz processor, 128 of GB RAM, 2 TB of HDD, and 2×8-core processors running CentOS and Oracle Java 1.8.0_131. The worker nodes have Intel(R) Xeon(R) CPU E5-2603 v4 @1.70GHz processor, 64 GB of RAM, 10 TB of HDD, and 2×6-core processors running CentOS and Oracle Java 1.8.0_31-b04. All the datasets we use in this paper are publicly available on UCR-Star [30]. Table 1 summarizes the datasets that we use in this section.

10.1 Data scanning

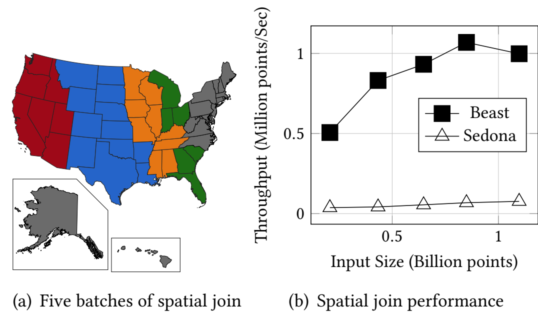
As Beast focuses on in-situ processing, this first experiment shows the performance of scanning and parsing data in various file formats. For CSV, WKT, ad Shapefiles, we run on both decompressed and compressed versions. Figure 7 shows the ingestion throughput. In this experiment, we load the dataset and run a `count` operation

[6] https://bitbucket.org/bdlabucr/beast/src/master/doc/visualization.md



(a) Throughput (b) Block utilization

Figure 8: Partitioning and indexing performance



(a) Five batches of spatial join (b) Spatial join performance

Figure 9: Spatial join on buildings ⋈ roads

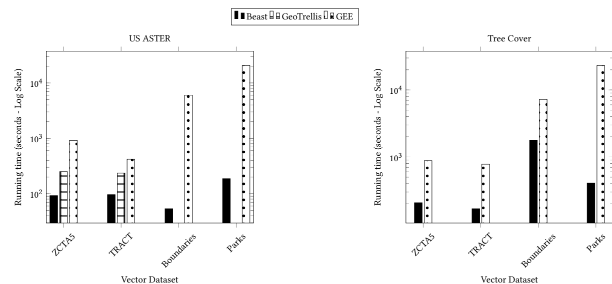


Figure 10: Zonal Statistics

to measure the throughput in terms of millions of points per second. All systems are scalable but Beast stands out with its performance and supported file formats. It is interesting to see that the throughput with Shapefile almost doubles when the input file is compressed due to the smaller file size and the efficient decompression. This is not the case with compressed text files which are compressed using the bzip2 which is more expensive to decompress. AsterixDB does not support Shapefile or GeoJSON input formats. While Sedona does support these two formats, it requires each record to appear in a separate line in GeoJSON files which is not the case with the MS Buildings dataset. For Shapefile, it can only read one file at a time while the roads dataset contains thousands of files. For the rest of the experiments, we convert the non-supported formats so that we can run all experiments.

10.2 Data partitioning and indexing

This experiments measures the performance of the partitioning and indexing steps of Beast, Sedona, and AsterixDB. Beast uses R*-Grove partitioner and R*-tree to index each partition. Sedona uses Quad-tree partitioner and R-tree as a local index. AsterixDB supports only a hash partitioner on the primary key and R-tree local index. While the indexing step is not always required, it is usually recommended to get the best performance.

To control the input size, we split the all-objects dataset into six batches of 16 GB each. Figure 8(a) shows that all systems are scalable but Beast is consistently faster. Figure 8(b) reveals the block utilization of different partitioning techniques calculated as the ratio between the partition size and block capacity as defined in [78]. The higher the block utilization, the better the load balance. R*-Grove is a clear winner when compared to other techniques, since it is the only method that takes the record's write size into account in its partition's boundaries computation step.

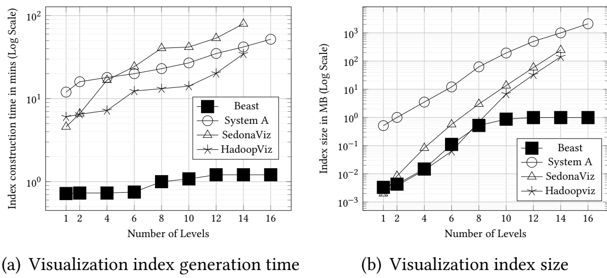
10.3 Spatial join operation

Figure 9 shows the performance of the spatial join operation between Roads and MS Buildings datasets. To vary the input size, we split both datasets by state into five batches as shown in Figure 9(a). The total number of points in both inputs increase from 200 million to a little over one billion and we measure the throughput in terms of millions of points per second. We did not include the results of AsterixDB since it does not yet have a scalable spatial join query plan. For both Beast and Sedona, we measure the overall running time including data loading, partitioning, indexing, and processing. In Sedona, we use quad-tree partitioning and R-tree indexing. In Beast we use PBSM since the inputs are not indexed. Beast is significantly faster than Sedona due to the spatial join optimizations and the load balanced partitioning.

Figure 10 shows the performance of the zonal statistics operation as compared to Google Earth Engine (GEE) [32] and GeoTrellis [43] for two big raster datasets, US Aster and Treecover, with up-to a trillion pixels. This experiment clearly shows up-to two orders of magnitude speedup over the baselines with some systems, e.g., GeoTrellis, failing to finish the process. Interested readers can refer to [71] for more details.

10.4 Visualization

This experiment studies the scalability of the multilevel visualization algorithm for all-nodes dataset. We compare AID* in Beast [27] to SedonaViz [85], HadoopViz [23], and a commercial system, termed System A, when they generate all required tiles in a desired range of zoom levels. In Figure 11(a), as we increase the number of zoom levels, AID* becomes two orders of magnitude faster since it produces only the expensive tiles while the baselines generate all tiles in the range of zoom levels. This is further clarified in Figure 11(b) which shows that the generated index size of AID* up-to three orders of magnitude smaller than baselines and it stabilizes after level 12 since all additional tiles in deeper levels are skipped.



(a) Visualization index generation time (b) Visualization index size

Figure 11: Multilevel visualization performance

Table 2: Summary of related work

Feature	Beast	Sedona[86]	AsterixDB[6]	GeoMesa[26]	BBoxDB[54]
Data access (I=Input, O=Output)					
CSV/WKT	I/O	I/O	I/O	I/O	I/O
Shapefile	I/O	I*	–	I/O	I/O
KML/KMZ	O	–	–	–	–
GeoJSON	I/O	I/O*	I/O*	I/O	I/O
GPX	I	–	–	I	I
GeoTIFF	I/O	–	–	I	–
HDF+	I	I	–	–	–
Generators	✓	–	–	–	–
Summarization					
Sample	✓	✓	✓	✓	–
Histogram	✓	✓	✓	✓	–
Bloom Filter	✓	–	✓†	✓	✓†
Partitioning and indexing					
Partitioning	✓	✓	–	–	✓
Local indexing	✓	✓	✓	✓	✓
Dimensions	Multi	2	3	3	Multi
Join Optimizations					
Filter	✓	✓	✓	✓	✓
Refinement	✓	–	–	–	–
Visualization					
Client-side	✓	✓	✓	✓	✓
Single-level	✓	✓	–	–	–
Multilevel	✓	✓§	–	–	–

* Limitations on the file format apply
+ Hierarchical Data Format
† Bloom filter supported for non-spatial data only
§ Limited scalability with zoom levels

11 RELATED WORK

Table 2 summarizes the related work in big spatial data processing to some of the available systems. To highlight some parts in the table, we mention that Beast is the most comprehensive system in terms of input and output formats. It is the only system that provides a distributed spatial data generator. It is also one of a few systems that provide multidimensional data types and indexing which helps with spatio-temporal data. In addition, it provides both filter and refinement spatial join techniques to efficiently handle big variety data. Finally, it provides a scalable multilevel visualization index that is deployed in a live system (UCR-Star) that hosts hundreds of datasets with terabytes of size.

Interactive exploratory analytics on non-spatial data: There has been a large body of work to support interactive exploration on non-spatial data from industry [31, 56, 61, 80] and academia [3, 4, 12, 41]. Other systems focus on the visual *exploration* aspect [15, 33, 36, 47, 49, 65]. These systems proved to be very powerful and expressed the power of the interactive exploration method but none of them focused on spatio-temporal data, queries, and visualization. They only provide a limited set of geospatial functionality such as rendering the results on a map or perform specific spatial operations such as point-in-polygon queries. Beast provide end-to-end support for spatial data including data loading, partitioning, load balancing, join, and visualization.

Spatial data visualization: First, there is a lot of research on non-map visualization using bar charts, pie charts, and the like [15, 18, 39, 40, 46, 47, 80–82] which are different from the map visualization that Beast supports. Switching to map visualization, existing work focus on either client-side rendering or server-side rendering. Client-side rendering works only on small data, so the server needs to reduce the data size using sampling, aggregation or selection before visualization [15, 31, 38, 64, 65, 81, 84, 84]. Server-side rendering [22, 23, 50, 60, 85, 87] can support large-scale data visualization. However, it consumes a lot of resources on the server to either prerender millions of images or cache data in memory to maintain interactivity of the visualized data. Beast focuses on server-side rendering to support big data, but it uses an adaptive approach that minimizes the resource usage on the server which allows it to scale to thousands of datasets and terabytes of data.

Big spatial data: The work in big spatial data is more than can be covered in detail in this paper [2, 5, 7–10, 20, 35, 51–55, 62, 73, 74, 83, 86]. However, these systems either focus on batch processing [20, 25, 53–55, 73, 83, 86] to handle big *volume* data, or stream processing [1, 8, 14, 16, 51] to handle big *velocity* data. Beast focuses on big *variety* data by providing an end-to-end framework for in-situ data exploration that can work with input files in a variety of formats. It uses data summarization to provide interactive query processing, various scalable spatial join algorithms for data integration, and map-based exploratory interface for exploring the results. We refer interested readers to these recent surveys [21, 58].

12 CONCLUSION

This paper introduces Beast, a Spark-based system for exploratory data science on spatio-temporal data with big variety. Beast is designed with maximum compatibility with Spark core to ease in-situ processing and exploratory queries. Beast extends Spark RDD class to provide SpatialRDD functions, spatial partitioning, and spatial dependencies between RDDs. For in-situ processing Beast supports efficient readers for a wide array of geospatial file formats for both vector and raster data. Additionally, Beast provides a scalable data generator for benchmarking and testing. Beast also adds a partitioning and indexing framework for parallelization and load balancing. In addition, Beast provides an efficient spatial join query processor to combine multiple datasets together. Finally, Beast contains a visual exploratory interface that allows users to visualize big data on a map. Experiments with large scale data show a clear evidence of the scalability and usefulness of Beast. We believe that Beast will be of a great help to researchers and developers in the data science field.

ACKNOWLEDGMENTS

This work is supported in part by NSF under grants IIS-2046236, IIS-1954644, IIS-1838222 and CNS-1924694 and by Agriculture and Food Research Initiative Competitive Grant no. 2019-67022-29696 from NIFA.

REFERENCES

[1] Ahmed S. Abdelhamid, Ahmed R. Mahmood, Anas Daghistani, and Walid G. Aref. 2020. Prompt: Dynamic Data-Partitioning for Distributed Micro-batch Stream Processing Systems. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2455–2469. https://doi.org/10.1145/3318464.3389713

[2] Ahmed S. Abdelhamid, MingJie Tang, Ahmed M. Aly, Ahmed R. Mahmood, Thamir Qadah, Walid G. Aref, and Saleh M. Basalamah. 2016. Cruncher: Distributed in-memory processing for location-based services. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. 1406–1409. https://doi.org/10.1109/ICDE.2016.7498356

[3] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. 1999. Aqua: A Fast Decision Support Systems Using Approximate Query Answers. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. 754–757. http://www.vldb.org/conf/1999/P76.pdf

[4] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*. 29–42. https://doi.org/10.1145/2465351.2465355

[5] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. 2013. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *PVLDB* 6, 11 (2013), 1009–1020. http://www.vldb.org/pvldb/vol6/p1009-aji.pdf

[6] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.* 7, 14 (2014), 1905–1916. https://doi.org/10.14778/2733085.2733096

[7] Ahmed M. Aly, Hazem Elmeleegy, Yan Qi, and Walid G. Aref. 2016. Kangaroo: Workload-Aware Processing of Range Data and Range Queries in Hadoop. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, San Francisco, CA, USA, February 22-25, 2016*. 397–406. https://doi.org/10.1145/2835776.2835841

[8] Ahmed M. Aly, Ahmed R. Mahmood, Mohamed S. Hassan, Walid G. Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. 2015. AQWA: Adaptive Query-Workload-Aware Partitioning of Big Spatial Data. *PVLDB* 8, 13 (2015), 2062–2073. http://www.vldb.org/pvldb/vol8/p2062-Aly.pdf

[9] Furqan Baig, Hoang Vo, Tahsin M. Kurç, Joel H. Saltz, and Fusheng Wang. 2017. SparkGIS: Resource Aware Efficient In-Memory Spatial Query Processing. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2017, Redondo Beach, CA, USA, November 7-10, 2017*. 28:1–28:10. https://doi.org/10.1145/3139958.3140019

[10] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. 1998. The Multidimensional Database System RasDaMan. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. 575–577. https://doi.org/10.1145/276304.276386

[11] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*. Atlantic City, NJ, 322–331.

[12] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2001. Approximate query processing using wavelets. *VLDB J.* 10, 2-3 (2001), 199–223. https://doi.org/10.1007/s007780100049

[13] Harry Chasparis and Ahmed Eldawy. 2017. Experimental Evaluation of Selectivity Estimation on Big Spatial Data. In *Proceedings of the Fourth International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data (GeoRich 2017), collocated with ACM SIGMOD 2017*. Chicago, IL, 8:1–8:6. https://doi.org/10.1145/3080546.3080553

[14] Zhida Chen, Gao Cong, and Walid G. Aref. 2020. STAR: A Distributed Stream Warehouse System for Spatial Data. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2761–2764. https://doi.org/10.1145/3318464.3384699

[15] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2015. Vizdom: Interactive Analytics through Pen and Touch. *PVLDB* 8, 12 (2015), 2024–2027. http://www.vldb.org/pvldb/vol8/p2024-crotty.pdf

[16] Anas Daghistani, Walid G. Aref, Arif Ghafoor, and Ahmed R. Mahmood. 2020. SWARM: Adaptive Load Balancing in Distributed Streaming Systems for Big Spatial Data. *CoRR* abs/2002.11862 (2020). arXiv:2002.11862 https://arxiv.org/abs/2002.11862

[17] Jens-Peter Dittrich and Bernhard Seeger. 2000. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, David B. Lomet and Gerhard Weikum (Eds.). IEEE Computer Society, 535–546. https://doi.org/10.1109/ICDE.2000.839452

[18] Muhammad El-Hindi, Zheguang Zhao, Carsten Binnig, and Tim Kraska. 2016. VisTrees: fast indexes for interactive data exploration. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 5. https://doi.org/10.1145/2939502.2939507

[19] Ahmed Eldawy, Louai Alarabi, and Mohamed F. Mokbel. 2015. Spatial Partitioning Techniques in Spatial Hadoop. *PVLDB* 8, 12 (2015), 1602–1605. https://doi.org/10.14778/2824032.2824057

[20] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *Proceedings of the IEEE International Conference on Data Engineering, IEEE ICDE*. Seoul, South Korea, 1352–1363. https://doi.org/10.1109/ICDE.2015.7113382

[21] Ahmed Eldawy and Mohamed F. Mokbel. 2016. The Era of Big Spatial Data: A Survey. *Foundations and Trends in Databases* 6, 3-4 (2016), 163–273. https://doi.org/10.1561/1900000054

[22] Ahmed Eldawy, Mohamed F. Mokbel, Saif Al-Harthi, Abdulhadi Alzaidy, Kareem Tarek, and Sohaib Ghani. 2015. SHAHED: A MapReduce-based system for querying and visualizing spatio-temporal satellite data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. 1585–1596. https://doi.org/10.1109/ICDE.2015.7113427

[23] Ahmed Eldawy, Mohamed F. Mokbel, and Christopher Jonathan. 2016. HadoopViz: A MapReduce Framework for Extensible Visualization of Big Spatial Data. In *32nd IEEE International Conference on Data Engineering, (ICDE 2016)*. Helsinki, Finland, 601–612. https://doi.org/10.1109/ICDE.2016.7498274

[24] Ahmed Eldawy, Lyuye Niu, David Haynes, and Zhiba Su. 2017. Large Scale Analytics of Vector+Raster Big Spatial Data. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, (SIGSPTAIAL 2017)*. Redondo Beach, CA, 62:1–62:4. https://doi.org/10.1145/3139958.3140042

[25] Ahmed Eldawy, Ibrahim Sabek, Mostafa Elganainy, Ammar Bakeer, Ahmed Abdelmotaleb, and Mohamed F. Mokbel. 2017. Sphinx: Empowering Impala for Efficient Execution of SQL Queries on Big Spatial Data. In *Advances in Spatial and Temporal Databases - 15th International Symposium, SSTD 2017, Arlington, VA, USA, August 21-23, 2017, Proceedings*. 65–83. https://doi.org/10.1007/978-3-319-64367-0_4

[26] Anthony D. Fox, Christopher N. Eichelberger, James N. Hughes, and Skylar Lyon. 2013. Spatio-temporal indexing in non-relational distributed databases. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*. 291–299. https://doi.org/10.1109/BigData.2013.6691586

[27] Saheli Ghosh and Ahmed Eldawy. 2020. AID*: A Spatial Index for Visual Exploration of Geo-Spatial Data. (2020). https://doi.org/10.1109/TKDE.2020.3026657

[28] Saheli Ghosh, Ahmed Eldawy, and Shipra Jais. 2019. AID: An Adaptive Image Data Index for Interactive Multilevel Visualization. In *35th IEEE International Conference on Data Engineering, (ICDE 2019)*. Macau, China, 1594–1597. https://doi.org/10.1109/ICDE.2019.00150

[29] Saheli Ghosh, Akil Sevim, and Ahmed Eldawy. 2020. A Demonstration of Interactive Exploration of Big Geospatial Data on UCR-Star. ACM. https://doi.org/10.1145/3397536.3422334

[30] Saheli Ghosh, Tin Vu, Mehrad Amin Eskandari, and Ahmed Eldawy. 2019. UCR-STAR: The UCR Spatio-Temporal Active Repository. *SIGSPATIAL Special* 11, 2 (Dec. 2019), 34â€“40. https://doi.org/10.1145/3377000.3377005

[31] Hector Gonzalez, Alon Y. Halevy, Christian S. Jensen, Anno Langen, Jayant Madhavan, Rebecca Shapley, and Warren Shen. 2010. Google fusion tables: data management, integration and collaboration in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*. 175–180. https://doi.org/10.1145/1807128.1807158

[32] Noel Gorelick et al. 2017. Google Earth Engine: Planetary-scale geospatial analysis for everyone. *Remote sensing of Environment* 202 (2017), 18–27.

[33] Jeffrey Heer, Stuart K. Card, and James A. Landay. 2005. prefuse: a toolkit for interactive information visualization. In *Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI 2005, Portland, Oregon, USA, April 2-7, 2005*, Gerrit C. van der Veer and Carolyn Gale (Eds.). ACM, 421–430. https://doi.org/10.1145/1054972.1055031

[34] Nicolaus Henke, Jacques Bughin, Michael Chui, James Manyika, Tamim Saleh, Bill Wiseman, and Guru Sethupathy. 2016. *The Age of Analytics: Competing in a Data-driven World*. Technical Report. McKinsey Global Institute.

[35] Fei Hu, Chaowei Yang, John L. Schnase, Daniel Q. Duffy, Mengchao Xu, Michael K. Bowen, Tsengdar Lee, and Weiwei Song. 2018. ClimateSpark: An in-memory distributed computing framework for big climate data analytics. *Computers & Geosciences* 115 (2018), 154–166. https://doi.org/10.1016/j.cageo.2018.03.011

[36] Kevin Zeng Hu et al. 2019. VizML: A Machine Learning Approach to Visualization Recommendation. In *Processing of the ACM International Conference on Human*

Factors in Computing Systems, CHI, Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos (Eds.). ACM, Glashow, Scotland, UK, 128.
[37] Edwin H. Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Transactions on Database Systems* 32, 1 (2007), 7. https://doi.org/10.1145/1206049.1206056
[38] Jianfeng Jia, Chen Li, Xi Zhang, Chen Li, Michael J. Carey, and Simon Su. 2016. Towards interactive analytics and visualization on one billion tweets. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*. 85:1–85:4. https://doi.org/10.1145/2996913.2996923
[39] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. 2014. M4: A Visualization-Oriented Time Series Data Aggregation. *PVLDB* 7, 10 (2014), 797–808. http://www.vldb.org/pvldb/vol7/p797-jugel.pdf
[40] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. 2016. VDDA: automatic visualization-driven data aggregation in relational databases. *VLDB J.* 25, 1 (2016), 53–77. https://doi.org/10.1007/s00778-015-0396-z
[41] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 631–646. https://doi.org/10.1145/2882903.2882940
[42] Puloma Katiyar, Tin Vu, Sara Migliorini, Alberto Belussi, and Ahmed Eldawy. 2020. SpiderWeb: A Spatial Data Generator on the Web. In *28th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL 2020)* (Seattle, Washington, USA). ACM. https://doi.org/10.1145/3397536.3422351
[43] Ameet Kini and Rob Emanuele. 2014. Geotrellis: Adding Geospatial Capabilities to Spark.
[44] Scott T Leutenegger et al. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the IEEE International Conference on Data Engineering, IEEE ICDE*. IEEE, 497–506.
[45] Lauro Lins, James T. Klosowski, and Carlos Scheidegger. 2013. Nanocubes for Real-Time Explorations of Spatiotemporal Datasets. In *Proceedings of the IEEE Transactions on Visualization and Computer Graphics, TVCG*.
[46] Zhicheng Liu and Jeffrey Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Trans. Vis. Comput. Graph.* 20, 12 (2014), 2122–2131. https://doi.org/10.1109/TVCG.2014.2346452
[47] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. 2013. *imMens*: Real-time Visual Querying of Big Data. *Comput. Graph. Forum* 32, 3 (2013), 421–430. https://doi.org/10.1111/cgf.12129
[48] Ming-Ling Lo and Chinya V. Ravishankar. 1994. Spatial Joins Using Seeded Trees. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, Richard T. Snodgrass and Marianne Winslett (Eds.). ACM Press, 209–220. https://doi.org/10.1145/191839.191881
[49] Yuyu Luo, Chengliang Chai, Xuedi Qin, Nan Tang, and Guoliang Li. 2020. Interactive Cleaning for Progressive Visualization through Composite Questions. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 733–744. https://doi.org/10.1109/ICDE48307.2020.00069
[50] Mengyu Ma, Anran Yang, Ye Wu, Luo Chen, Jun Li, and Ning Jing. 2020. DiSA: A Display-driven Spatial Analysis Framework for Large-Scale Vector Data. In *SIGSPATIAL '20: 28th International Conference on Advances in Geographic Information Systems, Seattle, WA, USA, November 3-6, 2020*, Chang-Tien Lu, Fusheng Wang, Goce Trajcevski, Yan Huang, Shawn D. Newsam, and Li Xiong (Eds.). ACM, 147–150.
[51] Ahmed R. Mahmood, Ahmed M. Aly, Thamir Qadah, El Kindi Rezig, Anas Daghistani, Amgad Madkour, Ahmed S. Abdelhamid, Mohamed S. Hassan, Walid G. Aref, and Saleh M. Basalamah. 2015. Tornado: A Distributed Spatio-Textual Stream Processing System. *PVLDB* 8, 12 (2015), 2020–2023. https://doi.org/10.14778/2824032.2824126
[52] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. 2015. Distributed SECONDO: A Highly Available and Scalable System for Spatial Data Processing. In *Advances in Spatial and Temporal Databases - 14th International Symposium, SSTD 2015, Hong Kong, China, August 26-28, 2015. Proceedings*. 491–496. https://doi.org/10.1007/978-3-319-22363-6_28
[53] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. 2017. Distributed secondo: an extensible and scalable database management system. *Distributed and Parallel Databases* 35, 3-4 (2017), 197–248. https://doi.org/10.1007/s10619-017-7198-9
[54] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. 2018. BBoxDB - A Scalable Data Store for Multi-Dimensional Big Data. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*. 1867–1870. https://doi.org/10.1145/3269206.3269208
[55] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. $\mathcal{MD}$ -HBase: design and implementation of an elastic data infrastructure for cloud-scale location services. *Distributed and Parallel Databases* 31, 2 (2013), 289–319. https://doi.org/10.1007/s10619-012-7109-z
[56] omnisci [n.d.]. OmniSci. https://www.omnisci.com/.
[57] osmtiles 2019. OpenStreetMap disk usage. https://wiki.openstreetmap.org/wiki/Tile_disk_usage.
[58] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How Good Are Modern Spatial Analytics Systems? *Proc. VLDB Endow.* 11, 11 (2018), 1661–1673. https://doi.org/10.14778/3236187.3236213
[59] Jignesh M. Patel and David J. DeWitt. 1996. Partition Based Spatial-Merge Join. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 259–270. https://doi.org/10.1145/233269.233338
[60] Gary Planthaber, Michael Stonebraker, and James Frew. 2012. EarthDB: scalable analysis of MODIS data using SciDB. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2012, Redondo Beach, CA, USA, November 6, 2012*. 11–19. https://doi.org/10.1145/2447481.2447483
[61] qlik [n.d.]. Qlik: Business Intelligence. https://www.qlik.com/.
[62] Ramon Antonio Rodriges Zalipynis. 2018. ChronosDB: Distributed, File Based, Geospatial Array DBMS. *PVLDB* 11, 10 (2018), 1247–1261. http://www.vldb.org/pvldb/vol11/p1247-zalipynis.pdf
[63] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *Comput. Surveys* 16, 2 (1984), 187–260.
[64] Arvind Satyanarayan and Jeffrey Heer. 2014. Lyra: An Interactive Visualization Design Environment. *Comput. Graph. Forum* 33, 3 (2014), 351–360. https://doi.org/10.1111/cgf.12391
[65] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Vis. Comput. Graph.* 23, 1 (2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030
[66] A. B. Siddique and Ahmed Eldawy. 2018. Experimental Evaluation of Sketching Techniques for Big Spatial Data. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 2018), Carlsbad, CA, USA, October 11-13*. Carlsbad, CA, 522. https://doi.org/10.1145/3267809.3275464
[67] Abu Bakar Siddique, Ahmed Eldawy, and Vagelis Hristidis. 2019. Comparing Synopsis Techniques for Approximate Spatial Data Analysis. *PVLDB* 12, 11 (2019), 1583 – 1596. https://doi.org/10.14778/3342263.3342635
[68] A. B. Siddique, Ahmed Eldawy, and Vagelis Hristidis. 2019. Euler++: Improved Selectivity Estimation for Rectangular Spatial Records. In *4th IEEE International Workshop on Big Spatial Data (BSD 2019)* (Los Angeles, CA, USA). IEEE. https://doi.org/10.1109/BigData47090.2019.9006498
[69] Samriddhi Singla and Ahmed Eldawy. 2018. Distributed zonal statistics of big raster and vector data. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, (SIGSPATIAL 2018)*. Seattle, WA, 536–539. https://doi.org/10.1145/3274895.3274985
[70] Samriddhi Singla and Ahmed Eldawy. 2020. Flexible Computation of Multidimensional Histograms. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Spatial Gems (SpatialGems 2020)* (Seattle, Washington, USA). ACM.
[71] Samriddhi Singla and Ahmed Eldawy. 2020. Raptor Zonal Statistics : Fully Distributed Zonal Statistics of Big Raster + Vector Data. In *Proceedings of the 2020 IEEE International Conference on Big Data (IEEE BigData 2020)* (Atlanta, Georgia, USA). IEEE.
[72] Samriddhi Singla, Ahmed Eldawy, Rami Alghamdi, and Mohamed F. Mokbel. 2019. Raptor: Large Scale Analysis of Big Raster and Vector Data. *PVLDB* 12, 12 (2019), 1950 – 1953. https://doi.org/10.14778/3352063.3352107
[73] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. *Proc. VLDB Endow.* 9, 13 (2016), 1565–1568. https://doi.org/10.14778/3007263.3007310
[74] Hoang Vo, Yanhui Liang, Jun Kong, and Fusheng Wang. 2018. iSPEED: a Scalable and Distributed In-Memory Based Spatial Query System for Large and Structurally Complex 3D Data. *PVLDB* 11, 12 (2018), 2078–2081. http://www.vldb.org/pvldb/vol11/p2078-vo.pdf
[75] Tin Vu, Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. 2020. Using Deep Learning for Big Spatial Data Partitioning. 3 (2020). https://doi.org/10.1145/3402126
[76] Tin Vu and Ahmed Eldawy. 2018. R-Grove: growing a family of R-trees in the big-data forest. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, (SIGSPATIAL 2018)*. Seattle, WA, 532–535. https://doi.org/10.1145/3274895.3274984
[77] Tin Vu and Ahmed Eldawy. 2020. DeepSampling: Selectivity Estimation with Predicted Error and Response Time. In *DeepSpatial 2020: ACM SIGKDD Workshop on Deep Learning for Spatiotemporal Data, Applications, and Systems* (San Diego, CA, USA). ACM.
[78] Tin Vu and Ahmed Eldawy. 2020. R*-Grove: Balanced Spatial Partitioning for Large-Scale Datasets. (Aug. 2020). https://doi.org/10.3389/fdata.2020.00028
[79] Tin Vu, Sara Migliorini, Ahmed Eldawy, and Alberto Bulussi. 2019. Spatial Data Generators. In *1st ACM SIGSPATIAL International Workshop on Spatial Gems (SpatialGems 2019)* (Chicago, IL, USA). ACM.
[80] Richard Michael Grantham Wesley, Matthew Eldridge, and Pawel Terlecki. 2011. An analytic data engine for visualization in tableau. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens,*

Greece, June 12-16, 2011. 1185–1194. https://doi.org/10.1145/1989323.1989449

[81] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock D. Mackinlay, Bill Howe, and Jeffrey Heer. 2016. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Trans. Vis. Comput. Graph.* 22, 1 (2016), 649–658. https://doi.org/10.1109/TVCG.2015.2467191

[82] Eugene Wu, Leilani Battle, and Samuel R. Madden. 2014. The Case for Data Visualization Management Systems. *PVLDB* 7, 10 (2014), 903–906. http://www.vldb.org/pvldb/vol7/p903-wu.pdf

[83] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1071–1085. https://doi.org/10.1145/2882903.2915237

[84] Jia Yu and Mohamed Sarwat. 2020. Turbocharging Geospatial Visualization Dashboards via a Materialized Sampling Cube Approach. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1165–1176. https://doi.org/10.1109/ICDE48307.2020.00105

[85] Jia Yu, Anique Tahir, and Mohamed Sarwat. 2019. GeoSparkViz in Action: A Data System with Built-in Support for Geospatial Visualization. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 1992–1995. https://doi.org/10.1109/ICDE.2019.00222

[86] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. 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, Bellevue, WA, USA, November 3-6, 2015*, Jie Bao, Christian Sengstock, Mohammed Eunus Ali, Yan Huang, Michael Gertz, Matthias Renz, and Jagan Sankaranarayanan (Eds.). ACM, 70:1–70:4. https://doi.org/10.1145/2820783.2820860

[87] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. 2018. GeoSparkViz: a scalable geospatial data visualization framework in the Apache spark ecosystem. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management, SSDBM 2018, Bozen-Bolzano, Italy, July 09-11, 2018*, Dimitris Sacharidis, Johann Gamper, and Michael H. Böhlen (Eds.). ACM, 15:1–15:12. https://doi.org/10.1145/3221269.3223040

[88] Yaming Zhang and Ahmed Eldawy. 2020. Evaluating Computational Geometry Libraries for Big Spatial Data Exploration. In *Proceedings of the Sixth International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data (GeoRich 2020), collocated with ACM SIGMOD 2020* (Portland, OR, USA). ACM. https://doi.org/10.1145/3403896.3403969