The Era of Big Spatial Data

Ahmed Eldawy Mohamed F. Mokbel
Computer Science and Engineering Department
University of Minnesota, Minneapolis, Minnesota 55455
Email: {eldawy,mokbel}@cs.umn.edu

***Abstract*—The recent explosion in the amount of spatial data calls for specialized systems to handle big spatial data. In this paper, we discuss the main features and components that needs to be supported in a system to handle big spatial data efficiently. We review the recent work in the area of big spatial data according to these four components, namely, *language*, *indexing*, *query processing*, and *visualization*. We describe each component, in details, and give examples of how it is implemented in existing work. After that, we describe a few case studies of systems for big spatial data and show how they support these four components. This assists researchers in understanding the different design approaches and highlights the open research problems in this area. Finally, we give examples of real applications that make use of these systems to handle big spatial data.**

I. Introduction

In recent years, there has been an explosion in the amounts of spatial data produced by several devices such as smart phones, space telescopes, medical devices, among others. For example, space telescopes generate up to 150 GB weekly spatial data [1], medical devices produce spatial images (X-rays) at a rate of 50 PB per year [2], a NASA archive of satellite earth images has more than 500 TB and is increased daily by 25 GB [3], while there are 10 Million geotagged tweets issued from Twitter every day as 2% of the whole Twitter firehose [4], [5]. Meanwhile, various applications and agencies need to process an unprecedented amount of spatial data. For example, the Blue Brain Project [6] studies the brain's architectural and functional principles through modeling brain neurons as spatial data. Epidemiologists use spatial analysis techniques to identify cancer clusters [7], track infectious disease [8], and drug addiction [9]. Meteorologists study and simulate climate data through spatial analysis [10]. News reporters use geotagged tweets for event detection and analysis [11].

Unfortunately, the urgent need to manage and analyze big spatial data is hampered by the lack of specialized systems, techniques, and algorithms to support such data. For example, while big data is well supported with a variety of Map-Reduce-like systems and cloud infrastructure (e.g., Hadoop [12], Hive [13], HBase [14], Impala [15], Dremel [16], Vertica [17], and Spark [18]), none of these systems or infrastructure provide any special support for spatial or spatio-temporal data. In fact, the only way to support big spatial data is to either treat it as non-spatial data or to write a set of functions as wrappers around existing non-spatial systems. However, doing so does not take any advantage of the properties of spatial and spatio-temporal data, hence resulting in sub-par performance.

The importance of big spatial data, which is ill-supported in the systems mentioned above, motivated many researchers to extend these systems to provide distributed systems for big spatial data. These extended systems natively support spatial data which makes them very efficient when handling spatial data. This includes (1) MapReduce systems such as Hadoop-GIS [19], ESRI Tools for Hadoop [20], [21], and SpatialHadoop [22]; (2) Parallel DB systems such as Parallel Secondo [23]; (3) Systems built on key-value stores such as $\mathcal{MD}$-HBase [24] and GeoMesa [25]; and (4) Systems that use resilient distributed datasets (RDD) such as SpatialSpark [26] and GeoTrellis [27].

In this paper, we describe the general design of a system for big spatial data. This design in inspired by the existing systems and as it covers the functionality provided by these systems. Our goal is to allow system designers to understand the different aspects of big spatial data in order to help them in two directions. First, it is helpful to researchers pursuing research in existing systems to decide the possible directions of advancing their research. To better support them, we provide examples of how each feature is implemented in different systems, which makes it easy to choose the most suitable approach depending on system architecture. Second, this paper helps researchers who are planning to build a new system for big spatial data, to figure out the components that needs to be there to efficiently support big spatial data. As more distributed systems for big non-spatial data are emerging, we expect that there will be more work to extend those systems to support spatial data. In this case, this paper serves as a guideline of which components should be implemented in the system.

The system design we propose in this paper contains four main components, namely, *language*, *indexing*, *query processing*, and *visualization*. The *language* is a simple high level language that allows non-technical users to interact with the system. The *indexing* component adapts existing spatial indexes, such as R-tree and Quad tree, to the distributed environment so that spatial queries would run more efficiently. The *query processing* component contains a set of spatial queries that are implemented efficiently in the system. Finally, the *visualization* component allows data to be displayed as an image which makes it easy for users to explore and understand big datasets. Based on these four components, we study the available systems for big spatial data and discuss how they are designed based on these four components. We also discuss some applications that can make use of big spatial data systems and how the corresponding systems are designed to meet the requirements of this application.

The rest of this paper is organized as follows. Section II gives an overview of the system design. The four components are described in Sections III-VI. Sections VII and VIII provide use cases of *systems* and *applications* for big spatial data, respectively. Finally, Section IX concludes the paper.

II. Overview

In this section we provide an overview of the proposed system design for big spatial data. The system consists of four components, namely, *language*, *indexing*, *query processing*, and *visualization*, as described briefly below.

The language component hides all complexities of the system by providing a simple high level language which allows non-technical users to access system functionality without worrying about its implementation details. This language should contain basic spatial support including standard data types and functions. Section III contains more details about the language.

The indexing component is responsible of storing datasets using standard spatial indexes in the distributed storage. One way to adapt existing indexes to distributed environments is through a two-level structure of *global* and *local* indexes, where the global index partitions data across machines, while the local indexes organize records in each partition. In addition to storing indexes on disk, there should be additional components which allow the query processing engine to access these indexes while processing a spatial query. Details of spatial indexing is given in Section IV.

The query processing component encapsulates the spatial operations that are implemented in the system using the constructed spatial indexes. This component also needs to be extensible to allow developers to implement new spatial operations that also access the spatial indexes. Details of the query processing in discussed in Section V.

The visualization component allows users to display datasets stored in the system by generating an image out of them. The graphical representation of spatial data is more common for end users as it allows them to explore and visualize new datasets. The generated image should be of a very high quality (i.e., high resolution) to allow users to zoom into a specific region and see more details. This is particularly important for big spatial data because a small image of a low resolution would not contain enough details to describe such a big dataset. Details of the visualization is described in Section VI.

III. Language

As most users of systems for big spatial data are not from computer science, it is urgent for these systems to provide an easy-to-use high level language which hides all the complexities of the system. Most systems for big non-spatial data are already equipped with a high level language, such as, Pig Latin [28] for Hadoop, HiveQL for Hive [13], and Scala-based language for Spark [18]. It makes more sense to reuse these existing languages rather than proposing a completely new language for two reasons. First, it makes it easier to adopt by existing users of these systems as they do not need to learn a totally new language. Second, it makes it possible to process data that has both spatial and non-spatial attributes through the same program because the introduction of spatial constructs should not disable any of its existing features of the language.

Extending a language to support spatial data incorporates the introduction of *spatial data types* and *spatial operations*. The Open Geospatial Constortium (OGC) [29] defines standards for spatial data types and spatial operations to be supported by this kind of systems. Since these standards are already adopted by existing spatial databases including PostGIS [30] and Oracle Spatial [31], it is recommended to follow these standards in new systems to make it easier for users to adopt. It also makes it possible to integrate with these existing systems by exporting/importing data in OGC-standard formats such as Well-Known Text (WKT). OGC standards are already adopted in three languages for big spatial data, Pigeon [32] which extends Pig Latin, ESRI Tools for Hadoop [20] which extends HiveQL, and the contextual query language (CQL) used in GeoMesa [25].

IV. Indexing

Input files in traditional systems are not spatially organized which means that spatial attributes are not taken into consideration to decide where to store each record. While this is acceptable for traditional applications for non-spatial data, it results in sub-performance for spatial applications. There is already a large number of index structures designed to speed up spatial query processing (e.g., R-tree [33], Grid File [34], and Quad Tree [35]). However, migrating these indexes to other systems for big data is challenging given the different architectures used in each one. In this section, we first give an example of how spatial indexes are implemented in the Hadoop MapReduce environment and then show how to generalize the described method to support other indexes in other environments. After that, we show how these indexes are made available to the query processing engine.

A. Spatial Indexing in Hadoop

In the Hadoop MapReduce environment, there are two main limitations which make it challenging to adopt a traditional index such as an R-tree. (1) These data structures are designed for *procedural* programming where a program runs as sequential statements while Hadoop employs a *functional* MapReduce programming where the program consists of a *map* and *reduce* functions and Hadoop controls how these two functions are executed. (2) A file in Hadoop Distributed File System can be written in append only manner and cannot be modified which is very limiting compared to traditional file systems where files can be modified.

In [19], [21], [22], these two challenges are overcome in Hadoop by employing a two-layer index structure consisting of one global index and multiple local indexes. In this approach, the input file is first partitioned across machines according to a global index, and then each partition is independently indexed using a local index. (1) This approach lends itself to the MapReduce programming paradigm where partitions can be processed in parallel using a MapReduce job. (2) It overcomes the limitations of HDFS where the small size of each partition allows it to be bulk loaded in memory and then written to HDFS in a sequential manner. Figure 1 shows an example of an R-tree index where each rectangle represents a partition in the file stored as one HDFS block (64MB). To keep the size of each partition within 64MB, dense areas (e.g., Europe) contain smaller rectangles while sparse areas (e.g., oceans) contain larger rectangles. When a range query, for example, runs on that index, it can achieve orders of magnitude better performance by quickly pruning partitions that are outside query range as they do not contribute to the answer.

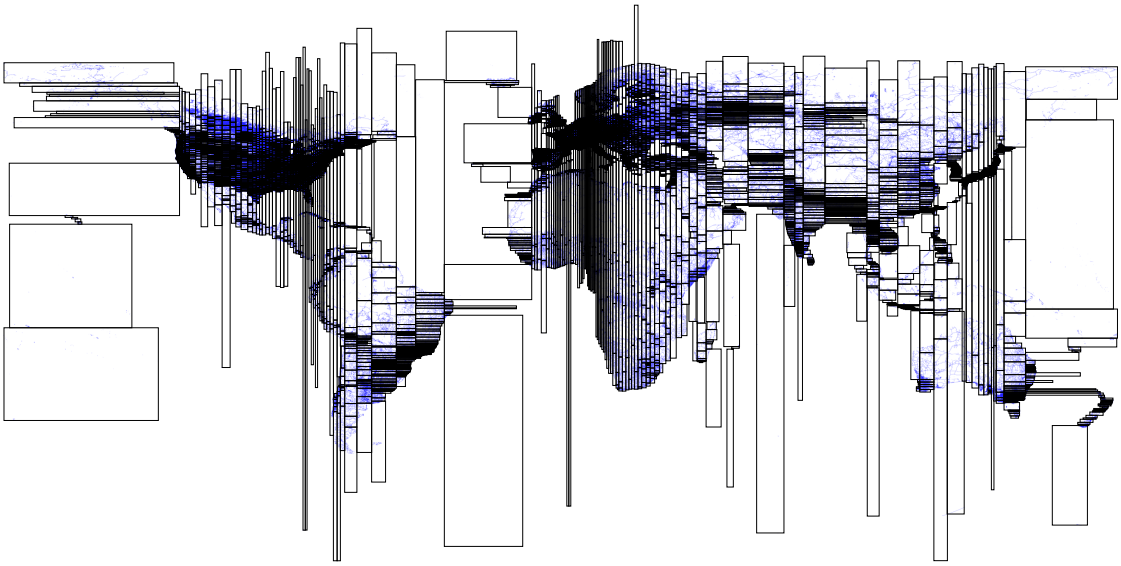


Fig. 1. R-tree partitioning of a 400GB road network data

B. Other Spatial Indexes

Most systems for big spatial data share the same two limitations of Hadoop. For example, Hive, Spark, and Impala, all use HDFS as a storing engine. Also, Spark employs RDD as a another functional programming paradigm. Thus, the two-layer design described above can be employed in other systems. However, there are three design decisions that should be considered when designing a distributed spatial..

1. **Global/Local index types:** An index might have a global index only, local indexes only, or both, and the types of global and local indexes do not have to be the same. For example, a uniform grid index is constructed using only a global index of a uniform grid [19], [22], an R-tree distributed index uses R-tree for both global and local indexes [22], a PMR Quad tree uses Z-order partitioning in the global index and Quad tree in local indexes [21], and a K-d tree-based index uses only a global index of K-d tree [24], [36].

2. **Static or Dynamic:** A static index is constructed once for a dataset and records cannot be inserted nor deleted from it. On the other hand, a dynamic index can accommodate both insertions and deletions. A static index is useful for data that does not change frequently such as archival data. It also matches the architecture of HDFS where a file, once uploaded, cannot be modified. This makes it reasonable to use when constructing various indexes in Hadoop [19], [21], [22], [37]. A dynamic index is more suitable for highly dynamic data that is rapidly changing such as moving objects. In this case, it has to be stored on a storage engine that supports updates. For example, a dynamic Quad tree and K-d tree indexes are constructed in HBase [38] and a geohash index is constructed in Accumulo [25].

3. **Primary or Secondary:** In a primary index, the actual value of records are stored in the index. In a secondary index, the values of records are stored separately from the index while the index contains only references or pointers to the records. The primary index is more suitable for a distributed file system such as HDFS as it avoids the poor performance of random access. However, as its name indicates, one file can accommodate only one primary index and possibly multiple secondary indexes. Therefore, secondary indexes could be inevitable if multiple indexes are desired. Most index structures implemented for big spatial data are primary indexes including grid index [19], [22], R-tree [22], [37], R+-tree [22], and PMR Quad tree [21]. A secondary PMR Quad tree is also implemented and shown to be much worse in performance than a primary index [21].

C. Access Methods

Organizing data in the file system is just the first part of indexing, the second part, which completes the design, is adding new components which allow the indexes to be used in query processing. Without these components, the query processing layer will not be able to use these indexes and will end up scanning the whole file as if there were no index constructed. Back to our example with Hadoop, the index is made accessible to MapReduce programs through two components, namely, SpatialFileSplitter and SpatialRecordReader. The SpatialFileSplitter accesses the global index with a user-defined filter function to prune file partitions that do not contribute to answer (e.g., outside the user-specified query range). The SpatialRecordReader is used to process non-pruned partitions efficiently by using the local index stored in each one. These two components allow MapReduce programs to access the constructed index in its two levels, global and local. To make these indexes accessible in other systems, e.g., Spark, different components need to be introduced to allow RDD programs to access the constructed index.

This separation between the index structure in the file system and the access methods used in query processing provides the flexibility to reuse indexes. For example, all of Hadoop, Hive, Spark, and Impala can read their input from raw files in HDFS. This means that one index appropriately stored in HDFS can be accessed by all these systems if the correct access methods are implemented. This also means that we can, for example, construct the index using a Hadoop MapReduce job, and query that index from Hive using HiveQL.

V. Query Processing

A main part of any system for big spatial data is the query processing engine. Different systems would probably use different processing engines such as MapReduce for Hadoop and Resilient Distributed Dataset (RDD) for Spark. While each application requires a different set of operations, the system cannot ship with all possible spatial queries. Therefore, the query processing engine should be extensible allowing users to express custom operations while making use of the spatial indexes. To give some concrete examples, we will describe three categories of operations, namely, basic query operations, join operations, and fundamental computational geometry operations.

A. Basic Query Operations

To give example of basic query operations, we will describe range and k-nearest neighbor queries, and we will use the MapReduce environment as an example. The challenge in these two queries is that the input file in Hadoop Distributed File System (HDFS) is traditionally a heap file which requires a full table scan to answer any query. With the input file spatially indexed, we should rewrite these queries to avoid accessing irrelevant data to improve the performance [22].

Range Queries. In a range query, the input is a set of records R a rectangular query range A while the output is the set of all records in R overlapping A. In traditional Hadoop, all records in the input are scanned in parallel using multiple machines, each record is compared to A, and only matching records are produced in the output [39]. With the two-level spatial index discussed in Section IV, a more efficient algorithm is proposed which runs in three phases [19], [21], [22]. (1) The *global index* is first accessed with a range query to select the *partitions* which overlap A and prune the ones that are disjoint with A as they do not contribute to the final answer. (2) The partitions selected by the first step are processed in parallel where each machine processes a *local index* in one partition with a range query to find all *records* that overlap A. (3) Since the indexed file might have some replication according to the index type, a final *duplicate avoidance* step might be necessary to ensure each matching record is reported only once.

k-nearest neighbor (kNN). The kNN query takes a set of points P, a query point Q, and an integer k as input while the output is the k closest points in P to Q. A traditional implementation without a index would scan the input set P, calculate the distance of each point $p \in P$ to Q, sort the points based on the calculated distance, and choose the top-k points [39]. With the spatial index, a more efficient implementation is provided which runs runs in two rounds [22]. In the first round, the global index is accessed to retrieve only the partition that contains the query point Q. The local index in the matching partition is used to answer the kNN query using a traditional single machine algorithm. The answer is tested for correctness by drawing a *test circle* centered at Q with a radius equal to the distance to the k^{th} farthest neighbor. If the circle overlaps only one partition, the answer is correct as all other partitions cannot contain any points that are closer to points in the answer. If the circle overlaps more partitions, a second round executes a range query to retrieve all points in the test circle and chooses the closest k points to Q.

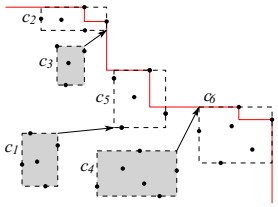


Fig. 3. Pruning in skyline

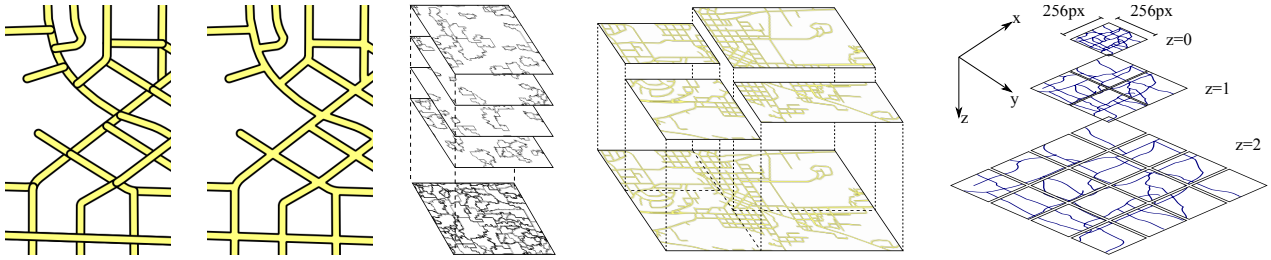
B. Join Operations

In spatial join, the input is two sets R and S and a spatial join predicate θ (e.g., touches, overlaps or contains), and the output is the set of all pairs $\langle r, s\rangle$ where $r \in R$, $s \in S$ and the join predicate θ is true for $\langle r, s\rangle$. If the input files are not indexed, SJMR [40] is used as the MapReduce implementation of the Partition Based Spatial-Merge join (PBSM) [41]. In this technique, the two files are co-partitioned using a uniform grid and the contents of each grid cell are joined independently. If the input files are spatially indexed, we can provide a more efficient algorithm which runs in three phases [22]. (1) The *global join* step joins the two global indexes to find each pair of overlapping partitions. (2) The *local join* step processes each pair of overlapping partitions by running a spatial join operation on their two local indexes to find overlapping records. (3) The *duplicate avoidance* step is finally executed if the indexes contain replicated records to ensure that each overlapping pair is reported only once.

C. Computational Geometry Operations

The area of computational geometry is rich with operations that are used extensively when processing spatial data, such as, polygon union, skyline and convex hull. Traditional CG algorithms rely on a single machine which makes them unscalable to work with big spatial data. A spatial index constructed in a distributed environment provide a room for improvement if the algorithms are redesigned to make use of them. In the following part, we give two examples of operations that are implemented efficiently in a MapReduce environment, namely, skyline, and convex hull [42].

Skyline. In the skyline operation, the input is a set of points P and the output is the set of *non-dominated* points. A point p dominates a point q if p is greater than q in all dimensions. There exist a divide and conquer algorithm for skyline which can be ported to MapReduce but it would require to scan the whole file. This algorithm is improved in [42] by applying a pruning step based on the global index to avoid processing partitions that do not contribute to answer. A partition c_i is pruned if *all* points in this partition are dominated by at least one point in another partition c_j, in which case we say that c_j dominates c_i. For example in Figure 3, c_1 is dominated by c_5 because the top-right corner of c_1 (i.e., best point) is dominated by the bottom-left corner of c_5 (i.e., worst point). The transitivity of the skyline domination rule implies that *any* point in c_5 dominates *all* points in c_1. In addition, the partition c_4 is dominated by c_6 because the top-right corner of c_4 is dominated by the top-left corner of c_6 which means that any point along the top edge of c_6 dominates all points in c_4. Since the boundaries of each partition are tight, there has to be at least one point along each edge.



(a) Not smoothed (b) Smoothed (c) Overlay (d) Stitch (e) Multilevel Image

Fig. 2. Visualization

Convex Hull. In the convex hull operation, the input is a set of points P, and the output is the points that form the minimal convex polygon that contains all points in P. To apply the pruning step in convex hull, we utilize a property which states that a point on the convex hull must be part of one of the four skylines (min-min, min-max, max-min, and max-max). Therefore, we apply the skyline pruning technique four times for the four skylines, and prune partitions that do not contribute to any of the four skylines. Apparently, if a partition does not contribute to any of the four skylines, it cannot contribute to the convex hull.

VI. Visualization

The visualization process involves creating an image that describes an input dataset. This is a natural way to explore spatial datasets as it allows users to spot interesting patterns in the input. Traditional visualization techniques rely on a single machine to load and process the data which makes them unable to handle big spatial data. For example, visualizing a dataset of 1.7 Billion points takes around one hour on a single commodity machine. GPUs are used to speed up the processing but they are still limited to the memory and processing capacity of a single machine and cannot scale out to multiple machines. We can categorize the visualization techniques based on the structure of the generated image into two categories, *single level images* and *multilevel images*. In single level images, the produced image consists of one file that contains an image of a specified resolution. In multilevel images, the generated image consists of a set of *image tiles* at different zoom levels which allows users to zoom in to see more details.

A. Single Level Image Visualization

In single level image visualization, the input dataset is visualized as a single image of a user-specified image size ($width \times height$) in pixels. To generate a single image in a MapReduce environment, we propose an algorithm that runs in three phases, *partitioning*, *rasterize*, and *merging*. (1) The *partitioning* phase partitions an input file using either the default Hadoop partitioner, which does not take spatial attributes into account, or using a spatial partitioner which groups nearby records together in one partition. (2) The *rasterize* phase starts by applying an optional *smoothing* function which fuses nearby records together to produce a cleaner image. Figures 2(a) and 2(b) give an example of visualizing a road network *without* and *with* smoothing, respectively, where intersecting road segments are fused together to provide a more realistic look. Notice that a smoothing function can be applied only if a spatial partitioning is used. After that, the rasterize phase creates a partial image and visualizes all records in that partition on that partial image. (3) The *merging* phase collects all partial images together to produce one final picture which is written to the output as a single image. This phase either *overlays* partial images (Figure 2(c)) or *stitches* them together (Figure 2(d)) depending on whether we use the default Hadoop partitioner or spatial partitioner. The final image is then written to disk as a single image.

B. Multilevel Image Visualization

The quality of a single level image is limited by its resolution which means users cannot zoom in to see more details. On the other hand, multilevel images provide multiple zoom levels which allows users to zoom in and see more details in a specific region. Figure 2(e) gives an example of a multilevel image of three zoom levels 0, 1, and 2, where each level contains 1, 4, and 16 image tiles, respectively. Each tile is a single image of a fixed resolution 256×256. Most modern web maps (e.g., Google Maps and Bing Maps) use this technique where all image tiles are generated in an offline phase while the web interface provides a convenient way to view the generated image by allowing the user to zoom in/out and pan through the image. The goal of the multilevel image visualization algorithm is to generate all these image tiles efficiently for an input dataset.

The input to this algorithm is a dataset and a range of zoom levels $[z_{min}, z_{max}]$ and the output is all image tiles in the specified range of levels. A naïve approach is to use the single level image algorithm to generate each tile independently but this approach is infeasible due to the excessive number of MapReduce jobs to run. For example, at zoom level 10, there will be more than one million images which would require running one million MapReduce jobs. Alternatively, we can provide an algorithm that understands the structure of the multilevel image and is able to generate all image tiles in one MapReduce job. To generate a multilevel image, we can provide an algorithm that runs in two phases, *partition* and *rasterize*. The partition phase scans all input records and replicates each record r to all overlapping tiles in the image according to the MBR of r and the MBR of each tile. This phase produces one partition per tile in the desired image. The *rasterize* phase processes all generated partitions and generates a single image out of each partition. Since the size of each image tile is small, a single machine can generate that tile efficiently. This technique is used in [45] to produce temperature heat maps for NASA satellite data.

System	Architecture	Data types	Language	Indexes	Queries	Visualization
SpatialHadoop [22]	MapReduce	Vector	Pigeon* [32]	Grid, R-tree, R+-tree	RQ, KNN, SJ, CG	Single level, Multilevel
Hadoop GIS [19]	MapReduce	Vector	HiveQL	Grid	RQ, KNN, SJ	-
ESRI Tools for Hadoop [20], [21]	MapReduce	Vector	HiveQL*	PMR Quad Tree	RQ, KNN	-
$\mathcal{MD}$-HBase [24]	Key-value store	Point-only	-	Quad Tree, K-d tree	RQ, KNN	-
GeoMesa [25]	Key-value store	Vector	CQL*	Geohash	RQ	Through GeoServer
Parallel Secondo [23]	Parallel DB	Vector	SQL-Like	Local only	RQ, SJ	-
SciDB [36], [43]	Array DB	Point, Raster	AQL	K-d tree	RQ, KNN	Single level
SpatialSpark [26]	RDD	Vector	Scala-based	On-the-fly	SJ	-
GeoTrellis [27], [44]	RDD	Raster	Scala-based	-	Map Algebra	Single level

∗ OGC-compliant

TABLE I. CASE STUDIES OF SYSTEMS

VII. CASE STUDIES: SYSTEMS

In this section, we provide a few case studies of systems for big spatial data. It is important to mention that this does not serve as a comprehensive survey of all systems nor it provides a detailed comparison of them. Rather, this section gives a few examples of how a few systems for big spatial data cover the four aspects that we discussed earlier in this paper, namely, *language*, *indexing*, *query processing*, and *visualization*. The goal of this section is to help system designers understand the possible alternatives for building a system for big spatial data.

Table I summarizes the case studies covered in this section. Each row in the table designates a system for big spatial data while each column represents one aspect of the system as described below.

Architecture. Each system mentioned in the table is built on an existing system for big data and it follows its architecture. We can see that this column is quite diverse as it contains MapReduce-based systems, key-value stores, parallel DB, Array DB, and resilient distributed dataset (RDD). This shows that interest of processing spatial data across a wide range of systems. It is worth mentioning that we did not find any notable work for integrating spatial data into the core of a distributed column-oriented database such as Vertica [17], Dremel [16], or Impala [15]. Although these systems can process points and polygons due to their extensibility, this processing is done as an on-top approach while the core system does not understand the properties of spatial data [26].

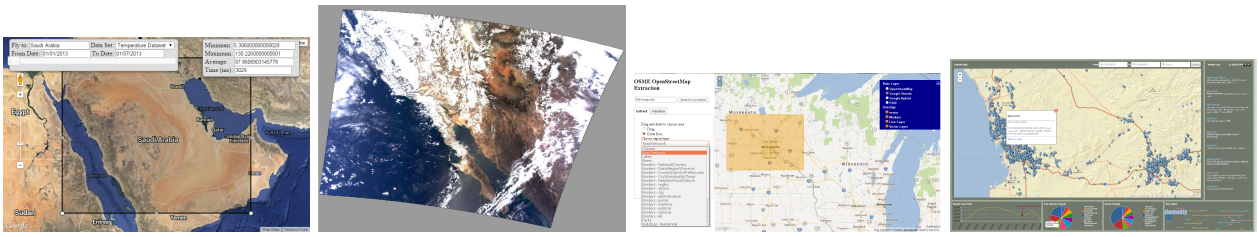
Data types. Most systems are designed to work with vector data (i.e., points and polygons) while only two systems, SciDB and GeoTrellis, handle raster data. SciDB employs an array-based architecture which allows it to deal with a raster layer as a two-dimensional array. GeoTrellis natively supports raster data and it distributes its work on a cluster using Spark. Although other systems do not natively support raster data, they can still process it by first converting it to vector data where each pixel maps to a point. Even though SciDB can handle polygons, the system internally does not treat it spatially; for example, it cannot index a set of polygons.

Language. While most systems provide a high level language, only three of them provide OGC-compliant data types and functions. Most of them are declerative SQL-like languages including HiveQL, Contextual Query Language (CQL), Secondo SQL-like language, and Array Query Language (AQL). Other languages are based on Scala and Pig Latin which are both procedural languages. Although there might not be a deep research in providing an OGC-compliant language, it is very important for end users to adopt the system especially that many users are not from the computer science field.

Indexes. The indexes implemented in systems vary and they include both flat indexes (grid and geohash) and hierarchical indexes (R-tree, R+-tree, Quad tree, PMR Quad tree and K-d tree). Notice that Parallel Secondo implements a local-only index by creating an index in each Secondo instance. However, it does not provide a global index which means that records are partitioned across machines in a non-spatial manner. This means that it has to access all partitions for each query which is more suitable for the parallel DB architecture. SpatialSpark provides an on-the-fly index which is constructed in an ad-hoc manner to answer a spatial join query but is never materialized to HDFS. This allows each machine to speed up the processing of assigned partitions but it cannot be used to prune partitions as the data is stored as heap files on disk. This leaves the RDD architecture of Spark open for research to implement spatial indexes.

Queries. The queries supported by the systems cover the three categories described in Section V, namely, basic operations, join operations and computational geometry (CG) operations. Both $\mathcal{MD}$-HBase and GeoMesa focus on the basic operations, range and kNN queries, as they are both real-time queries which makes them more suitable for the key-value store architecture. Spatial join is an analytical query which is suitable for MapReduce, parallel DB and RDD architectures. While Hadoop-GIS supports range query, kNN and spatial join, the constructed index is only used when processing the range query and self join, while the other two operations are done through full table scan. As GeoTrellis primarily work with raster data, it supports Map Algebra operations [46] which are perfect to parallelize in distributed systems as most of them they are embarrassingly parallel. GeoTrellis natively works on a single machine, so it uses Spark to parallelize the work over multiple machines.

Visualization. Unlike all other aspects, visualization is only supported by a few systems and only one of them covers both single- and multi-level images. Notice that the two systems that work with raster data support visualization as raster data is naturally a set of images which makes it reasonable to support visualization. GeoMesa supports visualization through GeoServer [47], a standalone web service for visualizing data on maps. This means that GeoMesa provides a plugin to allow GeoServer to retrieve the data from there while the actual visualization process runs on the single machine running GeoServer. This technique works only with small datasets but cannot handle very large datasets due to the limited capabilities of a single machine. The other approach is to integrate the visualization algorithm in the core system which makes it more scalable by parallelizing the work over a cluster of machines.



(a) SHAHED (b) EarthDB (c) TAREEG (d) TAGHREED

Fig. 4. Application case studies

VIII. Case Studies: Applications

This section provides a few case studies of applications that use the systems described in Section VII to handle big spatial data. These applications help readers understand how these systems are used in a real end-user application.

A. SHAHED

SHAHED [45] is a MapReduce system for analyzing and visualizing satellite data. It supports two main features, spatio-temporal selection and aggregate queries, and visualization. It makes these features available through a user-friendly web interface as shown in Figure 4(a). In this interface, users can navigate to any area on the map and choose either a specific point or a rectangle on the map. In addition, they can choose a time range from the date selectors. Based on user choice, the system runs a spatio-temporal selection query to retrieve all values (e.g., temperature) in the specified range, a spatio-temporal aggregate query to retrieve the minimum, maximum, and average in the range, or visualizes all values in the specified range as a temperature heat map.

SHAHED internally uses SpatialHadoop where all input datasets are indexed using a uniform grid index as the data is uniformly distributed. A SpatialHadoop MapReduce job constructs the indexes efficiently while the queries are processed directly on the index, without MapReduce, to provide real-time answers while avoiding the overhead of MapReduce. For example, it runs an aggregate query for a small area over a dataset of total size 2TB in less than a second. Temperature heat maps are generated using the visualization component of SpatialHadoop. If one day is selected, the generated heat map is visualized as a still image, while if a range of dates is selected, an image is created for each day and they are then compiled into a video. The efficiency of the visualization component allows it to visualize a dataset of 6.8 Billion points in around three minutes.

B. EarthDB

EarthDB [43] is another system that deals with satellite data and it uses SciDB as an underlying framework. It uses the functionality provided by SciDB to process satellite data and visualize the results as an image (Figure 4(b)). It supports two queries, (1) it reconstructs the true-color image by combining the values of the three components RGB, (2) it generates a vegetation heat map from raw satellite data. In both examples, the query and visualization are expressed in SciDB's Array Query Language (AQL) which processes the data in parallel and generates the desired image. The use of AQL allows users to play with the query in an easy way to make more advanced processing techniques or produce a different image.

C. TAREEG

TAREEG [48] is a MapReduce system for extracting OpenStreetMap [49] data using SpatialHadoop. It provides a web interface (Figure 4(c)) in which the user can navigate to any area in the world, select a rectangular area, and choose a map feature to extract (e.g., road network, lakes, or rivers). TAREEG automatically retrieves the required data and sends it back to the user via email in standard data formats such as CSV file, KML and Shapefile. The challenge in this system is extracting all map features from a single extremely large XML file provided by OpenStreetMap, called Planet.osm file. The Planet.osm file is a 500GB XML file which is updated weekly by OpenStreetMap. Using a standard PostGIS database to store and index the contents of that file takes days on a single machine. To process it efficiently, TAREEG uses Pigeon, the spatial high level language of SpatialHadoop, to extract all map features using MapReduce in standard format (e.g., Point, Line, and Polygon). The extracted files are then indexed using R-tree indexes to serve user requests more efficiently. The extraction and indexing steps happen once in an offline phase and it takes only a few hours on a 20-node cluster instead of days. In the online phase, the system issues range queries on the created indexes based on user request. The retrieved values are then put in standard file format and is sent back to the user in an email attachment.

D. TAGHREED

TAGHREED [50] is a system for querying, analyzing and visualizing geotagged tweets. It continuously collects geotagged tweets from Twitter [5] and indexes them using SpatialHadoop. Since SpatialHadoop does not support dynamic indexes, it creates a separate index for each day and periodically merges them into bigger indexes (say, weekly or monthly) to keep them under control. In addition to the spatial index, TAGHREED also constructs an inverted index to search the text of the tweets. The users are presented with a world map (Figure 4(d)) where they can navigate to any area of the world, choose a time range and a search text. TAGHREED automatically retrieves all tweets in the specified spatio-temporal range matching the search text, and runs some analyses on the retrieved tweets, such as, top hashtags and most popular users. Both the tweets and the analyses are visualized on the user interface where users can interact with them, e.g., choose a tweet to see more details.

IX. Conclusion

In this paper, we study the distributed systems designed to handle big spatial data. We show that this is an active research area with several systems emerging recently. We provide general guidelines to show the four components that are needed in a system for big spatial data. The high level *language* allows non-technical users to use the system using standard data types and operations. The spatial *indexes* store the data efficiently in a distributed storage engine and allow the queries to use them through suitable access methods. The *query processing* engine encapsulates a set of spatial queries to be used by end users. Finally, the *visualization* component helps users explore the datasets or query results in a graphical form. We give a few case studies of systems for big spatial data and show how they cover those four components. Finally, we provide some real applications that use those systems to handle big spatial data and provide end-user functionality.

References

[1] Telescope Hubbel site: Hubble Essentials: Quick Facts. http://hubblesite.org/the_telescope/hubble_essentials/quick_facts.php.

[2] (2012, Sep.) European XFEL: The Data Challenge. http://www.eiroforum.org/activities/scientific_highlights/201209_XFEL/index.html.

[3] (2012) MODIS Land Products Quality Assurance Tutorial: Part:1. https://lpdaac.usgs.gov/sites/default/files/public/modis/docs/MODIS_LP_QA_Tutorial-1.pdf.

[4] GnipBlog. https://blog.gnip.com/tag/geotagged-tweets/.

[5] Twitter. The About webpage. https://about.twitter.com/company.

[6] H. Markram, "The Blue Brain Project," *Nature Reviews Neuroscience*, vol. 7, no. 2, pp. 153–160, 2006.

[7] L. Pickle, M. Szczur, D. Lewis, , and D. Stinchcomb, "The Crossroads of GIS and Health Information: A Workshop on Developing a Research Agenda to Improve Cancer Control," *International Journal of Health Geographics*, vol. 5, no. 1, p. 51, 2006.

[8] A. Auchincloss, S. Gebreab, C. Mair, and A. D. Roux, "A Review of Spatial Methods in Epidemiology: 2000-2010," *Annual Review of Public Health*, vol. 33, pp. 107–22, Apr. 2012.

[9] Y. Thomas, D. Richardson, and I. Cheung, "Geography and Drug Addiction," *Springer Verlag*, 2009.

[10] J. Faghmous and V. Kumar, *Spatio-Temporal Data Mining for Climate Data: Advances, Challenges, and Opportunities*. Advances in Data Mining, Springer, 2013.

[11] J. Sankaranarayanan, H. Samet, B. E. Teitler, and M. D. L. J. Sperling, "TwitterStand: News in Tweets," in *SIGSPATIAL*, 2009.

[12] Apache. Hadoop. http://hadoop.apache.org/.

[13] A. Thusoo, J. S. Sen, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A Warehousing Solution over a Map-Reduce Framework," *PVLDB*, pp. 1626–1629, 2009.

[14] HBase. http://hbase.apache.org/.

[15] Impala. http://impala.io/.

[16] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Commun. ACM*, vol. 54, no. 6, pp. 114–123, 2011.

[17] "C-store: A column-oriented DBMS," in *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, 2005, pp. 553–564.

[18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," 2010. [Online]. Available: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[19] 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," in *VLDB*, 2013.

[20] ESRI Tools for Hadoop. http://esri.github.io/gis-tools-for-hadoop/.

[21] R. T. Whitman, M. B. Park, S. A. Ambrose, and E. G. Hoel, "Spatial Indexing and Analytics on Hadoop," in *SIGSPATIAL*, 2014.

[22] A. Eldawy and M. F. Mokbel, "SpatialHadoop: A MapReduce Framework for Spatial Data," in *ICDE*, 2015.

[23] J. Lu and R. H. Guting, "Parallel Secondo: Boosting Database Engines with Hadoop," in *ICPADS*, 2012.

[24] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi, "$\mathcal{MD}$-HBase: Design and Implementation of an Elastic Data Infrastructure for Cloud-scale Location Services," *DAPD*, vol. 31, no. 2, pp. 289–319, 2013.

[25] A. Fox, C. Eichelberger, J. Hughes, and S. Lyon, "Spatio-temporal Indexing in Non-relational Distributed Databases," in *International Conference on Big Data*, Santa Clara, CA, 2013.

[26] S. You, J. Zhang, and L. Gruenwald, "Large-Scale Spatial Join Query Processing in Cloud," The City College of New York, New York, NY, Tech. Rep.

[27] A. Kini and R. Emanuele. Geotrellis: Adding Geospatial Capabilities to Spark. http://spark-summit.org/2014/talk/geotrellis-adding-geospatial-capabilities-to-spark.

[28] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-so-foreign Language for Data Processing," in *SIGMOD*, 2008, pp. 1099–1110.

[29] Open Geospatial Consortium. http://www.opengeospatial.org/.

[30] PostGIS. http://postgis.net/.

[31] R. Kothuri and S. Ravada, "Oracle spatial, geometries," in *Encyclopedia of GIS.*, 2008, pp. 821–826.

[32] A. Eldawy and M. F. Mokbel, "Pigeon: A Spatial MapReduce Language," in *ICDE*, 2014.

[33] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," in *SIGMOD*, 1984.

[34] J. Nievergelt, H. Hinterberger, and K. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *TODS*, vol. 9, no. 1, 1984.

[35] R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys," *Acta Inf.*, vol. 4, pp. 1–9, 1974.

[36] M. Stonebraker, P. Brown, D. Zhang, and J. Becla, "SciDB: A Database Management System for Applications with Complex Analytics," *Computing in Science and Engineering*, vol. 15, no. 3, pp. 54–62, 2013.

[37] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on Processing Spatial Data with MapReduce," in *SSDBM*, 2009.

[38] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "$\mathcal{MD}$-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services," in *MDM*, 2011.

[39] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng, "Spatial Queries Evaluation with MapReduce," in *GCC*, 2009.

[40] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu, "SJMR: Parallelizing spatial join with MapReduce on clusters," in *CLUSTER*, 2009.

[41] J. Patel and D. DeWitt, "Partition Based Spatial-Merge Join," in *SIGMOD*, 1996.

[42] A. Eldawy, Y. Li, M. F. Mokbel, and R. Janardan, "CG_Hadoop: Computational Geometry in MapReduce," in *SIGSPATIAL*, 2013.

[43] G. Planthaber, M. Stonebraker, and J. Frew, "EarthDB: Scalable Analysis of MODIS Data using SciDB," in *BIGSPATIAL*, 2012.

[44] GeoTrellis. http://geotrellis.io/.

[45] A. Eldawy, M. F. Mokbel, S. Alharthi, A. Alzaidy, K. Tarek, and S. Ghani, "SHAHED: A MapReduce-based System for Querying and Visualizing Spatio-temporal Satellite Data," in *ICDE*, 2015.

[46] S. Shekhar and S. Chawla, *Spatial Databases: A Tour*. Prentice Hall Upper Saddle River, NJ, 2003.

[47] GeoServer. http://geoserver.org/.

[48] L. Alarabi, A. Eldawy, R. Alghamdi, and M. F. Mokbel, "TAREEG: A MapReduce-Based System for Extracting Spatial Data from OpenStreetMap," in *SIGSPATIAL*, Dallas, TX, Nov. 2014.

[49] [Online]. Available: \url{http://www.openstreetmap.org/}

[50] A. Magdy, L. Alarabi, S. Al-Harthi, M. Musleh, T. Ghanem, S. Ghani, and M. F. Mokbel, "Taghreed: A System for Querying, Analyzing, and Visualizing Geotagged Microblogs," in *SIGSPATIAL*, Nov. 2014.