DBSJ Journal Vol. 13, No. 1
March 2015

Invited Paper

The Era of Big Spatial Data: A Survey

Ahmed ELDAWY[♡]
Mohamed F. MOKBEL[◇]

The recent explosion in the amount of spatial data calls for specialized systems to handle big spatial data. In this paper, we survey and contrast the existing work that has been done in the area of big spatial data. We categorize the existing work in this area from three different angles, namely, *approach*, *architecture*, and *components*. (1) The approaches used to implement spatial query processing can be categorized as *on-top*, *from-scratch* and *built-in* approaches. (2) The existing works follow different *architectures* based on the underlying system they extend such as MapReduce, key-value stores, or parallel DBMS. (3) We also categorize the existing work into four main 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. At the end, we give cast studies of real applications that make use of these systems to provide services for end users.

1. 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 [12], medical devices produce spatial images (X-rays) at a rate of 50 PB per year [14], a NASA archive of satellite earth images has more than 500 TB and is increased daily by 25 GB [15], while there are 10 Million geotagged tweets issued from Twitter every day as 2% of the whole Twitter firehose [7, 13]. Meanwhile, various applications and agencies need to process an unprecedented amount of spatial data. For example, the Blue Brain Project [45] studies the brain's architectural and functional principles through modeling brain neurons as spatial data. Epidemiologists use spatial analysis techniques to identify cancer clusters [51], track infectious disease [19], and drug addiction [57]. Meteorologists study and simulate climate data through spatial analysis [30]. News reporters use geotagged tweets for event detection and analysis [53].

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 [3], Hive [58], HBase [8], Impala [9], Dremel [46], Vertica [54], and Spark [66]), 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 handle big spatial data. In this paper, we survey the existing work in the area of big spatial data. The goal is to cover the different approaches of processing big spatial data in a distributed environment which would help both existing and future researchers to pursue reasearch in this important area. First, this survey helps existing researchers to identify possible extensions to their work by checking the broad map of all work in this area. Second, it helps future researchers who are going to explore this area by laying out the state of the art work and highlighting open research problems.

In this survey, we explore existing work from three different angles, namely, *implementation approach*, *underlying architecture*, and *spatial components*. The *implementation approaches* are classified as *on-top*, *from-scratch*, and *built-in*. The *on-top* approach uses an existing system for non-spatial data as a black box while spatial queries are implemented through user-defined functions (UDFs). This makes it simple to implement but possibly inefficient as the system is still internally unaware of spatial data. The *from-scratch* approach is the other extreme where a new system is constructed from scratch to handle a specific application which makes it very efficient but difficult to build and maintain. The *built-in* approach extends an existing system by injecting spatial data awareness in its core which achieves a good performance while avoiding building a new system from scratch. The *underlying architectures* of most surveyed systems follow that of existing systems for non-spatial data such as MapReduce [23], Resilient Distributed Dataset (RDD) [65], or Key-Value stores [22]. We also categorize the *spatial components* implemented by these systems into *language*, *indexing*, *query processing*, and *visualization*. We give insights of how each of these components are supported using different implementation approaches and in different architectures. Finally, we provide some key applications of big spatial data that combine different components in one system to provide services for end-users.

The rest of this paper is organized as follows. We start by givin an overview of the surveyed work in Section . Section describes the three different approaches to handle big spatial data. Section discusses the different underlying architectures and how spatial queries are implemented in each one. The existing works of the four spatial components, namely, *language*, *indexing*, *query processing*, and *visualization*, are provided in Sections -. After that, Section provides a few examples of end-user applications that work with big spatial data. Finally, Section concludes the paper.

2. Overview

Table 1 provides a high level overview of the works discussed in this paper. Each row in the table designates a system for big spatial data while each column represents one aspect of the system. This section provides an overview of these systems and highlights the main differences between them. The rest of this paper delves into the details of each aspect (i.e., column) and provide more insight about the different work done from that aspect.

Approach. The three implementation approaches used in related work are *on-top*, *from-scratch*, and *built-in*. Due to its simplicity, the *on-top* approach is used more than anything else. On the contrary, only a few systems are built from scratch and their features are very limited compared to other systems. Due to their complexity, most of them are not active in research anymore. Only SciDB is still active, however, it is designed mainly for scientific applications dealing with multidimensional arrays rather than spatial applications dealing with lines and

[♡] Nonmember Department of Computer Science and Engineering, University of Minnesota, Twin Cities
eldawy@cs.umn.edu

[◇] Nonmember Department of Computer Science and Engineering, University of Minnesota, Twin Cities
mokbel@cs.umn.edu

Table 1: Case studies of systems

	Approach	Architecture	Language	Indexes	Queries	Visualization
[21]	On-top	MapReduce	-	R-tree	Image quality	-
[62, 68, 69]	On-top	MapReduce	-	R-tree	RQ, KNN, SJ, ANN	-
[33]	On-top	MapReduce	-	-	Multiway SJ	-
[60]	On-top	MapReduce	-	-	-	Single level
[71]	On-top	MapReduce	-	-	K-means	-
[17]	On-top	MapReduce	-	-	Voronoi, KNN, RNN, MaxRNN	-
[42]	On-top	MapReduce	-	-	KNN Join	-
[67]	On-top	MapReduce	-	-	KNN Join	-
BRACE [61]	From-scratch	MapReduce	BRASIL	Grid	SJ	-
PRADASE [43]	Built-in	MapReduce	-	Grid	RQ	
ScalaGiST [41]	Built-in	MapReduce	-	GiST	RQ, KNN	-
SpatialHadoop [26–29]	Built-in	MapReduce	Pigeon*	Grid, R-tree, R+-tree	RQ, KNN, SJ, CG	Single level, Multilevel
Hadoop GIS [16]	Built-in	MapReduce	QL^{SP}	Grid	RQ, KNN, SJ	-
ESRI Tools for Hadoop [4, 63]	Built-in	MapReduce	HiveQL*	PMR Quad Tree	RQ, KNN	-
$\mathcal{MD}$-HBase [48]	Built-in	Key-value store	-	Quad Tree, K-d tree	RQ, KNN	-
GeoMesa [32]	Built-in	Key-value store	CQL*	Geohash	RQ	Through GeoServer
Paradise [24]	From-scratch	Parallel DB	SQL-Like	Grid	RQ, NN, SJ	-
Parallel Secondo [40]	Built-in	Parallel DB	SQL-Like	Local only	RQ, SJ	-
SciDB [52, 56]	From-scratch	Array DB	AQL, AFL	K-d tree	RQ, KNN	Single level
[64]	On-top	RDD + Impala	Scala-based	On-the-fly	SJ	-
GeoTrellis [6, 36]	On-top	RDD	Scala-based	-	Map Algebra	-
[70]	From-scratch	Other	-	K-d tree, R-tree	RQ	-

$*$ OGC-compliant

polygons. The built-in approach is used with a few systems and most of them are based on MapReduce due to the popularity of Hadoop. More details about the different approaches are given in Section .

Architecture. Most system discussed in this survey are built on existing systems for big data and, hence, they follow their architectures. We can notice in the table that this column is quite diverse as it contains MapReduce-based systems, key-value stores, parallel DB, Array DB, resilient distributed dataset (RDD), and others. This shows a great 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 [54], Dremel [46], or Impala [9]. Although these systems can process points and polygons due to their extensibility, this kind of processing is done as an on-top approach while the core system does not understand the properties of spatial data [64]. The different architectures are described in details in Section .

Language. A high level language allows non-technical users to use the system without much knowledge of the system internal design. The Open Geospecial Consortium (OGC) defines standard data types and functions to be supported by such a high level language. While many systems provide a high level language for spatial processing, 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. Section provides more details about the language component.

Indexes. Spatial indexes allow system to store the data in the file system in a spatial manner taking the spatial attributes into consideration. The goal is to allow queries to run faster by making use of the index. 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 some systems implement local-only indexes by creating an index in each cluster node. This technique is relatively easy but is also limited as it cannot be used in queries where nearby records need to be processed together. This means that it has to access all file partitions to process each query which is more suitable for the parallel DB architecture such as Parallel Secondo [40]. 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 spatial indexing in the RDD architecture an open research problem. Section gives more details about the distributed spatial indexes.

Queries. The main component of any system for big data processing is the query processing component which encapsulates the spatial queries supported by the system. The queries supported by the systems cover a wide range of categories including; *basic queries* such as range query and *k*NN; *join queries* including spatial join and kNN join; *computational goemetry* queries such as polygon union, skyline, convex hull and Voronoi diagram construction; *data mining* queries such as K-means; and *raster operations* such as image quality. The underlying architecture affects the choice of operations to implement. For example, Hadoop, is geared towards analysis operations such as kNN join and spatial join, while HBase and Accumulo are designed for point queries which make them more suitable for interactive queries such as point queries and nearest neighbor queries. SciDB works natively with multidimensional arrays which makes it more suitable for raster operations working with satellite data. Section gives more details about the supported queries.

Visualization. Visualization is the process of creating an image that describes an input dataset such as a heat map for temperature. There are mainly two types of images, *single level* image which is generated with a fixed size and users cannot zoom in to see more details, and *multilevel* image which is generated as multiple images at different zoom levels to allow users to zoom in and see more details. Unlike all other aspects, visualization is only supported by a few systems and only one of them covers both single level and multilevel 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 [5], 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 as done in [29, 52, 60]. More details about visualization are given in Section .

3. Implementation Approach

In this section, we describe the different approaches of implementing a system for big spatial data. There are mainly three different approaches used in existing works, namely, *on-top*, *from-scratch*, and *built-in*, as detailed below.

3.1 On-top Approach

In the *on-top* approach, an underlying system for big data is used as a black box while spatial data awareness is added through user-defined functions which are written *on top* of the system. The advantage of this approach is its simplicity as we do not need to delve into the implementation details of the system. Most existing system are flexible and provide an easy way to add third party logic through standard APIs. For example, Hadoop can be extended by defining custom *map* and *reduce* functions [23]. In Spark [66], developers can write custom logic in Java or Scala through the resilient distributed dataset (RDD) abstraction [65]. Hive [58] exposes a SQL-like language which can also be extended through UDFs.

This approach is used to implement several queries using the brute-force technique in Hadoop. In this case, all input records are scanned using a MapReduce program to compute the answer. For example, a range query operation scans the whole file and tests each record against the query range [68]. Similar techniques are used to implement other operations such as k-nearest neighbor [68], image quality computation [21], and computational geometry operations [26]. Binary operations are done in a similar fashion where all *pairs* of records are scanned. For example, in spatial join, even pair of records is tested with the join predicate to find matching pairs. Due to the huge sizes of input files, scanning all pairs is usually unpractical. Therefore, an alternative technique is employed where the files are spatially co-partitioned on-the-fly, such that each partition can be processed independently. This technique is used to answer spatial join query [69], all nearest neighbor (ANN) [68], approximate [67] and exact [42] kNN-join queries. For example, SJMR [69] is proposed as a MapReduce spatial join algorithm which resembles the partition based spatial-merge (PBSM) join algorithm for distributed environments. In this algorithm, the map function partitions the data according to a uniform grid while the reduce function finds overlapping records in each grid cell.

This technique is used in other systems as well. For example, ESRI proposes a set of user-defined functions (UDFs) [4] which extends Hive to support standard data types and operations. This allows writing SQL-like queries to process spatial data in a similar way to spatial database systems such as PostGIS and Oracle Spatial. A similar technique is used to build Pigeon [27], an extension to Pig Latin [49] to write spatial MapReduce queries. The spatial join operation is implemented in a similar way to the method described above in Hive [16], Spark, and Impala [64]. Raster operations have been also scaled out on Spark by combining it with GeoTrellis [36]

3.2 From-scratch Approach

The second approach to support big spatial data in a distributed environment is to build a new system *from scratch* to support a specific application. This gives the full flexibility to design the best technique to store the data and process it. However, it has two main drawbacks. First, the system requires a huge effort to build and maintain as all components have to be created from scratch. Second, users that already use existing systems to process non-spatial attributes do not want to throw away their systems and use a new one to process spatial attributes, rather, they want one system to process both spatial and non-spatial attributes.

One of the early systems that were designed from scratch to support spatial data is Paradise [24]. It was proposed as a parallel DBMS for spatial data and was designed to support both vector and raster data. Unfortunately, it is no longer active in research and was not updated in more than a decade. A more recent system is BRACE [61] which is proposed to perform behavioral simulations based on the MapReduce architecture and it consists of three layers. The *language* layer contains a high level language, termed BRASIL, for defining the logic of the simulation. The *storage* layer stores the data in a distributed grid index stored in the main memory of cluster machines. The *query processing* layer, termed BRACE, applies a series of distributed spatial joins to perform the behavioral simulation. Although this system is very efficient, it is not suitable to perform any queries other than the behavioral simulation which makes it very limited. Similarly, another system is built from scratch which stores data in distributed K-d trees and R-trees and perform both point and range queries [70]. SciDB [56] is another example of a system build from scratch to handle multidimensional data. It is originally designed for scientific applications with high-dimensional data which means it can handle two or three dimensional spatial data such as satellite [52] or astronomic data [59].

3.3 Built-in Approach

The third approach to build a system for big spatial data is the *built-in* approach in which an existing system is extended by injecting spatial data awareness inside the core of the system. This is expected to combine the advantages of the two other approaches. First, it is relatively easier than building a new system from scratch as it makes use of an existing system. Second, it achieves a good performance as the core of the system is aware of spatial data and handles it efficiently. An extended system with built-in spatial support should be backward compatible with the original system which means it can still handle non-spatial data as before but it adds special handling for spatial data.

For example, PRADASE [43] extends Hadoop to work with trajectory data where the data is stored in HDFS as a grid and an efficient spatio-temporal range query is implemented on it. $\mathcal{MD}$-HBase [48] introduces a K-d tree and quad tree inside HBase [8] and uses it to run both range and kNN queries efficiently. GeoMesa [32] follows a similar approach in Accumulo by building a geohash index. Hadoop-GIS [16] extends Hive with a grid index and efficient query processing for range and self-join queries. SpatialHadoop [28] extends Hadoop with grid index, R-tree, and R+-tree, and uses them to provide efficient algorithms for range query, kNN, spatial join, and a number of computational geometry operations [26]. ScalaGiST [41] is an attempt to provide a wide range indexes in Hadoop using a GiST-like abstraction. Parallel Secondo [40] extends a spatial DBMS, Secondo, to provide a parallel spatial DBMS system with a SQL-like query language.

4. Architecture

There are different architectures used in systems for big spatial data. Since most of these systems extend existing systems for big data, they follow their underlying architecture. The features and limitations of each system affect the scope of spatial applications and queries that can be supported in it. For example, systems that are designed for large analytical queries, such as Hadoop, Hive, and Impala, are more suitable to handle long-running spatial queries such as spatial join and kNN join. On the other hand, systems that are designed for interactive queries, such as key-value stores, are better to use if we want to support small queries such as point or nearest neighbor queries. In this section, we categorize the existing work in big spatial data according to the underlying system architecture and highlight the types of queries that are better suited for each system architecture and how they are implemented.

4. 1 MapReduce

In the MapReduce architecture, the data sits in a distributed file system while the query processing is done through the MapReduce abstraction [23]. Typically, the *map* function scans the whole file and generates a set of intermediate key-value pairs. These pairs are shuffled across machines and grouped by the key where each group is reduced separately. Although this abstraction is very generic and can be applied in different system architectures, it was originally designed to handle long-running analytical queries. There are three notable open source systems that support this architecture for non-spatial data, namely, Hadoop [3], the original open source MapReduce system, Hive [58], a data warehousing system built on-top of Hadoop, and Pig Latin [49], a high level language for Hadoop. There are two main limitations to these systems which limit its applicability for different queries. (1) They all use the Hadoop Distributed File System (HDFS) which does not support file edits making it suitable for static data. (2) There is a significant overhead for starting each MapReduce job making it unsuitable for interactive queries which should run in a sub-second, and iterative algorithms where hundreds of iterations might run for each algorithm and the overhead accumulates for each iteration (i.e., MapReduce job). Since Hadoop was the first open source system for distributed processing that is relatively easy to install and use, most work in big spatial data is based on it.

As Hadoop is designed for analytical jobs, most operations built for Hadoop are long-running analytical jobs. This includes spatial index construction [16, 21, 28, 39, 43, 63], image quality computation for raster data [21], all nearest neighbor (ANN) [21], spatial join [16, 28, 69], and kNN join [42, 67]. Also, several computational geometry queries are implemented for Hadoop [26] including polygon union, skyline, convex hull, farthest and closets pairs. In addition, visualization techniques have been proposed for spatial data in the Hadoop environment [29, 60]

Although Hadoop is not designed for interactive queries, some works proposed MapReduce algorithms for a few interactive queries for two reasons. First, users of Hadoop might occasionally need to run this type of queries and it would be better to run them as efficient as possible. Second, as mentioned above, Hadoop has been the main system for distributed processing for a few years and it is worth, from a research point of view, to test it with all types of queries. The implemented queries include range queries [16, 21, 28, 43, 63], kNN queries [16, 17, 21, 28, 63], and reverse nearest neighbor queries (RNN) [17]. Although most of these systems optimize the MapReduce job to return the result as fast as possible, there is a significant overhead in starting the query making all of them unsuitable for an interactive application. As a work around, some systems construct a spatial index using MapReduce and process it directly from the HDFS to avoid the overhead of the MapReduce job [29, 44].

Similar to interactive queries, some iterative spatial queries can be also implemented in MapReduce. For example, the k-means clustering algorithm was implemented in Hadoop [71] where each iteration runs as a separate MapReduce job. As expected, the significant overhead on each iteration makes the algorithm not very scalable with cluster size. For example, as reported in [71], using four machines reduces the processing time by only 25% instead of the ideal 75%.

There is only one system that uses the MapReduce engine to process spatial iterative queries efficiently for behavioral simulation [61]. However, this system does not use Hadoop at all as it builds a system from scratch for this application where all data resides in memory which reduces the total overhead. This at least shows that the limitations are coming from the Hadoop environment not the MapReduce abstraction.

4. 2 Key-value Store

An alternative architecture is the key-value store where data is abstracted as a set of key-value records. This architecture is used in some open source systems such as HBase [8] and Accumulo [2] and was inspired by BigTable [22] designed by Google. In this architecture, data is manipulated in a per-record basis where each record is identified by a key and holds one or more values. Unlike Hadoop, HBase and Accumulo allow modifying and deleting records. In addition, they provide quick access to a single record by keeping all records sorted by the key. Unfortunately, this efficiency with single records makes them less efficient than Hadoop in accessing (i.e., scanning) a very large file making them less efficient with analytical queries.

Using this architecture, it was possible to implement spatial indexes that support insertions and deletions in real-time including K-d tree, quad tree [48], and Geohash index [32]. In both cases, the underlying order of key-value pairs is exploited by linearizing spatial records using a space filling curve, such as the Z-curve or Geohash, and using the linearized value as part of the key. This ensures that spatially nearby records are stored close together on disk. On-top of these indexes, point, range and kNN queries were implemented efficiently by limiting the search space to a very small range of keys in the index.

4. 3 Parallel Database

In parallel database architecture, there is one master node and multiple slave nodes where the master node issues the query and the slave nodes execute the query. Each slave node runs a spatial DBMS instance which acts as a storage and query processing engine. For example, Parallel Secondo [40] runs multiple Secondo, a spatial DBMS, instances as one per node while using the task scheduler of Hadoop to coordinate these nodes. In this case, HDFS and MapReduce query processing are both overridden by Secondo storage and query processing engine. This makes it easy to parallelize embarrasingly parallel problems to multiple nodes but this solution is still limited as it does not incorporate global indexes.

4. 4 Array Database

Array databases were proposed mainly for scientific applications which deal with high dimensional data [55]. Since spatial data is natively multi-dimensional, systems that employ this architecture can also support spatial data. In particular, raster data is a good candidate to be supported in such architecture as each raster layer can be represented as a two-dimensional array. The main queries that are supported by array databases include selection (i.e., N-dimensional range query) and analytical queries using linear algebra. A drawback with this data model is that it cannot efficiently support lines or polygons as they cannot be directly stored in a common array structure.

SciDB implements efficient multidimensional selection queries using a K-d tree index. In addition, its array data model makes it more suitable for raster datasets, such as satellite images, which are naturally represented as a two-dimensional array. For raster datasets, it supports aggregation queries, iterative algorithms, and convolution operations which combine multiple images [59].

4. 5 Resilient Distributed Dataset (RDD)

RDD [65] is a programming paradigm designed to support complex analytical queries using distributed in-memory processing. In this programming model, data is loaded from a distributed file system, goes through a pipeline of multiple in-memory operations, and the result is finally stored back in the distributed file system. This is mainly proposed as an improvement to Hadoop to avoid the huge overhead associated with MapReduce programs by avoiding excessive interaction with disk. This makes it more suitable with iterative queries by processing the iterations while data is in memory and finally writing the answer to disk. However, it still suffers from the limitations of HDFS as it is used as the main file system.

The main system that uses RDD is Spark [66] which is available as open source. Since this system is relatively newer than Hadoop, there has not been much work done in the area of big spatial data using Spark. In [64], a spatial join query is implemented in Spark by implementing a variation of the PBSM algorithm [50] to distribute the work across machines and then it uses a GPU-based algorithm to do the join on each machine. In [36], Spark is combined with GeoTrellis [6], a system for raster data processing, to parallelize raster operations. This is particularly useful for raster operations because most of them are very localized and embarrassingly parallel. Although Spark is optimized for iterative processing, we did not find any work proposing RDD implementations for iterative spatial operations such as the k-means clustering algorithm.

5. 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. Although providing a language for spatial data might not be of great interest to researchers due to the limited research challenges, it is of a great importance for end-users to adopt the system especially that most of them are not from the computer science field. Most systems for big non-spatial data are already equipped with a high level language, such as, Pig Latin [49] for Hadoop, HiveQL for Hive [58], AQL for SciDB [56], and Scala-based language for Spark [66]. 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) [10] 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 [11] and Oracle Spatial [37], 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 [27] which extends Pig Latin, ESRI Tools for Hadoop [4] which extends HiveQL, and the contextual query language (CQL) used in GeoMesa [32]. Hadoop-GIS [16] proposes QL^{SP} which extends HiveQL but it does not follow the OGC standards. In [61], an actor-based high level language, termed BRASIL, which is designed specifically for behavioral simulations. SciDB provides an array query language (AQL) which is not designed specifically for spatial data but can be extended through user-defined functions (UDFs).

6. Indexing

Input files in a typical system for big data are not spatially organized which means that the 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 [34], Grid File [47], and Quad Tree [31]). However, migrating these indexes to other systems for big data is challenging given the different architectures used in each one. In this section, we survey the existing work in spatial indexing in distributed systems for big data. First, we describe the general layout of distributed spatial indexes used in most systems. Then, we describe the existing techniques for spatial indexing in three categories, namely, *index bulk loading*, *dynamic indexing*, and *secondary indexes*. Finally, we give a brief discussion of how such an index is made available to query processing.

6. 1 Index Layout

The general layout of spatial indexes created in distributed systems is a two-layer index of one global index and multiple local indexes. The global index determines how the data is partitioned across machines while local indexes determine how records are stored in each machine. This two-layer index lends itself to the distributed environment where there is one master node that stores the global index and multiple slave nodes that store local indexes. These two levels are orthogonal which means a system can implement a global-only index, a local-only index, or both. Besides, there is a flexibility in choosing any type of index at each of the two levels. Figure 1 gives an example of an R-tree global index constructed on a 400GB dataset that represents the road network in the whole world. The blue points in the figure represent road segments while the black rectangles represent partition boundaries of the global index. As shown in figure, this index handles the skewness very well by adjusting the size of the partitions such that each partition contains, roughly, the same amount of data. For example, dense areas in Europe contain very small rectangles while sparse areas in the oceans contain very large rectangles.

6. 2 Index Bulk Loading

The most common file system used in open source distributed systems is the Hadoop Distributed File System. It is already used in Hadoop, Hive, HBase, Spark, Impala and Accumulo. HDFS has a major limitation that files can only be written in a sequential manner and, once

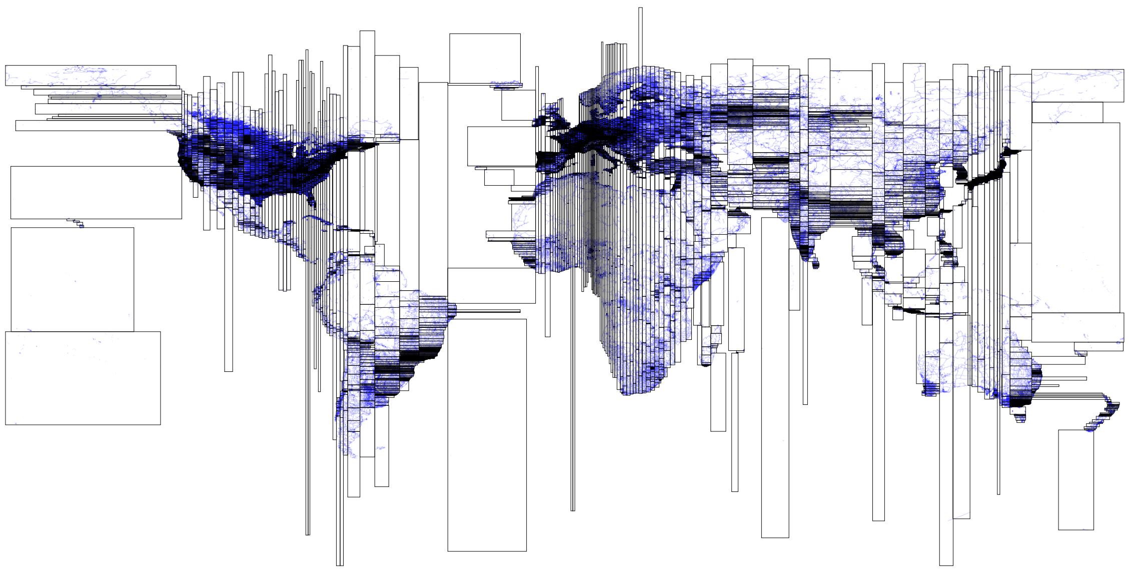


Figure 1: R-tree partitioning of a 400GB road network data

written, cannot be further modified. This rules out most traditional index building techniques as they rely on inserting records one-by-one or in batches and the index structure evolves as records are inserted. Since HDFS is designed mainly for static data, most techniques focus on bulk loading the index which are described below.

To overcome the limitations of HDFS, most bulk loading techniques use a three-phase approach. In the first phase, the space is subdivided into *n* partitions by reading a sample of the input file which is then partitioned into *n* partitions of roughly equal sizes. It is expected that the sample is a good representative of the data distribution in the original file. In the second phase, the input file is scanned in parallel, and each record is assigned to one or more partitions based on its location and the index type. Records in each partition are then loaded into a separate *local* index which is stored in the file system. It is expected that the size of each partition is small enough to be indexed by a single machine. In the third phase, the *local* indexes are grouped under a common *global* index based on their corresponding MBRs; i.e., the MBR of the root node of each local index.

In [16, 28], a uniform grid index is constructed by subdividing the space using a uniform grid. Notice that in this case, no sample needs to be drawn from the input file as the space is always divided using a uniform grid. Then, each record is assigned to all overlapping partitions and each partition is written as a heap file; i.e., no local indexing is required. Finally, the global index is created by building a simple in-memory lookup table that stores where each grid cell is stored on disk.

This technique is also used in [21] to build an R-tree where the space is subdivided by mapping each point in the random sample to a single number, using a Z-curve, sorting them based on the Z value, and then subdividing the sorted range into *n* partitions equal to number of machines in the cluster. A similar technique is used in [28] to build both R-tree and R+-tree, where *n* is first determined by dividing the file size over the HDFS block capacity to calculate the expected number of blocks in the output file. Then, the sample is bulk loaded into an in-memory R-tree using the STR bulk loading algorithm [38]. While partitioning the file, if R-tree is used, each record is assigned to exactly one partition, while in R+-tree, each record is assigned to all overlapping partitions. This technique is further generalized in [41] to bulk load any tree index described by the GiST abstraction [35].

In [63], a slightly modified technique is used to build a PMR Quad tree. First, a random sample is drawn form the file, linearized using a Z-curve, and partitioned into *n* partitions. Then, without spatially partitioning the file, each machine loads part of the file and builds an in-memory PMR Quad tree for that partition. The nodes of each partial quad tree are partitioned into *n* partitions based on their Z-curve values. After that, each machine is assigned a partition and merges all nodes in that partition into a *locally consistent* quad tree. Finally, these *locally consistent* quad trees are merged into one final PMR Quad tree.

6.3 Dynamic Indexes

Some applications require a dynamic index that accommodates insertions and deletions of highly dynamic data, such as geotagged tweets, moving objects, and sensor data. In this case, static indexes constructed using bulk loading cannot work. HBase [8] and Accumulo [2] provide a layer on top of HDFS that allows key-value records to be dynamically inserted and deleted. Modification of records is accommodated by using a multi-versioning system where each change is appended with a new timestamp. In addition, these systems keep all records sorted by the key which allows efficient access to a single record or small ranges of consecutive records. These systems are utilized to build dynamic indexes for spatial data as follow.

$\mathcal{MD}$-HBase [48] extends HBase to support both quad tree and K-d tree indexes where the index contains points only. In this approach, each point is inserted as a record in HBase where the key is calculated by mapping the two-dimensional point location to a single value on the Z-curve. This means that all points are sorted based on the Z-curve. After that, the properties of the Z-curve allows the sorted order to be viewed as either a quad tree or a K-d tree. This structure is utilized to run both range and kNN queries efficiently. This technique is also applies in [32] to build a geohash index in Accumulo but it extends the work in two directions. First, it constructs a spatio-temporal index by interleaving the time dimension with the geohash of the spatial dimensions. Second, it supports polygons and polylines by replicating each record to all overlapping values on the Z-curve. Although these systems provide dynamic indexes, they are designed and optimized for point queries which inserts or retrieves a single record. They can still run a MapReduce job on the constructed index, but the performance is relatively poor compared to MapReduce jobs in Hadoop.

SciDB [56] supports an efficient dynamic K-d tree index as it is designed for high dimensional data. Similar to HBase, SciDB uses multi-versioning to accommodate updates and records are kept sorted using their keys. However, unlike HBase and Accumulo, the key is allowed to be multidimensional which makes it ready to store spatial points. This technique is not directly applicable to lines or polygons as a line or polygon cannot be assigned a single key.

6. 4 Secondary Indexes

Similar to traditional DBMS, distributed systems can build either a primary index or a secondary index. In the primary index, records are physically reordered on the disk to match the index, while in secondary index, records are kept in their original order while the index points to their offset in the file. In HDFS, secondary indexes perform very poorly due to the huge overhead associated with random file access [39]. Therefore, most existing indexing techniques focus on primary indexing. There are only two notable works that implement secondary indexes [41, 63]. In both cases, the index is bulk loaded as described earlier but instead of storing the whole record, it only stores the offset of each record in the file. As clearly shown in [63], the performance of the secondary index is very poor compared to a primary index and is thus not recommended. However, it could be inevitable to have a secondary index if users need to build multiple indexes on the same file.

6. 5 Access Methods

Creating the index on disk is just the first part of the indexing process, 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. Most of the works discussed above do not mention clearly the abstraction they provide to the query processing logic and describe their query processing directly. This is primarily because they focus on specific queries and they describe how they are implemented. However, it is described in [28] that the index constructed in Hadoop 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.

This separation between the index structure on 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.

7. Querying

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 to allow users to express custom operations while making use of the spatial indexes. To give some concrete examples, we will describe five categories of operations, namely, basic query operations, join operations, computational geometry operations, data mining operations, and raster operations.

7. 1 Basic Query Operations

The basic spatial query operations include, point, range, and nearest neighbor queries. We give examples of how these operations are implemented in different systems, with, and without indexes.

Point and 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. A point query is a special case where the query range has a zero width and height. In [68], a brute force algorithm for range queries is implemented in MapReduce by scanning the whole file and selecting records that match the query area. In [16, 28, 63], the constructed index is utilized where the global index is first used to find partitions that overlap the query range and then the local indexes, if constructed, are used to quickly find records in the final answer. The reference point [25] duplicate avoidance technique is used to eliminate redundant records in the answer if the index contains replication. Although this technique is efficient in design, it performs bad for point queries and small ranges as it suffers from the overhead of starting a MapReduce job. This overhead is avoided in GeoMesa [32] and $\mathcal{MD}$-HBase [48], as they run on a key-value store which is more efficient for this kind of queries. In Hadoop, some applications [18, 29, 44] achieve an interactive response for range queries by bypassing the MapReduce engine and running the query against the index on the file system directly.

nearest neighbor (NN) queries: There are different variations of NN queries but the most common one is the kNN query. 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. In [16, 68], a brute force technique is implemented in MapReduce where the input is scanned, the distance of each point $p \in P$ to Q is calculated, points are sorted based on distance, and finally top-k points are selected. In [28, 48, 63], the constructed indexes are used by first searching the partition that contains the query point and then expanding the search, as needed, to adjacent partitions until the answer is complete. In [17], a different approach is used where a Voronoi diagram is constructed for the input file first, and then the properties of this diagram is used to answer kNN queries. In addition to kNN query, this Voronoi diagram is also used to answer both reverse NN (RNN) and maximal reverse NN (MaxRNN) queries. In [62], the all nearest neighbor (ANN) query is implemented in MapReduce which finds the nearest neighbor for each point in a given set of points. It works as two MapReduce jobs where the first one partitions the data using a Z-curve to group nearby points together, and finds the answer for points which are colocated with their NN in the same partition. The second MapReduce job finds the NN for all remaining points using the brute-force technique.

7. 2 Join Operations

Spatial Join: 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, the partition based spatial-merge (PBSM) join algorithm is used where the input files are copartitioned using a uniform grid and the contents of each grid cell are joined independently. This technique is implemented in Hadoop [69], Impala and Spark [64], without major modifications. A more efficient algorithm is provided in [16] for the special case of self-join when the input file is indexed using a uniform grid. In this algorithm, the partition step is avoided and the records in each grid cell are directly joined. In [28], a more efficient algorithm is implemented which provides a more general join algorithm for two files when one

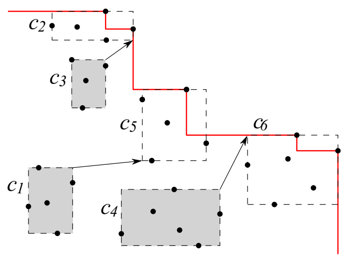


Figure 2: Pruning in skyline

or both files are indexed. If both files are indexed, it finds every pair of overlapping partitions and each pair is processed independently by a map task which applies an in-memory spatial join algorithm, such as the plane-sweep algorithm. If only one file is indexed, it partitions the other file on-the-fly such that each partition corresponds to one partition in the indexed file. This allows a one-to-one mapping between the partitions of the two files making it very efficient to join each pair independently.

kNN Join: Another join operation is the kNN join where the input is two datasets of points R and S, and we want to find for each point $r \in R$, its k nearest neighbors in S. In [67], a brute force technique is proposed which calculates all pairwise distances between every pair of points $r \in R$ and $s \in S$, sorts all of them, and finds the top-k for each point r. A more efficient technique is proposed in the same work but it provides an approximate answer. The later technique first partitions all points based on a Z-curve, and finds the kNN for each point within its partition. In [42] an efficient and exact algorithm is provided for the kNN join query which runs in two MapReduce jobs. In the first job, all data is partitioned based on a Voronoi diagram and a partial answer is computed only for points which are colocated with their kNN in the same partition. In the second phase, the kNN of all remaining points is calculated using the brute-force technique.

7. 3 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 for 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. Many computational geometry operations have a divide and conquer algorithm which can be adapted to work in a distributed environment where the problem is divided over cluster nodes, each node generates a partial answer, and a single machines combines all these in one final answer. If the input file is spatially indexed, this algorithm can be improved by pruning partitions that do not contribute to the answer. In the following part, we describe a Voronoi diagram construction algorithm which does not use a spatial index, and then give two examples of pruning techniques used with skyline and convex hull operations.

Voronoi Diagram. A Voronoi diagram for a set of vertices is a partitioning of the space into disjoint polygons where each polygon is associated with one vertex in the input. All points inside each polygon are closer to the associated vertex than any other vertex in the input. In [17], the Voronoi diagram is constructed using a MapReduce job where each mapper constructs a partial Voronoi diagram for a partition, and one reducer merges them all into one final Voronoi diagram. The drawback of this method is that the machine that merges them could be a bottleneck for very large datasets.

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 process the whole file. This algorithm is improved in [26] 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 2, 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.

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.

7. 4 Spatial Data Mining Operations

Most data mining techniques rely on iterative processing where the answer is refined in each iteration until an answer of an accepted quality is reached. For a long time, Hadoop was the sole player in the area of big data. Unfortunately, Hadoop is ill-equipped to run iterative algorithm due to the significant overhead associated with each iteration [20]. Therefore, there is no much work in this area for Hadoop. The K-Means clustering algorithm is implemented in a straight forward manner in MapReduce where each iteration is implemented as a separate MapReduce job [71]. However, the performance was very poor due to the overhead imposed by Hadoop in each iteration. Although Spark was proposed as an alternative system with better support to this kind of algorithms, we did not find any notable work for spatial data mining with Spark which leaves this area open for research.

7. 5 Raster Operations

All the operations described above are vector operations which deal with point, lines and polygons. Another class of operations are the raster operations which are used with raster data such as images. Unlike vector operations, raster operations are much easier to parallelize due to their locality in processing. Most of these operations deal with a single pixel or a few pixels that are close to each other. This makes these operations embarrassingly parallel and can be easily parallelized in a shared-nothing system. In [36], GeoTrellis, a system for raster data processing, is combined with Spark to parallelize the computation over a cluster of machines. In [52, 59], raster operations are parallelized using SciDB, an array database, where each raster dataset is represented as a two-dimensional array. Each of these systems implement specific raster operations but it would be interesting to build a system that supports a wide range of raster operations over a cluster of machines.

8. 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

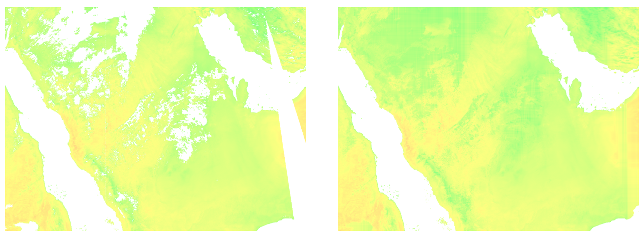


Figure 3: Single level visualization

data which makes them unable to handle big spatial data. 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.

8. 1 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* × *height*) in pixels. Existing visualization algorithms for single level image can be categorized based on the partitioning technique they apply into *pixel-level* partitioning and *grid-level* partitioning.

In *pixel-level* partitioning, records are partitioned according to the image pixel they affect in the final image. This technique is used to render 3D triangles [60] using Hadoop and to visualize satellite data [52] using SciDB. In [60], 3D triangles are partitioned and grouped by the pixel they affect in the image. In other words, a partition is created for every pixel in the generated image and it contains all triangles that could possibly determine the color of that pixel. In each partition, triangles are sorted by their *z*-dimension and the color of the pixel is determined based on the triangle on the top. In [52], satellite data from multiple raster layers are partitioned and grouped using pixel-level-partitioning. For each partition, the values are combined together, using some user-defined logic, into an RGB value which represents the color of the corresponding pixel. Finally, all pixel values are compiled into the final image. The pixel-level-partitioning technique is suitable for applications where there is some complex computation associated with each value that needs to be done for each pixel separately. The drawback is that it might create too many partitions, as one per pixel, which can be overwhelming for large images.

In *grid-level* partitioning, records are partitioned according to a uniform grid such that each grid cell covers a part of the image. In [29], a MapReduce algorithm is proposed to visualize satellite data (e.g., temperature) as a heat map. Records are first partitioned using a uniform grid. For each grid cell, a preprocessing step is applied to recover missing values in the input data which are caused due to clouds blocking satellite sensors or misalignment of satellites. Figure 3 shows an example of a heat map for temperature before and after recovery of missing points. The recovery technique uses a two-dimensional interpolation function which estimates missing values based on other nearby values. After that, a partial heat map is created for each grid cell by mapping each point to a pixel in the image and coloring it according to the temperature value. Finally, the partial heat maps are *stitched* together to form the final image. This technique reduces the number of partitions by applying the coarser-grained grid partitioning. Furthermore, it allows the interpolation technique to be applied as it

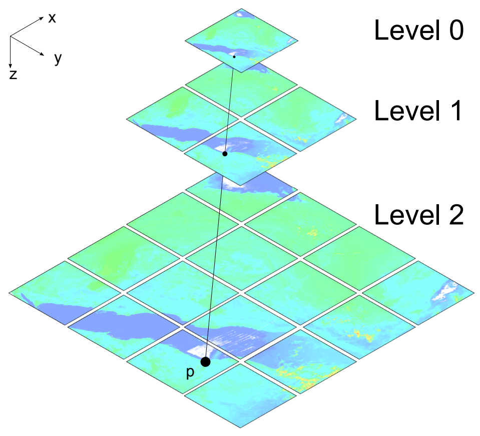


Figure 4: Mutlilevel Image

groups many points in one partition. The drawback is that the grid size must be chosen carefully to ensure load balancing and avoid too many records in one partition.

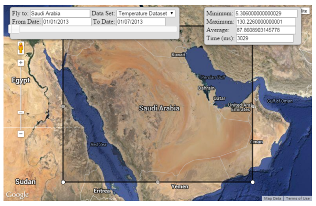
8. 2 Multilevel Image Visualization

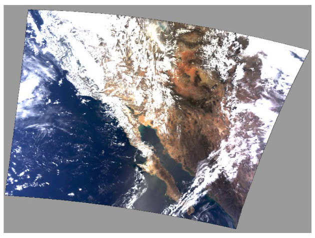
The quality of a single level image is limited by its resolution which means users cannot zoom in to see more details. On the other hand, multilevel images provide multiple zoom levels which allows users to zoom in and see more details in a specific region. Figure 4 gives an example of a multilevel image 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.

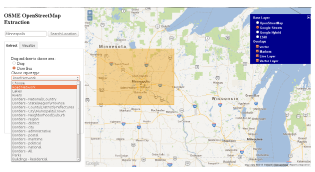
The input to this algorithm is a dataset and a range of zoom levels $[z_{min}, z_{max}]$ and the output is all image tiles in the specified range of levels. A naïve approach is to use any of the single level image algorithms to generate each tile independently but this approach is infeasible due to the excessive number of jobs to run. For example, at zoom level 10, there will be more than one million images which would require running one million jobs to generate all of them. A more efficient MapReduce algorithm is provided in [29] where the map function partitions the data using a *pyramid-partitioning* technique where it replicates each point to every overlapping tile in all zoom levels. For example, in Figure 4, the point *p* is replicated to three tiles in the three zoom levels. The reduce function groups points by tile and generates a single image that corresponds to that tile. A drawback to this technique is that tiles at lower zoom levels (e.g., zoom level zero) would have too many records as they cover larger regions in the input. To solve this problem, an adaptive sampling technique [29] is applied which down-samples the data according to the zoom level such that there is an average of one point per pixel in each tile. This ensures that each tile contains roughly the same number of records while covering the whole space.

9. Applications

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


(a) SHAHED


(b) EarthDB


(c) TAREEG

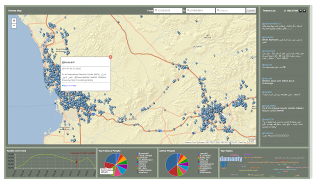

(d) TAGHREED

Figure 5: Application case studies

9.1 SHAHED

SHAHED [29] 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 5(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.

9.2 EarthDB

EarthDB [52] 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 5(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.

9.3 TAREEG

TAREEG [18] is a MapReduce system for extracting OpenStreetMap [1] data using SpatialHadoop. It provides a web interface (Figure 5(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.

9.4 TAGHREED

TAGHREED [44] is a system for querying, analyzing and visualizing geotagged tweets. It continuously collects geotagged tweets from Twitter [13] 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 5(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.

10. Conclusion

In this paper, we surveyed the state-of-the-art work in the area of big spatial data. We studied the existing system from three different angles, *implementation approach*, *underlying architecture*, and *spatial components*. We categorized the spatial components of a system for big spatial data into four components, namely, *language*, *indexing*, *query processing*, and *visualization*. For each component, we highlighted the recent work and described the different approaches to support each one. We also identified several open research problems which could be highly important for researchers pursuing research in this area. Finally, we provide some real applications that use those systems to handle big spatial data and provide end-user functionality.

Acknowledgment

This work is supported in part by the National Science Foundation, USA, under Grants IIS-0952977 and IIS-1218168.

[Bibliography]

[1] `http://www.openstreetmap.org/`.
[2] Accumulo.

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

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

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

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

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

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

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

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

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

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

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

[14] European XFEL: The Data Challenge, Sept. 2012. http://www.eiroforum.org/activities/scientific_highlights/201209_XFEL/index.html.

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

[16] 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.

[17] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-based Geospatial Query Processing with MapReduce. In *CLOUDCOM*, 2010.

[18] 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.

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

[20] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

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

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of ACM*, 51, 2008.

[24] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J. Yu. Client-Server Paradise. In *VLDB*, pages 558–569, 1994.

[25] J. Dittrich and B. Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE*, pages 535–546, 2000.

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

[27] A. Eldawy and M. F. Mokbel. Pigeon: A Spatial MapReduce Language. In *ICDE*, 2014.

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

[29] 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.

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

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

[32] 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.

[33] H. Gupta, B. Chawda, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. Mohania. Processing multi-way spatial joins on map-reduce. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT, pages 113–124, New York, NY, USA, 2013.

[34] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

[35] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB*, pages 562–573, 1995.

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

[37] R. Kothuri and S. Ravada. Oracle spatial, geometries. In *Encyclopedia of GIS.*, pages 821–826. 2008.

[38] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*, pages 497–506, 1997.

[39] H. Liao, J. Han, and J. Fang. Multi-dimensional Index on Hadoop Distributed File System. *ICNAS*, 0, 2010.

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

[41] P. Lu, G. Chen, B. C. Ooi, H. T. Vo, and S. Wu. ScalaGiST: Scalable Generalized Search Trees for MapReduce Systems. *PVLDB*, 7(14):1797–1808, 2014.

[42] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient Processing of k Nearest Neighbor Joins using MapReduce. *PVLDB*, 2012.

[43] Q. Ma, B. Yang, W. Qian, and A. Zhou. Query Processing of Massive Trajectory Data Based on MapReduce. In *CLOUDDB*, 2009.

[44] 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.

[45] H. Markram. The Blue Brain Project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.

[46] 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*, 54(6):114–123, 2011.

[47] J. Nievergelt, H. Hinterberger, and K. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *TODS*, 9(1), 1984.

[48] 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*, 31(2):289–319, 2013.

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

[50] J. Patel and D. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD*, 1996.

[51] 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*, 5(1):51, 2006.

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

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

[54] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. 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*, pages 553–564, 2005.

[55] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The Architecture of SciDB. In *SSDBM*, 2011.

[56] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.

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

[58] 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*, pages 1626–1629, 2009.

[59] J. VanderPlas, E. Soroush, K. S. Krughoff, and M. Balazinska. Squeezing a Big Orange into Little Boxes: The AscotDB System for Parallel Processing of Data on a Sphere. *IEEE Data Eng. Bull.*, 36(4):11–20, 2013.

[60] H. T. Vo, J. Bronson, B. Summa, J. L. D. Comba, J. Freire, B. Howe, V. Pascucci, and C. T. Silva. Parallel Visualization on Large Clusters using MapReduce. In *IEEE Symposium on Large Data Analysis and Visualization, LDAV*, 2011.

[61] G. Wang, M. A. V. Salles, B. Sowell, X. Wang, T. Cao, A. J. Demers, J. Gehrke, and W. M. White. Behavioral Simulations in MapReduce. *PVLDB*, 3(1):952–963, 2010.

[62] K. Wang, J. Han, B. Tu, J. D. amd Wei Zhou, and X. Song. Accelerating Spatial Data Processing with MapReduce. In *ICPADS*, 2010.

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

[64] S. You, J. Zhang, and L. Gruenwald. Large-Scale Spatial Join Query Processing in Cloud. Technical report, The City College of New York, New York, NY.

[65] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. pages 15–28, 2012.

[66] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. 2010.

[67] C. Zhang, F. Li, and J. Jestes. Efficient Parallel kNN Joins for Large Data in MapReduce. In *EDBT*, 2012.

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

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

[70] X. Zhang, J. Ai, Z. Wang, J. Lu, and X. Meng. An efficient multi-dimensional index for cloud data management. In *CIKM*, pages 17–24, Hong Kong, China, 2009.

[71] W. Zhao, H. Ma, and Q. He. Parallel *K*-Means Clustering Based on MapReduce. In *CloudCom 2009*, pages 674–679, 2009.

Ahmed ELDAWY

is a fifth year PhD candidate at the department of Computer Science and Engineering, University of Minnesota. He obtained his B.Sc. and MS in Computer Science from Alexandria University in 2005 and 2010, respectively. His research interest lies in the broad area of data management systems. Ahmed released his ongoing PhD work, SpatialHadoop, as an open source project which has been used by several companies, organizations and research institutes around the world. During his PhD, he visited IBM T.J. Watson research center, Microsoft Research in Redmond, Qatar Computing Research Institute, and GIS Technology Innovation Center in Saudi Arabia. He has been awarded the University of Minnesota Doctoral Dissertation Fellowship in 2014.

Mohamed F. MOKBEL

(Ph.D., Purdue University, USA, MS, B.Sc., Alexandria University, Egypt) is an associate professor at University of Minnesota. He is also the founding Technical Director of the KACST GIS Technology Innovation Center, Umm Al-Qura University, Saudi Arabia. His current research interests focus on database systems, GIS, and big spatial data. His research work has been recognized by four best paper awards and by the NSF CAREER award 2010. Mohamed is/was general co-chair of SSTD 2011, program co-chair of ACM SIGSPATIAL GIS 2008-2010, and IEEE MDM 2011, 2014. He is in the editorial board of ACM Transactions on Spatial Algorithms and Systems and ACM Transactions on Database Systems. Mohamed has held various visiting positions at Microsoft Research, USA and Hong Kong Polytechnic University. Mohamed is a founding member of ACM SIGSPATIAL and is currently serving as an elected chair of ACM SIGSPATIAL for the term 2014-2017. For more information, please visit: www.cs.umn.edu/˜mokbel