Efficient Spark-Based Framework for Big Geospatial Data Query Processing and Analysis

Isam Mashhour Aljawarneh, Paolo Bellavista, Antonio Corradi, Rebecca Montanari, Luca Foschini, Andrea Zanotti
University of Bologna, Italy,
{isam.aljawarneh3, paolo.bellavista, antonio.corradi, rebecca.montanari, luca.foschini}@unibo.it, andrea.zanotti9@studio.unibo.it

Abstract—The exponential amount of geospatial data that has been accumulated in an accelerated pace has inevitably motivated the scientific community to examine novel parallel technologies for tuning the performance of spatial queries. Managing spatial data for an optimized query performance is particularly a challenging task. This is due to the growing complexity of geometric computations involved in querying spatial data, where traditional systems failed to beneficially expand. However, the use of large-scale and parallel-based computing infrastructures based on cost-effective commodity clusters and cloud computing environments introduces new management challenges to avoid bottlenecks such as overloading scarce computing resources, which may be caused by an unbalanced loading of parallel tasks. In this paper, we aim to fill those gaps by introducing a generic framework for optimizing the performance of big spatial data queries on top of Apache Spark. Our framework also supports advanced management functions including a unique self-adaptable load-balancing service to self-tune framework execution. Our experimental evaluation shows that our framework is scalable and efficient for querying massive amounts of real spatial datasets.

Keywords—querying spatial data; MapReduce; big data; spark

I. INTRODUCTION

Today's proliferation of ubiquitous positioning devices and technologies has simplified the collection of spatial data at an exponential rate. Also, the large-scale spread of mobile devices, such as smartphones and sensor-enabled devices, has encouraged the participatory collection of a massive amount of geospatial data, specifically in the so-called Smart Cities [1]. Consequently, the demand for analyzing such big spatial data has become increasingly a necessity, to extract knowledge that facilitates better decision making, which is beneficial for diverse life fields.

Advanced analysis techniques are essential to derive a useful knowledge from big data, which led to the emergence of several big data management and processing systems. Those are typically based on a functional-based programming paradigm called MapReduce [2]. The first open-source realization of MapReduce was Hadoop [3]. However, using Hadoop nearly for a decade, it was found that it has its limitations, which motivated the introduction of alternatives, among which one has soon became a benchmark in big data processing, the so-called Apache Spark [4]. Spark has introduced Resilient Distributed Datasets (RDDs) [5], as sets of objects partitioned among machines of a cluster. Various features of RDDs enabled Spark to outperform its predecessors. For example, RDDs are lazily evaluated, meaning that the execution of a transformation operation is delayed until an action operation is invoked on the same RDD, therefore avoiding storage and IO overheads, and thus optimizing overall performance. However, while Spark works nicely with big data of various fields, spatial data processing is intrinsically more complex and needs an integrated support for customized implementations. Therefore, Spark's spatial extensions have been developed to simplify geospatial data processing. For example GeoSpark [6], which basically runs exactly as Spark, but with the awareness for geospatial data. Some other competitors are based on Hadoop like SpatialHadoop [7], and Hadoop-GIS [8].

The accelerated advancement of sensor-enabled data collection technologies and the high availability of cost-effective commodity computers have enabled the capture of an extra tremendous amount of geospatial data, which is beneficial for various fields of scientific research. For example, smart city environments have emerged and became common in the last two decades, where the examination and analysis of data generated by participants enables solving problems related to the continuous population growth, aiming to provide citizens with useful and efficient services, and thus improving the quality of their lives. However, to realize this complex system, it is necessary to develop an efficient infrastructure to collect information in the urban context, which requires dividing tasks among participants in an effective way that ensures their willingness to participate. To achieve this, a clustering algorithm has to be utilized. However, this involves a complex querying for a huge amount of geospatial data in an efficient manner in terms of time and storage. Such queries often involve tens of millions of geospatial objects and expensive geometric calculations to be processed with an efficient response time. Due to their incapability to scale, traditional database management systems are no longer useful for querying such huge amount of spatial data [9]. Also, extensions like GeoSpark do not include an integrated support for clustering methods, or customizable modules for specific application requirements. Further, GeoSpark did not consider important issues like data load balancing in specific application scenarios. Hence, it is essential to add a top-layer that provides services for alleviating costly geometric computations and minimizing the overhead that may be caused by unbalanced data loads.

To tackle the aforementioned problems, we have designed a framework for supporting distributed, scalable and self-adaptable querying of big geospatial data. To the best of our knowledge, this is the first work that addresses the querying of big geospatial data using a framework that utilizes Spark,

together with its geospatial extensions, aiming to efficiently minimize query response time. Our framework introduces a novel service layer on top of GeoSpark, which facilitates geospatial processing. For experimental purposes, we have used a real dataset. Running experiments on an Amazon's EC2 cloud shows that our framework has optimized the overall performance.

II. BACKGROUND

Systematic processing of big spatial data involves complex and repetitive geometric computations, where traditional systems face bottlenecks. For example, clustering participants in smart cities Mobile CrowdSensing (MCS) campaigns offers an opportunity to devise solutions for optimized participant's selection in a process that typically involves complex data management, querying and clustering [10]. To answer these new needs, some seminal geospatial extensions have recently been introduced, aiming to integrate with current big data processing systems.

A. Spark and Geospatial Extensions

Apache Spark [4] is an open-source realization for the MapReduce framework that aims to process huge amount of data efficiently in a parallel fashion. It is an efficient general-purpose solution for processing disk-resident, memory-resident and big data streams (micro-batches). The core programming abstractions of Spark are RDDs, which are groups of objects partitioned across multiple computing resources for parallel manipulation. Spark's jobs include constructing new RDDs, RDD's transformation, or calculating a result by invoking a function on RDDs [11].

Spark provides high-level APIs for various programming languages such as Java, Scala, and Python. It allows programmers to develop a complex data pipeline system, parallelizing multiple statement's processing flows through the Directed Acyclic Graph (DAG) pattern. Also, it supports sharing of in-memory type information through the DAG, so that multiple processes can run efficiently on shared data. It has been developed to overcome major limitations of the Hadoop MapReduce implementation. For example, the inefficiency of processing iterative jobs in Hadoop, as many common machine learning algorithms apply functions repeatedly on the same dataset for optimizing one or more parameters, while each iteration can be expressed as a MapReduce process, each time the job executes, it must reload the data from the disk, causing a significant loss in performance.

Even though Spark outperforms its predecessors for processing big data, it is still not optimized for specific application scenarios, like geospatial data analysis, and that led to the emergence of specific extensions built on top of Spark. GeoSpark [6] is an open-source framework, designed to be able to process large amounts of spatial data on a large-scale. It extends the classical Spark's RDD in the form of SpatialRDD (SRDD) and partitions their elements across multiple machines, through introducing concepts of space operations in parallel. Those are geometric operations that obey the recognized Open Geospatial Consortium (OGC) standards. GeoSpark also extends the SRDD layer to perform spatial queries, as the Type Range, KNN and Join, on a set of large-scale geospatial data.

GeoSpark consists of three layers, namely; i) Apache Spark, ii) spatial RDD and iii) spatial query processing layers, from the bottom to the top layer. The bottom layer is responsible for performing basic Spark functions, in addition to the recovery and storage of persistent data (e.g. Local disk or HDFS). The middle layer contains four new types of RDDs; those are; PointRDD, RectangleRDD, PolygonRDD and CircleRDD, where for each of them there is a dedicated library of geometric operations. This layer also uses spatial indexes such as Quad-Tree and R-Tree. The top layer is the one that deals with executing spatial queries over large scale datasets. In particular, after creating an SRDD, the user can invoke query operations directly, through this layer, where GeoSpark processes query requests and returns results to user.

GeoSpark has been designed to be a generic framework for processing big spatial datasets. However, many optimization issues are still to be solved in this context. For example, the load-balancing, which can be defined as a problem where various nodes of a parallel-processing system might be assigned data loads in an unbalanced manner, causing an overall response-time lateness. Another issue is that every spatial management system collects data in a format that may significantly differs from others, thus preparation of the data before sending it to the computing nodes is indispensable. However, GeoSpark did not provide services that can be specialized for various spatial datasets preparation's scenarios. Moreover, GeoSpark did not encapsulate a full-fledge library of spatial data querying services. For example, density-based clustering is an interesting method for clustering data in various fields, including geospatial processing applications. Simply put, in this method, clusters are those regions which have densities higher than surrounding areas, where latters are considered noise or border points that separate clusters. However, GeoSpark does not contain customizable querying services that might be utilized for an efficient application of density-based algorithms on spatial datasets. To address these issues, it is essential to build a framework that extends Spark and GeoSpark, with the principle goal of optimizing parallel big spatial data querying and analysis.

The next subsection shows potential queries that can be used by any algorithm for processing big spatial data.

B. Spatial Query Types and Associated Challenges

A support for various spatial query types is essential to optimize the running time of calculation-intensive algorithms. Spatial queries include, selection, join, proximity detection and pattern discovery. Selection queries can involve filtering of some spatial objects based on predefined constraints. Join queries, like intersection and union, discover interrelations between subsets from spatial objects. Proximity queries detects proximity among spatial objects that forms a point of interest. For example, MCS's participants who are in a spatial proximity normally construct groups near interesting locations of a city, where they share correlated interests. An example query may try to find participant's density distribution for

various locations in a city. Pattern discovery means identifying patterns that significantly differ from others.

Nowadays, traditional systems failed to respond to the needs for querying unprecedented increasing amounts of spatial data, and the pressing needs for minimizing query response time. As a consequence, the community shifted to parallelizing spatial queries by leveraging the MapReduce approach, aiming to optimally enhance query response time.

Moreover, algorithms for processing spatial data can involve one or many queries of various types. For example, a clustering algorithm may employ selection, proximity and join queries. Based on this fact, with the addition to the fact that such algorithms normally employ repetitive structures, it is obvious that processing big spatial data is tremendously expensive in such a way that there is always a space for further enhancements and optimizations. To put it in a precise way, the application of big spatial data processing algorithms differs from batch processing, when applying them in a distributed manner. As those algorithms need to partition datasets across various nodes in a distributed processing environment, load-balancing is an optimization problem, where the aim is to distribute datasets across various computing resources in a way which guarantees that all nodes will nearly complete processing at the same time, thus avoiding an unbalanced loading, and therefore decreasing the response time, increasing the throughput, and enhancing the overall performance. Our work falls into this context, specifically designing a framework that incorporates various mechanisms for optimizing spatial query performance. In the next section, we elaborately describe our framework with a comprehensive real scenario.

III. FRAMEWORK ARCHITECTURE

A. Architectural Overview

Spatial datasets are heavily skewed, which means that the density of object's in some regions of a geographical space is higher than that of boundaries, thus one of the challenging tasks is to properly balance loads when distributing spatial data across multiple nodes for parallel querying. Further, parallel spatial queries incur extra overheads caused by the need to repeatedly exchange data among processing nodes.

In this paper, we have designed a framework for supporting costly big spatial data queries, with the awareness for density-based spatial data partitioning, especially in heavily skewed datasets. Also, a special treatment has been introduced for facilitating an efficient data load balancing. The main objective of our framework is to provide a customizable service for load balancing across computing resources, which is mainly based on the distribution density of spatial objects.

Our framework integrates and significantly extends Spark and GeoSpark, and consists of the layered architecture depicted in Figure 1. At the lowest layer resides Spark, providing tools for data discovery, RDD's creation, and standard operations to process them, including many different primitives from MapReduce approach. At the middle layer, there is GeoSpark, providing a fully integrated support with the underlying framework, and offering advanced processing methods by defining the generic data acquired by Spark as geospatial data, through the Point class defined herein, and the resulting PointRDD container. In addition, at this stage there are also modules for data mining analysis such as the SpatialKNNQuery. At the top layer, there is our novel service layer, consisting of many modules as better explained in the following.

Services include geospatial dataset preparation, self-adaptation, and querying. First, some spatial datasets contain objects in a format that differ from those acceptable by GeoSpark. Preparation means applying customizable modules for reformatting input dataset so that it complies with specific application requirements and GeoSpark input's constraints.

Self-adaptation service also exists in this layer, which enables adaptive partitioning of spatial datasets, with the final goal to minimize the processing time. To be more specific, for every session of an application, a self-tuning module developed by us will be utilized to optimize division factors for subsequent sessions. This means an adaptive partitioning mechanism for subsequent sessions, which will notably balance loads among participating nodes, hence optimizing usage of node's resources to avoid overloading specific nodes on behalf of others. In other words, a typical situation in the execution of a MapReduce task is that an unbalanced distribution of data might cause a processing delay at one of the nodes in a cloud environment, and that in its turn will cause a delay for the whole process, despite that other nodes might have already completed their parts, so a self-adaptive service is required to optimize the whole process. Finally, querying comprises the interconnection between the developer and lower layers, where spatial queries will be forwarded to GeoSpark, while other queries will be sent to lower layer directly. This service also includes a module to translate spatial queries into corresponding GeoSpark's jobs.

In the next subsection, we describe the application of our framework for spatial query optimization in a real scenario.

B. Example Application Scenario

To test the capabilities of our framework in supporting the querying and analysis of geospatial data-intensive sets, we have applied it to a real scenario. ParticipAct is a project of the University of Bologna, which aims to study the potential cooperation between citizens, leveraging smartphones as a tool for interaction and interconnection [12]. The project had achieved a large-scale spatial data collection using smartphone's sensors. Yet, only few experimental efforts have been made to address the problem of assigning MCS data collection campaigns to users in real scenarios [10]. This problem can be defined as identifying potential participants for each task, hence increasing the probability of a willingness of selected users in participation. One method for achieving this is to employ an efficient clustering algorithm that can cluster users based on historical records captured earlier. To be more specific, clustering here signifies the grouping of users on a location-based style, based for example on the density of their distribution. However, with a huge amount of geospatial data to be experimented, applying traditional clustering methods in a batch-processing and local style is not

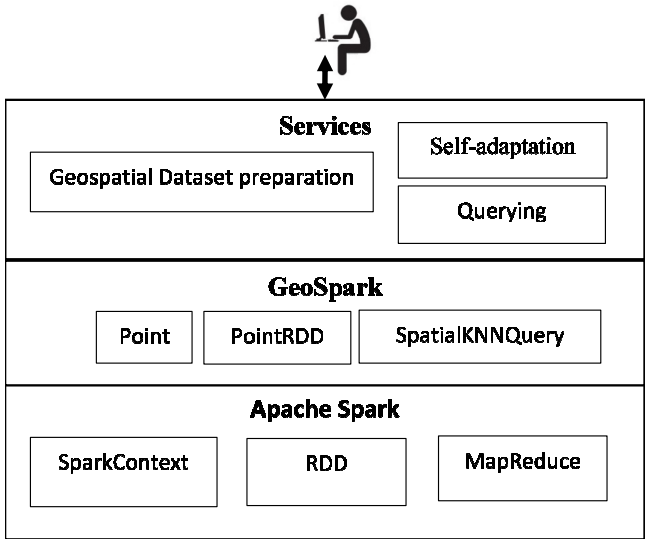

Fig. 1. Framework Architecture.

feasible. Consequently, tailoring a clustering method so that it works in compliance with the MapReduce fashion is necessary. Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [13] is an efficient clustering algorithm that is based on identifying clusters as high density regions, separated by areas with a low density, referred to as a noise. However, the performance of DBSCAN degrades dramatically with an increasing large source datasets, because of the execution complexity derived from repeated iterations on the initial dataset.

As a solution for the aforementioned traditional DBSCAN's limitation, [14] proposed a variation called DBSCAN-MapReduce (DBSCAN-MR), which aims to solve the scalability problem of the traditional technique for scenarios that contain increasing number of input elements, subdividing the initial dataset on multiple nodes, and executing on each of them the classical DBSCAN algorithm. In this way, the complexity of the problem is reduced, since it is no longer the complexity of running on the entire dataset, but only of a single portion, where each partition is processed within MapReduce operations. However, DBSCAN-MR has not been designed with the goal of optimally solving the load-balancing problem. Add to this the fact that it is a generic application for the MapReduce, hence preparation of spatial datasets has not been considered. For these reasons, we have specialized three services from our top layer: i) preparation as *filtering service*; ii) *querying service* that optimizes several spatial query types and transforms queries into GeoSpark's, and sequentially, Spark's jobs; and iii) *self-adaptation service*.

The *filtering service* reads input data from a geospatial dataset and utilizes GeoSpark to construct a PointRDD. Since not all points satisfy requirements of our scenario, filters that operate on all parameters of each point have been provided in the service layer. First, a user filter allows to filter the input database by selecting only points related to a single user or a set of users. Then, an accuracy filter enables threshold values of accuracy that can be set so that only points that satisfy will be considered. Moreover, a date filter allows to define start date and end date within which to consider the input elements, making the analysis restricted to a specific period. Finally, a motion filter supports filtering the items that are in motion, by eliminating consecutive elements that are recorded for the same user. Elements that meet all desired constraints are saved in an RDD; we use a module called ParticipActPoint, specifically designed for this purpose. Thereafter, GeoSpark's Point class will be used for simplifying the structure of each element keeping only the longitude and latitude.

Focusing on the second *querying service*, it realizes the DBSCAN-MR algorithm that in its turn incorporates various types of spatial queries. First, the partitioning phase breakdowns internal elements of **`PointRDD`** on different nodes in a cluster or cloud, where every node clusters a local subset of data. We have defined a **`ManagePartitions`** module with two different methods of data partitioning; **`prepareVertical`**, which subdivides the RDD to form parallel partitions, and the second one makes a grid partitioning. Both methods require an RDD input of points, and a parameter for the operation of the algorithm DBSCAN, namely Eps used later to determine the maximum distance for the creation of the cluster, and in particular, to decide the width of the edges to replicate (2Eps). In the subsequent mapping phase, classical DBSCAN is applied to local data of each node. Once the algorithm examined all points in all nodes, the output of the mapping returns a new RDD, this time with the key ID of the point and the **`CompletePoint`** (a module we defined to reformat points). Finally, the **`Reduce`** function groups together all elements that share the same ID replicated on multiple partitions, which determine the union of temporary clusters located in different partitions that will be merged in a later stage. Results from the reduce phase are merged to find out the cluster's global structure. In the **`Relabeling`** phase, each core local point that belongs to a global cluster is relabeled to identify the resulting cluster. We have used Spark's broadcast variable for this purpose.

Finally, the *self-adaptation service* is applied to enforce the load balancing of data across the computing nodes. At each attempt, a new configuration of cuts that balances the timing within partitions is automatically calculated. In particular, this will define new cutting factors for subsequent sessions of the algorithm, thus balancing the local execution time in all partitions, therefore reducing the overall time and optimizing the overall performance of the algorithm.

IV. Experimental Results

Moving from boards to the reality, to analyze the performance of our framework, we have tested the application of the DBSCAN-MR utilizing service from the top-layer of our framework, with a focus on self-adaptation, and using the scenario discussed earlier. By doing this, we test the integration of our added features with those offered by GeoSpark.

Our experimental setup utilized Amazon AWS cloud's computing services, specifically Amazon's EC2 service, where 5 nodes have been used for deployment, one master and four slaves. On each node, Spark 1.6.2 was installed, and Ganglia 3.7.2 was used for performance analysis. Our input database consisted of 250,000 spatial objects collected through

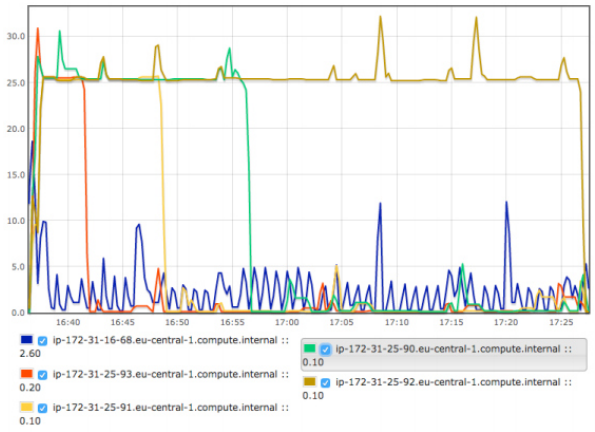

Fig. 2. First Execution of DBSCAN-MR.

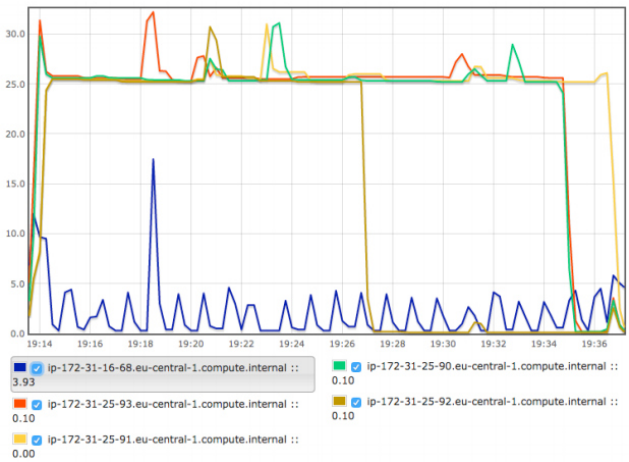

Fig. 3. Tenth Execution of DBSCAN-MR.

the ParticipAct project [12]. Each point in the ParticipAct database includes the following attributes: i) the *data_id* parameter is a unique sequential integer for each record in the database, it is the id of the point in the ParticipAct database; ii) two timestamps, *received_timestamp* and *sample_timestamp*, the first marks the time when the point has been received from ParticipAct platform, while the second represents the time when the data was actually sampled; iii) the *accuracy* represents the degree of accuracy, and the reliability of the information received; iv) *latitude* and *longitude* that represent the coordinates of the point; v) he *provider* parameter that can be of three types: gps, network or fused, depending on the type of detection instrument used to locate the spatial object; and vi) the *user_id* parameter, which represents a single user.

We performed ten experimental sessions on the same dataset, keeping the parameters of DBSCAN-MR algorithm fixed, but automatically changing the cut factors with each session, aiming to optimize the size of partitions, and therefore the position of latitude's edges, and consequently speeding up the entire process of determining the resulting cluster. This process was carried out automatically by our support through partition configurations provided by the self-adaptation service that stopped after ten iterations because there was no further major benefit (the overall time improvement was below 1%) in further adjusting the configuration cuts.

Particularly, Fig. 2 and Fig. 3 show the percentage of CPU usage at the four slaves for the whole experiment duration for the first (Fig. 2) and tenth (Fig. 3) execution iterations. It is evident that the total time has been optimized in terms of the termination time on each partition; in fact, after ten iterations each partition completed its local execution of the classic DBSCAN with nearly a synchronized time comparing to others, thus reducing the total running time of the DBSCAN-MR.

Delving into finer details, Fig. 2 shows the first (non-optimized) iteration where the total length is 50.1 minutes. Each line represents a node and it is possible to note the end of execution of each of them. In particular, the first three ended execution early, meaning that in the fourth node there is a higher concentration of the cluster. Initial configuration of the cuts, chosen as default and evenly distributed values are, respectively, 0.25, 0.50, and 0.75.

Fig. 3, instead, shows the tenth iteration, and thanks to our automatic management operations the total time length has more than halved to 22.7 minutes. This is because our self-adaptation service successfully balanced the data distribution in order to have the execution time of each partition approximately similar. The resulting final configurations of the cuts are 0.27, 0.43 and 0.52.

Finally, Fig. 4 shows the timing in relation to three execution iterations: the first, the fifth, and the tenth. As shown, the execution time continuously improves and reduces from 50.1 minutes (first iteration) to 22.7 minutes (tenth iteration), obtaining a percentage of speed-up improvement equivalent to 54.7%.

V. Related Works

Many Hadoop-based frameworks can be tackled in the literature, [15] presented HadoopDB, which is an integration between Hadoop and Postgres Spatial, aiming to facilitate computations on huge spatial data. However, their system employs Hadoop native libraries for processing geospatial data extracted from a Postgres spatial database, hence a big effort is required for executing spatial geometric computations. In addition, they did not provide services for load-balancing, thus efficiency degrades at execution time.

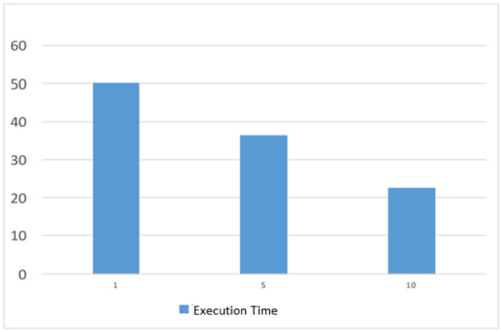

Fig. 4. Query Performance Optimization using our Framework.

Also, [16] implemented a VegaGiStore system on top of Hadoop, aiming to support various concurrent spatial queries for real-life applications. However, their spatial data partitioning method did not consider dependable load-balancing, which leads to significant loss of efficiency in case of applying computation-intensive clustering algorithms.

A method has been developed by [17] to efficiently process big spatial data queries by introducing a hierarchical spatial index, that is implemented on top of HBase. Also, it provides prefix matching filtering service, to filter spatial objects based on some constraints. However, their implementation did not consider the possible incorporation of other services, so it is not a general-purpose framework and cannot be adapted to be incorporated with emerging big data processing systems like Spark.

A geospatial data processing framework was designed by [18], aiming to better manage and process spatial data in a distributed fashion with an awareness for extensibility and adaptability. They based the framework on Hadoop, with the goal of an effortless adaptation of existing spatial algorithms to the distributed paradigm. However, it does not consider a self-adaptation service in the top layer. Also, it is based on Hadoop, thus extending it to handle online spatial data streams will require enormous efforts.

Nowadays, shelves have become increasingly saturated with geospatial query processing engines, thus choosing the correct solution become harder. Yet, most systems and frameworks of the literature have considered only Hadoop. As far as we know, there is no fully optimized solution that is built on top of Spark and its spatial extensions, that aims to optimize geospatial query processing with a prioritization for load balancing.

VI. Conclusions and Future Work

The ever-increasing diffusion of sensor-enabled portable devices enabled the collection of an unprecedented huge amount of geospatial data, where analysis has become essential for knowledge discovery. However, traditional database processing systems are unable to analyze such massive amount of data reliably. Also, despite that big data processing systems appeared like Hadoop and Spark, they do not provide specific support for spatial data processing, which, in turns, led to the appearance of extensions like GeoSpark. However, Geospark does not provide a full-fledged top layer service, requiring the developer to spend more efforts in the integration with GeoSpark while executing complex query-intensive algorithms. In this paper, we have introduced a framework for efficiently querying and analyzing big geospatial data, which was plugged on top of GeoSpark, with a motivation to optimize the performance. Several experiments using real spatial data on a 5-nodes Amazon's EC2 cloud have shown a unique performance.

In the future, we envision a contribution that incorporates more services to the top layer of our framework. Those may include, other types of queries, machine learning and data mining services with awareness for geospatial analysis. Another research direction is to provide services that simplify developer's interaction with the top layer, by providing a full-fledged native library of services that are customizable for various application scenarios. Finally, there are many application scenarios nowadays that require online streams of spatial data to be analyzed on-the-fly. In this paper, we have only considered offline disk-resident datasets, but another promising research avenue is to adapt this framework so that it applies to online geospatial data streams.

Acknowledgment

This research was supported by the Sacher project (no. J32I16000120009) funded by the POR-FESR 2014-20 program through CIRI.

References

[1] G. Cardone *et al.*, "The participact mobile crowd sensing living lab: The testbed for smart cities," *IEEE Communications Magazine,* vol. 52, pp. 78-85, 2014.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM,* vol. 51, pp. 107-113, 2008.

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

[4] M. Zaharia *et al.*, "Spark: cluster computing with working sets," in *Proc. of the 2nd USENIX conference on Hot topics in cloud computing*, Boston, MA, 2010.

[5] M. Zaharia *et al.*, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, San Jose, CA, 2012.

[6] J. Yu *et al.*, "GeoSpark: a cluster computing framework for processing large-scale spatial data," in *Proc. of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, Seattle, Washington, 2015.

[7] A. Eldawy and M. F. Mokbel, "SpatialHadoop: A MapReduce framework for spatial data," in *Proc. of 2015 IEEE International Conference on Data Engineering*, 2015, pp. 1352-1363.

[8] A. Aji *et al.*, "Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce," in *Proc. of VLDB Endowment,* vol. 6.

[9] S. You *et al.*, "Large-scale spatial join query processing in Cloud," in *2015 31st IEEE International Conference on Data Engineering Workshops*, 2015, pp. 34-41.

[10] A. Corradi *et al.*, "Smartphones as smart cities sensors: MCS scheduling in the ParticipAct project," in *Proc. of the 2015 IEEE Symposium on Computers and Communication (ISCC)*, 2015.

[11] H. Karau *et al.*, *Learning Spark: Lightning-Fast Big Data Analytics*: O'Reilly Media, Inc., 2015.

[12] G. Cardone *et al.*, "ParticipAct: A Large-Scale Crowdsensing Platform," *IEEE Transactions on Emerging Topics in Computing,* vol. 4, pp. 21-32, 2016.

[13] J. Sander *et al.*, "Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications," *Data Min. Knowl. Discov.,* vol. 2, pp. 169-194, 1998.

[14] B. R. Dai and I. C. Lin, "Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition," in *2012 IEEE Fifth International Conference on Cloud Computing*, 2012, pp. 59-66.

[15] U. Bellur, "On Parallelizing Large Spatial Queries Using Map-Reduce," in *Proc. of Int. Symp. on Web and Wireless Geographical Information Systems (W2GIS)*, South Korea, May 2014.

[16] Y. Zhong *et al.*, "Towards Parallel Spatial Query Processing for Big Spatial Data," in *Proc. of IEEE 26th Inter. Parallel and Distributed Processing Symp. Workshops & PhD Forum*, 2012.

[17] K. Lee *et al.*, "Efficient spatial query processing for big data," in *Proc. of the 22nd ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, Dallas, Texas, 2014.

[18] R. Giachetta, "A framework for processing large scale geospatial and remote sensing data in MapReduce environment," *Computers & Graphics,* vol. 49, pp. 37-46, 2015.