Distributed Execution of Spatial SQL Queries

Konstantinos Giannousis*, Konstantina Bereta†, Nikolaos Karalis‡ and Manolis Koubarakis§
Dept. of Informatics and Telecommunications
National and Kapodistrian University of Athens, Athens, Greece
**kgiann@di.uoa.gr,* †*konstantina.bereta@di.uoa.gr,* ‡*nkaralis@di.uoa.gr,* §*koubarak@di.uoa.gr*

***Abstract*—The volume of available spatial data that is generated and collected has significantly increased in the last few years. A number of applications based on Map-Reduce-like systems and cloud infrastructure have emerged. These applications offer a variety of features, however they differ in terms of spatial functions, partitioning and indexing. In this paper we present our own implementation that enables spatial support for distributed execution of spatial SQL queries as part of the system Exareme. Then, we evaluate some of the State-of-the-Art existing geospatial distributed systems, emphasizing on systems based on Apache Spark. We conduct detailed functional and performance benchmarks that include corner cases that stress the systems in comparison and reveal their advantages and weaknesses in both functionality and performance.**

***Keywords*- geospatial; big data; spatial data; apache spark; exareme;**

I. Introduction

Large amounts of spatial data is currently available from multiple sources, GPS-enabled mobile phones, Sensor Observation Services, Automatic Identification System (AIS) sensors used in maritime and many more. This vast spatial data is collected and analyzed, as for example, the exploitation of a large volume of historical AIS data to estimate the location and connections of the major trade routes [1].

Domain experts (e.g., earth scientists, meteorologists, etc.) use extensively Geographic Information Systems (GIS) as they offer a user interface for visualizing spatial data and performing spatial operations. Spatial data is often stored in geospatial databases. Decades of research focused in developing techniques of efficiently storing and querying geospatial data in relational databases. Most of these research results were adopted by RDBMSs and currently there is a variety of open source and commercial RDBMSs with geospatial support (e.g., PostGIS [2], SpatiaLite [3], Oracle-Spatial [4]).

In the era of big data and with the contributions of efforts made by multiple domains, the spatial data is first class citizen (e.g., Earth observation, AIS tracking etc). Recent efforts extend distributed frameworks (e.g., Apache Spark [5], Apache Hadoop [6]) with spatial support. The authors in [7] reviewed the recent work in the area of big spatial data according to a set of key components and they are presenting many of the systems that are supporting them.

Our contributions to the state-of-the-art are the following:

- We describe the development of a spatial extension to our own Exareme system, to support large-scale spatial SQL queries.
- We evaluate the performance of our implementation using the Jackpine Benchmark [8].

This paper is organized as follows. In Section II we describe the current state-of-the-art in the area of distributed spatial data management systems which forms the ground on which our work is built. Next, in Section III we present our approach for developing a system that enables a combination of partitioning and indexing mechanisms to support spatial SQL queries efficiently. In Section IV we perform a functional and performance evaluation of our implementation compared with other, state-of-the-art distributed systems that offer spatial support. Last, in Section V we summarize the findings of our study.

II. Related work

Exareme [9] operates as a paralleled RDBMS that uses a master-worker architecture. Each parallel node is running an instance of Exareme. The core component of Exareme is a data flow system named MadIS which is based on an APSW wrapper of SQLite [10] to process queries over data that are stored (or can be virtually seen) as tables. To ensure backwards compatibility with the centralized relational model, it can be used as centralized DBMS, supporting all SQL-92 functionality that SQLite supports and serves as its back-end.

From a users point of view, the system is used as a traditional database system: create/drop tables or indexes, import external data, issue queries, while its language is based on SQL to express both intra-worker and inter-worker data-flows described with simple parallelism primitives. It uses UDFs and an inverted syntax to easily express local pipelines and complex computations.

Until now, Exareme did not provide geospatial support. In the following section we describe how we extended it to support spatial SQL queries efficiently.

Some of the recent renowned systems for distributed spatial query processing were implemented as extensions for Hadoop MapReduce such as SpatialHadoop [11], Hadoop-GIS[12], and Hadoop with ESRI's spatial extensions [13] supported with Hive. Hadoop provides a very fault tolerant environment for parallel execution, but storing intermediate results to disk increases the execution time for spatial operations. Hence, the in-memory execution model of Spark became very popular as it reduces the execution time drastically, compared to MapReduce jobs [14]. Following to this trend, many Spark-based systems included geospatial support.

In the context of this work, we will only consider some notable Spark-based systems, such as STARK [15], GeoSpark [16], Magellan [17], Spatial-Spark [18] and Simba [19] as they reportedly achieve better performance [14], [15] than the Hadoop-based systems.

STARK [15] is built on top of Spark and provides a domain specific language (DSL) that seamlessly integrates into any (Scala) Spark program. It also includes an expressive set of spatio-temporal operators for filter, join with various predicates as well as k nearest neighbor search. A density based clustering operator allows to find groups of similar events.

It is tightly integrated into the Apache Spark API so that users can directly invoke the spatial-temporal operators and their RDDs. Spatial-temporal partitioning means that partitions are created by taking into consideration the location in space and/or time. This is achieved by creating new data type and operator classes that make use of already existing Spark operations, but also extend internal Spark classes.

Two spatial partitioners are implementing Sparks Partitioner interface so that they can be used to spatially partition an RDD with the RDDs *partitionBy* method. The *Grid Partitioner* is a fixed grid partitioner that divides data space into a number of intervals per dimension resulting in a grid of rectangular cells (partitions) with equal dimensions. Because all cells have equal size, some of them may contain more data items than others. To overcome this problem, the *Cost-Based Binary Space Partitioner*, based on [20] divides the space into two partitions with equal cost (number of contained items). When the items of one partition are exceeding a threshold, it is recursively divided into two partitions of equal cost. This way, large regions with only a few items will belong to the same partition, while dense regions are split into multiple partitions.

STARK uses an R-Tree implementation for indexing the contents of a partition. In our experiments we use *Live Indexing*. While processing a partition for evaluating a predicate, the contents of that partition are indexed on-the-fly while the query is evaluated.

Spatial Joins and filters can be called directly as transformations on standard RDDs. Additionally, it allows defining custom distance functions and predicates for its operators. More specifically, it supports the following operators: `intersect`, `contains` and `containedBy`.

GeoSpark [16] is an in-memory cluster computing framework for processing large-scale spatial data. GeoSpark is used to load, process, and analyze large-scale spatial data in Apache Spark. It provides a set of out-of-the-box Spatial Resilient Distributed Dataset (SRDD) types (e.g., Point RDD and Polygon RDD) that provide in-house support for geometrical and distance operations.

GeoSpark partitions Spatial RDDs by creating one global grid file. The SRDDs are using the Apache Spark layer by extending Spark's partitioner. Global grids are low cost in either file-size or data partitioning but constructing a global grid file demands multiple coordinate sorting on the same datasets. In our experiments we used Equalgrid.

Spatial indexing uses the Sort-Tile-Recursive (STR) algorithm. It is a simple and efficient bulk-loading method for spatial or multidimensional data management using R-tree. GeoSpark's index can be persisted either in memory or in disk for later from the same program.

The following operators are supported as spatial join conditions: `Overlap`, `Inside`, and `Disjoint`. Construct operators such as `MinimumBoundingRectangle` and group operators such as `Union` are also supported.

Magellan [17] is a distributed execution engine that extends Spark SQL to provide a relational abstraction for geospatial analytics on big data. It uses database techniques like efficient data layout, code generation and query optimization in order to optimize geospatial queries. Developers can write either standard SQL or data frame queries for spatial operations, while the execution engine efficiently takes care of laying data out in memory during query processing.

It does not perform spatial partitioning but provides spatial indexing. It uses a Z-Order Curve, which is finally treated as a linear or pointer based Quad-Tree. By extending Spark SQL it inherently supports relational spatial joins. However, these joins are not handled as spatial joins by default, unless a spatial join rule is injected. Otherwise, the join will be evaluated as a Cartesian Join followed by a predicate [1]. Magellan supports the following topological functions that can be used in SQL queries : `Intersects`, `Contains` and `Within`.

Spatial-Spark [18] provides a large-scale spatial join query processing on two leading in memory Big Data systems, namely Apache Spark and Cloudera Impala [18]. To achieve this, they focus on data processing on parallel hardware like multi-core CPUs and GPUs. Spatial-Spark implements an extension of broadcast join, where the right relation is read into memory and distributed to all workers. If the relation is too big to fit in memory then a spatial partitioner is used. Two different kinds of spatial joins are supported: Broadcast Spatial Join and Partitioned Spatial Join.

Spatial-Spark can be used via the command line and run single operations or as a library in Scala programs. It currently supports the following operators: `intersect`, `contains`, `within`, `within distance`, `overlaps` and `nearest distance`.

Simba [19] extends the Spark SQL engine across the system stack to support rich spatial queries and analytics

[1]As mentioned in [17] at paragraph Spatial Joins

Table I: Functionality overview

System	Spatial Index	Spatial Partitioning	Language
Exareme	R*-Tree	Grid	SQL
STARK	R-Tree	Grid	DSL
Geospark	R-Tree, Quad-Tree	Grid	Java API
Magellan	Z-order		SQL
Spatial-Spark	R-Tree	FGP, BSP, STP	Scala

through both SQL and DataFrame query interfaces. Due to the fact that indexing of more complex geometries than points (e.g., polygons) has not yet been implemented and due to its very limited spatial support[2] we decided not to include it in our functionality and performance benchmarking which is presented in Section IV.

In the aspect of benchmarking, Jackpine Benchmark [8] is a state-of-the-art benchmark for spatial SQL queries that can support any database that can be accessed via JDBC. Since it is considered a robust, state-of-the-art benchmark platform for spatial databases, our idea is to extend the usages of its operations into distributed spatial systems to assess the performance of individual spatial SQL functions.

It is the first time that a well-established benchmark like Jackpine is used to evaluate the performance of distributed systems with spatial support.

III. Implementation

As mentioned above, Exareme follows the master-worker model. In an Exareme cluster, all nodes are running a MadIS instance. For the parts of the SQL queries that can be run in parallel, the master node of Exareme distributes these parts to the workers.

For the spatial extension of Exareme, SpatiaLite [3], [21] is used instead of SQLite.

Users can define partitioned tables, incorporating Exareme's dedicated primitives declaring the property based on which the table will be partitioned (e.g., a hash value of a column). A table partition is a set of tuples of a table. When two tables are partitioned into the same property, resulting into the same number of partitions that need to be joined, a `direct` join can be performed instead of a Cartesian product. In a direct join, the corresponding partitions of the two (or more) tables are evaluated in parallel and the union of the results is returned (elimination of duplicates is also considered). However, when joining two tables that are partitioned into different number of partition on the same property, Exareme performs re-partitioning: One of two tables will be re-partitioned on-the-fly so that a direct join can be performed or the join is implemented using Cartesian product.

Partitioning geometries, geometrical objects such as points, polygons, linestrings etc, is a more challenging task, as geometries are typically multi-dimensional values and geometric computations are typically more costly. That could be avoided if all tables that are expected to be involved in spatial joins are partitioned into the same number of partitions on the same property. But this is knowledge that we do not always have beforehand (i.e., at the time when declaring the tables).

[2]As noted in [19], at Table 1, row 'Geometric object' and at the corresponding notation 'Simba is being extended to support general geometric objects', but until now it supports spatial operations only between points, p. 3, section 2.4 Spatial Operations

Addressing the need for partitioning geometry data in a way that the need of re-partitioning is avoided as much as possible is an even bigger challenging task. When the key-column is a column containing a geometry object, re-partitioning is not trivial. Especially when geometries are polygons, as we should ensure that there are no overlapping geometries belonging to different partitions either belong in the same or different geometry tables. Also, when defining a distributed table, the table(s) that will be joined with it are not known a-priori.

In literature, several well-known spatial indices have been employed for data partitioning, either dynamic data structures such as R-trees [22], quad-trees, k-d trees, or static like grids. Using a dynamic data structure is not optimal in this setting, as maintaining a centralized index structure, e.g., an R-tree, would require that all geometries of all partitions would be stored in a single R-tree, which is both a performance overkill and it also introduces a bottle-neck in the distributed spatial query processing.

Maintaining a distributed dynamic spatial index structure, however, does not fit our case either, as given the fact that the number of partitions are defined a-priori, the fact that all overlapping geometries from all the tables that will be potentially spatially joined will be located in the same index node is not ensured.

As we want to avoid as much as possible the need to re-partition the geometry tables and at the same time to be able to execute distributed spatial queries, we chose to employ the following partitioning scheme that combines both static and dynamic well-known spatial index structures:

- We define a grid that is given to the system as a configuration parameter. The grid is partitioned into a number of $M \times N$ cells. By this way, all spatial tables that we want to partition will be partitioned using a common reference.
- When a table is defined as distributed and partitioned on the geometry column, it is partitioned in $M \times N$ parts that correspond to the respective parts of the grid that the geometries of the table overlap with. If a geometry overlaps with more than one cells of the grid, it is replicated to all cells it overlaps with. The grid will be the same for every new table we want to partition on a geometry column, so all geometries will be partitioned having the same reference point: the grid. This is how we ensure that all geometries in our distributed database will be located in partitions that correspond to the same grid cell.
- Exareme by default creates a temporary table for each

partition. In the geospatial case, we build an R-tree index on each partition. By this way, our whole structure is essentially a **grid of R-trees**.

The spatial support that is described above has been added to the functionality of Exareme not only by changing the APSW's underlying SQLite to SpatiaLite but also by extending Exareme's UDFs and the Madis Process executor. The spatial extension of Exareme is available online at: https://github.com/nkaralis/exareme.

IV. Functionality and performance benchmark

A. Functionality benchmark

In this section we describe the functionalities of the main state-of-the-art distributed systems that support spatial queries and we compare them with our implementation. Table I briefly describes the indexing and partitioning mechanisms supported in these systems, as well as the respective programming languages that they support.

Following Jackpine's approach, we are using the Dimensionally Extended Nine-Intersection Model (DE-9IM) which proposes the relationships: Equals, Disjoint, Intersects, Touches, Crosses, Within, Contains and Overlaps. The DE-9IM has been adopted by the Open Geospatial Consortium. More specifically, we are using a subset of Jackpine's micro benchmark topological relationships among geometric objects such as polygons, lines and points. The goal of the micro benchmark is to test the basic topological relationships and spatial analysis functions. A topological relationship describes how two spatial objects relate to each other in terms of topological constraints.

The queries included in the micro benchmark should provide complete, yet minimal, coverage of topological relations.

Those concerning topological relations and pair joins are shown in [8] Table III *Micro Benchmark Queries* section *Topological relations, all pair joins*. Table III shows the data models that are used for our evaluation.

The supported operations per system are shown in table II.

B. Performance benchmark

For our experiments we used the Okeanos Cloud Infrastructure[3]. The experiments were executed using a cluster of 3 VMs, consisted by 8 Cores CPU, 8 GB RAM and 60 GB Storage capacity each with Ubuntu 16.04.4 LTS, Hadoop 2.7.3, and Apache Spark 2.2.0.

The same cluster was used also for the evaluation of Exareme but without HDFS and Apache Spark. For spatial support we installed SpatiaLite 4.3.0a, based on SQLite 3.11.0.

Apache Spark's mechanisms are responsible for the distribution of the resources and partitioning of data.

[3]www.okeanos.grnet.gr/

Figure 1. Exareme compared to STARK - Line scalability with indexers

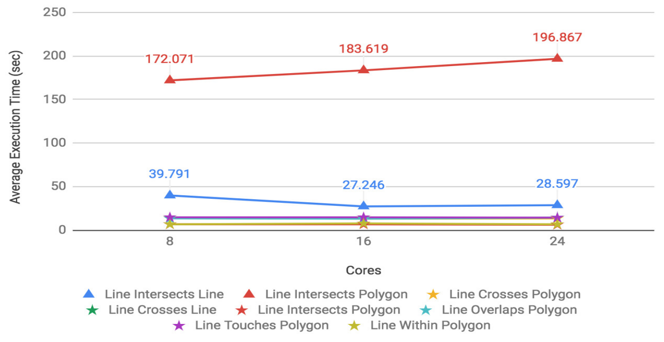
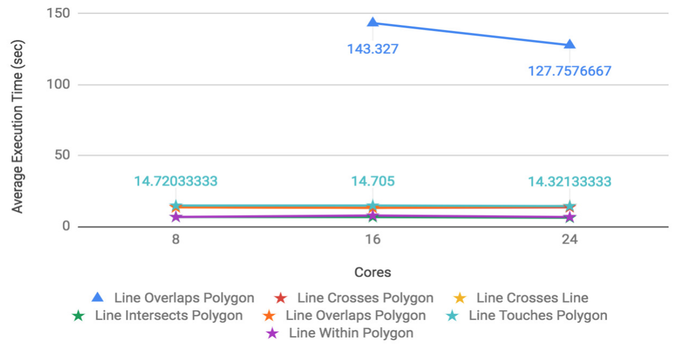


Figure 2. Exareme compared to GeoSpark - Line scalability with indexers



We set the amount of memory for the driver process (spark.driver.memory) to 10 GB and the default timeout for all network interactions (spark.network.timeout) to 1000 seconds. Finally, we set the limit of total size of serialized results of all partitions for each Spark action (spark.driver.maxResultSize) to 5 GB. This value was set in order to protect the driver from out-of-memory errors.

In Exareme, each worker is occupying a VM and consumes its resources to process the operations. The configuration and the installation was ready out-of-the-box.

C. Evaluation results

The results of the benchmark queries are described in this section. In each benchmark we run experiments 4 times, considering the first as the warmup execution and measuring the average execution time for the last 3 runs. Each benchmark is run for all systems multiple times but with different configurations. We used three partition configurations (None, 8 or 16 partitions) and three core configurations (8, 16 or 24 Cores) per run. The scenarios are comprised of 18 pairwise spatial join queries among polygon, line and point objects as in Table II and the results are shown categorized by these configurations. For the experiments we used the indexing and partitioning techniques as shown in Table I.

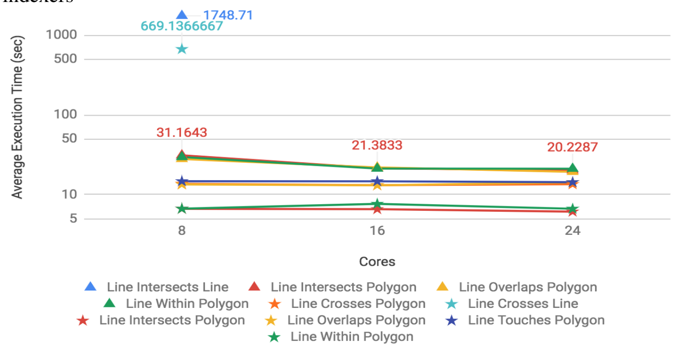
Table II: Micro Benchmark supported operations per system (Polygons)

	Exareme	GeoSpark	Stark	Magellan	Spatial-Spark
Polygon Contains Point	Y	Y	Y	Y	Y
Polygon Contains Polygon	Y	Y	Y		Y
Polygon Disjoint Polygon	Y				
Polygon Equals Polygon	Y				
Polygon Overlaps Polygon	Y				Y
Polygon Touches Polygon	Y				
Polygon Within Polygon	Y				Y
Line Crosses Polygon	Y				
Line Crosses Line	Y				
Line Intersects Polygon	Y		Y		Y
Line Intersects Line	Y		Y		Y
Line Overlaps Polygon	Y	Y			Y
Line Touches Polygon	Y				
Line Within Polygon	Y				Y
Point Equals Point	Y				
Point Intersects Line	Y		Y		Y
Point Intersects Polygon	Y		Y	Y	Y
Point Within Polygon	Y			Y	Y

Table III: Datasets used for micro Benchmark

Dataset name	Geometry	Cardinality	Data file size (Bytes)
edges_merge	line	5895060	2,034,869,136
pointlm_merge	point	13441	821,049
arealm_merge	polygon	5963	3,756,243

Figure 3. Exareme compared to Spatial-Spark - Line scalability with indexers



Due to space considerations, we could only include part of the results in the paper. All results are also available online[4]. In this paper we decided to present the results of queries that involve spatial joins with lines because, as in the respective experiment of the Jackpine benchmark, the lines' dataset is too big to fit in memory. So, we wanted to examine the behaviour of the Spark-based systems in this stressed scenarios in comparison to Exareme. For these queries, systems like Spatial-Spark and GeoSpark needed to be configured differently than others. Spatial-Spark's Broadcast Spatial Join had to be configured with more partitions and GeoSpark had to be run with enabled disk persistence for line objects to avoid error messages for insufficient memory during tests.

In figures 1, 2 and 3 we provide diagrams of the results for experiments related to line scalability. Magellan didn't return any result on these queries. Diagrams with indexers are comparing each Spark-based system (shown in each diagram with a triangle pointer) with Exareme (shown in each diagram with the star pointer). The values used to produce the results are referring to experiments with datasets of 8 partitions. This number has been selected because all systems give useful results that can be compared to each other.

In STARK we are using live index, which means that in each experiment it creates an index on the fly. In case a geometric object extends in multiple partitions, an extent information is hold for the minimum and maximum values of the object. The content of a partition is first put into an R-tree and then, this index is queried using the query object. These objects may contain extent informations if they belong to multiple cells of the partition grid. So, the elements of the R-Tree have to be checked again to see if they really match the query object.

For small datasets, this performs better but for larger datasets, depending the query object, it creates some latency due to the number of elements and the network communication as it can be seen at experiment *Line intersects Polygon* fig. 1.

GeoSpark, is using an R-Tree index per each partition, after data are partitioned. This is supposed to be fast but in our experiments it increases the communication between different partitions. Especially, for larger datasets, it has a significant performance impact i.e. *Line overlaps polygon* fig. 2).

In general, each system performed differently in various tests. Considering the number of operations, Exareme implements almost all of them, apart of 'Line Intersects Line' which never finished. Exareme's results are within a steady

[4]https://github.com/kgiann78/GeoSpatial-Distributed-Systems

range for all different configurations and in most of the experiments.

Considering the number of operations, Spatial-Spark comes right after Exareme. When the datasets are small, Spatial-Spark with Spatial Broadcast Join performes better than others. Unfortunately, it performes poorly in cases where a large dataset was member of a spatial operation.

On the other hand, STARK implements less operations than Exareme and Spatial-Spark but performs better in many cases. Of course, it should be taken into consideration that the number of partitions is larger than other systems with the same nominal number, due to it's grid partitioning. GeoSpark and Magellan implement the less operations, without many differences to Exareme or other systems.

Magellan, provides a clever way to enhance Spark SQL with spatial support but the lack of spatial partitioning and the limited number of operations didn't give much to the comparisons.

V. Conclusions

In this paper we introduced an extension to enable spatial support for the Exareme system and process spatial SQL queries. We also performed a detailed functional and performance evaluation of systems that offer similar functionality. The outcome of the benchmarks shows that each system supports different spatial operations that can be used in SQL queries, with Exareme implementing almost all of them. Exareme shows stable performance in query execution times, despite the fact that other systems were too close in execution times.

References

[1] G. Spiliopoulos, K. Chatzikokolakis, D. Zissis, E. Biliri, D. Papaspyros, G. Tsapelas, and S. Mouzakitis, "Knowledge extraction from maritime spatiotemporal data: An evaluation of clustering algorithms on big data," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec 2017, pp. 1682–1687.

[2] PostGIS. (2018) PostGIS homepage. [Online]. Available: https://postgis.net/

[3] A. Furieri. (2018) SpatiaLite homepage. [Online]. Available: https://www.gaia-gis.it/fossil/libspatialite/home

[4] Oracle. (2018) Oracle-Spatial Homepage. [Online]. Available: https://www.oracle.com/technetwork/database/options/spatialandgraph/overview/index.html

[5] Apache Spark. (2018) Apache Spark Homepage. [Online]. Available: https://spark.apache.org

[6] Apache Hadoop. (2018) Apache Hadoop Homepage. [Online]. Available: https://hadoop.apache.org

[7] A. Eldawy and M. F. Mokbel, "The era of big spatial data," in *Proceedings of the International Conference on Very Large Databases, VLDB 2017, Munich, Germany, August, 2017.*, 2017.

[8] S. Ray, B. Simion, and A. D. Brown, "Jackpine: A benchmark to evaluate spatial database performance," in *2011 IEEE 27th International Conference on Data Engineering*, April 2011, pp. 1139–1150.

[9] Y. Chronis, Y. Foufoulas, V. Nikolopoulos, A. Papadopoulos, L. Stamatogiannakis, C. Svingos, and Y. E. Ioannidis, "A relational approach to complex dataflows," in *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016.*, 2016.

[10] Hive. (2018) SQLite Homepage. [Online]. Available: http://www.sqlite.org/

[11] Spatial-Hadoop. (2018) Spatial-Hadoop Homepage. [Online]. Available: http://spatialhadoop.cs.umn.edu/

[12] Hadoop-GIS. (2018) Hadoop-GIS Homepage. [Online]. Available: https://sites.google.com/site/hadoopgis/

[13] ESRI. (2018) Spatial framework for Hadoop. [Online]. Available: https://github.com/Esri/spatial-framework-for-hadoop

[14] S. Hagedorn, P. Götze, and K. Sattler, "Big spatial data processing frameworks: Feature and performance evaluation," in *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, 2017, pp. 490–493.

[15] S. Hagedorn and T. Räth, "Efficient spatio-temporal event processing with STARK," in *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, 2017, pp. 570–573.

[16] J. Bao, C. Sengstock, M. E. Ali, Y. Huang, M. Gertz, M. Renz, and J. Sankaranarayanan, Eds., *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015.* ACM, 2015.

[17] R. Sriharsha. (2018) Magellan github page. [Online]. Available: https://github.com/harsha2010/magellan

[18] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17, 2015*, 2015, pp. 34–41.

[19] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 1071–1085.

[20] Y. He, H. Tan, W. Luo, S. Feng, and J. Fan, "MR-DBSCAN: a scalable mapreduce-based DBSCAN algorithm for heavily skewed data," *Frontiers Comput. Sci.*, vol. 8, no. 1, pp. 83–99, 2014.

[21] Spatialite. (2018) SpatialLite Index. [Online]. Available: https://www.gaia-gis.it/fossil/libspatialite/index

[22] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, 1984, pp. 47–57.