Poster Paper

Big Spatial Data Processing Frameworks: Feature and Performance Evaluation

- Experiments & Analyses -

Stefan Hagedorn
TU Ilmenau, Germany
stefan.hagedorn@tu-ilmenau.de

Philipp Götze
TU Ilmenau, Germany
philipp.goetze@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Nowadays, a vast amount of data is generated and collected every moment and often, this data has a spatial and/or temporal aspect. To analyze the massive data sets, big data platforms like Apache Hadoop MapReduce and Apache Spark emerged and extensions that take the spatial characteristics into account were created for them. In this paper, we analyze and compare existing solutions for spatial data processing on Hadoop and Spark. In our comparison, we investigate their features as well as their performances in a micro benchmark for spatial filter and join queries. Based on the results and our experiences with these frameworks, we outline the requirements for a general spatio-temporal benchmark for Big Spatial Data processing platforms and sketch first solutions to the identified problems.

1. INTRODUCTION

In the Big Data era, almost every piece of information produced is also stored and used for analyses. The produced information can be of any kind: plain web server logs, sensor readings from home or environment monitoring, (mobile) location-aware devices that periodically report their position, complex entities in Open Data sets like Wikipedia/WikiData, or structured event information extracted from news articles and other text sources. Often these types have at least two features in common: a time and a location component. For scalable processing of large datasets, data parallel architectures like Hadoop MapReduce and Apache Spark have been introduced and have widely been accepted. However, their general data model does not take spatial or temporal relations of the data items into account and therefore cannot efficiently answer spatial, temporal, or spatio-temporal queries.

In this paper, we present results of an experimental study comparing existing solutions for spatial data processing on Apache Hadoop and Apache Spark. Particularly, we consider the Hadoop extensions Hadoop-GIS [1] and SpatialHadoop [2] as well as the Spark-based systems SpatialSpark [3], GeoSpark [4], and our own implementation STARK[1]. We investigate their features and also compare their performance in a micro benchmark for spatial filter and join queries. Finally, we will conclude with an outlook to a general spatial and spatio-temporal benchmark.

2. EXISTING SOLUTIONS FOR BIG SPATIAL DATA PROCESSING

The first approach to implement spatial operations as an extension for Hadoop MapReduce is SpatialHadoop [2, 5]. It is built on top of Hadoop and provides spatial operators for range queries, k nearest neighbors, and joins that can be integrated into any Hadoop MapReduce program. Furthermore, spatial partitioning and indexing is available, too.

Another approach that extends the plain Hadoop MapReduce framework with spatial operators is Hadoop-GIS [1]. Similarly to SpatialHadoop, Hadoop-GIS utilizes a two-level indexing: a global partition indexing and an optional local spatial indexing. The query processing engine RESQUE (written in C++), uses these indexes to identify partitions to load and to speed up processing the required partitions. The RESQUE engine provides spatial operators for filters and joins and is integrated into the Hive ecosystem.

While Hadoop is a very fault tolerant environment for parallel execution, writing all intermediate results to disk makes the execution slow. Hence, the in-memory execution model of Spark became very popular as it reduces the execution time drastically, compared to MapReduce jobs. Currently, there are two systems that implement spatial operators for Spark: GeoSpark and SpatialSpark.

GeoSpark [4,6] is a Java implementation that comes with four different RDD types: `PointRDD`, `RectangleRDD`, `PolygonRDD`, and `CircleRDD`. These special RDDs internally maintain a plain Spark RDD that contains elements of the respective type, i.e., points, rectangles, polygons, and circles. GeoSpark supports k nearest neighbor queries, range queries, as well as joins and each of these queries can be executed with or without using an index.

The main goal of the SpatialSpark approach described in [3] is to provide a parallel join technique for large spatial data sets. For this, they focus on data processing on parallel hardware like multi-core CPUs and GPUs. SpatialSpark implements a broadcast join, where the right relation is read into memory and distributed to all workers. If the relation is

[1] https://github.com/dbis-ilm/stark

Table 1: Feature comparison of Hadoop- and Spark-based Big Spatial Data processing platforms

	Hadoop-GIS	SpatialHadoop	GeoSpark	SpatialSpark	STARK
Query Language / DSL	✓	✓	×	×	✓
Spatio-Temporal Data	×	×	×	×	✓
Spatial Partitioning	✓	✓	✓	✓	✓
Indexing	✓	✓	✓	✓	✓
Persistent Indexes	✓	✓	×	✓	✓
Filter				no partitioning	
Contains	✓	✓	✓	(✓- w/o Index)	✓
ContainedBy	✓	✓	×	(✓- w/o Index)	✓
Intersects	✓	✓	✓	(✓- w/ Index)	✓
WithinDistance	✓	✓	×	(✓- w/o Index)	✓
Join	✓	✓	(✓- pred. limit.)	(✓- returns IDs)	✓
k Nearest Neighbors	✓	✓	✓	×	✓
Clustering	×	×	×	×	✓

too big for main memory, a spatial partitioning and indexing is utilized [7].

In the next sections, we will have a deeper look into the supported features and limitations of the mentioned systems and will also compare the performances of their operators.

3. FEATURE COMPARISON

The DE-9IM [8] defines all possible relations between two spatial objects and the Open Geospatial Consortium released a standard for spatial data types and operations, which is implemented in many spatial DBMS. In Table 1 we compare the five engines based on a subset of these standards and additionally include aspects like query language, spatial partitioning, indexing, and data analysis operators.

Query Language.

Hadoop-GIS is integrated into Hive and implements the SQL/Spatial MM standard and includes a complete set of predicates, according to DE-9IM that can be used with filter and join operators. For SpatialHadoop the authors introduced the Pigeon [9] language that extends Pig Latin with spatial functions. In Pigeon, fields of type `bytearray` are implicitly converted to a `geometry` type when needed. SpatialHadoop programs can also be written as plain MapReduce programs, but as we did not find any documentation we found it hard to set the correct classes to parse spatial data and set parameters for a simple range query correctly. Furthermore, SpatialHadoop provides a command line script that can be used to run a single query/join.

GeoSpark can be used via its Java API, which however does not integrate well into the Spark API. Unlike in Spark where transformations are defined as methods on an RDD, in GeoSpark users have to create extra instances of, e.g., `PointRDD` and pass in the base `JavaRDD`. For the operations, again a new instance of the operator class, e.g., `RangeQuery` has to be created which accepts the GeoSpark-RDD to work on. This makes it tedious to write complex programs and to represent a data flow. The main drawbacks of GeoSpark are these special RDDs, which can only hold geometries of one certain type (points in `PointRDD`, polygons in `PolygonRDD`, ...). On the one hand, this makes it impossible to load a dataset that contains different geometry types in one column and, on the other hand, all other columns are removed when putting the data into these spatial RDDs. This also means that it is not possible to process the data in subsequent steps since related columns such as an ID are not available anymore.

It seems that SpatialSpark should be used only via the command line and run single queries/operations. However, the internal Scala classes can be used in other programs as well, although there is very little documentation.

STARK provides an integrated DSL (domain specific language) for spatio-temporal query processing that seamlessly integrates into any (Scala) Spark program. 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.

To the best of our knowledge from the found literature and provided documentation, only SpatialHadoop and STARK are able to also process temporal or spatio-temporal data.

Spatial Partitioning and Indexing.

Hadoop-GIS comes with a recursive grid partitioning and a global index (R-tree, R*-tree). This index is stored in the HDFS and used to identify partitions that need to be loaded, i.e., that contribute to the result. Furthermore, objects within a partition (tile) can be indexed as well on demand. SpatialHadoop also employs two index levels: on a global level an index partitions data across all nodes while a local index organizes data inside each partition. The indexes hold a copy of the data to avoid random HDFS lookups [10]. The number of generated partitions is calculated depending on the input file size, the HDFS block size, and an overhead ratio. These indexes are used on read to eliminate records that do not contribute to the final result. As index structures, SpatialHadoop supports grid files, R-tree, and R+-trees.

GeoSpark comes with several partitioning techniques: R-tree partitioning as well as Voronoi, Hilbert, and fixed grid partitioning. As described in [4], it supports R-trees and quadtrees to create an ad hoc index on the RDDs. However, during the evaluation, we found that choosing quadtrees is not possible and respective settings are ignored. Persisting indexes in GeoSpark seems not to be possible, since there is no way to load and index or assign it to an RDD.

SpatialSpark includes a fixed grid partitioning, binary space partitioning (BSP), as well as tile partitioning. Indexes have to be created and written to disk/HDFS before they can be

used within a program, i.e., there is no possibility for live on-the-fly indexing.

STARK includes a fixed grid partitioner as well as a cost BSP, which ensures partitions with almost same cost (amount of data items). Indexes in STARK can be created on-the-fly within a program and can also be materialized for later use. However, in STARK an index that should be materialized can also be used within the same program, while in the other frameworks, this index has to be created in an extra run and then has to be explicitly loaded. Currently, STARK uses only R-trees for indexing.

Spatial Filter & Join Operators.

Hadoop-GIS and SpatialHadoop are DE-9IM compatible and spatial filter and join operators can be used with many predicates.

GeoSpark only provides a *contains* and *intersects* predicate for spatial filters. For spatial joins only *contains* and, for joining two point data sets, *withinDistance* is supported. For joins spatial partitioning is obligatory, but indexing cannot be used.

SpatialSpark supports spatial filter queries with the predicates *contains*, *within* (*containedBy*), and *withinDistance*. However, a spatial partitioning cannot be applied in combination with the filter operator. When querying a persistent index for these range queries the *intersects* predicate is compulsorily used. Internally, they expect RDDs with an ID and a geometry object, which are processed when calling the specific query object (like `RangeQuery` or `BroadcastSpatialJoin`). SpatialSpark does not allow other payload fields but the ID and, furthermore, the result of a join returns only the matched pairs IDs, which requires additional joins afterwards to retrieve the complete tuple in the application.

STARK includes a wide range of spatial predicates (that can also be used for spatio-temporal data) which are applicable for filter and join. While other systems neglect payload data and only work with IDs and geometries, STARK keeps any payload data throughout all operations.

Other Data Analysis Operators.

All considered frameworks support a k-nearest neighbor operator, except SpatialSpark. However, they provide a 1-nearest-neighbor join predicate. A clustering operator is only available in STARK, which implements DBSCAN.

4. PERFORMANCE EVALUATION

In the following experiments, we focus on a micro benchmark comparing the execution times for single operators with different settings. The benchmarks are executed on our cluster of 16 nodes with an Intel Core i5 processor and 16 GB RAM on each node. The nodes are interconnected with a 1 Gbit/s network. The Spark jobs are executed with 32 executors and 2 cores for each executor. The data generator, test programs, settings, as well as more experiments and results are available on GitHub[2] . To run our experiments, we first had to fix issues in GeoSpark[3]. The most important problem was that operations that use an index for querying returned the candidate set returned by the index (R-tree). We added the candidate pruning step to obtain the correct results. Furthermore, the *contains* predicate was actually a *containedBy* (the operands were swapped).

The first experiment executes a spatial filter operator over a 50,000,000 polygon data set (880 GB, uniform distribution) with a *contains* predicate to find those polygons that contain a given query point. We used all available spatial partitioners of each framework and executed the operation without indexing as well as with live (on-the-fly) index creation, if possible. In this experiment, STARK performed best with only 47 (BSP, live index) or 52 seconds (BSP, no index). SpatialSpark is very limited in its usability as a spatial partitioning is not allowed in combination with a filter. The run without spatial partitioning and indexing took 3866 seconds (more than 1 hour). GeoSpark needed 1237 seconds (20 minutes) without partitioning but was not able to process this data set at all with a spatial partitioner and crashed after several hours for each partitioner. While investigating this problem we executed the program on a smaller polygon data set with 1,000,000 entries (17,6 GB). In the best case, it took 54 seconds with Hilbert partitioning. That is the same time that STARK needed to process 880 GB. For SpatialHadoop (as a representative of the Hadoop based systems) we used their command line program, but were not able to receive a correct result: The program finished after 39 minutes with zero results. The problem is that a point query option is not available and so we provided a point as query range. A visualization of the execution times along with a more detailed analysis that includes the overhead of the partitioning can be found in our GitHub repository[2] .

Our next experiment examines the influence of the query range size to the execution time. For this, we used a point data set with 50,000,000 points and executed a filter operator with a *containedBy* predicate to find all points contained by a given polygon. While the data set has a value range of [-180, 180] for `x` coordinates and [-90,90] for `y`, we execute the filter with 5 different squares created as polygons. These squares have the side lengths: 1, 5, 10, 50, and 100. Additionally, there is a query range that covers the complete data space. Figures 1 to 3 show the execution times for all partitioner/indexing combinations. Partitioners that require setting the number of partitions in advance all use the same amount (8100). This shows the impact of the pruning step that the frameworks can take. If the query region is small, only a single or very few partitions may contain result objects and other partitions do not have to be checked. STARK makes use of this partition pruning approach where ever possible and thus, is able to outperform the others that do not seem to perform this action as they need the same time for all six queries.

In the last experiment that we show here, we analyzed the spatial join operation on two point data sets (1,000,000 points, uniform distribution) that finds equal points (same exact location) in the two data sets. Figure 4 shows the result for the Spark based frameworks with their best partitioner for the join with and without using an index. We were not able to perform this test with SpatialHadoop as the command line program crashed with an error and we did not find any helpful documentation. For GeoSpark and SpatialSpark the same partitioner performed best in both cases. However, GeoSpark has a bug for Grid and R-tree partitioning as in the final result 1 and ca. 10,000 tuples respectively were missing (we also encountered different result sizes for in each repetition of the experiment). It can also be seen that for these frameworks, one cannot benefit from

[2] https://github.com/dbis-ilm/spatialbm

[3] We further had to use version 0.3 as the newer version 0.3.2 crashed with out-of-memory errors for the same settings.

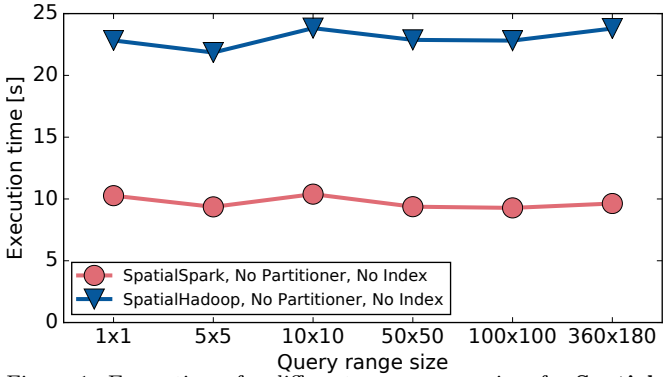

Figure 1: Exec. times for different range query sizes for **SpatialHadoop** & **SpatialSpark**

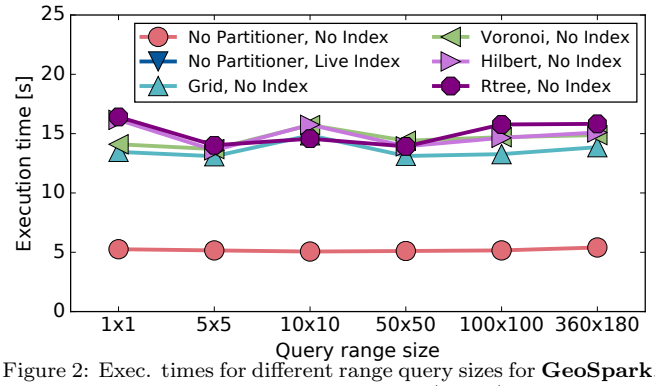

Figure 2: Exec. times for different range query sizes for **GeoSpark**. *No Partitioner, Live Index* is out of range (40 sec) for all queries.

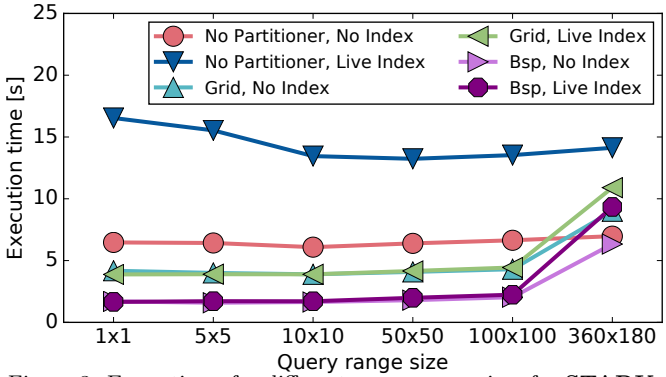

Figure 3: Exec. times for different range query sizes for **STARK**.

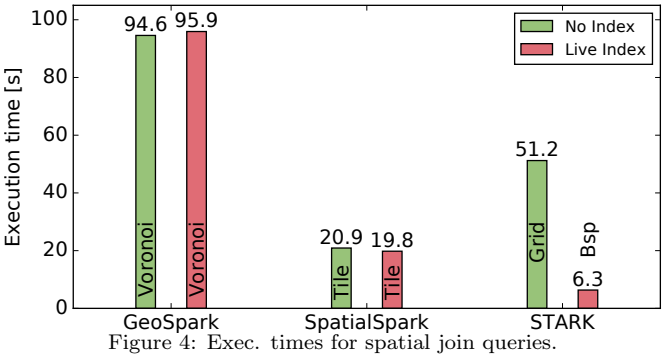

Figure 4: Exec. times for spatial join queries.

live indexing. This maybe because the speedup of querying the point index is not big enough to compensate the time required to build the index. For STARK, without using an index the Grid partitioner performed best, but was slower than SpatialSpark. With live indexing, however, the BSP was best and outperformed the others. The reason here may be that the BSP created partitions with an equal number of elements and thus equal workload on the executors.

5. CONCLUSION

In this paper, we introduced Hadoop and Spark based engines that allow processing Big Spatial Data. As the results of our feature comparison and micro benchmark show, they all differ in supported operations as well as in implementation and thus, in performance. However, the performed micro benchmark should just be a starting point for a more exhaustive spatial benchmark. For this, a more flexible data generator is needed to easily create clustered data of different sizes. Furthermore, a good benchmark should contain micro benchmark tests, as shown in the previous section, as well as a macro benchmark performing real world queries. The macro benchmark is needed to (1) evaluate the real performance and (2) to compare the functionality and usability of the system. Although the queries may be formulated in natural text leaving the task of the implementation to the authors and developers of an engine, we believe that the Pigeon [9] extension for Pig Latin in combination with our Piglet [11] for code generation will provide a good and portable user interface for such a benchmark.

Acknowledgments.

This work was partially funded by the German Research Foundation (DFG) under grant no. SA782/22

6. REFERENCES

[1] A. Aji, F. Wang *et al.*, "Hadoop-GIS: A High Performance Spatial Data Warehousing System over Mapreduce," *VLDB*, pp. 1009–1020, 2013.

[2] A. Eldawy and M. F. Mokbel, "A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data," in *VLDB*, 2013.

[3] S. You, J. Zhang, and L. Gruenwald, "Large-Scale Spatial Join Query Processing in Cloud," in *ICDE*, 2015.

[4] J. Yu, J. Wu, and M. Sarwat, "GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data." SIGSPATIAL, 2015, p. 70.

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

[6] J. Yu, J. Wu, and M. Sarwat, "A Demonstration of GeoSpark: A Cluster Computing Framework for Processing Big Spatial Data," *ICDE*, 2016.

[7] S. You, K. Gorlo *et al.*, "Big Spatial Data Processing using Spark," https://git.io/vXiMd, accessed Nov. 14, 2016.

[8] M. J. Egenhofer and R. D. Franzosa, "Point-set topological spatial relations," *IJGIS*, 1991.

[9] A. Eldawy and M. F. Mokbel, "Pigeon: A spatial MapReduce language," in *ICDE*, 2014.

[10] R. T. Whitman, M. B. Park *et al.*, "Spatial Indexing and Analytics on Hadoop." SIGSPATIAL, 2014.

[11] S. Hagedorn and K.-U. Sattler, "Piglet: Interactive and Platform Transparent Analytics for RDF & Dynamic Data," in *WWW*, April 2016, pp. 187–190.