Spatial Join Query Processing in Cloud: Analyzing Design Choices and Performance Comparisons

Simin You
Dept. of Computer Science
CUNY Graduate Center
New York, NY, USA
syou@gc.cuny.edu

Jianting Zhang
Dept. of Computer Science
The City College of New York
New York, NY, USA
jzhang@cs.ccny.cuny.edu

Le Gruenwald
Dept. of Computer Science
The University of Oklahoma
Norman, OK, USA
ggruenwald@ou.edu

***Abstract*—Data volumes of GPS recorded locations and many other types of geospatial data are fast increasing. Processing large-scale spatial joins in Cloud for performance and scalability is becoming increasingly popular. In this study, we compare three leading Cloud-based spatial data management systems, namely HadoopGIS, SpatialHadoop and SpatialSpark, both conceptually through analysis of design choices and empirically through experiments using real world datasets. Using both a workstation serving as a single-node cluster and up to 10 nodes Amazon EC2 clusters, the results show that the combined factors, including Cloud platforms, data access models and the underlying geometry libraries, have significant impacts in their realized performance. While *SpatialHadoop* generally wins on robustness, *SpatialSpark* is the clear winner of efficiency due to in-memory processing.**

Keywords— Spatial Join, Query Processing, Cloud Computing, Design, Performance

I. INTRODUCTION

The recent development of Big Data systems has motivated processing large-scale geospatial data on commodity cluster computers in a distributed manner. Spatial joins [1], such as matching taxi pickup/drop-off locations with road segments through point-to-nearest polyline distance computation or assigning species occurrence records to ecological zones through point-in-polygon test, are not only data intensive but also computing intensive. Although efficient parallel algorithms on shared memory architectures have been explored previously [2, 3], it is desirable to scale spatial join to distributed computing nodes to process large-scale spatial data that are beyond the capacities of single computing nodes. While serial spatial join algorithms typically have two phases, i.e., spatial filtering to pair up spatial data items based on the spatial relationship of their Minimum Bounding Rectangles (MBRs) and spatial refinement to remove false positives due to MBR approximation by using exact geometry to test spatial relationships between paired spatial data items, distributed spatial joins require an additional partition phase and a global join phase to distribute workload across multiple computing nodes.

Several research prototype systems have been developed for large-scale spatial query processing in Cloud. *HadoopGIS*[1] is a pioneering system that allows spatial joins and several other spatial operations on Hadoop by adopting Hadoop Streaming[2] framework and integrating several open source software packages for spatial indexing and geometry computation [4]. *SpatialHadoop*[3], on the other hand, tightly integrates spatial operations including indexing and joins into Hadoop [5]. While *HadoopGIS* is mostly a collection of modules (Java, python and C++ wrappers) that can be nicely plugged into Hadoop, *SpatialHadoop* has 14,000 lines of Java code which re-implements several popular spatial operations from scratch in Hadoop. *SpatialSpark*[4], which is a lightweight implementation of several spatial join algorithms on top of the Apache Spark[5] in-memory Big Data system, recently becomes available [6]. Different from *HadoopGIS* and *SpatialHadoop* that were built on top of Hadoop and utilize Hadoop Distributed File System (HDFS[6]) to store intermediate results for scalability and fault tolerance, *SpatialSpark* targets at in-memory processing for higher performance. As a lightweight implementation, *SpatialSpark* has a very small codebase (<2,000 lines) which makes it easy to understand, use and maintain. The three systems differ significantly in terms of distributed computing platforms (Hadoop/Spark), data access models (streaming/random/functional), languages (Java/mixed/Scala) and the underlying geometry libraries (GEOS[7]/JTS[8]).

Despite the similarities among *HadoopGIS*, *SpatialHadoop* and *SpatialSpark* from an end-user perspective where end-to-end runtime is the primary concern, it is interesting to analyze their design choices, compare their realized performance and understand how the designs and implementations affect performance. We believe this is the first work towards comparing systems for large-scale geospatial processing (or Big Spatial Data systems) and gain insights for future enhancements. Our work shares similar motivation of several previous works on performance comparisons on Big Data systems for relational data [7] and graph data [8]. In addition, although [9] provided a comprehensive evaluations of batched queries on point data (static and moving objects) using a spatial join framework on a shared memory machine, the systems we evaluate target at much larger scale spatial joins in distributed computing platforms for more complex geospatial data (such as polylines and polygons) and higher generality (both sides of a join can be any type of geospatial data).

Our technical contributions are twofold. First, we analyze the components and stages of distributed spatial join implementations in *HadoopGIS*, *SpatialHadoop* and *SpatialSpark* and provide a generalized framework to set the context for discussing various design and implementation choices that have been exploited in the three systems. Second, we prepare several datasets derived from public sources and use them to evaluate the end-to-end performance of the three systems. The rest of the paper is arranged as follows. After a brief background introduction, Section 2 provides the generalized framework and introduces the design and implementation choices adopted by *HadoopGIS*, *SpatialHadoop* and *SpatialSpark*. Section 3 presents our experiments and results. Finally, Section 4 is the summary and future work directions.

II. GENERALIZED FRAMEWORK FOR ANALYZING DESIGN AND IMPLEMENTATION CHOICES

Techniques on distributed spatial joins exist long before Hadoop and Spark were developed [1]. However, the significant burden of handling distributed data communications and spatial join logic in a tightly intertwined manner has made the traditional techniques difficult to adopt in practical applications. Both Hadoop and Spark separate data communication and processing logic to make distributed data management much easier for application developer, although in a related but different way. All the three systems to be evaluated are built on top of either Hadoop or Spark. As a consequence, their design and implementation choices are significantly impacted by the underlying platforms. Similar to the discussions in [5], we divide a complete distributed spatial join process into three stages: (1) preprocessing (2) global join and (3) local join. The components of the implementations of a complete distributed spatial join in the three systems, however, have a complex relationships with respect to the three stages.

Assuming neither of the two input datasets in a spatial join is indexed, the preprocessing stage is responsible for loading data into a distributed file system, performing data format transformation if necessary, building indexes at one or multiple levels and logically or physically partition the datasets into chunks (referred as partitions hereafter) that may be beneficial to the subsequent stages. The global join stage can be considered as a distributed extension to spatial filtering in serial spatial join algorithms. In all the three systems, input data items are first grouped into partitions and a spatial join on the MBRs of the partitions is first performed to pair up partitions whose MBRs spatially intersect, typically on a single computing node (e.g. the master node in a Hadoop or Spark cluster). The list of paired partitions is then dispatched to distributed computing nodes for the final local spatial join stage. Fig. 1 represents a generalized framework that are shared by the three systems and serves as the context for the discussions of the implementation choices.

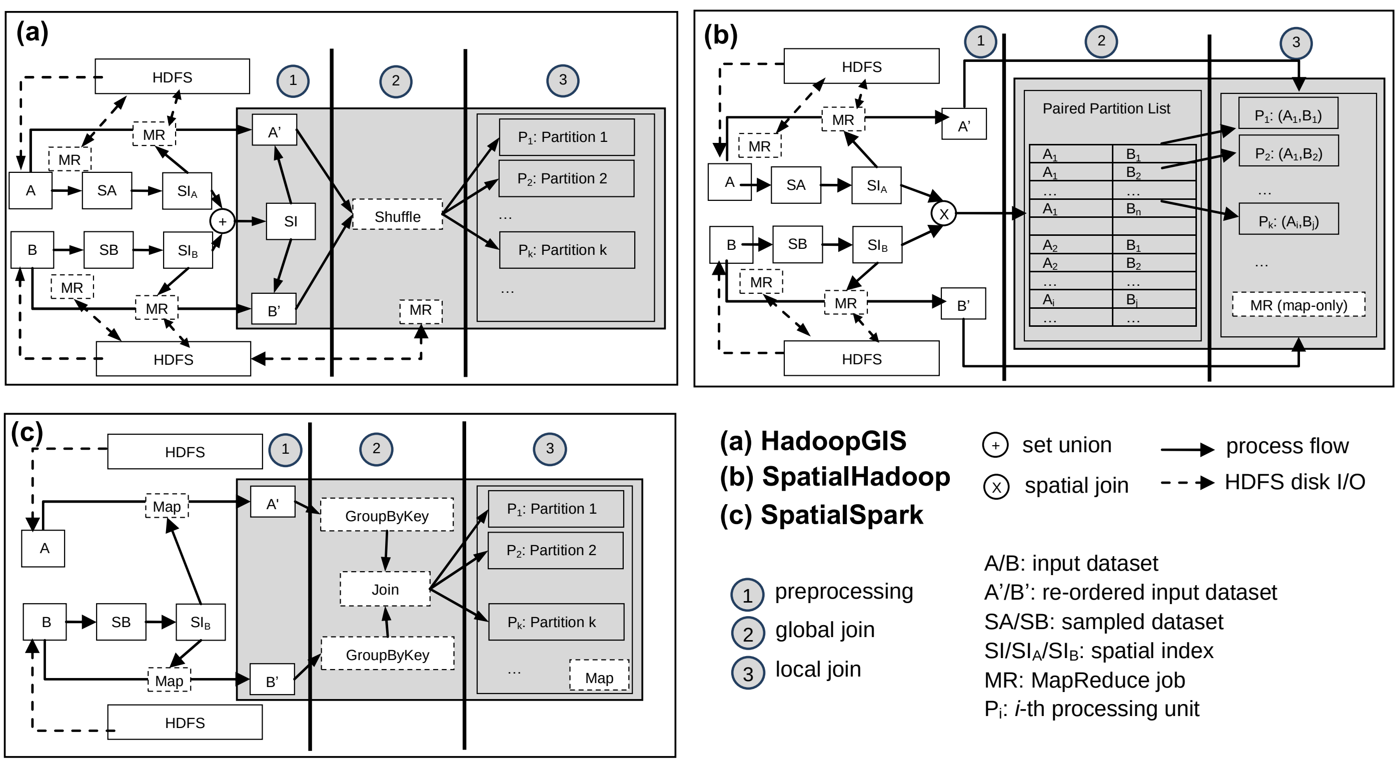

Fig. I Illustration of a Generalized Framework for Analyzing Design Choices of HadoopGIS, SpatialHadoop and SpatialSpark

Given two input datasets (A and B, respectively), subplots of Fig. 1 illustrate how the three stages are implemented in *HadoopGIS* (Fig. 1(a), top left), *SpatialHadoop* (Fig. 1 (b), top right) and *SpatialSpark* (Fig. 1 (c), bottom left), respectively. We use SA and SB to represent the sampled dataset of A and B, respectively. We also use SI to represent the spatial index that is used for global join, which can be built either from both SA and SB (*HadoopGIS* and *SpatialHadoop*) or from SA or SB (*SpatialSpark*), and used either in both the Map phase and Reduce phase (*HadoopGIS*) or the Map phase only (*SpatialHadoop*) in a MR job, or a standalone step that uses an implicit Join parallel primitive which is equivalent to the shuffle phase in a MR job (*SpatialSpark*). Note that the shaded region in Fig. 1(a) and Fig. 1(b) represent the most important (and the final) MR job in the end-to-end distributed spatial join process. The shaded region in Fig. 1(c) represents steps that are functionally equivalent to the MR job in the shaded region in Fig. 1(a) and Fig. 2(b).

While *HadoopGIS* processes data items sequentially and is transparent to partition boundaries (streaming data access), *SpatialHadoop* is aware of partition boundaries and explicitly pair up partitions based on the spatial relationships among the MBRs of partitions (random data access). *SpatialSpark*, on the other hand, adopts a data parallel (functional) model due to the underlying Spark platform. While *SpatialSpark* is similar to *HadoopGIS* with respect to sequential data access within a partition, it is aware of partition boundary as partitions are defined by keys. However, different from *SpatialHadoop*, *SpatialSpark* does not have the ability to randomly access data items within a partition and we use “functional data access” to reflect the fact that both Spark and *SpatialSpark* use Scala which is a functional programming language. The different data access models largely imposed by the underlying system infrastructures have significant impacts on many design choices and performance subsequently. While the implementation choices will be discussed in more details in the subsequent three subsections on the three stages, respectively, from Fig. 1, it is clear that *SpatialHadoop* and *HadoopGIS* have much more interactions (including reading inputs, writing outputs and shuffling intermediate results) with HDFS while *SpatialSpark* touches HDFS only when input data are read from HDFS to memory of computing nodes. As we shall see in the experiment section, reducing disk I/O to a minimum level contributes significantly to the efficiency of *SpatialSpark*.

A. Preprocessing

HadoopGIS uses the following steps to partition an input dataset: (1) a map-only job to convert the input to a tab separated file while the dataset is being loaded to HDFS, (2) a map-only job to sample data items and extract MBRs of the sampled data items, (3) a MR job to compute the spatial extent of the dataset based on the samples (with a single reducer), (4) a map-only job to normalize the sample MBRs, (5) a local program to generate partitions based on the sample MBRs, (6) a MR job to assign each input data items with an partition by querying the partitions it intersects. Although step 2-4 involve only MBRs of samples which can be significantly smaller than the input dataset (assuming sample rate is far less than 1.0), step 1 and 5 require reading and/or writing all the input data items with all attributes, which can be very expensive with respect to disk I/O. Step 6 is the most expensive step with respect to both I/O and computation for three reasons. First, each mapper needs to build a spatial index from sample MBRs. Second, since data items are stored as strings (due to the requirement imposed by Hadoop Streaming), each input data item needs to be parsed and the parsed MBR needs to query against the spatial index. Third, each input data item, in addition to its assigned partition ID, needs to be written out to HDFS to be processed in the reduce step. The reducer is currently implemented as a python script which uses a pipelined *cat-sort-unique* combination to remove duplicated data items in a partition. Note that Linux command *cat* is used to retrieve data from HDFS to feed the subsequent *sort* and *unique* modules. However, removing duplication through sorting is typically suboptimal. Furthermore, Step 5 uses a local serial program and requires copy data back and forth between a local file system and HDFS, which can be expensive as well. Conceptually, Step 1-5 can be combined to simplify both the process workflow, and more importantly, to reduce disk I/O overheads. Disk I/Os are known to be excessive (and quite often unnecessary) in Hadoop when compared with Spark which is designed for in-memory processing.

Compared with *HadoopGIS* that adopts a Hadoop Streaming framework which requires it to process input data items sequentially (as a string per line) in both a mapper and reducer, *SpatialHadoop* is tightly integrated with Hadoop which allows it to use low-level APIs for more efficient processing. Assuming that neither of the input datasets has been indexed, the preprocessing stage in *SpatialHadoop* is implemented using two MapReduce (MR) jobs. The first MR job samples both input datasets and generate partitions based on the samples. The MBRs of the partitions for an input dataset are stored in a special HDFS file that is accessible to the subsequent MR jobs due to HDFS runtime (i.e., disk-based broadcast). In the second MR job, the mapper queries against a spatial index built from the MBRs to assign a partition ID for each data item from an input dataset. By using the partition ID as the key and the geometry as the value, after the shuffle stage of the MR job, all data items that belong to the same partition are naturally written to a same HDFS block file. A spatial index for all the data items within the block can be built and written to the beginning of the HDFS block file. From a developer’s perspective, since *SpatialHadoop* has random data accesses to block files in HDFS, it is both convenient and efficient than the Hadoop steaming approach adopted by *HadoopGIS*, although much deeper knowledge of Hadoop is required to develop *SpatialHadoop* than *HadoopGIS* which can be both an advantage and a disadvantage form a development perspective. As disk I/Os are typically more expensive than computing required for spatial indexing, the intra-partition indexes are built virtually for free.

SpatialSpark supports both spatial partitioning and sequence-based partitioning. While sequence-based (non-spatial) partition does not require preprocessing and is more efficient when the left side of a spatial join is a point dataset, we will focus on spatial partitioning in this study to make it comparable with *HadoopGIS* and *SpatialHadoop*. Similar to *HadoopGIS*, *SpatialSpark* supports creating partitions through sampling using different approaches as discussed in [10] and partially implemented (on top of Hadoop) in *HadoopGIS*. As the aggregated memory of a reasonably up-to-date cluster is typically larger than both of the input datasets in a spatial join, the end-to-end sampling process can be completed in-memory without touching HDFS, which is clearly more efficient. The built-in sampling function in Spark/Scala makes the implementation much easier. Different from *HadoopGIS* and *SpatialHadoop* where data items in a dataset (that may or may not participate in a distributed spatial join) are physically reordered and stored in HDFS, reordered data items are generated for query processing only in *SpatialSpark*. It may not need to be stored in HDFS which can be efficient in both computing and storage. Again, the advantage of *SpatialSpark* is due to the in-memory processing feature brought by Spark.

As a brief summary for the preprocessing stage, *HadoopGIS* samples and partitions both input datasets using several partition techniques. *SpatialHadoop* partitions sampled data items into grid cells which can be used to pair up partition MBRs to be detailed next. *SpatialSpark*, on the other hand, samples only one input dataset and uses partition MBRs to build a spatial index to assign partitions IDs for data items on both sides of a join. *HadoopGIS* is likely to be the most expensive technique among the three due to excessive geometry parsing and disk I/Os in multiple MR jobs which can be potentially combined or simplified.

B. Global Join

Global Join is to pair up spatial partitions of both input datasets before distributing the pairs to computing nodes for local join. Implementations of global join techniques are largely determined by the underlying distributed computing platforms (i.e., Hadoop and Spark) and data access models (random or streaming in Hadoop).

In *HadoopGIS*, after spatial partition, each data item in both input datasets is assigned a partition ID. Unfortunately, the spatial relationships among the partitions in the two joining datasets cannot be derived and used for pairing directly. To spatially pair up partitions, *HadoopGIS* reuses samples of these two datasets by concatenating them and creating new partitions based on the combined data. As this step is implemented on a local machine in a serial manner, the input sample files need to be copied from HDFS to local file system and the output partitions need to be copied back to HDFS. While the input datasets are sampled to reduce computing overheads, both I/O overhead and scalability are likely to be a serious performance issue when sampling rates are high. The HDFS file that stores the locally built partitions serves as one of the inputs to the distributed join MR job, which is essentially broadcast to all map tasks in the job. Each map task builds a spatial index (using R-Tree implemented in libspatialindex[9]) based on the MBRs of the new partitions. Each item in the input datasets query against the spatial index to compute its partition ID and append the ID to the item as the output of the map task in the MR job. By using the partition ID as the key, data items that spatially belong to the same partition in both input datasets are shuffled by Hadoop runtime and sent to a reducer task. The process is very similar to the last MR job in the preprocessing stage (spatial partition) in *HadoopGIS*, except that the reducer there only needs to write out shuffled data items to HDFS while the reducer in the distributed spatial join MR job needs to perform local spatial job (to be discussed in the next subsection). The design is smart in the sense that it avoids the limitation imposed by Hadoop Streaming framework. We note that it is very difficult (and may be even impossible) to pair partitions in the two input datasets in *HadoopGIS* otherwise, as this would require accesses to individual partitions whose information is invisible to Hadoop Streaming applications. However, a serious consequence is that the partition IDs assigned to data items in the two input datasets cannot be reused in the distributed spatial join job, which is wasteful. Querying each data item in both input datasets against the new spatial index is both disk I/O intensive (HDFS) and memory intensive (traversing R-Trees). Although data items that spatially belong to a same partition may have been physically shuffled to an HDFS block for both input datasets after the preprocessing stage, they still need to go through the shuffle step in the distributed spatial join MR job. The expensive shuffle operation, however, is again wasteful. The implementation decisions in *HadoopGIS* are clearly limited by Hadoop Streaming.

Global join in *SpatialHadoop* is much simpler and is likely to be more efficient. Pairing partitions in *SpatialHadoop* is actually implemented as a preprocessing step in a MR job by overloading the *getSplits*(…) function in *FileInputFormat* class in Hadoop. For the two partitioned input datasets, *SpatialHadoop* stores the MBRs of all their partitions as a HDFS file (indicated by *_master* keyword in its file name). After a MR job is started, the master node of the job will execute the *getSplits* function defined in a custom *FileInputFormat* class (*BinarySpatialInputFormat* in *SpatialHadoop* to be precise) to generate a list of splits that will be distributed to map tasks. Given the two MBR files, any in-memory spatial join technique can be applied to spatially pair up partitions based on their MBRs. For each split that is assigned to a map task, the HDFS block file names corresponding to the two paired partitions of the two input datasets are provided to the map task which allows it to perform local spatial join (to be detailed next) within the map task. As a map-only MR job, the implementation is very efficient and is quite different from *HadoopGIS* where local spatial joins are implemented in reducers. Different from *HadoopGIS* that only data items (lines of string text) are invisible to the mappers and reducers (due to Hadoop Streaming), *SpatialHadoop* has the location information of all the blocks in a HDFS file and can randomly access the data block files. The random data access flexibility allows

SpatialHadoop to easily adapt traditional serial spatial join techniques (see [1] for a review) for global join.

The programming model in Spark gives *SpatialSpark* more flexibility in pairing up partitions in the global join stage. Recall that, in the preprocessing stage, *SpatialSpark* only partitions one input dataset (assuming the right side of the spatial join) through sampling. A spatial index is built for the MBRs of the partitions. The index, as an in-memory data structure, can be sent (through broadcasting) to all computing nodes by Spark runtime without involving HDFS, which is more efficient than *HadoopGIS* (each map task builds its own index by reading the MBR file from HDFS) and could be more scalable than *SpatialHadoop* (the master node reads MBR files of both input datasets from HDFS and perform a serial spatial join). Subsequently, in *SpatialSpark*, the data items of both input datasets are used to query the index to compute that partition ID that each data item should be assigned to. We note that while the spatial query logic is expressed at the individual data item level in *SpatialSpark*, Spark runtime system actually divides the input data items into chunks (sequentially) and dispatch them for distributed execution. This step is functionally equivalent to the map step of the distributed MR job in *HadoopGIS* but is much more efficient due to in-memory processing. The partitioned data items are subsequently grouped based on partition IDs at the both sides, respectively. This can be easily achieved by using a *groupByKey* member function of RDDs (Resilient Distributed Dataset) that is natively supported by Spark. As a result of the step, each partition is associated with a list of data items that are assigned to the partition, for both input datasets. Finally, another RDD member function (i.e., *join*) is used to join {partition ID,{Left item ID}} and {partition ID,{Right item ID}} based on partition ID and generate the {partition ID, {Left item ID}X{Right item ID}} list. A disadvantage of *SpatialSpark* is that, similar to many Spark applications, available memory capacity is crucial to its success and efficiency. When available memory is insufficient, as presented in the experiment section, SpatialSpark may fail to work properly.

We note that the partition-based spatial join technique we discuss here is different from the broadcast-based spatial join that was originally implemented in the early version of *SpatialSpark* [6]. Although the built spatial index from sampled data items in the partition-based spatial join technique is broadcast to all computing nodes using Spark distributed computing infrastructure as discussed in [6], neither data items in the dataset of the right side of a spatial join nor their full index are broadcast to the left side. The data volume of the sample index to be broadcast can be controlled by adjusting sample rate which makes the partition-based spatial join more scalable at the expense of joining partitions based on partition IDs. In contrast, in broadcast-based spatial join technique, each data item can query against the broadcast full index tree (and/or the data items they index) to pair up data items directly. We leave a thorough comparison between broadcast-based and partition-based spatial join techniques in Cloud for future work. In this study, we focus on evaluating partition-based spatial join technique in the three systems.

The combination of the two steps in *SpatialSpark* is essentially equivalent to the shuffle step of the distributed MR Job in *HadoopGIS*. However, Spark is able to maximize the utilization of aggregated cluster memory capacity for fast in-memory processing, provided that there are sufficient memory available for the application. The join result can subsequently be treated as an RDD. A RDD map function can then be used to distribute a partition to a processing unit, in a way similar to the reducer in *HadoopGIS* and mapper in *SpatialHadoop* in their distributed spatial join MR jobs, respectively. Comparing *SpatialSpark* with *SpatialHadoop*, the work that is done in the global join stage in *SpatialSpark* is essentially equivalent to *SpatialHadoop* that requires re-partitioning [5]. We note that, in *SpatialHadoop*, even though both input datasets in a spatial join may have been indexed in the preprocessing stage, when the underlying grid configurations are not compatible and the cells in the two datasets cannot be directly used for pairing, repartition is required. On the other hand, the on-demand indexing in *SpatialSpark* makes re-partitioning unnecessary, although *SpatialHadoop* can run faster when re-partitioning can be skipped.

While it is possible for *SpatialSpark* to adopt a similar strategy as what has been done in *SpatialHadoop*, i.e., assigning data items of both input datasets to partitions in the preprocessing step and using a serial spatial join algorithm to pair up the partitions (at a master node), it would require *SpatialSpark* to incorporate quite some Hadoop/HDFS APIs which is likely to increase system complexity and reduce compatibility and interoperability. From a performance perspective, assigning partition IDs to input data items in *SpatialSpark* has the same level of overhead as re-partitioning in *SpatialHadoop*. However, we argue that *SpatialSpark* can be more I/O efficient as *SpatialHadoop* requires writing repartitioned datasets to HDFS before its data items are read back for local joins. We also argue that shuffling the grouped item lists in memory (in *SpatialSpark*) should be more efficient than shuffling lists of (*partition ID*, *Item*) pairs on disks (in *HadoopGIS* and *SpatialHadoop*), especially when *Item* has a large data volume. Furthermore, a (hash) join based on partition IDs (one-to-one integer matching) in parallel in *SpatialSpark* could be more efficient than a serial spatial join on a single computing node in *SpatialHadoop*. As such, we expect *SpatialSpark* to be more efficient in the global join stage.

C. Local Join

Data items that are aligned to a same partition (regardless how the partitions are generated) is assigned to a processing unit in all of the three systems. The implementations for this stage are much similar when compared with the other two stages: first build a spatial index for data items in a partition of one input dataset and the query the index for all the data items in the other input dataset to pair up data items based on their MBRs before the final local refinement.

As only a single processing unit is assigned to a partition pair in the current Cloud computing platforms running Hadoop/Spark, no more parallelisms need to be exploited to improve performance in this stage. *SpatialHadoop* actually provides both a plane-sweep based and a synchronized R-Tree traversal based serial spatial join implementation within a partition [1]. While the same module can be implemented in *HadoopGIS*, it is natural to use indexed nested loop join in *SpatialSpark*, due to the underlying Scala functional language. Implementing a plane-sweep based serial spatial join is more difficult than implementing an indexed nested loop join in Scala but can be an interesting improvement in its future work.

In a local join, a spatial refinement [1] step can be applied by using a geometry library, i.e., GEOS[7] (Geometry Engine - Open Source) for *HadoopGIS* and JTS[8] (Java Topology Suit) for *SpatialHadoop* and *SpatialSpark*, to check the spatial relationship between the paired data items using their exact geometry. While GEOS is a language port of JTS, we have found that JTS can be several times faster than GEOS [6]. As shown in the experiment section, this might be a major factor in causing *HadoopGIS* to have long runtimes, in additional to the inefficiency caused by Hadoop Streaming.

III. EXPERIMENT AND RESULTS

A. Experiment Setup

In order to conduct a reasonably comprehensive and fair performance study, we have prepared several real world datasets that are publically available for two experiments. The first experiment is designed to evaluate point-in-polygon test based spatial join, which uses pickup locations from New York City taxi trip data[10] in 2013 (referred as *taxi*) and New York City 2010 census blocks[11] (referred as *nycb*). The second experiment is designed to evaluate polyline-with-polyline intersection based spatial join using two US Census Bureau TIGER[12] datasets provided by *SpatialHadoop* data portal[13], namely *edges* and *linearwater*. In addition to utilizing full datasets for experiments, we have also derived three sampled datasets, which is 1 month data from the full taxi dataset (referred as *taxi1m*) and 10% sample of the TIGER datasets, including *linearwater0.1* and *edges0.1*. Table 1 lists the sizes of all datasets. The reason that we use 10% sample datasets is because not all experiment settings are able to handle all full datasets successfully. The performance of the sample datasets may provide an idea of the relative performance among the three prototype systems when one or more systems cannot handle the full datasets successfully.

We have prepared several hardware configurations. The first configuration (WS) is a single node cluster with a workstation that has dual 8 core CPUs at 2.6 GHz and 128 GB memory. The large memory capacity makes it possible to experiment spatial joins that require significant amount of memory. A 10-node Amazon EC2 cluster, in which each node is a *g2.2xlarge* instance consists of 8 vCPUs and 15 GB memory, is used to test scalability of the three systems. We vary the number of nodes from 10 to 6 for scalability test and term the configurations as EC2-10, EC2-8, EC2-6, respectively. We have excluded EC2-4 and EC2-2 configurations due to insufficient memory issue for most of the testing. Both clusters are installed with Cloudera CDH-5.2.0 for the Hadoop system for running *HadoopGIS*[1] (Github version as of 10/2014) and *SpatialHadoop*[3] Version 2.3. *SpatialSpark*[4] is deployed using Spark 1.1. No further parameter fine-tuning were attempted.

Table 1 Experiment Dataset Sizes and Volumes

Dataset	#of Records	Size
Taxi	169,720,892	6.9 GB
nycb	38,839	19 MB
linearwater	5,857,442	8.4 GB
edges	72,729,686	23.8 GB
linearwater0.1	585,809	852 MB
Edges0.1	7,271,983	2.3 GB

B. Results using Full Datasets

The end-to-end runtimes (in seconds) for the two experiments (*taxi-nycb* and *edge-linearwater*) under the four configurations on the three systems are listed in Table 2. The reported runtimes include indexing the two input datasets and performing the distributed join, i.e., end-to-end runtimes. It can be seen that *HadoopGIS* failed all the experiments using the full datasets, *SpatialHadoop* was successful in all the experiments while *SpatialSpark* was in between. The top reason for *HadoopGIS* to fail is broken pipeline, which is typical in Hadoop Streaming when the data that pipes through multiple processors is too big. The top reason for *SpatialSpark* to fail is out of memory and Spark is not able to spill data to external storage. While *SpatialSpark* was successful for both the workstation and EC2-10 configurations, it failed under EC2-8 and EC2-6 configurations. We note the workstation has 128 GB memory and the aggregated memory capacity of the EC2-10 cluster is 150 GB, which were sufficient for *SpatialSpark* to experiment on the full datasets.

Table 2 End-to-End Runtimes of Experiment Results of Full Datasets (in seconds)

		WS	EC2-10	EC2-8	EC2-6
taxi-nycb	HadoopGIS	-	-	-	-
	SpatialHadoop	3,327	2,361	2,472	3,349
	SpatialSpark	3,098	813	-	-
edge-linearwater	HadoopGIS	-	-	-	-
	SpatialHadoop	14,135	5,695	8,043	9,678
	SpatialSpark	4,481	1,119	-	-

When the available memory capacity is sufficient, it can be seen from Table 2 that *SpatialSpark* can be significantly faster than *SpatialHadoop*. Under EC2-10 configuration, *SpatialSpark* is 2.9X and 5.1X faster than *SpatialHadoop* for the two experiments, respectively. The results are different under the workstation configuration where *SpatialSpark* is 3.2X faster for the *edge-linearwater* experiment but is only 1.07X faster for the *taxi-nycb* experiment. A possible explanation is that the *taxi-nycb* experiment is much more disk I/O intensive than the *edge-linearwater* experiment and the performance of the workstation is significantly limited by its single-node disk I/O bandwidth. When disk I/O is not a limiting factor (either by using distributed I/O or the experiment is more computing bound in the *edge-linearwater* experiment), the speedups of *SpatialSpark* over *SpatialHadoop* have clearly demonstrated the efficiency of in-memory processing.

C. Results Using Sample Datasets

The runtimes of the *taxi1m-nycb* and *edge0.1-linearwater0.1* experiments are listed in Table 3. Since the performance of the three EC2 configurations are roughly the same for all the three systems (which may indicate poor scalability), we only show the results under the workstation and EC2-10 configurations. We list the breakdown runtimes to provide a better idea on the runtime distributions: column IA is the runtime for indexing the left side input dataset (*taxi1m* and *edge0.1*), column IB is the runtime for indexing the right side input dataset (*nycb* and *linearwater0.1*), column DJ is the runtime for distributed spatial join, and, column TOT is the summation of the three.

Table 3 Breakdown Runtimes of Experiment Results Using Sample Datasets (in seconds)

		WS				EC2-10			
		IA	IB	DJ	TOT	IA	IB	DJ	TOT
taxi1m-nycb	HadoopGIS	206	54	3,273	3,533				-
	SpatialHadoop	227	52	230	482	647	187	183	1,017
	SpatialSpark				216				67
edge0.1-linearwater0.1	HadoopGIS	1,550	488	1,249	3,287				-
	SpatialHadoop	1,013	307	220	1,540	756	596	106	1,458
	SpatialSpark				765				48

Although *HadoopGIS* still failed under the EC2-10 configuration for both experiments, it was successful under the workstation configuration. This makes it possible to compare its performance with *SpatialHadoop* and *SpatialSpark* directly. The runtimes for *SpatialSpark* are end-to-end times as it is difficult to measure each individual step due to the asynchronous data communication/computation in Spark. The results listed in Table 3 suggest that, while the indexing times are comparable in both *HadoopGIS* and *SpatialHadoop*, *SpatialHadoop* is 14X and 5.7X faster than *HadoopGIS* for distributed joins (as reported in the DJ column) in the two experiments, respectively. While excessive disk I/O and string parsing might be important factors in contributing to the poor performance of *HadoopGIS*, our previous results have identified that the C++ based GEOS geometry library used in *HadoopGIS* can be several time slower than the java-based JTS geometry library used in *SpatialHadoop* and *SpatialSpark* [6], which might be another major factor. We thus exclude *HadoopGIS* from further comparisons.

When comparing the end-to-end runtimes between *SpatialHadoop* and *SpatialSpark* using the sampled datasets, *SpatialSpark* is about 2.2X faster under the workstation configuration but is about 15X faster under the EC2-10 configuration for the *taxi1m-nycb* experiment. Similar results, i.e., 2.0X and 30X under the EC2-10 configuration, can be observed in the *edge0.1-linearwater0.1* experiment. The result was a surprise when compared with the speedups using the full datasets. A careful investigation revealed that indexing times under the EC2-10 configuration dominates in both experiments using the sampled datasets. These are quite different from the full dataset experiment results where distributed join (DJ) consumes most of the runtime, which are 1,950s out of 3,327s for *taxi-nycb* experiment under workstation configuration, 1,282s out of 2,361s for *taxi-nycb* experiment under EC2-10 configuration, 9,887s out of 14,135s for *edge-linearwater* under workstation configuration and 3,886s out of 5,695s for *edge-linearwater* under EC2-10 configuration. An explanation is that, indexing under EC2-10 configuration involve significant data shuffling among the 10 distributed computing nodes which can be very expensive for *SpatialHadoop*. In contrast, distributed joins under the EC2-10 configuration can be significantly sped up by distributed I/Os in *SpatialSpark*.

When comparing the distributed join times (DJ) only, *SpatialHadoop* takes only 220s in *edge0.1-linearwater0.1* experiment under the workstation configuration, which is significantly lower than the indexing runtimes. This may indicate the Hadoop infrastructure overheads for small datasets on a single computing node may be high. We note that the end-to-end runtime of *SpatialSpark* (765s), which is much larger than the distributed join (DJ) runtime but only half of the total (TOT) runtime of *SpatialHadoop*.

Under EC2-10 configuration, *SpatialSpark* is 2.7X and 2.2X faster than *SpatialHadoop* with respect to distributed join (DJ) runtimes for the two experiments, respectively. The results are consistent with the experiments using the full datasets, which are 1.8X (1282/712) and 3.5X (3886/1119) for the two experiments under EC2-10 configuration. However, similar to the workstation configuration, the indexing runtimes are several times larger than the distributed join runtimes for *SpatialHadoop*. The results may suggest that indexing in *SpatialHadoop* are quite expensive when compared with distributed joins, possibly due to distributed data shuffling and excessive disk I/Os. It is clear that the speedups of *SpatialSpark* over *SpatialHadoop* are mostly due to the ability to reduce unnecessary disk accesses by pipelining the process completely in memory.

IV. CONCLUSION AND FUTURE WORK

In this study, we have compared three leading Cloud-based spatial data management systems, both conceptually through architectural analysis and empirically through experiments using real world datasets on both a workstation serving as a single-node cluster and on Amazon EC2 clusters using up to 10 nodes. The analytical and experimental results suggest that, Cloud system platforms (Hadoop/Spark), data access models (streaming/random/functional), languages (Java/mixed/Scala) and the underlying geometry libraries (GEOS/JTS) have significant impacts in their realized performance. While *SpatialHadoop* generally is the winner of robustness, partially because it is built on top of the mature Hadoop platform, *SpatialSpark* is the winner with respect to efficiency in large-scale spatial join processing, due to the efficiency of in-memory processing provided by Spark.

All the three systems we have evaluated in this study run on Java Virtual Machines (JVMs), which do not support Single Instruction Multiple Data (SIMD) computing power yet. We have developed two sets of techniques, i.e., ISP-MC+ and ISP-GPU [11] by integrating spatial join techniques with Cloudera Impala [12] and LDE-MC+ and LDE-GPU [13] by developing distributed spatial join techniques directly on top of Apache Thrift[14] for distributed data communications. All of them are capable exploiting SIMD computing power on both multi-core CPUs and Graphics Processing Units (GPUs). For future work, we plan to include analyzing their data parallel designs and performance comparisons.

ACKNOWLEDGEMENT

This work is supported through NSF Grants IIS-1302423 and IIS-1302439.

REFERENCES

1. E. H. Jacox, and H. Samet (2007). Spatial join techniques. ACM Transaction on Database System 32(1), Article #7.
2. J. Zhang, S. You and L. Gruenwald (2014). Parallel Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs," Information Systems, vol. 4, 134–154.
3. J. Zhang and S. You. (2012). Speeding up large-scale point-in-polygon test based spatial join on GPUs. In Proc. ACM BigSpatial, 23-32.
4. A.Aji et al (2013) Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. In Proc. VLDB, 6(11), 1009-1020, 2013.
5. Eldawy and M. F. Mokbel, "SpatialHadoop: A MapReduce Framework for Spatial Data," in Proc. IEEE ICDE'15.
6. S. You, J. Zhang and L. Gruenwald (2015). Large-Scale Spatial Join Query Processing in Cloud," in *Proceedings of IEEE CloudDM'15.*
7. A. Floratou, U. F. Minhas and F. Ozcan (2014). SQL-on-Hadoop: Full Circle Back to Shared-nothing Database Architectures. Proc. VLDB Endow., 7(12), pp. 1295-1306
8. Y. Lu, J. Cheng, D. Yan and H. Wu (2014). Large-scale Distributed Graph Computing Systems: An Experimental Evaluation. Proc. VLDB Endow., 8(3), pp. 281-292
9. B. Sowell, M. V. Salles, T. Cao, A. Demers and J. Gehrke (2013). An Experimental Analysis of Iterated Spatial Joins in Main Memory," Proc. VLDB Endow.,6(14), pp. 1882-1893.
10. H. Vo, A. Aji and F. Wang (2014). SATO: a spatial data partitioning framework for scalable query processing. in Proc. ACM-GIS.
11. S. You, J. Zhang and L. Gruenwald (2015). Scalable and Efficient Spatial Data Management on Multi-Core CPU and GPU Clusters: A Preliminary Implementation based on Impala, in Proc. IEEE HardBD'15.
12. M. Kornacker, A. Behm, et al. "Impala: A Modern, Open-Source SQL Engine For Hadoop," in Proc. CIDR'15.
13. J. Zhang, S. You and L.Gruenwald (2015). A Lightweight Distributed Execution Engine for Large-Scale Spatial Join Query Processing. To appear in Proc. IEEE Big Data Congress.

---

[1] https://sites.google.com/site/hadoopgis/
[2] http://hadoop.apache.org/docs/r1.2.1/streaming.html
[3] http://spatialhadoop.cs.umn.edu/
[4] http://simin.me/projects/spatialspark/
[5] https://spark.apache.org/
[6] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[7] http://trac.osgeo.org/geos
[8] http://www.vividsolutions.com/jts/
[9] http://libspatialindex.github.io/
[10] http://www.andresmh.com/nyctaxitrips/
[11] http://www.nyc.gov/html/dcp/html/bytes/applbyte.shtml
[12] https://www.census.gov/geo/maps-data/data/tiger.html
[13] http://spatialhadoop.cs.umn.edu/datasets.html
[14] https://thrift.apache.org/