A SPARK BASED COMPUTING FRAMEWORK FOR SPATIAL DATA

Fei Xiao [a,*]

[a] Administrative Information Center, National Administration of Surveying, Mapping and Geoinformation of China, Engineer, 100830 Beijing, China - xiaof@sbsm.gov.cn

KEY WORDS: Spatial Data, Spark, Index, Spatial Operations

ABSTRACT:

In this paper, a novel Apache Spark-based framework for spatial data processing is proposed, which includes 4 layers: spatial data storage, spatial RDDs, spatial operations, and spatial query language. The spatial data storage layer uses HDFS to store large size of spatial vector/raster data in the distributed cluster. The spatial RDDs are the abstract logical dataset of spatial data types, and can be transferred to the spark cluster to conduct spark transformations and actions. The spatial operations layer is a series of processing on spatial RDDs, such as range query, k nearest neighbour and spatial join. The spatial query language is a user-friendly interface which provide people not familiar with Spark with a comfortable way to operation the spatial operation. Compared with other spatial frameworks based on Spark, it is highlighted that spatial indexes like grid, R-tree are used for data storage and query. Extensive experiments on real system prototype and real datasets show that better performance can be achieved.

1. INTRODUCTION

Geographic information systems(GIS) process two kinds of data including spatial and attribute data. The performance depends on the representation of the data and the extent to which they can integrate it (Samet , 2015). The traditional database management systems are designed to deal with attribute data and hence must be extended to handle spatial data. The requirements for an extended DBMS to fulfil the following objectives. (1) The logical data representation must be extended to geometric data. (2) The query language must integrate new functions to capture the rich set of possible operations applicable to geometric objects. (3) There should exist an efficient physical representation of the spatial data. (4) Efficient data access is essential for spatial databases, so new data structures are proposed for spatial indexing. (5) Some new relational query processing algorithms are needed, such as spatial join, k nearest neighbour etc. Several spatial database systems are developed based on specified DBMS, such as PostGIS on PostgreSQL, Oracle Spatial on Oracle.

But with rapid development of GIS and Internet information technology, organizations and enterprises in geoinformation field accumulate big datasets of spatial information. How to manage these data effectively and analysis them efficiently becomes highly tough problems. Distributed DBMS and NoSQL DBMS supplies GIS a new way to solve the big problems. Novel data storage structures and different choices of the trade-off of consistency, availability and partition tolerance (known as CAP) (Gilbert, Lynch, 2002). These masses of Not SQL DBMSs can also be extended to support spatial data. Spatial indexing structures such as grid, R-tree, R+-tree should be realized on them and spatial operations such as query and spatial analysis should also be supported for users.

Apache Hadoop and Spark are well known as the most effective solution for big data, and well accepted by most of industries and communities. The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models (Apache Hadoop Organization). It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures. Apache Spark is a fast and general engine for large-scale data processing (Apache Spark Organization). Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation. Spark takes new design philosophy to generalize MapReduce process. It adds two novel peculiarities to Hadoop MapReduce to support more general systems such as iterative, interactive and streaming apps. One is general task directed acyclic graph (DAG) model to schedule the execute stages, and the other is sharing data in cluster memory to achieve higher IO access speed. Proved by some experiments (Zaharia, Chowdhury, Das, …, 2012), Spark RDDs can outperform Hadoop by 20× for iterative jobs and can be used interactively to search a 1 TB dataset with latencies of 5–7 seconds.

In recent years, some computing framework for spatial data has been released. Hadoop-GIS is a scalable and high performance spatial query system over MapReduce, provides an efficient spatial query engine and an expressive SQL-like spatial query language to process spatial queries, data and space based partitioning, and query pipelines that parallelize queries implicitly on MapReduce (Ablimit, Xiling, Hoang, …, 2013). MD-HBase extends HBase, a Key-value store system, uses linearization and related indexes to store multi-dimensional spatial data in KV system (Nishimura, Das, Agrawal, 2011). Parallel-Secondo extends Secondo using Hadoop as a parallel distributed task scheduler (Lu, Guting, 2012). GeoTrellis provides high performance raster input/output, geoprocessing and web services using distributed processing to achieve quite amazing throughput for large raster datasets. It uses the Hadoop file system (HDFS) but replaces Hadoop's MapReduce with Spark for distributed processing (Kini, Emanuele, 2014). GeoMesa is a distributed spatio-temporal database built on top of Hadoop and column-family databases such as Accumulo and HBase, and it includes a suite of tools for indexing, managing and analysing both vector and raster data (Hughes, Annex, Eichelberger, …, 2015). SpatialHadoop is a full-fledged MapReduce framework with native support for spatial data by four layers of language, storage, MapReduce, and operations (Eldawy, Mokbel, 2015). GeoSpark proposes spatial resilient distributed datasets (SRDDs) and supplies geometrical operations over Apache Spark platform (Yu, Wu, Sarwat, 2015).

* Corresponding author

https://doi.org/10.5194/isprs-annals-IV-4-W2-125-2017 | © Authors 2017. CC BY 4.0 License.

SpatialSpark implements only spatial data join query process migrating from CUDA/Thrust compute platform to Spark (You, Zhang, Gruenwald, 2015).

In this paper, a novel Apache Spark based computing framework for spatial data is introduced. It leverages Spark as the under layer to achieve better computing performance than Hadoop. 4 layers architecture from low to high is proposed: spatial data storage, spatial RDDs, spatial operations and spatial query language. All managements of spatial data are mentioned around Apache Hadoop and Spark ecosystem. (1) The spatial data storage using HDFS to store large size of spatial data, vector or raster, in the distribute cluster. (2) The spatial RDDs are abstract logistical dataset of spatial data types and can be transferred to the spark cluster to do spark transformations and actions. (3) Spatial operations layer is a series of processing on spatial RDDs such as range query, k nearest neighbour and spatial join. (4) Spatial query language is a user-friendly interface which supplies people not major in computer a comfortable way to operation the spatial operation.

Comparing to other spatial framework based on spark, it is highlighted that two-layers spatial indexing approach of global and local indexes are used, which is inspired by SpatialHadoop (Eldawy, Mokbel, 2015). The spatial indexes are migrated to Spark successfully initiatively, so orders of magnitude better performance than GeoSpark can be achieved. The differences to other Hadoop and Spark spatial compute frameworks are the close integration, both logical and physical, between Hadoop HDFS and Spark spatial RDDs.

2. ARCHITECTURE

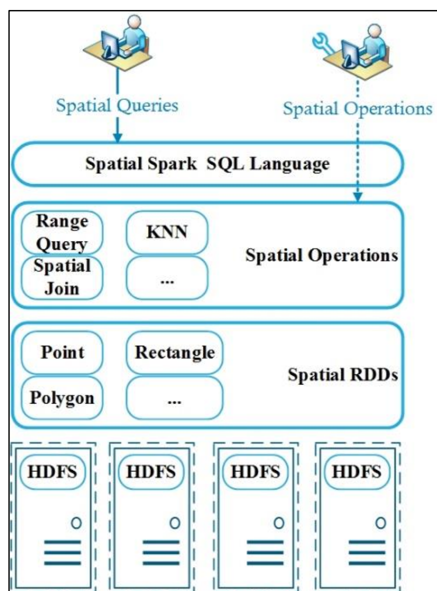

Figure 1. System Architecture

Figure 1 gives the high-level architecture of our system. 4 parts of the system present clearly. Two types of users can use this system in different level: (1) The casual user who knows the business more than the IT technologies can access system by the Spark SQL Language interface. (2) The developer who has professional knowledge of Spark framework and programming language can reuse of some operations in the system and can also extend them to meet their own requests.

2.1 Spatial Spark SQL Language

Spatial Spark SQL language layer is the top interface between the system and endpoint casual users. It can load data from a variety of structured sources (e.g., JSON, Hive, and Parquet), lets you query the data using SQL and provides rich extension of existing data types and functions (Todd, 2016).

Several spatial data types are implemented in system, such as point, rectangle and polygon. Some user-defined functions (UDFs) are implemented to help processing spatial operations, such as overlaps, distance, and boundary. Users can also extend and register themselves' UDFs in Python, Java and Scala.

The original work of introducing spatial data type and UDFs to Spark make the whole framework easy to use and integrate smoothly with existing non-spatial functions and operations such as Filter, Join and Group By, just like PostGIS to PostgreSQL.

2.2 Storage Layer

The storage layer supports persistent spatial data either on local disk or Hadoop file system (HDFS), but HDFS is recommended for using in cluster environment. Because raw spatial files in Hadoop do not support any indexes itself, we employ spatial index structures within HDFS as a means of efficient retrieval of spatial data. Indexing is the most important point in its superior performance over other Spark spatial computing framework.

The index structure in SpatialHadoop (Eldawy, Mokbel, 2015) is referenced in this paper, it is smart and elegant for spatial indexing in HDFS. Two level indexing approach of global and local indexes are proposed. Global index in HDFS name node indicates the minimum boundary rectangle (MBR) of each partitions of the spatial file. Local index is in each partition file that can be processed in parallel in both MapReduce job and Spark spatial RDDs transformations and actions.

Three phases process for indexing will be introduced. (1) Partitioning: big input file will be spatially split into n partitions, and n rectangles representing boundaries of the n partitions will be calculated. Each partition should fit in one HDFS block size, so an overhead ratio will be set for the overhead of replicating records overlapped and storing local indexes. (2) Local Indexing: requested index structure (e.g., Grid or R-tree) in each partition block will be built flowed by the spatial data. The index structure and spatial data is in the same partition block file. (3) Global Indexing: all local index files will be concatenated into one file that represents the final indexed file. It is constructed using bulk loading and is stored in the name node of the Hadoop clusters.

The framework supplies an easy way to build index by running a command line as following, then the spatial data file will be bulk loaded and be split to a few block files by the giving index type and block size.

shadoop index <input> <output> shape:<input format> sindex:<index> blocksize:<size> -overwrite

2.2.1 Build index: Different types of indexes are built in significant difference. The most two regular indexes, grid and R-tree will be introduced.

(1) Grid. Grids are frequently the simplest index in use. It partitions the data by a uniform grid, and the spatial object overlapping with the same grid will be in the same partition. In this paper, after the number of partitions n is calculated, the

https://doi.org/10.5194/isprs-annals-IV-4-W2-125-2017 | © Authors 2017. CC BY 4.0 License.

partition boundaries are computed by a $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ size grid. The spatial object is replicated to every grid which is overlapped with it. Because the grid index is a one-level flat index, the spatial data in each grid cell are stored in no particular order.

(2) R-tree. R-tree was proposed by Antonin Guttman in 1984. It is a balanced search tree, organizes the data in pages, and is designed for storage on disk. A Sort-Tile-Recursive (STR) algorithm is used to build an R-tree (Leutenegger, Scott T., Mario Lopez, …, 1997). The general process is similar to building a B-tree from a collection of keys by creating the leaf level first and then creating each successively higher level until the root node is created. After the number of partitions are calculated, we can tile the spatial data space using $\lceil\sqrt{n}\rceil$ vertical slices so that each slice contains enough rectangles to pack roughly $\lceil\sqrt{n}\rceil$ nodes. Then sort the spatial object by x-coordinate and partition them into $\lceil\sqrt{n}\rceil$ slices. If the spatial object is not point, the coordinates of center points of MBR is used in sort process. And then sort the spatial object of each slice by its y-coordinate and pack them into nodes of length $\frac{r}{n}$. In other words, the first $\frac{r}{n}$ records into the first node, the next $\frac{r}{n}$ into the second node, and so on.

What needs to highlight here is sample records of spatial data is used for partition process for efficiently reading very large input file. The size of random sample is set to a default ratio of 1% of the input file, with a maximum size of 100MB to ensure it fits in memory.

And in local indexing phase, spatial objects in each partition are bulk loaded into an R-tree also using the STR algorithm. Each block is represented by the MBR of it records, and all the partition blocks are concatenated into a global R-tree index using their MBRs as the index key by bulk loading process. Spatial records overlapping with multiple partitions will not be replicated, and are assigned to the smallest area it overlaps.

2.2.2 Index File Structure: The file structures of two indexes, grid and R-tree, are introduced for better understand the details of index files in HDFS. We take the comma-separated values (CSV) records of points as the example to see how organizes the index file.

(1) Grid. A spatial grid index is stored in one HDFS folder including one global index and partitioned blocks. Because the grid index has not local index, so each partition is the row spatial data format.

Serial Number	X1	Y1	X2	Y2	···
Records Counts	Block Size	Partition Name			···

Figure 2. Global index structure

Figure 2 shows the structure of a global index. Eight fields here describe the details of partition blocks. The first field is a natural number index. The second to fifth field is the MBR of each partition which is stored as left-bottom and right-top points coordinates. The MBR can be used for early pruning file blocks that do not contribute to required answer. The sixth field is the spatial object counts of each partition, and the seventh field is the size of each partition file in byte which can be used as the block reading offset of each partition. The eighth field is the partition name. The global structure is stored as one records each line with the End-Of-Line (EOL) as the mark character.

(2) R-tree. The R-tree index folder has the same style and the same global index structure as grid. Only differences are the local index structure saved with the spatial data in monolithic one file.

Figure. 3 shows the R-tree local index file structure. Every block file can be dived into three parts. The first part is the local R-tree specification stored in binary format. The first 8 bytes is a file type marker for verifying the R-tree file type and version. Then following 4 integer values and each integer is stored in 4 bytes binary format. These are tree size, height, degree, and element count in order. The middle part is the R-tree nodes information. Every R-tree node is stored in 36 bytes binary format. The first 4 bytes is the node sequence number saved in integer value, and the following 4 double type values are the MBR's coordinates of the node every of which occurs 8byte.The last part is the real spatial data storage. It is stored as the plain text format and each line one record.

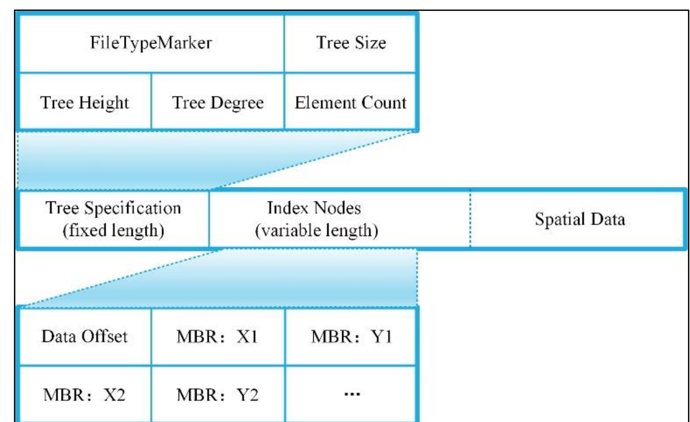

Figure 3. R-tree Local Index Structure

2.3 Spatial RDDs Layer

This Layer abstracts the storage of a variety of spatial data, and support the uniform interface to top layers. The Spark Resilient Distributed Datasets (RDDs) are extended to Spatial RDDs for adapting the Spark data structure to spatial data. New spatial RDD data types are implemented including point, line, rectangle and polygon. They provide advanced traits which are difficultly achieved in distributed computing environment, including partitioned collection, fault-tolerant, and simple programming interface.

Spatial RDDs are different with normal RDDs in two main aspects. First is the special customized partitioning method which can optimise the data distribution across the servers and accelerate the spatial options on them by using GeoHash encoding. Second is the spatial analysis functions support on spatial RDDs, which can be easily used for spatial operation.

2.3.1 Partitioning. Users can control two important aspects of SRDDs: persistence and partitioning. Users can indicate which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage). They can also ask that an RDD's elements be partitioned across machines based on a key in each record. This is useful for placement optimizations on spatial data sets, such as ensuring that two datasets that will be joined together are geohash-partitioned in the same way.

2.3.2 Fault tolerant. SRDDs are a fault-tolerant distributed memory abstraction that avoids high costs replication within network (Zaharia, Chowdhury, Das, …, 2012). They recognize computing processes as a directed acyclic graph (DAG). Each SRDD remembers the graph of operations used to build it, similarly to batch computing models, and can efficiently recompute data lost on failure. As long as the operations that create RDDs are relatively coarse-grained, i.e., a single operation applies to many data elements, this technique is much more efficient than replicating the data over the network. RDDs work well for a wide range of today's spatial data-parallel algorithms and programming models, all of which apply each

2.3.3 Programming interface. A spatial RDD is represented as an object. Transformations and actions can be invoked using methods on these objects. Programmers start by defining one or more RDDs through transformations on data in stable storage (e.g., map and filter). They can then use these SRDDs in actions, which are operations that return a value to the application or export data to a storage system. Examples of actions include count (which returns the number of elements in the dataset), collect (which returns the elements themselves), and save (which outputs the dataset to a storage system).

2.4 Spatial Operations Layer

This layer implements spatial proximity analyses and geometry processing. Three common spatial operation algorithms are proposed using SRDDs: range query, KNN query, and spatial join. They leverage spatial locality of data access technology and spatial index to achieve better performance.

Two aspects of SRDDs: persistence and partitioning can be controlled in spatial operations. We can indicate which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage). They can also ask that an RDD's elements be partitioned across machines based on a key in each record. This is useful for placement optimizations, such as ensuring that two spatial datasets that will be joined together are hash-partitioned in the same way.

Spatial operations can be divide into two types: plain operations and improvement ones. The first type processes query by full table scan, so it is bad in query efficiency. The second is improved by spatial index and can achieve better performance. We will reveal the differences in the following of this section with spark transformations and action. The pseudocodes are written by scala language in function programming style.

Plain operations are proposed firstly.

2.4.1 Range Query: A range query takes a set of points p and a query range r as input, and return the set of points in p that overlaps with r. Filter is a predefined function in Spark taking a predicate function as the parameter, in which contains operation is used. In consideration of lazy evaluation mechanism of Spark, the collect action is used to trigger the operation.

```
def RangeQuery(Points p, Range r) {
        p.filter(point=>r.contains(p)).collect()
}
```

2.4.2 K Nearest Neighbour: A kNN query takes a set of spatial points p, a query point q, and an integer k as input, and returns the k closest points in p to q. Map transformation is used to compute the distance between every point in p and the query point q, and then takeOrdered function of Spark action is used to get the top k elements of the RDD using either their natural order or a custom comparator.

```
def KNN(Points p, Point q, Int k){
        p.map(point=>(Distance(p,q),point)).
takeOrdered(k)
}
```

2.4.3 Spatial Join: A spatial join takes two sets of spatial records r and s and a spatial join predicate θ as input, and returns the set of all pairs (r,s) where $r \in R$, $s \in S$, and θ is true for all (r,s) tuple. The cartesian function of Spark is used to get all pairs of $R \times S$, and then all pairs should be filtered by the spatial join predicate θ.

```
def SpatialJoin(R r, S s, func predicateθ){
        r.cartesian(s).filter((r,s)=>predicateθ(r,s))
}
```

The improvement appears in two aspects. The first is GeoHash code can be used in user define partitioning. When using memory on a single machine, programmers worry about data layout mainly to optimize lookups and to maximizes colocation of information commonly accessed together. SRDDs give control over the same concerns for distributed memory, by letting users select a partitioning function and co-partition datasets, but they avoid asking having users specify exactly where each partition will be located. Thus, the runtime can efficiently place partitions based on the available resources, and the query performance is improved.

A GeoHash partition function GeoHashPartitioner is used to process partitioning, which needs a partition number as the parameter. It is recommended that partition number is equal to the total number of CPU cores in cluster. PartitionBy() function provide by Spark framework is used as following. In consideration of different type of space-filling curves, Peano curve is recommended to get better performance.

```
p.partitionBy(new GeoHashPartitioner(100)).persist()
```

The second is global and local indexes are used to reduce useless scan time. Generally global index is used to tailor points in the MBR which are certain not in the answer. For example, a range query job provides a filter function that prunes file blocks with MBRs completely outside the query range. Local index is used in refining phrases which will make it more efficient than scanning over all records.

3. EXPERIMENTS

This section provides experimental study for the performance of the following variables: partition and index. No partition, spark hash partition and spatial GeoHash partition are used in the partition experiment, and no index, grid index and R-tree index are used in the index experiment. Then a comparison experiment of the new framework and GeoSpark is emerged to prove getting a better performance.

Experimental Setup. Our cluster is running on Ali Cloud who is the biggest public cloud provider in China. The setting of worker

https://doi.org/10.5194/isprs-annals-IV-4-W2-125-2017 | © Authors 2017. CC BY 4.0 License.

server is as follows: (1) 4 Intel Xeon E5-2680 v3 (Haswell) CPU operating at 2.5GHz. (2) 8GB Memory. (3) Efficient cloud disk with 3000 max IOPS. The experiment computing cluster contains four workers.

Datasets. We use the TIGER file in these experiments, which is a real dataset presenting spatial features in the US, such as streets and rivers. We choose Zcta510 1.5GB dataset, Areawater 6.5GB dataset and Edges 62GB dataset.

4. IMPACT OF PARTITION AND DATA SIZE

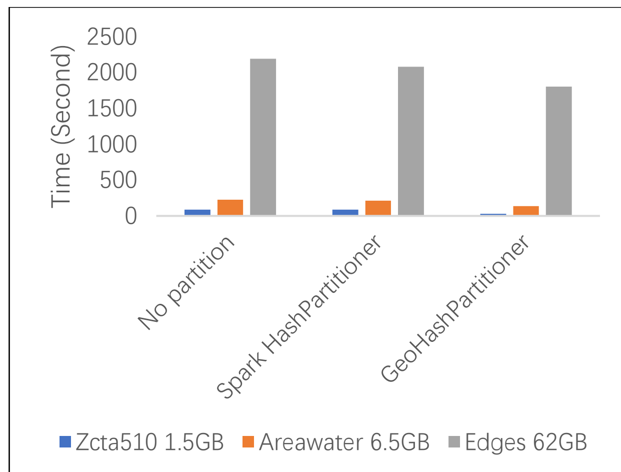

Figure 4. Range query experiments with different partition and data size

Different partitioners are used to do range query on the three-different size of dataset. The query time are shown in Figure 4. As depicted in Figure 4, Spark HashPartitioner gets not observably better performance than no partition, because regular hash function on spatial data does not help for data aggregation and data local, only GeoHash can achieve better performance because of its spatial feature. Another thing to draw attention is when the data size is over total memory of the cluster, query performance will significantly decline, and the advantage of GeoHash partition are not obviously. This because if data size is outdistance the cluster memory, data persistent takes much more time than data query. So it is recommend to get plenty of memory in big data memory computing.

5. PERFORMANCE OF DIFFERENT INDEX

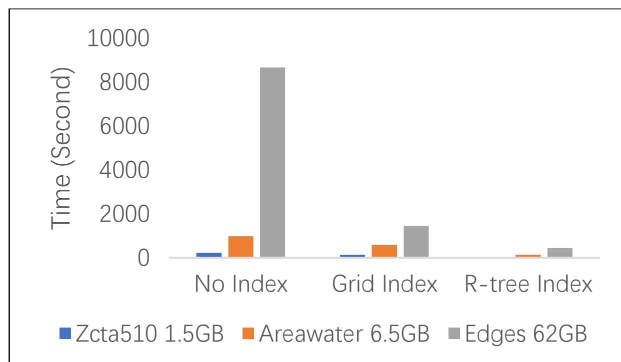

Figure 5. kNN experiments with different index and data size

As shown in Figure 5, kNN query using index can achieve better performance than no index on Spark framework, and the query time almost declines quickly with the increasing of data size. R-tree appears better performance than Grid index no matter of dataset size. This because in large dataset, the spatial data are in order of MBR and most of the uncorrelated partitions will not be read and scanned, and only the index will be read into memory which size is much less than dataset size.

6. PERFORMANCE COMPARISON

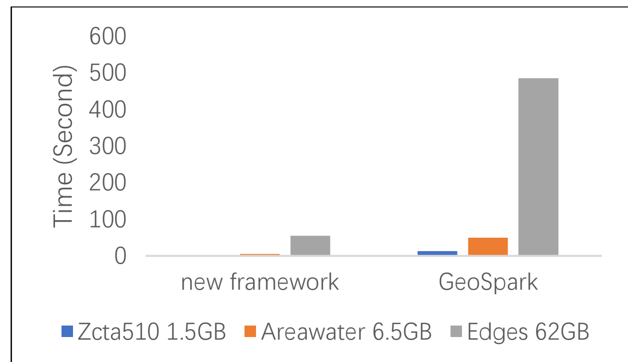

Figure 6. Spatial join experiments with different frameworks and data size

Different frameworks with R-tree index are used to evaluate the performance. As depicted in Figure 6, new framework described in this paper and GeoSpark cost more run time on the large dataset than that on the small one. However, new framework achieves much better run time performance than GeoSpark in both three datasets. This superiority is more obvious on the large dataset. The reason is that the new framework can use two-level index structure and better optimization by GeoHash partitioner. That accelerates the processing speed.

7. CONCLUSION

This paper introduced a new Apache Spark-based framework for spatial data processing is proposed, which includes 4 layers: spatial data storage, spatial RDDs, spatial operations, and spatial query language. The spatial data storage layer uses HDFS to store large size of spatial vector/raster data in the distribute cluster. The spatial RDDs are the abstract logical dataset of spatial data types, and can be transferred to the spark cluster to conduct spark transformations and actions. The spatial operations layer is a series of processing on spatial RDDs, such as range query, k nearest neighbour and spatial join. The spatial query language is a user-friendly interface which provide people not familiar with Spark with a comfortable way to operation the spatial operation. Compared with other spatial frameworks based on Spark, it is highlighted that spatial indexes like grid, R-tree are used for data storage and query. Extensive experiments on real system prototype and real datasets show that better performance can be achieved.

ACKNOWLEDGEMENTS

I give my thanks to NASG China and UN-GGIM who fund the project of *Geospatial Information Management Capacity Development in China and Other Developing Countries* and send me to GMU for studying. I would like to extend my sincere thanks to everyone in GMU CISC, who have helped me make this thesis possible and better. Thanks to Doctor Chaowei Yang, who gave me the vital direction of spatial big data and cloud computing, and I also would like to the PhDs, Min Sun, Han Qin, Fei Hu, Manzhu Yu, etc, I would miss the days working and studying together. Sincere thanks to my wife and my mom who

gave me great support to do the research, and thanks to my new born daughter NianNian, hope you grow healthily and happy.

REFERENCES

Samet, H., 2015. Sorting Spatial Data. *The International Encyclopedia of Geography.*

Gilbert, S. and Lynch, N., 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2), pp.51-59.

Apache Hadoop Orgnization. http://hadoop.apache.org

Apache Spark Orgnization. http://spark.apache.org/

Aji, A., Sun, X., Vo, H., Liu, Q., Lee, R., Zhang, X., Saltz, J. and Wang, F., 2013, November. Demonstration of Hadoop-GIS: a spatial data warehousing system over MapReduce. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (pp. 528-531). ACM.

Nishimura, S., Das, S., Agrawal, D. and El Abbadi, A., 2011, June. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on* (Vol. 1, pp. 7-16). IEEE.

Lu, J. and Guting, R.H., 2012, December. Parallel secondo: boosting database engines with hadoop. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on* (pp. 738-743). IEEE.

Kini, A., and R. Emanuele. Geotrellis: Adding geospatial capabilities to spark. In *Spark Summit* (2014).

Hughes, J.N., Annex, A., Eichelberger, C.N., Fox, A., Hulbert, A. and Ronquest, M., 2015, May. Geomesa: a distributed architecture for spatio-temporal fusion. In *SPIE Defense+ Security* (pp. 94730F-94730F). International Society for Optics and Photonics.

Eldawy, A. and Mokbel, M.F., 2015, April. SpatialHadoop: A MapReduce framework for spatial data. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on* (pp. 1352-1363). IEEE.

Yu, J., Wu, J. and Sarwat, M., 2015, November. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems* (p. 70). ACM.

You, S., Zhang, J. and Gruenwald, L., 2015, April. Large-scale spatial join query processing in cloud. In *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on* (pp. 34-41). IEEE.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S. and Stoica, I., 2012, April. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (pp. 2-2). USENIX Association.

Todd McGrath, 2016. “What Is Spark SQL?”, https://dzone.com/articles/what-is-spark-sql

Guttman, A., 2015. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data – SIGMOD '84* . p. 47.

Leutenegger, Scott T., Mario Lopez, and Jeffrey Edgington, 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings. 13th International Conference on. IEEE*.

U.S. Census Bureau, http://www.census.gov/geo/www/tiger/.