ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

UNIVERSITY OF PIRAEUS

Υλοποίηση υποστήριξης τρισδιάστατων δεδομένων με χρήση του SpatialHadoop και του Hadoop

3D Data Extension support using Hadoop and SpatialHadoop

Μεταπτυχιακή Διατριβή

Μανίκης Μιχάλης Α.Π.: ΜΠΣΠ12040

Επιβλέπων Καθηγητής: Πελέκης Νικόλαος

Τμήμα Πληροφορικής

Πανεπιστήμιο Πειραιώς, 2017

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Νικόλαος Πελέκης	Ιωάννης Θεοδωρίδης	Άγγελος Πικράκης

Table of Contents

Abstract 5

Περίληψη 5

1. Introduction 6

2. Description 8

3. Related work 12

4. Analysis 23

4.1 SpatialHadoop Structure 23

4.2 Analyzing SpatialHadoop Indexing 24

4.3 Analyzing SpatialHadoop queries 32

4.4 Process data and execute queries with Pigeon 41

5. SpatialHadoop 3D data extension implementation 47

5.1 3D Rtree Algorithm 47

5.2 Implementation on SpatialHadoop 48

6. Experimental Results 57

7. Conclusions and Future Work 62

7. References 63

Appendix 1 64

List of Figures

1. Hadoop MapReduce resource allocation
2. Hadoop MapReduce lifecycle
3. Hadoop GIS architecture
4. ESRI Tools architecture
5. GeoMesa architecture
6. GeoMesa tools system
7. GeoMesa Zcurve example
8. GeoMesa index key
9. LocationSpark architecture
10. MD-HBase architecture
11. IMIS 2d dataset plot
12. SpatialHadoop Grid index plot
13. SpatialHadoop RTree index plot
14. SpatialHadoop RangeQuery query result plot
15. SpatialHadoop KNN query result plot
16. First data split for SpatialJoin query
17. Second data split for SpatialJoin query
18. SpatialHadoop SpatialJoin query result plot
19. Pigeon introduction page
20. SpatialHadoop Visualizer
21. SpatialHadoop Visualizer query plot
22. 3DRtree example index plot
23. Trucks dataset plot
24. Command line result of 3DRtree indexing
25. Command line result of 3DRtree querying
26. Plot result of 3DRtree querying
27. 3d data test results
28. Indexing time of 3d data
29. Query time of 3d data
30. Indexing time of 2d data
31. Query time of 2d data

Abstract

We will describe an extension of Hadoop, (SpatialHadoop), the structure and the functionality. Then we will analyze the steps we took to expand this extension to spatial data with 3 dimensions, and present the experimental results using a range query on a dataset.

Περίληψη

Θα παρουσιάσουμε μία επέκταση για το Hadoop, (SpatialHadoop), τη δομή του καθώς και τη λειτουργία του. Έπειτα θα αναλύσουμε τα βήματα που πήραμε για να υλοποιήσουμε αυτή την επέκταση ώστε να υποστηρίζει τρείς διαστάσεις, και θα δείξουμε τα πειραματικά αποτελέσματα χρησιμοποιώντας ένα ερώτημα διαστήματος.

1. Introduction

In the recent days there has been an explosion of the amount of data we generate. Handling those data, analyzing and trying to derive decisions from them has been a challenge. From scientific studies to hospitals the amount of data that is generated cannot be handled by the tools that we already used. Big data poses a challenge and demands a new way of handling them. Tools such as Parallel DBMS and Hadoop have emerged to provide a way to handle these amounts of data. Parallel DBMS tries to solve the problem using parallel database systems and partitioning data through multiple disks. This way they can reduce the I/O operations, while on the other hand, Hadoop uses the MapReduce algorithm to provide a less expensive approach.

Spatial Big data is another challenge that needs to be tackled. Systems such as OpenStreetMap or highly detailed images in the range of 100.000x100.000 pixels pose a big challenge to our current computational technology. On top of that, queries on those data are highly compute-intensive and require a scalable and efficient solution to execute them. For example, filtering data using MBRs and/or executing spatial join queries is a highly cpu-intensive task.

Hadoop is a framework to run parallel code on cheap machines that are forming a cluster. Hadoop runs using the MapReduce framework. MapReduce is designed to support parallel execution of code. Upon this framework many extensions have been implemented in order to support new features. One drawback of Hadoop is its low speed in performing GIS queries. This is because it is unable to use spatial indexes. For this reason there are extensions that enable Hadoop to perform such queries much faster. One such extension is SpatialHadoop. It supports a wide array of queries, as well as indexes upon spatial data. It uses the MapReduce framework to take advantage of the parallel execution it supports, in order to speed up execution time. SpatialHadoop employs MapReduce to create a spatial index and then appropriately store the data in HDFS. This has the result that spatial queries will generate fewer MBRs for the shapes because the data are stored based on the index and thus nearby shapes are stored in the same node with the query result to be completed more quickly. Below we will present in more detail the above function.

The structure of the index is based on a global index (it can be a Grid or an Rtree), this index will be the master index across all nodes, and then the local index which will be stored in every node separately. SpatialHadoop will first consult the global index to identify which partitions have the data that we need, and then it will query the local index of every node to get the data. The local index is stored in a single file followed by the raw data. A small check in the code of SpatialHadoop will identify this header and skip it to retrieve the data lines from the file. In order to more effectively run spatial queries as well as run more complex queries, Pigeon was introduced in SpatialHadoop. Pigeon is an extension of Pig to easily create spatial queries. Using the language of Pig, who resembles SQL, we can add spatial features that may be used together with SpatialHadoop. The queries posed are converted into MapReduce form suitable for Hadoop. One limitation of SpatialHadoop is that it does not support 3D data operations.

We will implement an extension to SpatialHadoop so that it can support 3D data inputs as well as querying these data. The implementation starts with the Rtree index that needs to be modified to support a new Z variable. This variable will be able to represent either a Z coordinate or time data. In our case this variable represents time data using epoch time. The Rtree algorithm will be enhanced to take advantage of this new variable in order to index the 3d data. Continuing we will modify the query used to run with 2D data, to be able to query 3D data. Using this new extension we will run some test runs in order to check the speed to the queries and indexing process using various node sizes on a Hadoop cluster.

2. Description

In this dissertation we will present a way to run 3D data queries using Hadoop framework. Hadoop runs using the MapReduce framework. MapReduce is designed to support parallel execution of code. Hadoop is an application designed to run programs in large cluster of computers. It uses a different file system than usual operating systems, HDFS, which enables it to be acceptable to errors that can occur during operation. Specifically Hadoop is designed to run on low-cost computers that are likely to present errors. To run any algorithm on Hadoop, the code has to be implemented using the MapReduce framework. One computer has the role of the master who runs the NameNode, SecondaryNameNode and JobTracker, but alternatively also the DataNode and TaskTracker, and the other computers have the role of slaves and only run the DataNode and TaskTracker. The NameNode is responsible for coordinating the usage of HDFS, which is the storage layer of Hadoop. The JobTracker coordinates the execution of jobs using the MapReduce framework. Slave nodes run the corresponding slave parts of these modules. The DataNode is the slave of the NameNode, and the TaskTracker is the slave of the JobTracker.

A typical Hadoop workflow is like the following:

- Load data into the cluster (write to HDFS)
- Analyze the data (run MapReduce)
- Store results in the cluster (write to HDFS)
- Read the results (read from HDFS)

To write files to HDFS the client computer needs to first communicate with the NameNode. The NameNode will then provide a number of slaves that these data need to be written. The data will be broken into blocks and written in each slave by the DataNode. The DataNode will then replicate the block to other slaves based on the replication factor that was set. Hadoop also supports Rack Awareness. This means that an administrator can provide specific ids to the DataNodes in order to create a topology. This is done in order to be less affected from possible failures. The NameNode will occasionally send heartbeats to the DataNodes. Every 10^{th} heartbeat the DataNode needs to provide a block report, so the NameNode knows which

blocks are at which DataNodes. There is also a SecondaryNameNode which will connect to the NameNode and get a snapshot of the metadata that hold block information. This is not done frequently (about once per hour) and SecondaryNameNode cannot be considered a high availability backup of NameNode. To run MapReduce first the client submits a Map Reduce job to the Job Tracker. The Job Tracker asks the NameNode about which DataNodes have the necessary blocks for the file required to run the job. The Job Tracker then provides the TaskTrackers running on those nodes with the Java code required to execute the Map task on their local blocks. The Task Tracker starts a Map task and monitors the progress while sending regular heartbeats to the Job Tracker.

But before the Map task can begin, resources (CPU, memory, etc.) have to be allocated. This is done by the ResourceManager. When the MapReduce job is about to start, the ResourceManager will allocate a container where the ApplicationMaster for this job will run. A container is a collection of resources. When the ApplicationMaster starts, it will allocate more containers to actually run the Map and Reduce tasks. The process can also be seen in figure 1.

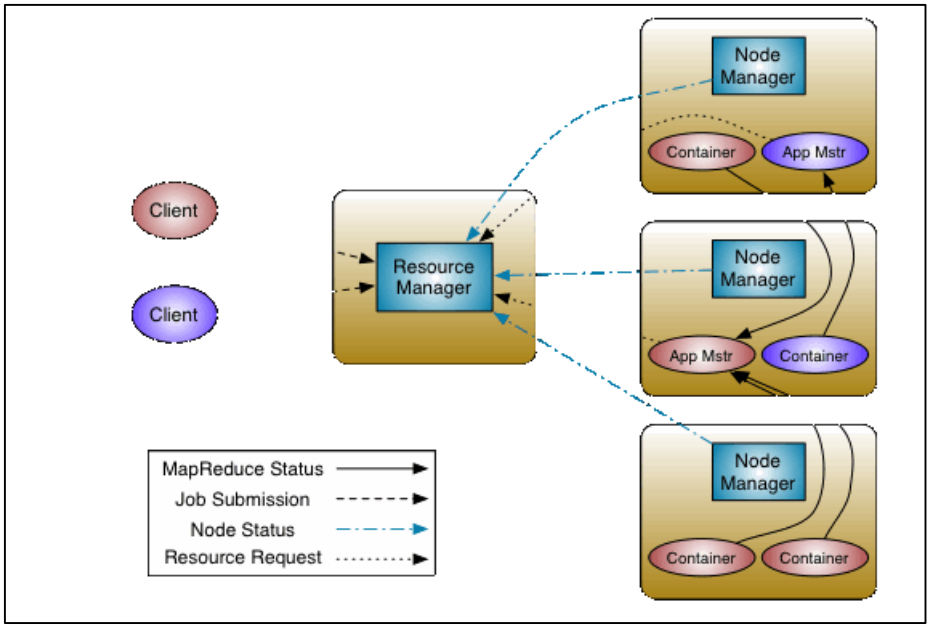

Figure 1: Hadoop MapReduce resource allocation

During the Map task the RecordReader will read the split data provided by the InputFormatter and output (key, value) pairs. When the Map task completes, each node stores the result of its local output in a temporary local storage. This is called

the intermediate data. These intermediate data (the (key, value) pairs) will be used in the next step, the reduce phase.

The Job Tracker starts a Reduce task on any one of the slaves in the cluster and instructs the Reduce task to go grab the intermediate data from all of the Mappers. There is a chance that the Map tasks may send these data to the Reducer almost simultaneously, resulting in a situation called Incast. Networks need to be able to handle possible incast situations.

The Reducer task now can begin the final computation. After the result is ready, the process to write the results to the HDFS (as described above) begins again. The RecordWriter and the OutputFormatter are responsible for this task. After this step the job completes. [11] On figure 2 there is an overview of the whole process.

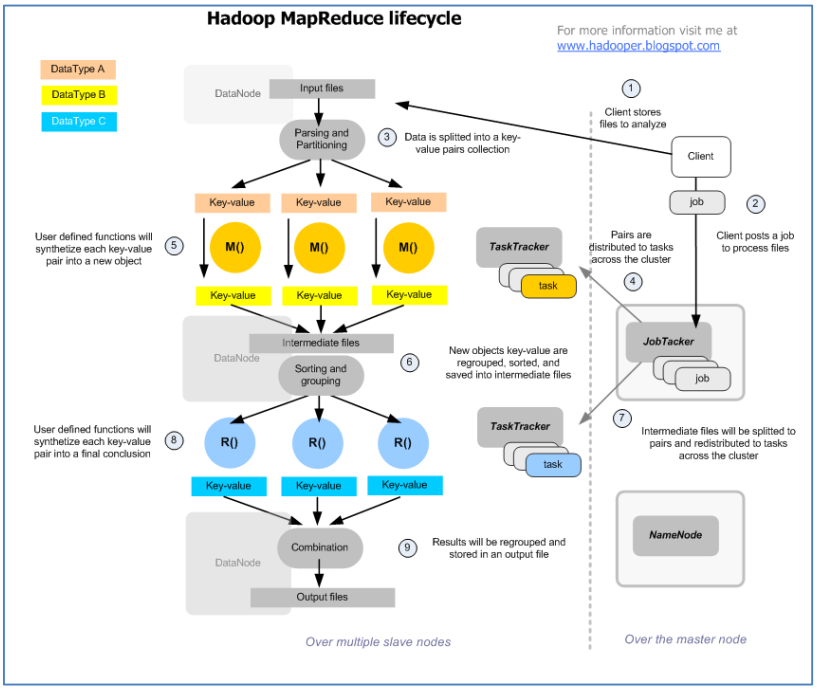

Figure 2: Hadoop MapReduce lifecycle

Upon this framework many extensions have been implemented in order to support new features.

This dissertation has the purpose of extending SpatialHadoop in order to be able to create spatial indexes and run spatial queries. On chapter **three** we will present a literature search of the systems that are similar to SpatialHadoop. We will provide a small description as well as their most important characteristics. On chapter **four** we will analyze SpatialHadoop and describe its structure as well as the underlying code. We will also provide a brief description of Pigeon, a module to support easier

execution of spatial queries using SpatialHadoop. On chapter **five** we will present the necessary changes that we implemented in order for SpatialHadoop to support 3D data queries. On chapter **six** we will show the experimental results after running the new extension. And finally on chapter **seven** we will provide suggestions for future improvements.

3. Related work

On this chapter we will present some of the work that has been done to use big spatial data with Hadoop. There are many different approaches and tools that differ based on their implementation and the features they provide.

The systems that we will present are: HadoopGIS, ESRI Tools, GeoMesa, SciDB, LocationSpark, MD-HBase.

The first system that we will present is **HadoopGIS**. HadoopGIS is structured on top of Hadoop and its goal is to provide an efficient solution capable of handling big spatial data in daily operations. It supports multiple spatial operations, as well as global indexing and local indexing to speed up queries. HadoopGIS provides spatial data partitioning to achieve task parallelization, an indexing-driven spatial query engine to process various types of spatial queries, implicit query parallelization through MapReduce, and boundary handling to generate correct results. [1] HadoopGIS is also integrated with Hive and through this interface it can provide an expressive spatial query language by extending HiveQL. It supports a lot of the basic spatial queries such as spatial join and nearest neighbor but it can also be extended by using Hive to provide more complex queries. The architecture of HadoopGIS is shown in the below image:

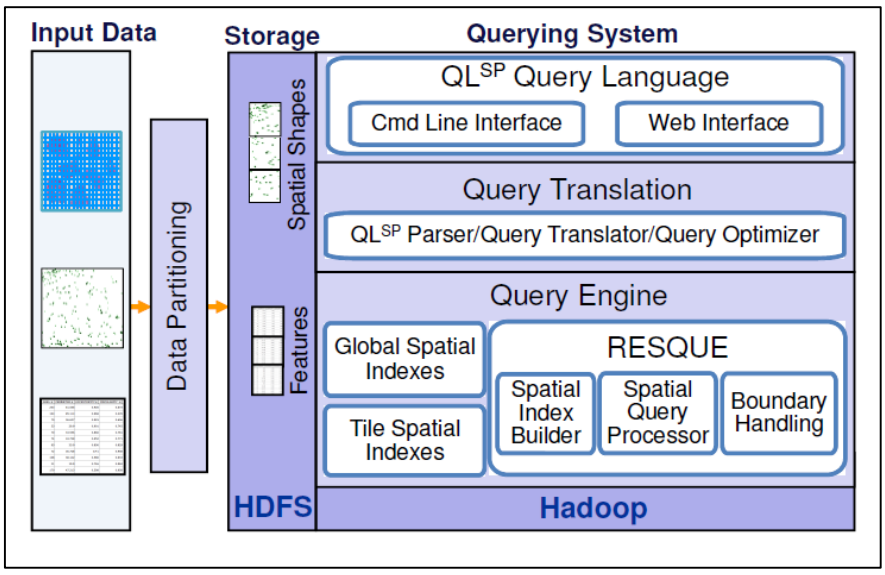

Figure 3: Hadoop GIS architecture

As shown in the Figure 3, the query engine used is called RESQUE (Real Time Spatial Query Engine). This special query engine was built because a general spatial query engine was not enough to its specialized purpose. RESQUE is a shared library and using it HadoopGIS can create queries in different levels. At a higher level it uses the global index to filter tiles and at tile level it supports indexing on demand in order to dynamically build indexes on the fly.

To be able to support a large variety of geometric computations, HadoopGIS has adopted a set of open source tools, for example SpatialIndex, GEOS and Hilbert RTree. In order for HadoopGIS to achieve greater speed while processing spatial queries it needs to identify the time consuming parts of the algorithm and break down those parts to be processed in parallel. These parts are called tiles or buckets and the goal is to run the algorithm on those tiles independently while also making sure that the results are consistent.

A typical spatial query using HadoopGIS and MapReduce then follows below algorithm:

First it partitions the data to generate the tiles. By doing this, HadoopGIS can process a large number of data in parallel without the need to communicate. The goal here is to create tiles with approximately the same number of data points in order to avoid data skewing. To achieve this, HadoopGIS breaks high density regions into smaller ones and uses a recursive partitioning approach to further break tiles.

After this step, the MBRs and the UID of the tile are stored in HDFS. The UID will be the key for the MapReduce algorithm. The MBRs will form the global index. Further indexing on tile level will be done on demand. Since modern hardware is widely available, the computational cost of such a task is small.

Since HDFS is more efficient using large files (default block size is 64MB), HadoopGIS merges multiple small tiles into large files and then stores them in HDFS. A pre-processing step can be run optionally here to perform further filtering. When this step is complete, the MapReduce algorithm can run on the data to perform the spatial queries. Boundary object handling is done using multiple assignment approach which means that objects that intersect in many different tiles will be stored in all of those tiles. After the algorithm is run, another post-processing step can be run optionally in case data needs for example another step of a MapReduce job, for example joining spatial query results in feature tables. Finally data is aggregated from all tiles and results are stored in HDFS. Performance results show that Join and

Containment queries are faster in HadoopGIS but aggregation queries are faster in RDMSs. Later was attributed to the fact that HadoopGIS has a record parsing overhead where it needs to parse every single record in real time but in RDMSs this step is already done and data are stored in binary format.
Overall HadoopGIS provides an effective partition approach and good boundary handling for correct results, as well as indexing in multiple levels (global and tile) and custom spatial queries using HiveQL.

ESRI Tools will be the second system that will be presented.
ESRI Tools is an open source toolkit [2]. At the core of the system is ESRI Geometry API written in Java. This API offers basic geometry objects such as point, lines and polygons, as well as spatial operations and indexing. It also offers 'acceleration' in spatial queries when a geometry type is used multiple times. It achieves this by building a quad tree on the geometry. This API can be used standalone with Hadoop to allow custom MapReduce applications with spatial data or it can also be used with other tools that the system offers.
One step above the API is the Spatial Framework for Hadoop. This framework is written in Java and extends the features by adding support for reading JSON files. It uses custom InputFormats and RecordReader classes to import JSON files in the MapReduce algorithm. InputFormats split the data into data-splits, these splits then act as input to the mappers. Mappers use RecordReader to read the split and run Map function on every record. The key for the Map function is the character offset from the start of the JSON file and the value is the JSON text. Below there is an example of a JSON file and how it is accepted by the Mappers.

```
/path/to/data/
data-1.json
json { "attributes" : { "Name" : "California" }, "geometry" : ... } { "attributes" : { "Name" : "Arizona" }, "geometry" : ... }
data-2.json json
{ "attributes" : { "Name" : "Utah" }, "geometry" : ... } { "attributes" : { "Name" : "Colorado" }, "geometry" : ... }
```

This JSON formatted text will then enter the Mappers:

```
Mapper 1 (data-1.json)
Record 1
Key 0
Value { "attributes" : { "Name" : "California" }, "geometry" : ... }
Record 2
Key 62
Value { "attributes" : { "Name" : "Arizona" }, "geometry" : ... }
```

Mapper 2 (data-2.json)
Record 1
Key 0
Value { "attributes" : { "Name" : "Utah" }, "geometry" : ... }
Record 2
Key 56
Value { "attributes" : { "Name" : "Colorado" }, "geometry" : ... }

ESRI tools also support user defined functions (UDFs) that extend Hive and offer new capabilities on top of ESRI API. Using these UDFs the user can create queries in Hive Query Language (HQL) and avoid the complexities of building queries using the MapReduce algorithm. Similar to Spatial Framework are the Geoprocessing tools that ESRI provides, written in Python. These tools add functionality to connect with ArcGIS. Using these tools a user can run Hadoop workflow jobs with ArcGIS and check the status, as well as exchange data between ArcGIS and Hadoop. This is especially useful if the user wants to visualize his data on ArcGIS.
Finally ESRI tools have a wide range of samples that the user can study in order to build new custom spatial tools.

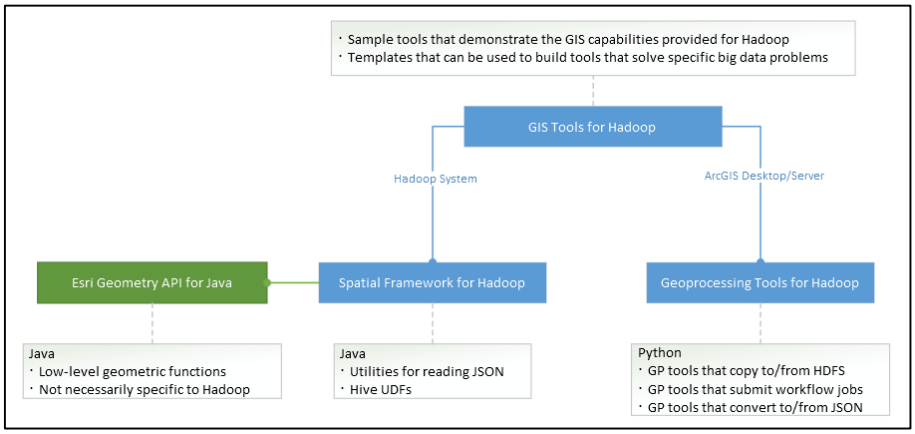

Figure 4: ESRI Tools architecture

GeoMesa [3] is the third spatial extension that will be examined, which is also open source. GeoMesa uses Accumulo, HBase and Cassandra to provide data persistence and storage of large amounts of geographic data such as point, line or polygons. It also supports real time stream processing of spatial data using Apache Kafka as a bottom layer.
Also through the use of GeoServer, the user can access all those capabilities using APIs such as OGC and protocols such as WFS and WMS.

The architecture of GeoMesa is shown in Figure 5.

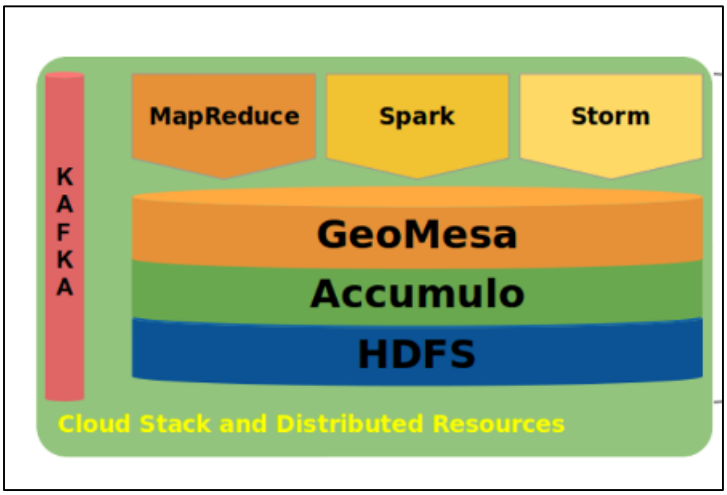

Figure 5: GeoMesa architecture

Specifically, GeoMesa supports cloud-based data storage such as Apache Accumulo, Apache HBase and Google Cloud Bigtable. For data streaming Apache Kafka is used.

Apache Storm helps user with processing and manipulating streaming data and Apache Spark allows analytics of the stored data or of real time streaming data.

To allow access to the stored data in Accumulo, GeoMesa implements GeoTools through HTTP access. Figure 5 shows an overview of the structure.

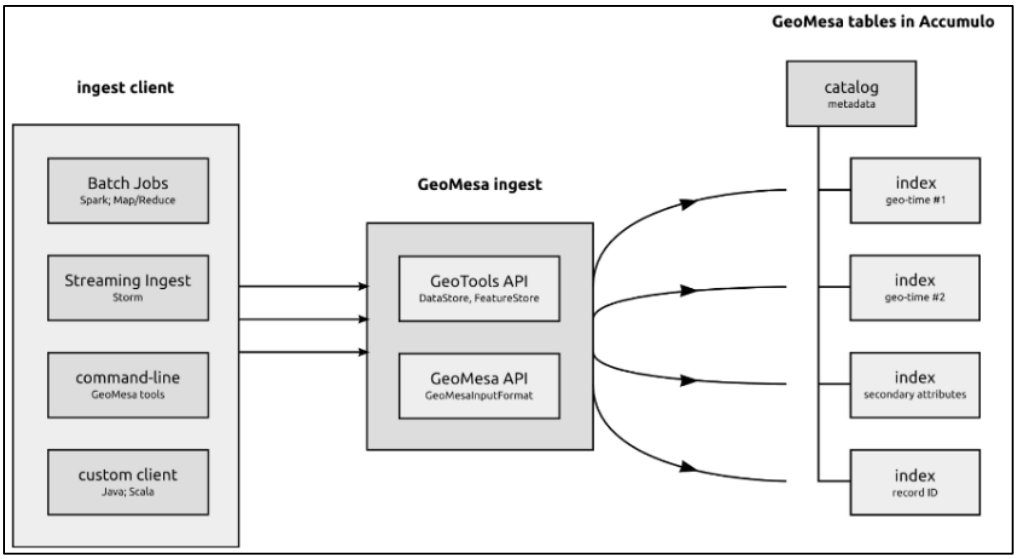

Figure 6: GeoMesa tools system

To ingest streaming or regular data, GeoMesa uses a range of tools (shown in Figure 6) such as Apache Hadoop or Apache Storm. GeoMesa uses a key-value approach to

store data in the database. It is a type of NoSQL database in which every record of data has a specific key that identifies the data. This key unlike relational databases is not just a sequential number; rather it stores properties of the actual data. For example orders from customer can have as key the order number. Then queries for customer orders can be more efficient. This can be expanded in spatio-temporal data. As explained in [3]:

"To store spatio-temporal data, we need to create a key that represents the time/space location of the record. GeoMesa uses this system to store locations as points along a special line that visits all the sectors of a map, like the red line shown in Figure 7:

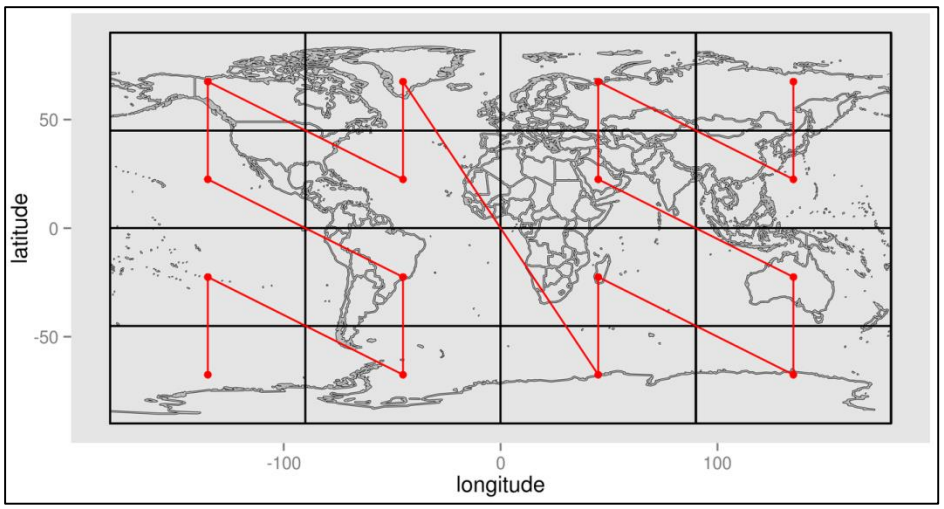

Figure 7: GeoMesa Zcurve example

This red line is known as a space-filling curve, or to be more specific, a Z-Curve. This line visits each cell exactly once, establishing a unique ordering of cells. Each point in this curve can be assigned a sequential value, letting GeoMesa represent what might have been a latitude-longitude pair as a single integer. This is great for representing two-dimensional data in a one-dimensional key, as is the case with a key-value datastore. Even more significantly, these space-filling curves can be adapted for n dimensions, letting GeoMesa linearize n dimensions of data in a single dimension."

The index used by GeoMesa uses a specifically constructed key by taking advantage of the three dimensions (latitude, longitude, timestamp) to create the following key as it appears in Figure 8.

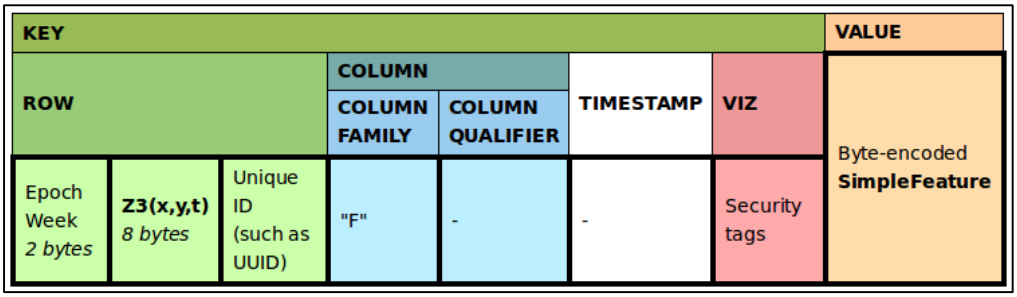

Figure 8: GeoMesa index key

Z3 encoding is used in the key section constructed by the three dimensions. The value is a byte-encoded SimpleFeature. SimpleFeature is a standard of OGC that defines a common storage and access model of two or more dimensional data. It is used by many popular DBs such as MySQL and PostGIS. [4]

Following GeoMesa, we will present **LocationSpark**. LocationSpark is a distributed framework that gives users the ability to work on in-memory data and is built upon Apache Spark. This is the biggest difference of LocationSpark and the previous tools that were analyzed. LocationSpark architecture is shown in Figure 9.

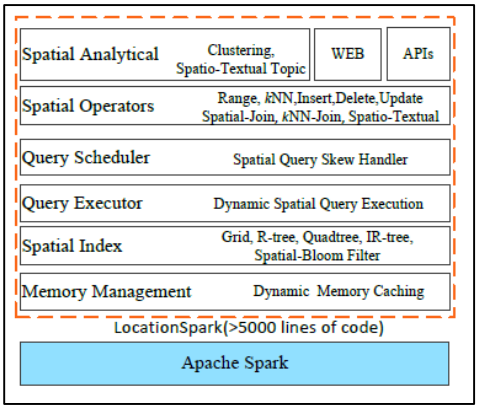

Figure 9: LocationSpark architecture

LocationSpark stores data in a key-value format. The key can be any two-dimensional point, for example latitude-longitude, rectangle etc. The value can be specified by the user that adds information about the key.

It supports a large number of spatial queries like KNN and spatial-join as well as spatial analytics like skyline computation. The main advantage of LocationSpark is its ability to use in-memory intermediate data instead of having to store data of each intermediate step in HDFS. It also uses access frequencies and timestamps which are recorded in the spatial index to determine the usage frequency of those data and

depending on the results it can decide which data should be stored into the disk or should be cached into memory. A problem with spatial data and queries is data skew. As shown in Figure 7 the query scheduler is responsible of minimizing data skew. Data skew means that data is unevenly distributed in the nodes and some specific nodes have a lot more data than others. This can cause performance problems when running the spatial queries. To solve this problem, LocationSpark collects statistical data about the use of each partition. Then the query scheduler will use these data to allocate more workers to specific partitions in order to increase speed. Next the query executor will decide how it will use the available spatial indexes to minimize memory usage and runtime delays. For example it can decide if it is worth to build an index over the query data and traverse both the query index and the data index at the same time.

LocationSpark uses a global index which has the target to ensure that each partition has the same amount of data. Each partition will also have its own local index. A good addition to LocationSpark involves spatial range queries. During those queries, it has to be determined which data partitions overlap the range query. This can cause a lot of network communication because data have to be moved between partitions. LocationSpark uses a filter called sFilter which is embedded into the global index and can better understand if the partition overlaps with the range query.

The next system that will be presented is **MD-HBase**. MD-HBase [6] is focused on supporting location based services. Location based services put a high load on both traditional DBMSs due to their high insertion rates and real time queries but also on key-value stores which despite the fact that can handle large scale operations, do not support multi-attribute access which location based services need. To overcome these issues, MD-HBase uses a two index approach. a K-d tree and a Quad tree over a key-value store. Using a key-value store MD-HBase can sustain a high insertion rate and can ensure fault tolerance and high availability. On the other hand the overlaid index layer can support real time data processing on multi-dimensional queries. The architecture of MD-HBase is shown in figure 10.

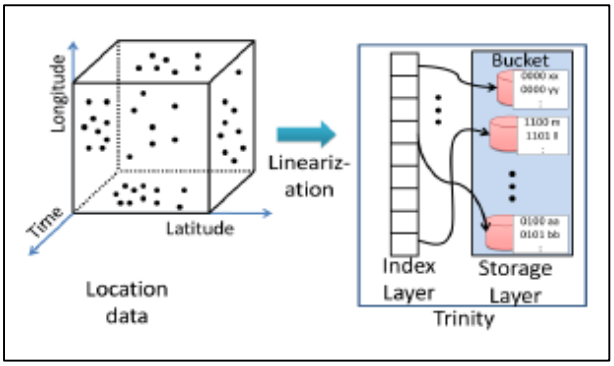

Figure 10: MD-HBase architecture

The index layer splits the space into splits called subspaces. These subspaces are mapped to abstract physical locations called buckets. This mapping can be one-to-one, many-to-one or many-to-many depending on the need of the application. Then the index layer uses a technique called 'longest common prefix naming scheme' to map the multi-dimensional index to a single dimension list which in turn points to buckets on the storage layer. On the other hand the storage layer is based on HBase. In HBase a table is a collection of splits, also called regions. Each of these regions stores a range partition of the space. MD-HBase uses three different techniques to store data:

1) Table share model. This model stores all buckets in a single HBase table. Data use their Z-value, which is the key, to be sorted.

2) Table per bucket model: This model stores buckets in their own HBase table.

3) Hybrid model: It uses both table share and table per bucket model.

4) Region per bucket model: In this model, each HBase region has its own bucket.

Experimental results, done by the authors, report a better performance than MapReduce systems.

Lastly we will examine a different that provides analysis for scientific or geospatial data which is **SciDB**. SciDB [5] is different than the previous tools we present because it does not necessarily rely on Hadoop to run. SciDB does support running on top of HDFS but user is free to choose.

The biggest difference between SciDB and regular databases is that SciDB uses arrays to maintain the data instead of tables. As described in [5] "*Each SciDB array has a name and consists of an ordered list of named dimensions that define the array's shape. Each cell in the space implied by the product of an array's dimensions contains a tuple which is a record of one or more named, typed,*

attributes." User of SciDB use algebra operations that are supported in order to do math on top of those arrays. Those operations include simple filtering of data, combination of multiple arrays and even array factorization and image processing. Some operators like regrid will even change the shape of the array and this includes many other operators that can change the shape and the attributes of the output array compared to their inputs.

One other difference with typical SQL databases is that SciDB handles missing data using up to 128 difference codes instead of just null. This means that the user can specify different level and meaning to missing data.

In order to run the queries SciDB uses a pipeline execution. This means that it reads the data from the disk and passes these data to every instance in a typical SciDB installation. These queries can have many operators and data are moving from one operator to the other. But in contrast with Hadoop and Map/Reduce, SciDB does not store intermediate results (similar to LocationSpark). Instead it immediately passes the data to the next operator in the pipeline until it reaches the final result. These SciDB instances can run using basic file system APIs and plain TCP/IP, so they do not necessarily depend on systems like HDFS, although the can run on top of them. When the query is complete, SciDB will store the data on its instances. The way this happens is again different than what happens with Hadoop. SciDB uses a share-nothing approach. Data are stored in specific instances and computations are done locally. Every instance does not know all the data that exist but only the part that is stored locally, in contrast with Hadoop and HDFS where that data are shared between all nodes. Before the query is run, SciDB will split the query and each instance will run its own share of it. SciDB uses a process known as Multidimensional Array Clustering to map data to the actual physical locations, this approach means that there is no global index, and the computations start immediately at the local level. Also when SciDB stores the data it will do it using a column-store strategy instead of the traditional row-store. This has a couple of advantages. It minimizes I/O for scientific data, which they have a lot of attributes, it stores the attributes physically close and encoding these data provides more efficient queries.

4. Analysis

We will analyze the code of SpatialHadoop and Pigeon through spatial queries. For the execution of all commands we use Linux.

4.1 SpatialHadoop Structure

SpatialHadoop is composed of four layers. From bottom to top we have the storage layer, the MapReduce layer, the operations layer and the language layer.

The storage layer is responsible for storing the spatial indexes. Traditionally, Hadoop works on non-indexed files. But in order to achieve good execution time we need indexes for spatial queries. Existing indexes cannot be applied to Hadoop. SpatialHadoop creates specialized spatial indexes for use on top of HDFS. The structure of the index includes a global index and local indexes. The global index will partition the data across multiple machines. So depending on the partition that each spatial object belongs, it will go to a specified machine. Below the global index we have a local index for each partition.

The MapReduce layer is the component that will use the spatial index for spatial queries. SpatialHadoop uses specialized components to take advantage of these spatial indexes. A SpatialFileSplitter is used for the global index, and a SpatialRecordReader for the local index. As we described above MapReduce framework consists of two phases, map and reduce. For SpatialHadoop during the map phase, the SpatialFileSplitter will receive the raw data and use a filter function to read the global index and prune data that are not part of the query, it will then split the data and send the splits to map tasks. Each map task will read the split using the SpatialRecordReader and create (k, v) pairs, based on the local index and depending on the query that was used. Each of these pairs corresponds to a record in the split.

The Operations layer consists of the various spatial operations that are supported by SpatialHadoop, for example RangeQuery or SpatialJoin. A RangeQuery will first use the SpatialFileSplitter to remove partitions that are not part of the query (using the global index), and then use the SpatialRecordReader to find the spatial objects that are inside the query range using the local index.

The language layer supports a specific language to be used in order to extend the below functionality. SpatialHadoop uses Pigeon, which is a spatial extension to Pig. Pigeon conforms to the OGC standard. Also it can use user defined functions

(UDFs) in order to extend the functionality of existing queries. Pigeon will be described in more detail on a next chapter. [10]

Following we will start analyzing the code of SpatialHadoop. We will start with the indexing process at chapter 4.2 and on chapter 4.3 we will analyze the query process. Finally on chapter 4.4 we will describe Pigeon. All these three chapters are based on SpatialHadoop 2.2.

4.2 Analyzing SpatialHadoop Indexing

Unlike Hadoop, SpatialHadoop trying to place, for example polygons, that are close together on the same node for faster queries. This process is achieved using the command index which uses already created GIS data that are in HDFS and the MapReduce framework saves them based on the index selected. The two supported indexes supported are the grid and RTree. SpatialHadoop makes use of a customized InputFormat, OutputFormat and RecordReader to read the data from the index and then place them accordingly.

In case of the grid index only a global index is used that is stored in the file _master, while in case of the RTree index a global index which resides in the _master file is used but also local indexes for each partition that resides at the beginning of each partition and before the actual data.

Before going on we have to clarify that when we talk about data file we mean the folder with the files or file containing our data. And when we talk about partition we mean each file created after the indexing and contains data and the local index if one exists. Finally the term shape means any line in the data file that contains a spatial shape.

Visualization of the data of the IMIS data (downloaded from http://chorochronos.datastories.org/) set are shown in figure 11 by executing the command **bin/shadoop plot imis_data output.png shape:ogc -vflip -fast**

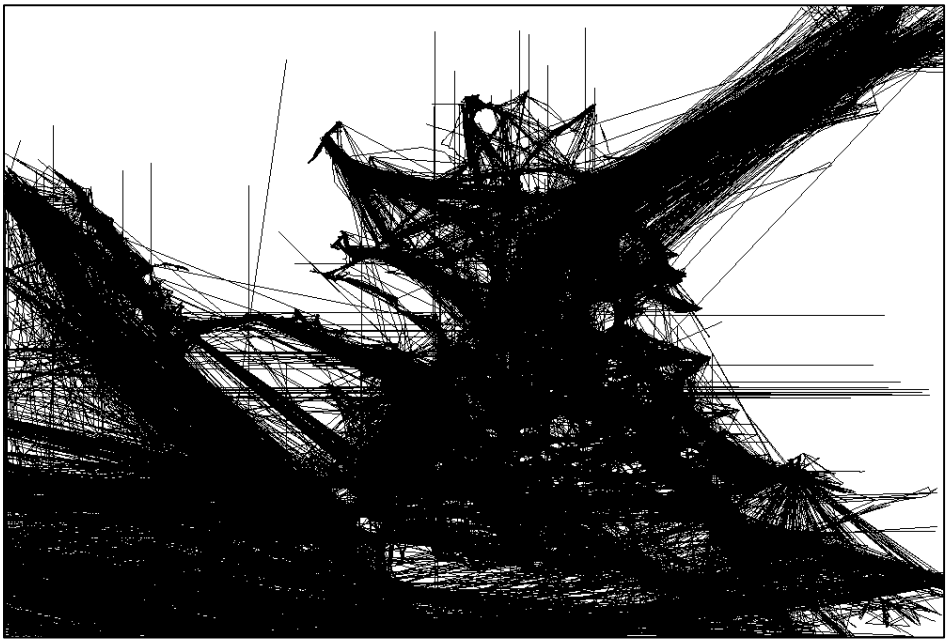

Figure 11: IMIS 2d dataset plot

Using all the data of imis dataset we create an RTree index by running **bin/shadoop index imis_data imis_data.rtree sindex: rtree shape: ogc**. **Attention should be paid to the fact that for the indexing to work, the geometry type must be at the beginning of each line of the file imis_data and that the separator has to be a comma (,)**. The above command creates an RTree index in imis_data.rtree folder.
This file contains _master file where the global index is, and the partitions (which the file _master points to), which each contain the local index and data.

The **Repartition.java** file is what will perform the indexing. If the data file is small, then this is done in local mode, otherwise it runs in MapReduce mode (as in our own).

For start MapReduce is configured and the appropriate classes are selected for InputFormat, OutputFormat, RecordReader and MapReduce, as we see in the following code fragment.

```
job.setMapperClass(RepartitionMap.class);
job.setInputFormat(ShapeInputFormat.class);
job.setOutputFormat(GridOutputFormat.class);
job.setReducerClass(RepartitionReduce.class);
job.setOutputCommitter(RepartitionOutputCommitter.class);
```

Now MapReduce is executed with the command.

```
JobClient.runJob(job);
```

Initially the method Repartition.repartitionMapReduce performed (Path, Path, OperationsParams) is executed. This method aims to create the cells that will form the global index if it is not already created. Depending on the type of index, the process selected is different. In the case of RTree, execution continues with the Repartition.packInRectangles method (FileSystem, Path [], FileSystem, Path, long, S), which does the following. Initially reads a part (sample) of the data file using the method Sampler.sampleMapReduceWithRatio (Path [], ResultCollector <T>, OperationsParams) and from the shapes that it reads, it chooses center points and generates an approximate MBR by expanding the MBR for each center point, and which MBR is used for an initial determination of the variable gridInfo which is of type global index (which at that time consists of only one cell, i.e. the specific MBR) and through which the method GridInfo.calculateCellDimensions (int) runs which will make an initial calculation of the number of cells that will form the global index.

The code of the above method is shown below.

```
public void calculateCellDimensions(int numCells) {
  int gridCols = 1;
  int gridRows = 1;
  while (gridRows * gridCols < numCells) {
    // ( cellWidth > cellHeight )
    if ((x2 - x1) / gridCols > (y2 - y1) / gridRows) {
      gridCols++;
    } else {
      gridRows++;
    }
  }
  columns = gridCols;
  rows = gridRows;
}
```

After running the above method we have an MBR that we parted on specific columns and rows, i.e. cells, and which is stored in a variable of type gridInfo.

Then that variable is passed as a parameter for Rtree.packInRectangles method (GridInfo, Point []) method, together with the table of all the center points calculated previously. This method by taking this original MBR, is trying to optimize the cells to contain approximately the same number of points from the sample. These

cells are created and stored in a variable of type CellInfo []. Part of these cells is shown below and also in figure 12.

```
Cell #1 Rectangle: (-1.7976931348623157E308,-1.7976931348623157E308)-(23.50675833333335,36.256275)
Cell #2 Rectangle: (-1.7976931348623157E308,36.256275)-(23.50675833333335,37.71416666666666)
Cell #3 Rectangle: (-1.7976931348623157E308,37.71416666666666)-(23.50675833333335,37.939775)
```

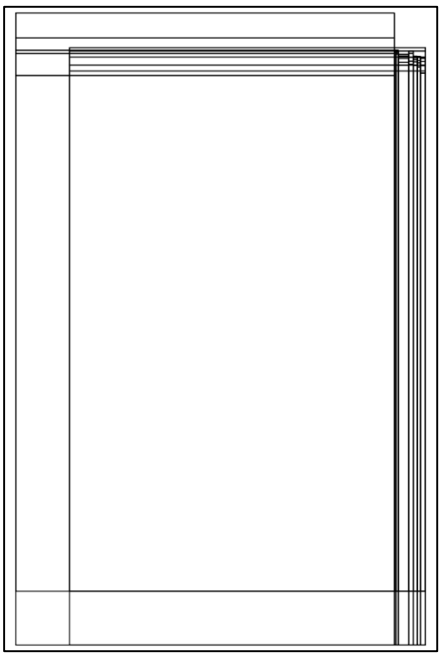

Figure 12: SpatialHadoop Grid index plot

Once these cells are calculated, and according to the index that we have chosen, the appropriate Mapper is chosen between *repartition with replication* and *repartition without replication*. In our case for RTree index the Mapper **RepartitionMapNoReplication.class** is used. As Reducer the **RepartitionReduce.class** is set which is also the same for each index. As input formatter the **ShapeInputFormat.class** is defined who is also identical for each index. The InputFormatter specifies an appropriate RecordReader to read our data in the file and pass the data to the map function. As OutputFormatter which will store the data set the **RTreeGridOutputFormat.class** is chosen.

Then the **Input Format phase** starts. In this phase the InputFormatter initializes the RecordReader. The RecordReader once initialized, begins to read the splits of the data files that have been returned by the `getSplits ()` method. It continues to read line by line the data checking each time the file if it has finished with the `SpatialRecordReader.nextLine ()` method. Inside this method we see

the special handling if we choose RTree index file, in which we should skip the RTree local index header stored at the beginning of the file.

```
pos += RTree.skipHeader(in);
```

The RecordReader after reading each line returns (key, value) pairs in the Mapper. The key contains the cellMBR which is the cell to which the partition belongs which contains any shape based on global index if there is already one, otherwise it contains an invalid shape, and the value contains the data line, i.e. each shape.

In the next stage the **Map phase** begins. Initially the method Repartition.RepartitionMapNoReplication performed <T extends Shape> runs, particularly the method Repartition.RepartitionMapNoReplication.map (Rectangle, T, OutputCollector <IntWritable, T>, Reporter) therein. The process map (), shown below, takes these (key, value) pairs and sends them to the RecordReader.

```
public void map(Rectangle cellMbr, T shape,

  OutputCollector<IntWritable, T> output, Reporter reporter)
  throws IOException {
Rectangle shape_mbr = shape.getMBR();
if (shape_mbr == null)
  return;
double maxOverlap = -1.0;
 int bestCell = -1;
// Only send shape to output if its lowest corner lies in the cellMBR
// This ensures that a replicated shape in an already partitioned file
// doesn't get send to output from all partitions
if (!cellMbr.isValid() || cellMbr.contains(shape_mbr.x1, shape_mbr.y1)) {
  for (int cellIndex = 0; cellIndex < cellInfos.length; cellIndex++) {
    Rectangle overlap = cellInfos[cellIndex].getIntersection(shape_mbr);
    if (overlap != null) {
      double overlapArea = overlap.getWidth() * overlap.getHeight();
      if (bestCell == -1 || overlapArea > maxOverlap) {
        maxOverlap = overlapArea;
        bestCell = cellIndex;
      }
    }
  }
}
if (bestCell != -1) {
  cellId.set((int) cellInfos[bestCell].cellId);
  output.collect(cellId, shape);
} else {
  LOG.warn("Shape: "+shape+" doesn't overlap any partitions");
```

```
    }
  }
```

If the key (i.e. cellMBR) is invalid or if it is valid and the lower left corner of the shapeMBR (which we count through the value, which contains the shape) is within the cellMBR (this check is done so as to not include more than one shapes, which are covered by many cellMBR) then it checks all the cells that we calculated previously and sees which one intersects with larger overlap with this shapeMBR. Using the cellID and the shape of this cell, a (key, value) pair is formed. This (key, value) will be sent by the Mapper to the Reducer. The Mapper will continue to read until all the shapes are read and sends the Reducer for each shape the cellID it belongs.

Below we see a sample of these intermediate (key, value) pairs created by the Mappers.

```
cellid: 5 shape: Rectangle: (981227.3105095724,332176.6021628698)-(981248.3105095724,332195.6021628698)
cellid: 4 shape: Rectangle: (497662.4341413035,985988.3460651748)-(497715.4341413035,986070.3460651748)
cellid: 1 shape: Rectangle: (214548.39263360525,388885.2276846203)-(214572.39263360525,388971.2276846203)
```

After finishing the Map phase, we proceed to the next phase which is the Partition phase. In this stage the intermediate (key, value) pairs created by the Mappers are directed to specific Reducers, so that for example, any shape that belongs to a specific cell is directed to a particular Reducer. So each Reducer takes all values (shapes) of a particular key (cellID). The process that controls this is the Partitioner, where Hadoop by default computes a hash value. SpatialHadoop does not use a customized Partitioner. During this phase the sorting of keys is also done.

After the Partition phase, we continue to the **Reduce phase** which is shown below.

```
  public void reduce(IntWritable cellIndex, Iterator<T> shapes,
      OutputCollector<IntWritable, T> output, Reporter reporter)
      throws IOException {
    T shape = null;
    while (shapes.hasNext()) {
      shape = shapes.next();
      output.collect(cellIndex, shape);
    }
    // Close cell
    output.collect(new IntWritable(-cellIndex.get()), shape);
  }
```

Here all the intermediate keys from the Mappers are taken from the Reducers. The Reducers receive a list with key a specific cellID and value a list of all shapes belonging to this cell. Then using Iterator it runs across this list and generates the same (key, value) pairs (already sorted). But every time it runs out of data from a list that contains a specific cell, an additional negative cellID is added, which will be used by the OutputFormatter to write data to files. A sample of these (key, value) pairs is shown below (the negative cellIndex is also shown).

```
cellIndex: 1 shape: Rectangle: (285046.9044709395,66127.0327746073)-(285099.9044709395,66136.0327746073)
cellIndex: 1 shape: Rectangle: (204314.60092697517,57576.037546577696)-(204394.60092697517,57584.037546577696)
cellIndex: -1 shape: Rectangle: (204314.60092697517,57576.037546577696)-(204394.60092697517,57584.037546577696)
cellIndex: 5 shape: Rectangle: (933426.9695659386,298756.318296197)-(933438.9695659386,298840.318296197)
```

Once the Reduce phase is completed we continue to the **Output Format phase**. In this phase the goal is to write the data files to the HDFS. It tries, in the case of RTree index, to create the local index and store data or in the case of grid index simply store the data. To achieve this, the method RTreeGridRecordWriter.writeInternal (int, S) is executed. It reads the (key, value) pairs of the Reducers and check whether we can insert it in the RTree by checking the degree of RTree. (The degree of the RTree is a constant number calculated by the formula: degree = 4096 / RTree.NodeSize, wherein RTree.NodeSize is equal to 36. However there is another method SpatialHadoop RTree.findBestDegree (int, int) for better calculating RTree degree, but which is not used yet).

If they can be imported, then the method GridRecordWriter.writeInternal (int, S) is executed, which keeps the shapes in an output stream. As shapes are imported in the appropriate cells, the MBR of these cells is changed accordingly. These cells in their final form will be the global index. When exceeding the degree of RTree or if you encounter a negative cellID (which means that the data for this cell is finished) then the retained shapes must be stored in the file and a new stream must be opened. This process is done by RTreeGridRecordWriter.flushAllEntries method (Path, OutputStream, Path). Initially all the data is written to a file. Then this file is read again to create a local rtree index. This is done by RTree.bulkLoadWrite method (byte [], int, int, int, DataOutput, boolean). After calculating the RTree, it is stored at the beginning of the file before the data. And finally a new entry is added in the file _master (where the global index

resides) which contains the number of the cell, the MBR of that cell and the file containing the data.
The process continues until the RecordReader reads all the data and completes the indexing.

The _master file of RTree global index is shown below. Each entry in the _master file shows which file contains the shapes of this MBR. Each of these files contains its own RTree local index. An example is shown below:

```
4,19.000025,34.60005,29.1073916666667,40.6386,part-00000_data_00004
4,19.0,34.6000666666667,29.39878,40.6373666666667,part-00000_data_00004_1
4,19.0000083333333,34.6,29.9890166666667,40.63796,part-00000_data_00004_2
4,19.0000266666667,34.6001166666667,29.3920166666667,40.5436333333333,part-00000_data_00004_3
```

In figure 13 we can see the final RTree global index which is created for the whole imis dataset.

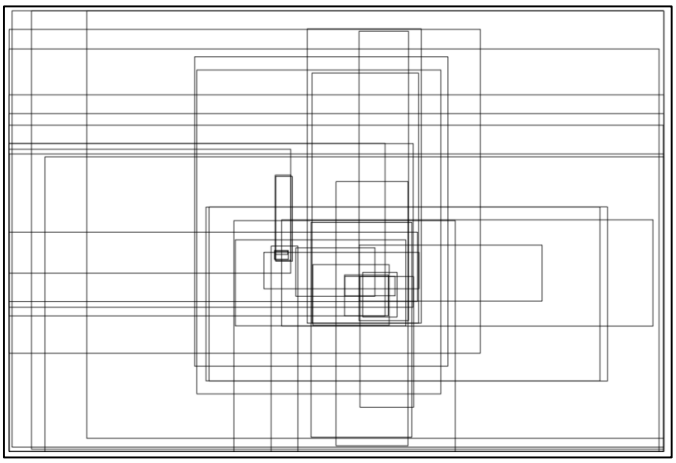

Figure 13: SpatialHadoop RTree index plot

The above procedure has the result that the files contain shapes that are close together (based on the index we used). These files will be stored in Hadoop to specific nodes so queries on the shapes will run faster, because using the indexed files, the generated MBRs are much less.

4.3 Analyzing SpatialHadoop queries

Then we will perform some spatial queries using SpatialHadoop and the index we created.

- The first query that we will perform is a **Range Query**.

The command is as follows **bin/shadoop rangequery imis_data.rtree rq_results rect: 25,37,24,36 shape: ogc.** Using this command we will find which shapes are inside or intersect with the square with coordinates 25.37 and 24.36. The file that executes the command is **RangeQuery.java**. Specifically, initially the method RangeQuery.rangeQuery (Path, Path, Shape, OperationsParams) defines the method that will filter the data. Filtering is the stage at which a selection is made of the partitions that will be used in the **Input Format phase**, depending on our query range using the global index. This method is RangeQuery.RangeFilter () and will run before Map Phase. Its results are the partitions that will be read by the RecordReader and send to the Mapper. To achieve this, the execution passes to SpatialInputFormat.listStatus (JobConf) method where a blockFilter is defined which is an object of class RangeFilter (). After that all the files in the folder we have defined are read during the execution of the method (in our case only imis_data.rtree) and for each one the method SpatialInputFormat.listStatus (FileSystem, Path, List <FileStatus>, BlockFilter) runs. This method checks for a global index on the data files, if yes then for each partition of the global index the method RangeQuery.selectCells (GlobalIndex <Partition>, ResultCollector <Partition>) runs, which checks if the MBR owned by each partition intersects or is contained in the query range rectangle. For each partition that has one of these criteria a customized collect method runs which is shown below, which adds the path of each partition in the variable result which will be returned to SpatialInputFormat.listStatus (JobConf) method.

```
public void collect(Partition partition) {
        try {
          Path cell_path = new Path(indexDir, partition.filename);
          if (!fs.exists(cell_path))
            LOG.warn("Matched file not found: "+cell_path);
          result.add(fs.getFileStatus(cell_path));
        } catch (IOException e) {
          e.printStackTrace();
```

```
    }
  }
```

These partitions will be read by the RecordReader which will send to the Mapper (key, value) pairs, where the key is the cellMBR and the value contains an RTree or shape (in our case RTree). Having found the partitions we need for our query via the global index, execution returns to RangeQuery.rangeQuery (Path, Path, Shape, OperationsParams) method, which according to the number of the partitions that we found, i.e. splits, will choose whether the query will run on local or MapReduce mode and pass on to the **Map Phase**. In the case of MapReduce, execution continues with RangeQuery.rangeQueryMapReduce (Path, Path, Shape, OperationsParams) method, where appropriate classes for the execution of the query are set. We check what type of index we have (in our case it is RTree and class **RTreeInputFormat.class** will be used as InputFormatter), and if we have a replicated index (in the case of RTree we don't, so using the Mapper will use the class **RangeQueryMapNoDupAvoidance.class**).

The Mapper will create (key, value) pairs with key Null value and a shape, while zero Reducers are defined. The method RangeQuery.RangeQueryMapNoDupAvoidance.map (Rectangle, Writable, OutputCollector <NullWritable, Shape>, Reporter) takes the pairs from the RecordReader and in the case of RTree where there is a local index it uses Rtree.search (Shape, ResultCollector <T>, int , int) method to check the RTree (i.e. the local index sent by the RecordReader) and return the shapes that satisfy the query range.

For the Range Query there is no **Reduce phase** and **Output Format phase** will only store the data. Plotting the result of the range query with the command **bin/shadoop plot rq_results output.png shape:ogc -vflip -fast,** we can see the results in figure 14.

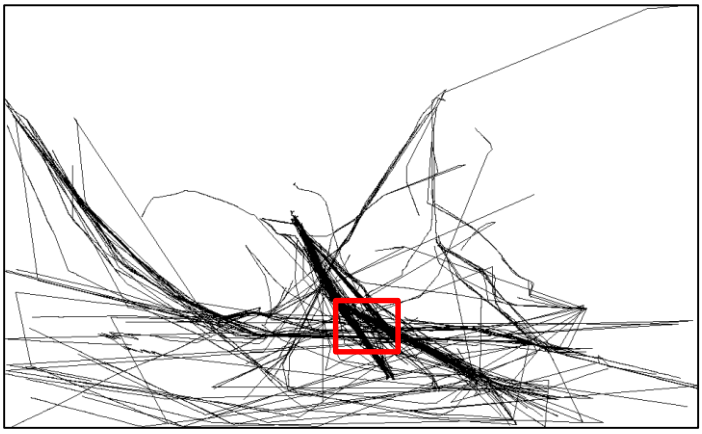

Figure 14: SpatialHadoop RangeQuery query result plot

- The next query that we will execute is the **KNN**.

The command is as follows **bin/shadoop knn imis_data.rtree knn_results point:25,37 k:10 shape:ogc .** We will find the 10 closest shapes at the point with coordinates 25.37. The file that will execute the command is **KNN.java**. Starting from KNN.main (String []) method, it is checked whether closeness is set otherwise the default value is -1. If yes, then the center points of the MBRs of the global index are calculated and with the following piece of code they are assigned in the variable queryPoints [i].

```
Partition partition = iterator.next();
double cx = (partition.x1 + partition.x2) / 2;
double cy = (partition.y1 + partition.y2) / 2;
double cw = partition.x2 - partition.x1;
double ch = partition.y2 - partition.y1;
int signx = ((direction & 1) == 0)? 1 : -1;
int signy = ((direction & 2) == 1)? 1 : -1;
double x = cx + cw * closeness / 2 * signx;
double y = cy + ch * closeness / 2 * signy;
queryPoints[i] = new Point(x, y);
```

Otherwise, the variable contains only the point we specified. Then the process KNN.knnMapReduce (Path, Path, Point, int, OperationsParams) runs wherein MapReduce is adjusted. First it checks for the type of index and set the appropriate InputFormatters and Mappers. Specifically for non RTree indexes a Combiner class must also be defined. Once MapReduce is set, the first iteration begins and we pass to the **Input Format phase** where, as with the Range Query, also here there is a selection of partitions associated with our query range using the filter **RangeQuery.RangeFilter.java** and the global index. After finding these partitions then the RecordReader passed them to the Mappers during **Map phase** in the form of

(key, value), wherein the key contains cellMBR and the value a shape or an RTree (depending on the index). The next method that runs is KNN.KNNMap <S extends Shape> containing specific separate map functions depending on whether the index contains local index (as herein). Specifically, for each (key, value) pair that the RecordReader sends, the Mapper runs the RTree.knn (double, double, int, ResultCollector2 <T, Double>) method on the RTree. This method after creating a query range, from the point we have set, with the following code

```
double query_area = ((getMBR().x2 - getMBR().x1) * (getMBR().y2 - getMBR().y1)) * k / getElementCount();
double query_radius = Math.sqrt(query_area / Math.PI);
Rectangle queryRange = new Rectangle();
queryRange.x1 = qx - query_radius / 2;
queryRange.y1 = qy - query_radius / 2;
queryRange.x2 = qx + query_radius / 2;
queryRange.y2 = qy + query_radius / 2;
```

it searches the local index of each partition with RTree.search (Shape, ResultCollector <T>, int, int) method as with the query Range Query, but now the query refers to the queryRange that was set. Upon returning the appropriate shapes it checks if the number is greater than k.

a) If not, then if there are other elements (shapes) in RTree, it doubles the query_radius and reruns the method.

b) If yes, then the shapes that are returned are sorted with QuickSort () and returns the k first shapes.

c) If there are no other elements in RTree then it returns.

The Mapper creates (key, value) pairs from the results with key Null and a value type TextWithDistance, which contains the shape in text form and its distance from the queryRange. **Reduce phase** does not exist and in the **Output Format phase** data are written to disk files. After that the command returns back to KNN.knnMapReduce (Path, Path, Point, int, OperationsParams) method wherein it is tested again that the number of results it is less than k.

a) If not, then a test circle is created with center the query point that we have defined and radius the distance to the k^{th} shape (i.e., the final shape that has been written in the partition). This is done by the method Tail.tail (FileSystem, Path, int, T, ResultCollector <T>). Then it is checked by GlobalIndex.rangeQuery

(Shape, ResultCollector <S>) method if there is a MBR in the global index that intersects with this test circle but it did not intersect with the original query point.

a1) If there is, then KNN.knnMapReduce (Path, Path, Point, int, OperationsParams) is executed for a second iteration, but this time with query range the new test circle that had been created.

a2) If there is no such MBR in the global index, the method ends and we find the k nearest neighbors.

b) If yes, then it checks how many MBRs intersect with the query range, and stores the distance of the MBR with the greatest distance.

b1) If the number of MBRs that do intersect is zero then the GlobalIndex.knn (double, double, int, ResultCollector2 <S, Double>) method, which is similar to the method we described for the RTree local index, doubles the query radius until we find k MBRs or until there are no other MBRs in global index, and again the maximum distance from to query range is returned. Having returned to the maximum distance, a circle is created with center the previous point and radius the maximum distance and KNN.knnMapReduce (Path, Path, Point, int, OperationsParams) runs again. In each new execution of KNN.knnMapReduce (Path, Path, Point, int, OperationsParams) the results that were stored in HDFS from the previous iteration are deleted.

After the successful execution of the KNN query we plot the results with the command **bin/shadoop plot knn_results output.png shape: ogc -vflip -fast** and which appear in figure 15.

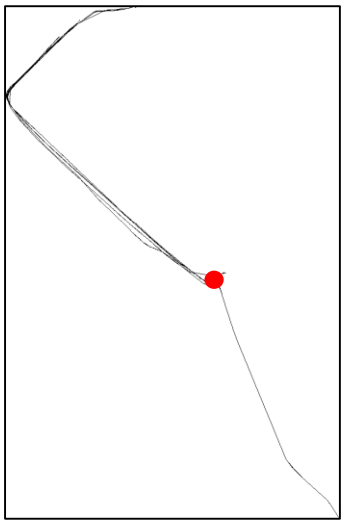

Figure 15: SpatialHadoop KNN query result plot

- Finally we will analyze a **Spatial Join** query.

We will split the data set in two pieces, removing lines from the original .txt file using the Linux commands:

```
sed -n '1,1000000p' imis1month.txt > imis1month_part1.txt
sed -n '2000000,3000000p' imis1month.txt > imis1month_part2.txt
```

These commands will create two data files. Again we use Pigeon to create the geometry columns from the data and using SpatialHadoop we create the RTree indexes for each one. Plotting those two indexes we get the following for the first in figure 16 and second in figure 17 respectively.

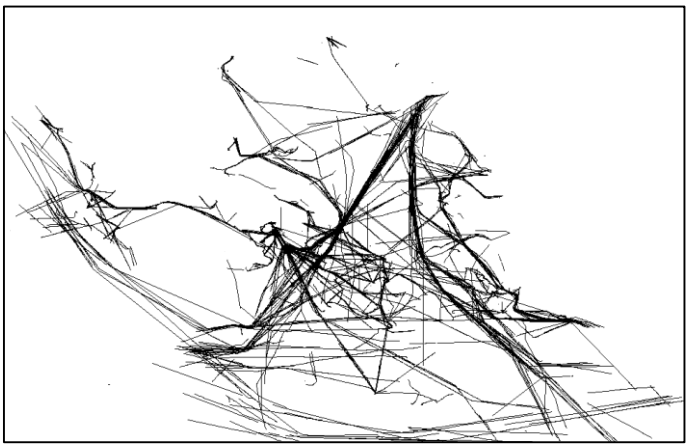

Figure 16: First data split for SpatialJoin query

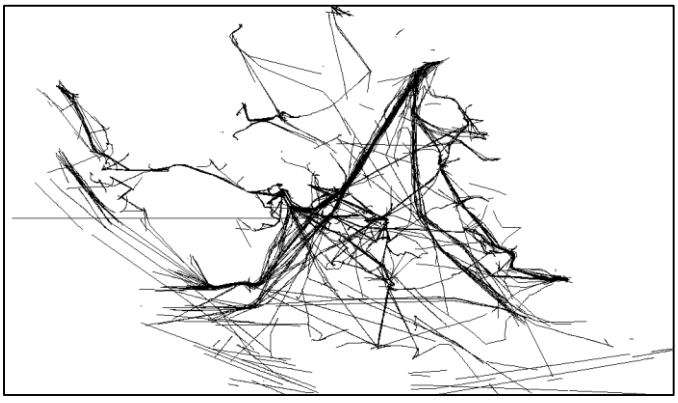

Figure 17: Second data split for SpatialJoin query

The command that will run for the spatial join query is **bin/shadoop dj imis_data_part1.rtree imis_data_part2.rtree sj_results shape:ogc**

The **DistributedJoin.java** file will run the command. Initially in method DistributedJoin.main (String []) it is checked if repartitioning needs to be done

(i.e. re-create the index) of one of the two data files, depending on what we have defined in the command.

a) If we choose to do repartitioning then firstly the method DistributedJoin.selectRepartition (Path [], OperationsParams) runs which will choose which of the two data files will be repartitioned. The data file to be selected depends on certain criteria. If only one data file has an index then the one that does not will be selected, otherwise the smaller file will be selected. Then the method DistributedJoin.repartitionStep (Path [], int, OperationsParams) runs which after extracting the global index of the first data file, it will repartition the second (which we choose by controlling the above criteria) based on the index of the first and using Repartition.repartitionMapReduce (Path, Path, OperationsParams) method which we have already described.

b) If we choose the option for auto repartitioning then the method DistributedJoin.distributedJoinSmart (Path [], Path, OperationsParams) undertakes the task to ascertain whether it should be repartitioned. Sorts the two data files based on their size and calculates the size of the global index for each one based on how many partitions each one contains, otherwise if there is no global index then it calculates the number of blocks in HDFS. It will compare the cost of join with and without repartition according to the following function.

```
cost_without_repartition = gIndexes[0] != null && gIndexes[1] != null ?
      GlobalIndex.spatialJoin(gIndexes[0], gIndexes[1], null) :
      (numBlocks[0] * numBlocks[1]);
cost_with_repartition = numBlocks[0] * 3 + numBlocks[1];
```

The variable numBlocks [] contains the number of blocks or partitions of each data file that we calculated previously. In the case of cost without repartitioning, if any global indexes exist for both files the cost will be calculated by the method GlobalIndex.spatialJoin (GlobalIndex <S1>, GlobalIndex <S2>, ResultCollector2 <S1, S2>) which only runs the method SpatialAlgorithms.SpatialJoin_planeSweep (S1 [], S2 [], ResultCollector2 <S1, S2>). This method takes all shapes of each global index and makes a spatial join on the two indexes and increases a counter for each shape that intersects. This counter will be the cost. If the cost of repartition is less than the cost without repartition then we continue to step **a)** otherwise we move to **c)**.

c) When step **a)** or **b)** is finished or if we did not choose to repartition then the method DistributedJoin.joinStep (Path [], Path, OperationsParams) runs which will set MapReduce. Adjusts the appropriate InputFormatter depending on whether we have RTree indexes (in our case the **DJInputFormatRTree.class** will be the InputFormatter), then the suitable Mapper is selected depending on what kind of index we have (if we have replicated index or not, in our case not, so the Mapper **RedistributeJoinMapNoDupAvoidance.class** is selected). As a filter class, which will filter the data that the RecordReader will receive, is defined the **SpatialJoinFilter.class**. And finally a Reduce phase does not exist here.

Then the **Input Format phase** begins during which a specific InputFormatter is used for binary data (i.e. data that contain duplicate data, as in our case with the spatial join). This InputFormatter is the **BinarySpatialInputFormat.java** and which will create CombineFileSplits from data files that we execute spatial join on. The CombineFileSplits is a specific variable which contains two splits, that is, two partitions from the two global indexes.

a) If an index exists for both data files the filter selects which partitions qualify the criteria. The filter (i.e. the method DistributedJoin.SpatialJoinFilter) makes a spatial join between the two indexes with the method GlobalIndex.spatialJoin (GlobalIndex <S1>, GlobalIndex <S2>, ResultCollector2 <S1, S2>) and tries to find partitions which will intersect and for which the intersection is greater than zero, which means that it skips the partitions that just touch. It returns pairs with these two partitions that satisfy the constraints, and the **BinarySpatialInputFormat.java** creates CombineFileSplits.

b) If even one data file has no index then a Cartesian product is generated from all partitions of each index and the generated pairs are stored in a variable of type CombineFileSplits. All these CombineFileSplits are returned to the RecordReader who is the DistributedJoin.DJInputFormatArray.DJRecordReader and who implements the BinaryRecordReader.BinaryRecordReader (Configuration, CombineFileSplit). The specific RecordReader uses two internal RecordReaders to create (key, value) pairs for the Mappers, wherein the key is a pair of two MBRs, i.e. the MBRs of each partition from the previous CombineFileSplit, and value a pair of the two RTree local indexes of each partition. These (key, value) pairs are sent to the Mappers and the **Map phase** begins. The

Mappers check if the value is an RTree or if it just contains simple array of shapes (i.e. no local index). In our case with an RTree, a spatial join between the RTrees is performed by the method RTree.spatialJoinDisk (RTree <S1>, RTree <S2>, ResultCollector2 <S1, S2>) which is shown below. This method searches both RTrees starting from the root node and traversing each RTree downwardly it checks which MBR pairs between the two RTree intersect using Cartesian product between all MBRs of the two RTrees.

With those that do intersect it checks whether we are in a leaf node level.

a) If yes, then it continues checking the shapes and which one intersects again using a Cartesian product. The ones that do intersect are stored in the output.

b) If not, then it continues to run down the RTree.

This process continues until it checks all the nodes/leaves which intersect.

After finishing the Map phase, there is no **Reduce phase** so we move to the **Output Format phase** where we just store the generated shapes.

We plot the results with the command **bin/shadoop plot sj_results output.png shape: ogc -vflip -fast** and we can see the result in figure 18.

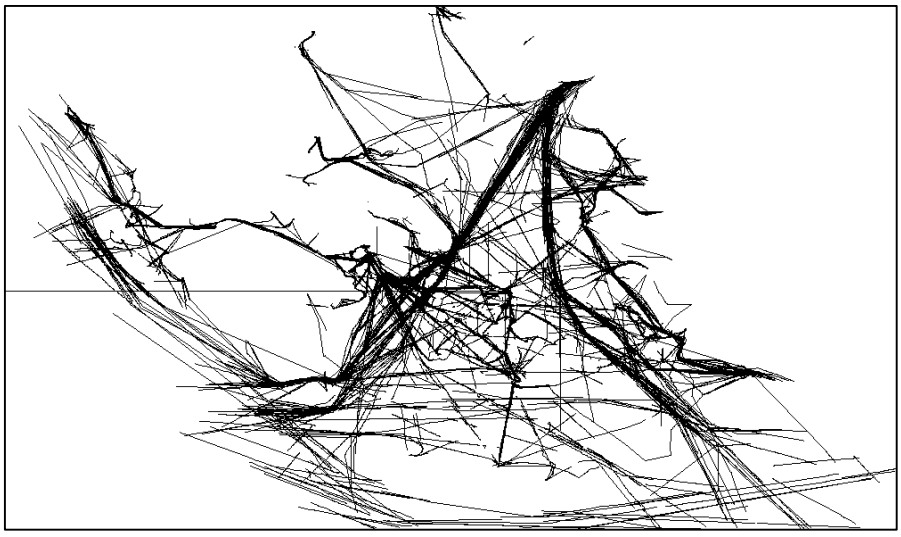

Figure 18: SpatialHadoop SpatialJoin query result plot

4.4 Process data and execute queries with Pigeon

To edit and run queries we used a dataset from http://chorochronos.datastories.org/. We will use SpatialHadoop and Pigeon for performing queries. We will carry out the following Pig script using the methods that Pigeon provides us for processing spatial data. You choose all objects containing more than 10,000 points. Below the script and the explanation is shown.

- We register the necessary .jar files so that we can use spatial methods of Pig from Pigeon in our script, and define the specific methods we use.

```
REGISTER /home/user/hadoop-1.2.1/pigeon.jar;
REGISTER /home/user/hadoop-1.2.1/lib/esri-geometry-api-1.1.1.jar;
DEFINE MakePoint edu.umn.cs.pigeon.MakePoint();
DEFINE MakeLine edu.umn.cs.pigeon.MakeLine();
DEFINE AsText edu.umn.cs.pigeon.AsText();
```

- Here we load the data file and add column names to variables.

```
file = LOAD 'imis1month.txt' USING PigStorage(',') AS (t, lon, lat, obj_id, flag, subtraj_id, subtraj_type, traj_id, port_id:int, port_title);
```

- We remove the first line of data as it contains header info.

```
rawdata = FILTER file BY t != 't';
```

- We choose which columns we want to keep from the data and from the columns lon and lat we create Point type geometry.

```
data = FOREACH rawdata GENERATE t AS time, MakePoint(lon,lat) AS position, obj_id AS vessel, subtraj_id AS sub_trajectory;
```

- We group the rows relating to each specific vessel and sub_trajectory. With this command Pig creates a bag data type that contains all the points we created for each object. This data type has the following format: <A, {<B, C>, <E, F>}>, where the first element is A and the is the element on which the grouping was done, while the second is a bag which contains one or more rows (<B, C>) by one or more elements (B, C), which belong to the A element. Specifically in our case A will be the vessel, while the B, C and E, F will contain the lon, lat coordinates.

```
group _data = GROUP data BY (vessel, sub_trajectory);
```

- We classify the points of each object according to the date.

```
sorted_group = FOREACH group_data {data_sorted = ORDER data BY time ASC; GENERATE group, data_sorted;};
```

- We find the number of points of each object and then the objects that have more than 10,000 points.

```
count_records = FOREACH sorted_group GENERATE group.vessel, group.sub_trajectory, COUNT(data_sorted) AS total;
filter_records = FOREACH (FILTER count_records BY total > 10000) GENERATE vessel, sub_trajectory;
```

- We join all our data to a table containing the records that have over 10,000 points, rest of data take NULL values.

```
nullify_non = JOIN sorted_group BY (group.vessel, group.sub_trajectory) LEFT OUTER, filter_records BY (vessel,sub_trajectory);
```

- We choose only the items that have over 10,000 points.

```
data_filter = FOREACH (FILTER nullify_non BY filter_distinct::vessel IS NOT NULL) GENERATE sorted_group::group, sorted_group::data_sorted;
```

- Creates a line geometry which consists of all the points of each object on our query and then all of data.

```
sub_trajectories_filter = FOREACH data_filter {GENERATE MakeLine(sorted_group::data_sorted.position) AS sub_trajectory_geom, sorted_group::group.sub_trajectory AS sub_traj, sorted_group::group.vessel AS vessel;};
sub_trajectories = FOREACH sorted_group {GENERATE MakeLine(data_sorted.position) AS sub_trajectory_geom, group.sub_trajectory AS sub_traj, group.vessel AS vessel;};
```

- After finishing our queries we convert the geometry column in WKT string format and store it in a file in HDFS. Also we store the dataset.

```
sub_traj_filter_wkt = FOREACH sub_trajectories_filter GENERATE AsText(sub_trajectory_geom) AS line, sub_traj, vessel;
sub_traj_wkt = FOREACH sub_trajectories GENERATE AsText(sub_trajectory_geom) AS line, sub_traj, vessel;
STORE sub_traj_filter_wkt INTO 'query';
STORE sub_traj_wkt INTO 'imis_data';
```

The below script is executed in the grunt shell of Pig. To get the grunt shell of Pig we execute the command pig. Then we can insert all the command of Pigeon we described. It can also be performed via the above website http://83.212.97.255:50070/pigeon.jsp, entering commands in Pigeon script field and

clicking Submit Query. Then we can see the results in the field Relations in figure 19.

Figure 19: Pigeon introduction page

Executing the command **bin/shadoop plot query output.png shape: ogc -vflip -fast** we plot the data. The same effect can be also achieved through another website that SpatialHadoop provides http://83.212.97.255:50070/visualizer.jsp and using Preprocess choice shown in figure 20. Then we can download from the HDFS and open the created image.

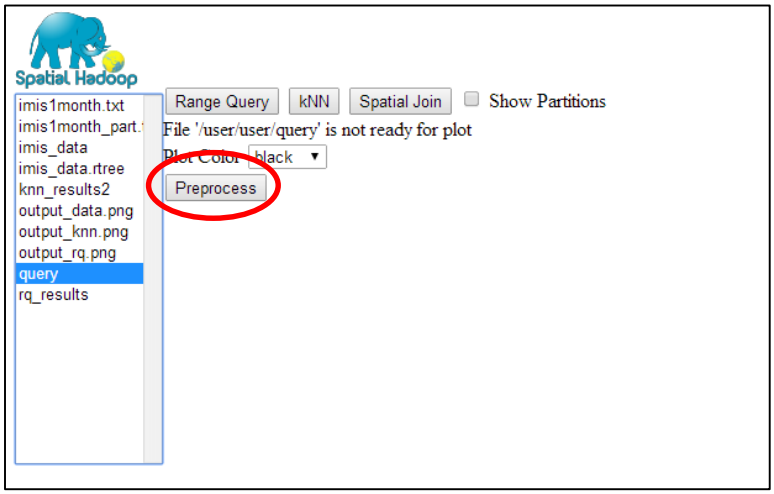

Figure 20: SpatialHadoop Visualizer

In figure 21 we can see the results of the query.

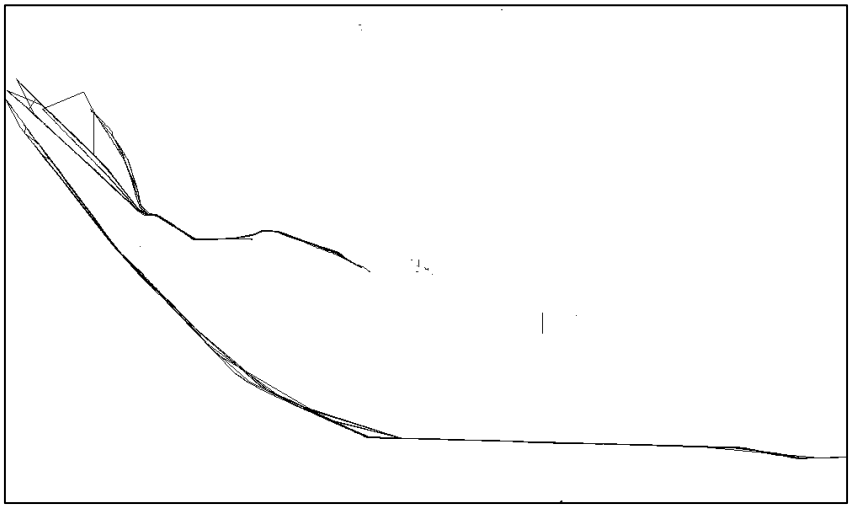

Figure 21: SpatialHadoop Visualizer query plot

At this point we do an analysis of Pigeon to describe how it creates these geometries. Pigeon uses **UDFs** (User Defined Functions) to add the necessary methods in Pig. For each spatial query there is a corresponding UDF. We will analyze two methods we used above, the MakePoint () and MakeLine (), which are responsible for creating lines. We begin with the MakePoint (). This method takes as input two points. Below we see the code, where we analyze each line.

- UDF extends the EvalFunc method in Pig which is the basis for all UDFs.

```
public class MakePoint extends EvalFunc<DataByteArray> {
```

- Implementing the exec method which is executed in each row (tuple) of the input.

```
public DataByteArray exec(Tuple input) throws IOException {
```

- Checking if each tuple contains two elements, as MakePoint () requires two numbers for the point x, y.

```
 if (input.size() != 2)
     throw new IOException("MakePoint takes two numerical arguments");
```

- Using the GeometryParser () we store the two values in two variables x, y.

```
double x = GeometryParser.parseDouble(input.get(0));
double y = GeometryParser.parseDouble(input.get(1));
```

- We create a Point using these two coordinates and esri-geometry-api.

```
Point point = new Point(x, y);
OGCPoint ogc_point = new OGCPoint(point, SpatialReference.create(4326));
```

- We return the result as DataByteArray.

```
return new DataByteArray(ogc_point.asBinary().array());
```

GeometryParser () checks the values that the Pig script returns and depending on the format, it stores the values in a suitable form. If the input is ByteArray then it returns WKB form, if it is string then it returns WKT format, and if it is a number, as here, it returns the same type. Then we analyze the MakeLine () method.

```
public class MakeLine extends EvalFunc<DataByteArray>{
    private GeometryParser geometryParser = new GeometryParser();
    @Override
    public DataByteArray exec(Tuple b) throws IOException {
```

- This creates a data type DataBag because as described above all points are in that form after performing GROUP.

```
DataBag points = (DataBag) b.get(0);
```

- Creating a table of points of size equal to the size of the DataBag.

```
Point[] coordinates = new Point[(int) points.size()];
int i = 0;
```

- For loop to import points from DataBag to the coordinates table.

```
for (Tuple t : points) {
coordinates[i++] = (Point) (geometryParser.parseGeom(t.get(0))).getEsriGeometry();
}
MultiPath multi_path = new Polyline();
```

- In the following for loop we inserts each line (segment) consisting of two elements in the final multi_path which contains all points.

```
for (i = 1; i <coordinates.length; i++) {
   Segment segment = new Line();
   segment.setStart(coordinates[i-1]);
   segment.setEnd(coordinates[i]);
   multi_path.addSegment(segment, false);
   }
```

- Create LineString geometry using esri-geometray-api and return data in the form of DataByteArray.

```
    OGCLineString linestring = new OGCLineString(multi_path, 0,
```

```
SpatialReference.create(4326));
    return new DataByteArray(linestring.asBinary().array());
```

5. SpatialHadoop 3D data extension implementation

5.1 3D Rtree Algorithm

In this section we will describe the structure of Rtrees. This type of index is what SpatialHadoop uses for local indexing and it will need to be modified in our case, in order to support 3D data.

An Rtree is a height balanced index which is structured using nodes in a tree form. Nodes are separated in non-leaf and leaf nodes. Non-leaf nodes point to other non-leaf nodes or leaf-nodes. They contain entries in the form (I, child-pointer). I is the MBR covering all the lower node rectangles and child-pointer is the address of the lower node. The first non-leaf node in the Rtree is called the root node. Leaf nodes point to the actual data objects. Leaf nodes contain entries in the form (I, tuple-identifier), where I is the bounding box of the spatial object and tuple-identifier is the tuple in the database. [8]

If M is the maximum number of entries in a node and m is the minimum number of entries in a node, then the Rtree has the following properties: [8]

- Every leaf node contains between m and M records unless it is the root
- For each index record (I, tuple-identifier) in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object represented by the indicated tuple
- Every non-leaf node has between m and M children unless it is the root
- For each entry (I, child-pointer) in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node
- The root node has at least two children unless it is a leaf
- All leaves appear on the same level

The height of the RTree is calculated by the following formula: ceil(logmN)-1. The RTree searching algorithm is simple. It uses bounding boxes to decide whether or not to descend into a node and search inside a subtree. This way, most of the nodes in the tree are never read. This makes R-trees suitable for large data sets and databases,

where nodes can be paged to memory when needed, and the whole tree cannot be kept in main memory. [9]

The search starts at the root node. The input for the search is bounding box. Every rectangle inside the root node is checked if it overlaps with the input bounding box. If it does then the connected node below needs to be searched also. This is done recursively for every node in the Rtree that overlaps with the input bounding box. When a leaf node is reached then all the spatial objects contained are queried against the input bounding box. [9]

In figure 22 we can see a finished Rtree for 3D data. [9]

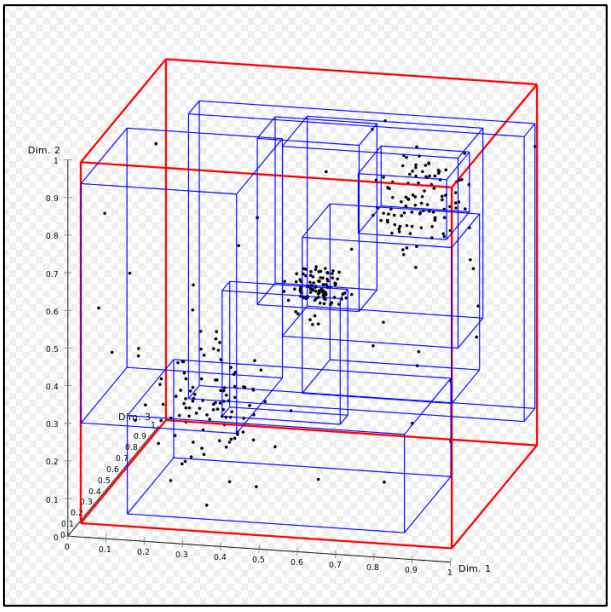

Figure 22: 3DRtree example index plot

5.2 Implementation on SpatialHadoop

In this section we will describe the implementation and the modifications we did using SpatialHadoop to support 3D data. We will use the Trucks.txt dataset from http://chorochronos.datastories.org/. This dataset consists of data lines of x,y coordinates along with a z coordinate that indicates time.

For SpatialHadoop to correctly parse the data, the geometry type needs to be first in every line. We use below python script to convert the Trucks dataset to the format we want. Also in this script we parse the date and time into an epoch object to convert it to a simple number for our Z coordinate.

The end result is a data line in the form of the following regular expression \d+,\d+,\d+ (x,y,epoch_time). Digits are important and affect the RTree header, so the format needs to be as mentioned above. If this format changes then the RTree header needs to be changed.

```
import time

dataFile = open("trucks.txt", "r")
outFile = open("trucks_modified.txt", "w")
datePattern = '%d/%m/%Y_%H:%M:%S'
for line in dataFile:
    line = line.rstrip().split(";")
    dateString = line[2] + "_" + line[3]
    dateEpoch = int(time.mktime(time.strptime(dateString, datePattern)))
    outString = line[6] + "," + line[7] + "," + str(dateEpoch)
    print >>outFile, outString
dataFile.close()
outFile.close()
```

Visualization of trucks dataset using SpatialHadoop command **bin/shadoop gplot trucks.txt output.png shape:point -vflip -fast** is shown in figure 23.

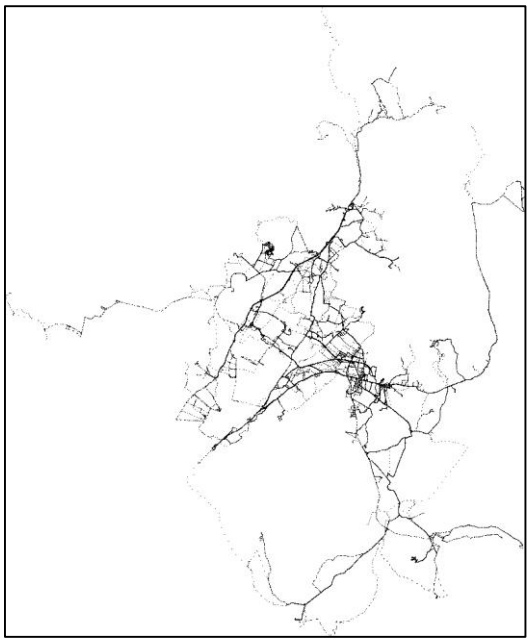

Figure 23: Trucks dataset plot

This is a 2D representation which ignores Z coordinate (epoch time).

In order for SpatialHadoop to support 3D data structures we need to change both the basic objects used by the extension as well as various data structures used to build

the index and run the queries. The spatial objects in SpatialHadoop need to support a new coordinate Z. The first modification starts with the implementation of the Z variable in the basic data objects. *Point* and Rectangle classes have been completely overridden to support a new Z coordinate, as well as Shape interface which is implemented by Cube class (Rectangle class has been renamed to Cube). This Z coordinate will store the time variable as an integer. Regular time format, for example time 15/10/2016 20:32:45, is converted to epoch time 1476552765, and stored in Z variable.

Below is the new constructor for both Point and Cube classes.

```
public Point(double x, double y, double z) {
 this.x = x;
 this.y = y;
 this.z = z;
}
```

```
public Cube(double x1, double y1, double z1, double x2, double y2, double z2) {
  this.set(x1, y1, z1, x2, y2, z2);
}
```

The mathematical formulas inside those classes have been changed to allow 3d data calculations, for example distance and MBR calculations.

CellInfo class has also been modified to use these new basic structures and support 3D data. New constructor is the following. This class holds information for the global index cells, which in our case will be cubes.

```
public CellInfo(int id, double x1, double y1, double z1, double x2, double y2, double z2) {
  super(x1, y1, z1, x2, y2, z2);
  this.cellId = id;
}
```

Storage Layer Modifications:

Storage layer is responsible for creating the global and local indexes and storing the indexes on HDFS. *GridInfo* is the object that corresponds to the global grid index. It holds information about the grid index. Methods have been modified to support grid calculations using new Z coordinate. A new dimension has been added called slices along with row and columns for 2D data. All calculations to support the 3D grid have been modified to use this new dimension. Below most important methods are shown.

```
public CellInfo getCell(int cellId) {
  int col = (cellId - 1) % columns;
```

```
  int row = ((cellId - col - 1) / columns) % rows;
  int slice = (((cellId - col - 1) / columns) - row) / rows;
  double xstart = x1 + (x2 - x1) * col / columns;
  double xend = col == columns - 1? x2 : (x1 + (x2 - x1) * (col + 1) / columns);
  double ystart = y1 + (y2 - y1) * row / rows;
  double yend = (row == rows - 1)? y2 : (y1 + (y2 - y1) * (row + 1) / rows);
  double zstart = z1 + (z2 - z1) * slice / slices;
  double zend = (slice == slices - 1)? z2 : (z1 + (z2 - z1) * (slice + 1) / slices);
  return new CellInfo(cellId, xstart, ystart, xend, yend, zstart, zend);
}

public int getCellId(int column, int row, int slice) {
  return (row * columns + column + slice * rows * columns) + 1;
}
```

These methods calculate the id of a partition (cell) or retrieve a partition given an id. Slice variable will hold the information for the third dimension.

GridRecordWriter, *OGCESRIShape*, *OGCJTSShape*, *SpatialAlgorithms* and *SpatialSite* have also been modified to use Cube class.

From indexing module, *GlobalIndex*, *GridPartitioner*, *IndexOutputFormat*, *Indexer*, *Partition*, *Partitioner*, *RTree*, *STRPartitioner* classes have been adapted to use 3D data.

GlobalIndex supports a basic spatial index, and was modified to support new coordinate.

GridPartitioner is the partitioner type class which will partition the data into multiple partitions that will go to various slaves. An important method here is:

```
public void createFromPoints(Cube mbr, Point[] points, int capacity) throws IllegalArgumentException {
```

This method will create the index given as input a number of Points from a sample (Sample class under operations module). We modified the method so it can create a 3d grid to partition the data based on that grid, using overlapping check. Method is below:

```
public int overlapPartition(Shape shape) {
  if (shape == null)
    return -1;
  Cube shapeMBR = shape.getMBR();
  if (shapeMBR == null)
    return -1;
  Point centerPoint = shapeMBR.getCenterPoint();
  int col = (int)Math.floor((centerPoint.x - x) / tileWidth);
  int row = (int)Math.floor((centerPoint.y - y) / tileHeight);
  int slice = (int)Math.floor((centerPoint.z - z) / tileAltitude);
```

```
  return getCellNumber(col, row, slice);
}
```

It will check with which partition each point overlaps. If our shapes are not points, the central point is calculated and given to the Sampler and later to the GridPartitioner class. In case we choose to replicate shapes another method support this:

```
public void overlapPartitions(Shape shape, ResultCollector<Integer> matcher) {
```

Replicate option will replicate shapes that overlap on two partitions, on both partition data files.

Also in order to correctly retrieve the correct partition below class is modified:

```
public CellInfo getPartition(int partitionID) {
  // Retrieve column and row of the given partition
  int col = partitionID % numColumns;
  int row = ((partitionID - col) / numColumns) % numRows;
  int slice = (((partitionID - col) / numColumns) - row) / numRows;
  return new CellInfo(partitionID, x + col * tileWidth, y + row * tileHeight, z + slice * tileAltitude, x + (col + 1) * tileWidth, y +
(row + 1) * tileHeight, z + (slice + 1) * tileAltitude);
}
```

The type of partitioner is set in Indexer class:

```
PartitionerClasses.put("rtree", GridPartitioner.class);
```

So this implies that if the rtree index is chosen then the partitioner will be a grid index. Note that this has nothing to do with the underlying local RTree index. The partitioner only splits the data to given slaves for further indexing. This can be changed if a user needs a different partitioner.

The RTree class is responsible for creating the local RTree index of every partition file. The following methods have been extensively modified to calculate the index based on 3D data and also use Z coordinate during the sorting algorithms. *bulkLoadWrite* method will now have a new direction called DIRECTION_Z to sort elements based also on this coordinate.

```
public static void bulkLoadWrite(final byte[] element_bytes,
  final int offset, final int len, final int degree, DataOutput dataOut,
  final Shape stockObject, final boolean fast_sort) {
```

```
public static Cube[] packInRectangles(GridInfo gridInfo, final Point[] sample) {
```

Additionally the header of the RTree for the node size has been changed. This is needed because we added a new dimension. We changed 4 to 6.

```
public static final int NodeSize = 4 + 8 * 6;
```

These changes compensate for the new increased size of the nodes inside the RTree due to the new dimension.

A similar approach was done for GridPartitioner, which is the index type that we used for the global index between all slave nodes. This class is responsible to partition the data into a grid. We have to modify it so the data are partitioned in a 3D grid instead. Below methods (but not exclusively) have been modified to both create the Grid index and calculate overlapping partitions using also the new coordinate.

```
public void createFromPoints(Cube mbr, Point[] points, int capacity) throws IllegalArgumentException {
```

```
public void overlapPartitions(Shape shape, ResultCollector<Integer> matcher) {
```

```
public int overlapPartition(Shape shape) {
```

We can also see the getCellNumber and getPartition function which are important for the calculations of the correct cell id for the 3D grid index as well as get the correct partition given a cellid. This is important during the creation of the index but also during the query process in order to retrieve the correct cells from the grid.

```
private int getCellNumber(int col, int row, int slice) {
   return row * numColumns + col + slice * numRows * numColumns;
  }
```

```
public CellInfo getPartition(int partitionID) {
   // Retrieve column and row of the given partition
   int col = partitionID % numColumns;
   int row = ((partitionID - col) / numColumns) % numRows;
   int slice = (((partitionID - col) / numColumns) - row) / numRows;
   return new CellInfo(partitionID, x + col * tileWidth, y + row * tileHeight, z + slice * tileAltitude, x + (col + 1) * tileWidth, y
+ (row + 1) * tileHeight, z + (slice + 1) * tileAltitude);
  }
```

MapReduce layer modifications:

MapReduce layer is responsible for running the map and reduce functions that control the way the jobs run. It splits the data and builds the indexes on slave nodes.

Both InputFormatters and RecordReaders will also need modifications. InputFormatters accept data and direct it to RecordReaders in order to read the data from files. They handle the partitioning of the data as well as the creation of the local index. Mathematical calculations inside those files need to adapt to the new Z coordinate. Index files will be modified to create a 3D index along with the provided data. The algorithm of the RTree will be used and will be extended to support a 3DRTree.

The OutputFormatters that will commit the job will have to be modified to create the files on disk using the correct format. These files are the partitioned data files that contain the data as well as the local index. For the purposes of this dissertation we will support only global index as grid and local indexes as a 3DRtree. The index structure for the global index is a grid index and for the local index is a 3DRTree.

Modifications have also been made to mapred module and classes:

BinarySpatialInputFormat, *CombinedSpatialInputFormat*, *GridOutputFormat3*, *GridRecordWriter3*, *RTreeRecordReader*, *RandomInputFormat*, *RandomInputGenerator*, *ShapeArrayInputFormat*, *ShapeArrayRecordReader*, *ShapeInputFormat*, *ShapeIterInputFormat*, *ShapeIterRecordReader*, *ShapeLineInputFormat*, *ShapeLineRecordReader*, *ShapeRecordReader*, *SpatialInputFormat*, *SpatialRecordReader*.

These classes will read the records from a data file and need support for 3D data.

From mapreduce module following classes have been modified:

RTreeRecordReader3, *SpatialInputFormat3*, *SpatialRecordReader3*.

Operations layer modifications:

From operations module which implements the supported queries *Main*, *FileMBR* and *RangeQuery* classes have been modified. Initially, as we explained, support exists only for Range type queries.

RangeQuery class that will run the query on top of the index will now use the new 3DRtree and 3D data to produce the query results.

From temporal module, *RepartitionTemporal* class is modified and from util, *FileUtil* class.

To support the large numbers of epoch time (Z variable) we modify function:

```
public static void serializeDouble(double d, Text t, char toAppend) {
```

in TextSerializerHelper file and parse the number as following:

```
byte[] bytes = (new BigDecimal(Double.toString(d))).toPlainString().getBytes();
```

Finally from main module *OperationsParams* and *RandomSpatialGenerator* classes were modified. We compile the module using maven with command:

mvn clean package

With these modifications, SpatialHadoop supports 3D data using 3DRTree index (specifically using STR algorithm to load the RTree), and range queries. After preparing the data as described above, the data are copied on HDFS using below commands:

bin/hadoop fs -mkdir /hadoop_user

bin/hadoop fs -copyFromLocal /home/hadoop_user/trucks.txt /hadoop_user

We now have the data on HDFS and we are ready to build the index. This is done with below command which will use MapReduce algorithm:

bin/shadoop index /hadoop_user/trucks.txt trucks3d.rtree sindex:rtree shape:point

sindex specifies the type of index and *shape* describes the type of our data. In our case, the data are 3D points, with x,y,epoch_time coordinates. Figure 24 shows part of the output of the command:

```
INFO mapred.FileInputFormat: Total input paths to process : 1
INFO mapred.SpatialRecordReader: Open a SpatialRecordReader to file: hdfs://master:8020/ha
Total time for sampling in millis: 55575
INFO indexing.Indexer: Finished reading a sample of 1146 records
INFO indexing.Indexer: Partitioning the space into 1 partitions with capacity of 32639
INFO mapreduce.SpatialRecordReader3: Open a SpatialRecordReader to split: /hadoop_user/tru
INFO indexing.RTree: Bulk loading a 3DRTree with 112203 elements
Total indexing time in millis 62687
```

Figure 24: Command line result of 3DRtree indexing

We can see the time it took for the index to be created. Here it took **62.687** milliseconds. Executing this command and depending on the size of our data, several files will be created. A **_master.rtree** file containing the master grid index. A **_rtree.wkt** file containing the same index in wkt format. And (in our case) a **part-00000.rtree** containing the data split and the local 3DRTree index along with the data. Larger data input files will result in more part- data output files. Next the range

query is run upon the RTRee index. We run the following command, **bin/shadoop rangequery trucks3d.rtree rq_results3d rect:470000,4200000,1031982663,480000,4210000,1032127416 shape:point** This command will select the points inside the rectangle **470000,4200000, 480000,4210000** but will also query the time variable based on the epoch time provided (**1031982663, 1032127416**). The query returns 183 results as shown in figure 25.

```
hadoop_user@master:~/hadoop-2.7.3$ bin/shadoop rangequery tr
17/02/11 23:34:48 INFO operations.RangeQuery: Selected 1 par
17/02/11 23:34:48 INFO mapreduce.SpatialInputFormat3: Spatia
17/02/11 23:34:48 INFO spatialHadoop.OperationsParams: Input
17/02/11 23:34:49 INFO operations.RangeQuery: Selected 1 par
17/02/11 23:34:49 INFO mapreduce.SpatialInputFormat3: Spatia
17/02/11 23:34:49 INFO mapreduce.RTreeRecordReader3: Open a
590
Time for 1 jobs is 4657 millis
Results counts: [183]
```

Figure 25: Command line result of 3DRtree querying

The query took **4.567** milliseconds to complete. In figure 26 we can the 2D representation of the 3D query result.

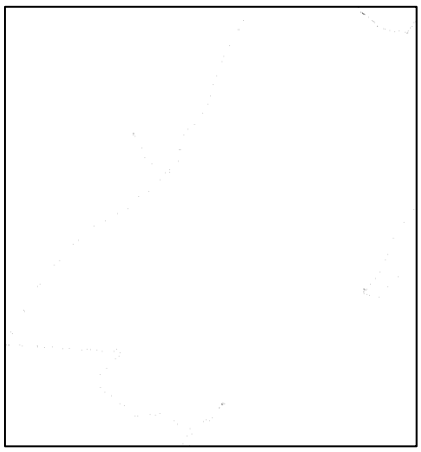

Figure 26: Plot result of 3DRtree querying

The above indexing and querying have been done using **1** slave node in a Hadoop cluster.

6. Experimental Results

In order to evaluate the new extension we used a cluster of vms running Hadoop on okeanos IAAS (https://okeanos.grnet.gr/home/). The cluster consists of 1 master node and 4 slave nodes. The specifications of the cluster are following:

For the master node:

- 8 Cores
- 8192 MB Memory
- 40GB System disk

For the slave nodes:

- 4 Cores
- 8192 MB Memory
- 10GB System disk

For all nodes:

- Hadoop-2.7.3
- SpatialHadoop 2.4.1

(All following commands should be run under {HADOOP_HOME} directory).

In order to test with various node sizes we used the following process to decommission nods from the cluster. We make sure that under etc/hadoop/hdfs-site.xml there is the following property:

```
<property>
  <name>dfs.hosts.exclude</name>
  <value>$HADOOP_CONF_DIR/exclude</value>
  <final>true</final>
</property>
```

And under etc/hadoop/yarn-site.xml the following property:

```
<property>
  <name>yarn.resourcemanager.nodes.exclude-path</name>
  <value>$HADOOP_CONF_DIR/exclude</value>
  <final>true</final>
</property>
```

We edit the file under etc/hadoop/exclude, and add the hostname of the node to be decommissioned. Then we run command **hdfs dfsadmin -refreshNodes** to refresh the nodes. We wait until decommissioning is over and then we remove the node from the etc/hadoop/include file. We run command **hdfs dfsadmin -refreshNodes** again. This will stop the datanode and the blocks on that node will be replicated on live nodes. Next we need to remove the node from yarn also. We run command **yarn rmadmin -refreshNodes**. Finally we remove the node from file etc/hadoop/slaves and we restart the cluster. After this process the node is decommissioned and is no longer used. We followed this process to do the tests with 1, 2, 3 and 4 nodes.

In our tests we run the index command and two queries.

The index command was:

{HADOOP_HOME}/bin/shadoop index /user/hduser/dataset_3d.txt test3d.rtree sindex:rtree shape:point

The query commands include one small query and one large query, shown below:

{HADOOP_HOME}/bin/shadoop rangequery test3d.rtree rq_results3d rect:500,500,1246562000,10000,10000,1246564000 shape:point

{HADOOP_HOME}/bin/shadoop rangequery test3d.rtree rq_results3d2 rect:500,500,1246562000,200000,200000,1246774000 shape:point

The small query will match with one partition from the data and the large query will match with three partitions. Along with the indexing time, the results from the queries can be seen on figure 27.

VMs	1	2	3	4
Indexing	771478	605899	567268	606442
Small Query With Index	5822	5815	5539	4707
Small Query Without Index	50203	39758	39624	45468
Large Query With Index	205613	194886	174935	154928
Large Query Without Index	55124	40344	40256	39882

Figure 27: 3d data test results

On figure 28 we can see the plot of the indexing times and on figure 29 we can see the plot of the query results.

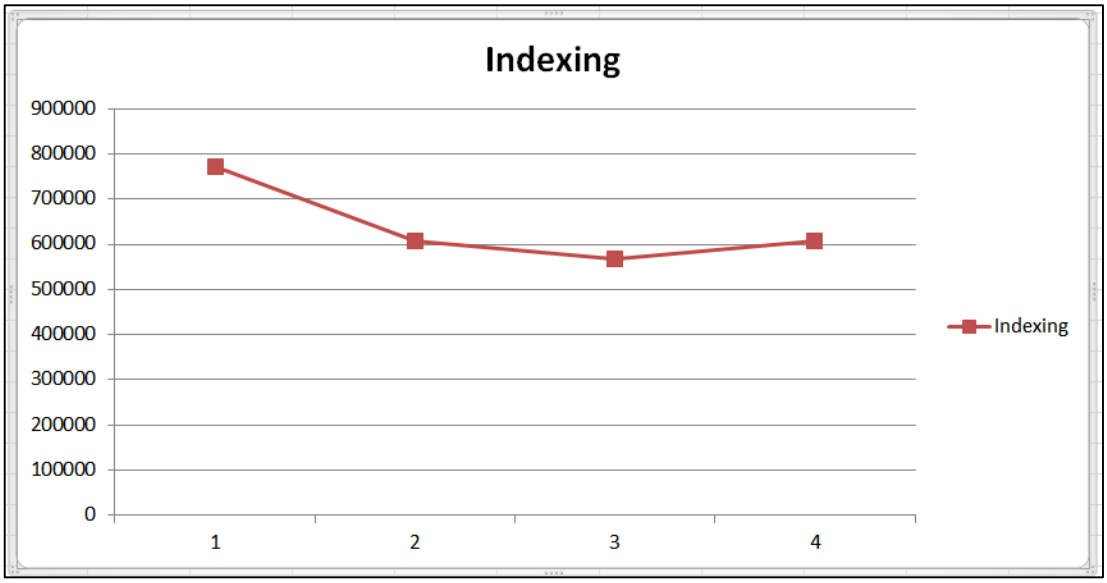

Figure 28: Indexing time for 3d data

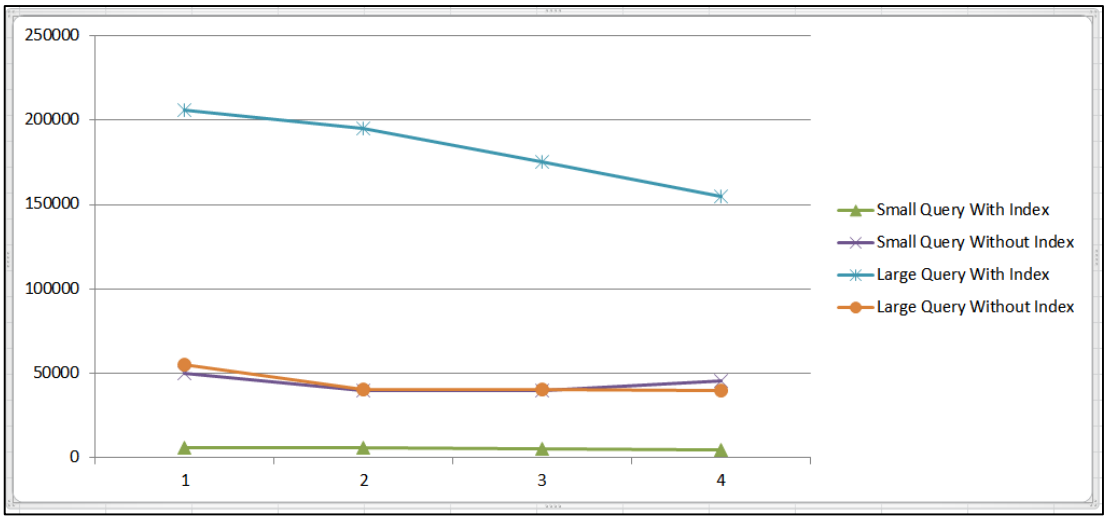

Figure 29: Query time for 3d data

On X axis is the number of nodes and on Y axis the time in milliseconds it took to run the command.

For the indexing time there is a decrease until we reach 4 nodes where it seems to stabilize. This can be attributed to the fact that HDFS stores index partitions in different nodes. But depending on the index this number of partitions is limited. It is expected that with a much larger data set the time it takes to index the data will be benefitted even more by an increased number of nodes.

We can also see that when not using the RTree index the time to do the queries does not change much when changing the node number. This time is fixed because

SpatialHadoop will have to always search the entire file to find the query results. The time to query the index keeps decreasing as the number of nodes increases. Searching the index seems to take more time than when not using the index. Although it is expected that with a much larger dataset and with an increased number of nodes the index query time will drop to levels lower than scanning the entire file. Scanning the entire file seems to not take so much benefit from using larger number of nodes. Also index query time can also be affected by the type of our data and the way they are organized by the global index and the local index. For example in our case we used a grid index as a global index and an RTree as a local index. This can be further improved to use also an RTree for global indexing. This will better distribute the partitions on different nodes.

As a control group and in order to exclude possible issues with our new code modifications we did a similar test with the unmodified SpatialHadoop module, using 2d data and queries.

The same results for indexing time and query time can be seen on figures 30 and 31 respectively.

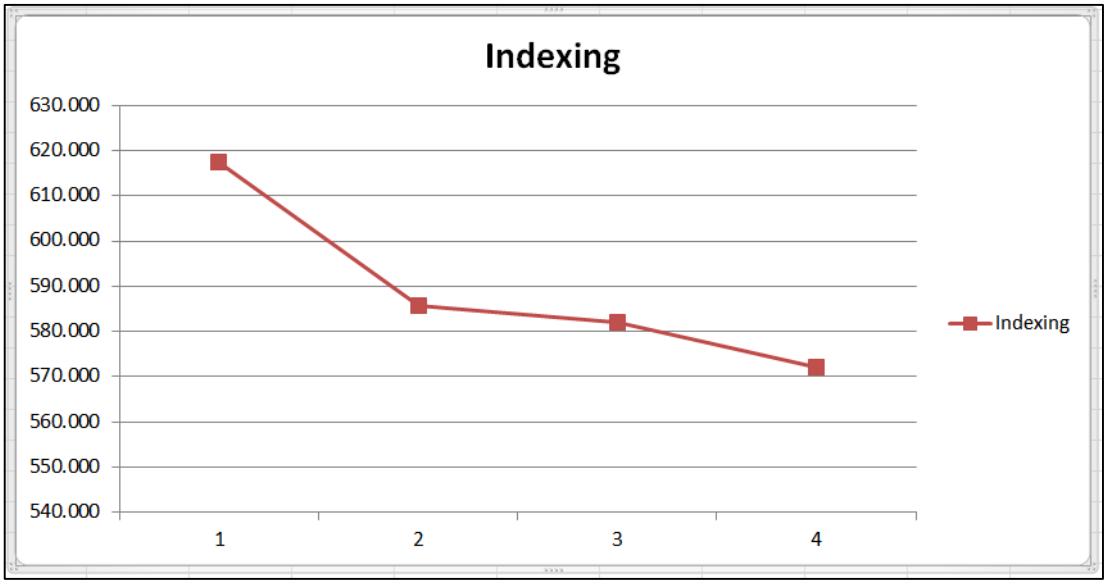

Figure 30: Indexing time of 2d data

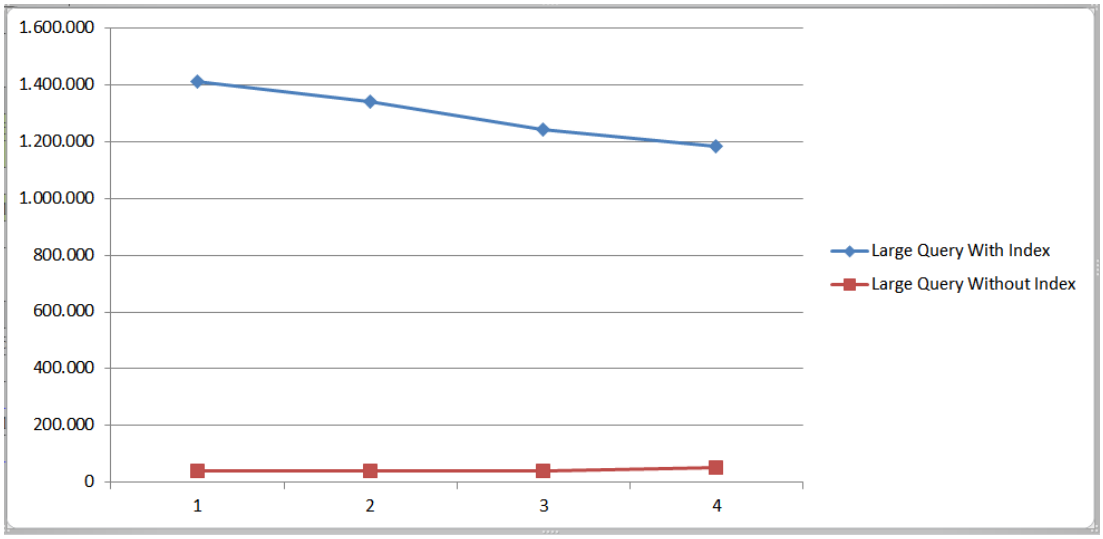

Figure 31: Query time of 2d data

Finally we verified the actual results from the queries using a python script to validate that the correct shapes are returned for each query.

7. Conclusions and Future Work

Results from our runs indicate that the new module works as expected and that the queries correctly use the new Z variable. Test runs to check the speed show a decrease in time when running queries especially for large queries. Scanning the entire file without the index seems to give faster results although with no benefit from node sizes. Increasing the node size positively affects the time it takes to query the index and also a benefit it takes to build the index. The way that the global index is created and the actual type of the data might be the cause of this issue. Also an improvement can be made to the Z variable. We used epoch time as a metric, but a different way of calculating the time might give better and faster results. Epoch time is a very large number which might cause performance issues when building the index or running the queries, especially for large volumes of data. The index can also be further improved to give better query times. The size and the type of the data can also affect this speed as we noticed that the same speed pattern appears even when using the unmodified SpatialHadoop module. This might indicate that the performance can be improved using a better index, contrary to, for example, grid index as a global index. Finally further test with a wide variety of data set can be done to verify the speed of the indexing as well as the time it takes to query non-index and indexed data. Performance modifications of the code can also be a potential topic for further time decrease.

7. References

1. Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop GIS: a high performance spatial data warehousing system over mapreduce. Proc. VLDB Endow. 6, 11 (August 2013), 1009-1020.
2. GIS Tools for Hadoop (http://esri.github.io/gis-tools-for-hadoop/)
3. GeoMesa Introduction (http://www.geomesa.org/documentation/user/introduction.html)
4. Simple Feature Access (https://en.wikipedia.org/wiki/Simple_Features)
5. Paradigm4. The Architecture and Motivation for Paradigm4's SciDB.
6. Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *Proceedings of the 2011 IEEE 12th International Conference on Mobile Data Management - Volume 01* (MDM '11), Vol. 1. IEEE Computer Society, Washington, DC, USA, 7-16.
7. Improving Sort-Tile-Recusive algorithm for R-tree packing in indexing time series. Conference: 2015 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future (RIVF)
8. Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (SIGMOD '84). ACM, New York, NY, USA, 47-57.
9. R-tree (https://en.wikipedia.org/wiki/R-tree)
10. SpatialHadoop: MapReduce Processing of Spatial Data in Hadoop (https://www.youtube.com/watch?v=A3Q2XlGvYcU)
11. Understanding Hadoop Clusters and the Network (http://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/)

Appendix 1

- Hadoop Installation and Configuration

The process begins with the installation of Hadoop. We will install the Hadoop as a multi-node cluster on two computers. One computer will function as the master, which will run all the daemons of Hadoop, and the second computer as the slave, which will run only TaskTracker and DataNode daemons. The operating system that we will use is Debian Linux.

After we download the Hadoop along with the source code, we install it on the master and all slave nodes. Then follow the steps below. At each step in brackets there are two labels, **m** which means that the step must be done only at the master node and **s** which means that the step must be done at all the slave nodes.

First and because we only have one public IP address to okeanos, we will adjust the master node in a way that the slave nodes have access to the internet. This is done so we can install the necessary packages. We create the file /etc/iptables.rules, add the following rules and execute the following command to load the rules. (**m**)

```
*nat
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
-A POSTROUTING -o eth2 -j MASQUERADE
COMMIT

*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -i lo -j ACCEPT
-A FORWARD -i eth1 -o eth2 -j ACCEPT
-A FORWARD -i eth2 -o eth1 -j ACCEPT
-A OUTPUT -o lo -j ACCEPT
COMMIT
```

iptables-restore < /etc/iptables.rules

Add the following in file /etc/network/interfaces so the rules are loaded in every restart. (**m**)

```
pre-up iptables-restore < /etc/iptables.rules
```

Enable IP forwarding in master node by adding in file /etc/sysctl.conf the variable *net.ipv4.ip_forward = 1* and running the command **sysctl -p /etc/sysctl.conf**. (**m**)

Install the following packages (**m**)

apt-get update

apt-get install openjdk-7-jdk

apt-get install vim

apt-get install libtool

apt-get install autoconf

apt-get install ivy

apt-get install maven

apt-get install sshpass

apt-get install unzip

apt-get install p7zip-full

Set the java version that we just installed as the default version.

update-alternatives --config java

Create a new Linux user specifically for Hadoop, for example user. The next steps will be performed as this new user, unless clearly specified. (**m**,**s**)

adduser hadoop_user

Add the following environmental variables in file /home/user/.bashrc: (**m**,**s**)

```
export CLASSPATH=.
export HADOOP_PREFIX=/home/user/hadoop-2.7.3
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
export PIG_HOME=/home/user/hadoop-2.7.3/contrib/pig-0.12.0
export PATH=$PATH:$PIG_HOME/bin:$JAVA_HOME/bin:$HADOOP_PREFIX/bin
```

Download and compile esri-geometry-api using Ant inside the lib folder of Hadoop. (**m**)

ant

Change the dependency inside file spatialhadoop/ivy.xml to *hadoop-2.7.3*. (**m**)

We download 2d version of SpatialHadoop and install it. (**m,s**)
In order to modify our packet to support 3d data we compile the new SpatialHadoop source code: (**m**)
mvn clean package

We copy spatialhadoop-2.4.1_SNAPSHOT.jar under directory spatialhadoop2-master/target/ to folder {HADOOP_HOME}/share/hadoop/common/lib/ spatialhadoop-2.4.1.jar (**m,s**)

We change execute permissions of shadoop file under {HADOOP_HOME}/bin directory with command: (**m,s**)
chmod +x shadoop

As a root user copy the dependency jar files of SpatialHadoop: (**m**)
cp /home/user/hadoop-2.7.3/share/hadoop/common/lib/esri-geometry-api-1.2.1.jar /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/ext/
cp /home/user/hadoop-2.7.3/share/hadoop/common/lib/jts-1.13.jar /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/ext/

Extract SpatialHadoop package (under/dist/package) that was just created inside Hadoop folder. (**m**)

Download and compile Pig inside main Hadoop folder: (**m**)
ant

Download and compile Pigeon inside main Hadoop folder, after adding the following in pom.xml file so it uses Java 1.7: (**m**)

```
<build>
  <plugins>
   <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
     <source>1.7</source>
     <target>1.7</target>
    </configuration>
```

```
    </plugin>
  </plugins>
</build>
```

mvn package. After the above we continue with the configuration of Hadoop.

If we want to enable remote debugging using an IDE we add the following in file etc/hadoop/hadoop-env.sh: (**m**)

```
export HADOOP_OPTS="$HADOOP_OPTS -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005"
```

If we want to add Hadoop to an Eclipse project we run the following command: (**m**)

ant eclipse

Increase the heap size of Hadoop by adding the following in file etc/hadoop/hadoop-env.sh: (**m**,**s**)

```
export HADOOP_HEAPSIZE=4096
```

Create an SSH key with the following commands: (**m**)

ssh-keygen -t rsa -P ""

cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys

Add names and IP addresses of the nodes inside file /etc/hosts. For example master and slave-X. (**m**)

Create a file named script.sh with the following contents (This step can be skipped if we are willing to make the configuration of nodes manually): (**m**)

```
#!/bin/bash
USERNAME=root
USERNAME2=user
#slave node names
HOSTS[0]="slave-1"
HOSTS[1]="slave-2"
HOSTS[2]="slave-3"
HOSTS[3]="slave-4"
HOSTS[4]="slave-5"
HOSTS[5]="slave-6"
HOSTS[6]="slave-7"
HOSTS[7]="slave-8"

#slave node passwords
```

```
PASS[0]="VJO5HrSiqP"
PASS[1]="ZOm0CQe08R"
PASS[2]="h5BDVLG9HR"
PASS[3]="uvvYaKMq9h"
PASS[4]="D3sGvSg4dQ"
PASS[5]="Fq7s1quZ4e"
PASS[6]="C30F1kCOtF"
PASS[7]="V5wZUuNuUu"
#/etc/hosts configuration
SCRIPT="echo '#Entries for hadoop nodes' >> /etc/hosts;
echo '192.168.0.3       master' >> /etc/hosts;
echo '192.168.0.2       slave-1' >> /etc/hosts;
echo '192.168.0.4       slave-2' >> /etc/hosts;
echo '192.168.0.5       slave-3' >> /etc/hosts;
echo '192.168.0.6       slave-4' >> /etc/hosts;
echo '192.168.0.7       slave-5' >> /etc/hosts;
echo '192.168.0.8       slave-6' >> /etc/hosts;
echo '192.168.0.9       slave-7' >> /etc/hosts;
echo '192.168.0.10      slave-8' >> /etc/hosts"
for ((i=0;i<${#HOSTS[@]};i++));
do
        echo 'Starting configuration of node' ${HOSTS[$i]}
        echo 'Copying hadoop'
        sshpass -p ${PASS[$i]} ssh -o StrictHostKeyChecking=no -l ${USERNAME2} ${HOSTS[$i]} 'rm -r /home/user/hadoop-2.7.3' >/dev/null
        sshpass -p ${PASS[$i]} scp -r /home/user/hadoop-2.7.3 ${USERNAME2}@${HOSTS[$i]}:/home/user >/dev/null
        echo 'Configuring SSH key'
        sshpass -p ${PASS[$i]} ssh-copy-id -i $HOME/.ssh/id_rsa.pub ${USERNAME2}@${HOSTS[$i]} >/dev/null
        echo 'Configuring /etc/hosts'
        sshpass -p ${PASS[$i]} ssh -o StrictHostKeyChecking=no -l ${USERNAME} ${HOSTS[$i]} ${SCRIPT} >/dev/null
        echo 'Configuring hostname'
        sshpass -p ${PASS[$i]} ssh -o StrictHostKeyChecking=no -l ${USERNAME} ${HOSTS[$i]} 'sed -i '1d' /etc/hostname' >/dev/null
        sshpass -p ${PASS[$i]} ssh -o StrictHostKeyChecking=no -l ${USERNAME} ${HOSTS[$i]} 'echo '${HOSTS[$i]}' >> /etc/hostname' >/dev/null
        sshpass -p ${PASS[$i]} ssh -o StrictHostKeyChecking=no -l ${USERNAME} ${HOSTS[$i]} 'hostname -F /etc/hostname' >/dev/null
        echo 'Configuring default gateway'
        sshpass -p ${PASS[$i]} ssh -o StrictHostKeyChecking=no -l ${USERNAME} ${HOSTS[$i]} 'route add default gw 192.168.0.3' >/dev/null
        echo 'Installing java 7'
        sshpass -p ${PASS[$i]} ssh -o StrictHostKeyChecking=no -l ${USERNAME} ${HOSTS[$i]} 'apt-get update && apt-get -y install openjdk-7-jdk' >/dev/null && sshpass -p ${PASS[$i]} ssh -o StrictHostKeyChecking=no -l ${USERNAME} ${HOSTS[$i]} 'echo 3 | update-alternatives --config java' >/dev/null && echo 'Node' ${HOSTS[$i]} 'configured'
        echo ' '
done
```

The above commands will copy the master node SSH key to all slave nodes, so master node can connect to all the slaves without password prompt. Additionally we

configure /etc/hosts and /etc/hostname files and install java 7 in all slave nodes after configuring the default gateway. We do the same for all slave nodes.

Make script executable: (**m**)

chmod u+x script.sh

./script.sh

SSH to localhost as well as the master node and all slave nodes so the SSH key fingerprint can be stored in the known_hosts file in master node. (**m**)

Add the following variable in etc/hadoop/hadoop-env.sh and etc/hadoop/yarn-env.sh files to configure Java. (**m**,**s**)

```
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
```

Add the following in file etc/hadoop/core-site.xml: (**m**,**s**)

```
<property>
<name>hadoop.tmp.dir</name>
   <value>/home/user/hadoop-2.7.3/tmp</value>
</property>
<property>
   <name>fs.defaultFS</name>
   <value>hdfs://master-node:8020</value>
</property>
```

Create a file name mapred-site.xml with the command **cp {HADOOP_HOME}/etc/hadoop/mapred-site.xml.template etc/hadoop/mapred-site.xml** and add the following: (**m**,**s**)

```
<property>
   <name>mapreduce.framework.name</name>
   <value>yarn</value>
</property>
<property>
   <name>mapreduce.map.java.opts</name>
   <value>-Xmx3072m</value>
</property>
<property>
   <name>mapreduce.reduce.java.opts</name>
   <value>-Xmx3072m</value>
</property>
<property>
<name>mapreduce.map.memory.mb</name>
<value>2048</value>
```

```
<description>No description</description>
</property>
<property>
<name>mapreduce.reduce.memory.mb</name>
<value>2048</value>
<description>No description</description>
</property>

Add the following in file etc/hadoop/hdfs-site.xml to configure replication: (m)
<property>
   <name>dfs.replication</name>
   <value>2</value>
</property>
<property>
   <name>dfs.namenode.name.dir</name>
   <value>/home/user/hadoop_store/hdfs/namenode</value>
</property>
<property>
   <name>dfs.datanode.data.dir</name>
   <value>/home/user/hadoop_store/hdfs/datanode</value>
</property>
```

Add the following in file etc/hadoop/hdfs-site.xml to configure replication: (**s**)

```
<property>
   <name>dfs.replication</name>
   <value>2</value>
</property>
<property>
   <name>dfs.datanode.data.dir</name>
   <value>/home/user/hadoop_store/hdfs/datanode</value>
</property>
```

Add the following in file etc/hadoop/yarn-site.xml: (**m**,**s**)

```
<property>
   <name>yarn.resourcemanager.resource-tracker.address</name>
   <value> master-node:8025</value>
</property>
<property>
   <name>yarn.resourcemanager.scheduler.address</name>
   <value> master-node:8030</value>
<property>
</property>
   <name>yarn.resourcemanager.address</name>
   <value> master-node:8050</value>
</property>
<property>
   <name>yarn.nodemanager.aux-services</name>
   <value>mapreduce_shuffle</value>
```

```
</property>
<property>
   <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
   <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
```

Add in file etc/hadoop/slaves all the hostnames of the slave nodes. For example slave-1. We add also add master node so it can also run DataNode and TaskTracker daemons. In file etc/hadoop/master we add the hostname of the master node. (**m**)

Add in file etc/hadoop/slaves the name of the slave node and in file etc/hadoop/master add the master node. (**s**)

Run the following commands to create the necessary HDFS folders:

mkdir -p /home/user/hadoop_store/hdfs/namenode

chmod 755 /home/user/hadoop_store/hdfs/namenode

mkdir -p /home/user/hadoop_store/hdfs/datanode

chmod 755 /home/user/hadoop_store/hdfs/datanode

Format HDFS with the command **{HADOOP_HOME}/bin/hdfs namenode -format.** (**m**)

Start cluster with command **{HADOOP_HOME}/sbin/start-all.sh** and stop it with command **{HADOOP_HOME}/sbin/stop-all.sh.** (**m**)

Start JobTracker with the command **{HADOOP_HOME}/sbin/mr-jobhistory-daemon.sh start historyserver**. To stop it, run the **command {HADOOP_HOME}/sbin/mr-jobhistory-daemon.sh stop historyserver.**

- Following steps describe the process to configure Eclipse IDE for remote debugging. Install java 7.

  Install Ant and in Eclipse go to window -> preferences -> java -> build path -> classpath variables and add a new variable with name ANT_HOME and as a path, the path to Ant folder.

Add (if not already) tools.jar file from java. Go to window -> preferences -> java -> installed jres. Select the correct java version and click edit -> add external jars. Navigate to jdk folder and in lib folder select tools.jar.

In remote VM and under Hadoop folder we run command **ant eclipse** to create the necessary files for the eclipse project.

Copy from the remote VM, the Hadoop folder, as well as the ivy2 jar dependencies which will be needed for compilation.

pscp -r -l user <remote VM IP>:<remote Hadoop path> <local path>

pscp -r -l user <remote VM IP>:<remote ivy2 path> <local path>

In Eclipse IDE go to file -> import -> general -> existing projects into workspace and select import to project.

Go to properties of the project and then java build path -> libraries, configure the path to the ivy2 jar dependencies. (The path where these files where downloaded in the previous step).

Copy SpatialHadoop source code to folder src/contrib under Hadoop project, and go to project properties, java build path -> source and add this folder in java build path variable of the project.

Finally to connect Eclipse IDE to the remote VM go to Run -> Debug Configurations -> Remote Java Application -> New. Under field Host and Port add the IP of the VM and the port which was previously configured in file conf/hadoop-env.sh.

- Generating experimental results dataset and running queries

The data that we used for the experimental results are generated with SpatialHadoop. We run command:

{HADOOP_HOME}/bin/shadoop generate test mbr:0,0,1000000,1000000 size:500.mb shape:point

It will generate a data file. This data file will be generated as a 2d file at first. Using a python script, shown below, we converted the data file by adding a third coordinate in the form of epoch time.

```
f = open("input_file", "r")
f2 = open("dataset_3d.txt ", "w")
start_epoch = 1246395600
i = 0
for line in f:
    if i == 10:
        i = 0
        start_epoch += 1
    else:
        i += 1
    line = line.rstrip()
    print >>f2, line + "," + str(start_epoch)
f.close()
f2.close()
```

After converting the data we store them into the HDFS using command:

{HADOOP_HOME}/bin/hadoop fs -copyFromLocal /usr/local/hadoop/dataset_3d.txt /user/hduser

The file will have data lines as following:

```
550112.8992165184,568785.044501004,1246427280
451068.2165806743,467595.5849949601,1246427280
537850.9298712214,621531.5314808576,1246427280
446283.4455272704,147200.20182653604,1246427280
```

To build the index we run command:

{HADOOP_HOME}/bin/shadoop index /user/hduser/dataset_3d.txt test3d.rtree sindex:rtree shape:point

and to run the queries we used following commands:

bin/shadoop rangequery test3d.rtree rq_results3d rect:500,500,1246562000,10000,10000,1246564000 shape:point

bin/shadoop rangequery test3d.rtree rq_results3d2 rect:500,500,1246562000,200000,200000,1246774000 shape:point

bin/shadoop rangequery dataset_3d.txt rq_results3d3 rect:500,500,1246562000,10000,10000,1246564000 shape:point

bin/shadoop rangequery dataset_3d.txt rq_results3d4 rect:500,500,1246562000,200000,200000,1246774000 shape:point