

# Skyline Query Processing in SpatialHadoop

*Dimitrios Pertesis*



Master of Science  
Department of Digital Systems  
University of Piraeus

2014

## Abstract

The MapReduce programming model allows us to process large data sets on a cluster of machines. A MapReduce job usually splits the input data set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks. The most popular open-source implementation is Apache Hadoop. Recently, an extension to Apache Hadoop has been developed called SpatialHadoop. SpatialHadoop is designed to handle large data sets of spatial data. SpatialHadoop contains spatial built-in data types but you can define your own data types. Moreover, it supports a variety of spatial operations and indexes.

In this project, we developed two efficient skyline computation algorithms and implemented on SpatialHadoop. Also, we compared them with an algorithm proposed in «CG\_Hadoop: Computational Geometry in MapReduce» paper. The object of this study is to implement algorithms that will be efficient in uniform, correlated and anti-correlated distributions of data. These algorithms should also be capable to work with indexed and non-indexed input files. In order to evaluate the efficiency of these three algorithms we ran a set of experiments in a cluster of 17 nodes.

## Περίληψη

Το MapReduce είναι ένα προγραμματιστικό μοντέλο που επιτρέπει την επεξεργασία μεγάλου όγκου δεδομένων σε ένα cluster από μηχανήματα. Ένα MapReduce job διαμοιράζει τα δεδομένα εισόδου σε ένα σύνολο από ανεξάρτητα κομμάτια τα οποία επεξεργάζονται από τις map διεργασίες παράλληλα. Το framework ταξινομεί τις εξόδους των map οι οποίες θα είναι στη συνέχεια είσοδοι στις reduce διεργασίες. Οι είσοδοι και έξοδοι ενός job αποθηκεύονται σε ένα σύστημα αρχείων. Το framework φροντίζει για τον προγραμματισμό και έλεγχο των διεργασιών καθώς και την επανεκτέλεση αποτυχημένων διεργασιών. Το πιο γνωστό ανοιχτού κώδικα λογισμικό είναι το Apache Hadoop. Πρόσφατα, έχει αναπτυχθεί μια επέκταση του Apache Hadoop με ονομασία SpatialHadoop. Το SpatialHadoop έχει σχεδιαστεί ειδικά να χειρίζεται μεγάλα σύνολα χωρικών δεδομένων. Το SpatialHadoop περιέχει έτοιμους χωρικούς τύπους δεδομένων αλλά μας επιτρέπει τη δημιουργία και δικών μας τύπων δεδομένων. Επιπλέον, υποστηρίζει ένα σύνολο από χωρικές λειτουργίες και δείκτες.

Σε αυτήν την εργασία, αναπτύξαμε δύο αποδοτικούς αλγόριθμους επεξεργασίας skyline ερωτημάτων και τους υλοποιήσαμε στο SpatialHadoop. Επίσης, τους συγκρίναμε με έναν αλγόριθμο που προτείνεται από το «CG\_Hadoop: Computational Geometry in MapReduce» paper. Το αντικείμενο αυτής της μελέτης είναι η υλοποίηση αλγορίθμων που θα είναι αποδοτικοί σε διαφορετικές κατανομές των δεδομένων όπως uniform correlated και anti-correlated. Οι αλγόριθμοι θα πρέπει να δουλεύουν σωστά ανεξάρτητα αν τα αρχεία που επεξεργαζόμαστε περιέχουν ή όχι δείκτες. Για να αξιολογήσουμε την απόδοση των τριών αλγορίθμων υλοποιήσαμε μια σειρά πειραμάτων σε ένα cluster με 17 μηχανήματα.

## Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Christos Doulkeridis, for his guidance, continuous support and for his invaluable help and advice throughout this project.

And last but not least, I would like to thank from the bottom of my heart my parents, brother and grandmother who always stood by me.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

To my parents, brother, grandmother and grandfather...

# Table of Contents

<b>Introduction</b> .....	1
1.1 Overview.....	1
1.2 Aims.....	3
1.3 Thesis outline .....	3
<b>Background</b> .....	5
2.1 Hadoop.....	5
2.1.1 MapReduce introduction.....	6
2.1.2 MapReduce Data Flow .....	6
2.1.3 Combiner Functions .....	11
2.1.4 Hadoop Streaming .....	12
2.1.5 Hadoop Pipes.....	13
2.2 SpatialHadoop .....	13
2.2.1 Extensible data types .....	14
2.2.2 Built-in data types.....	14
2.2.3 User-defined data types .....	15
2.2.4 Spatial Operations.....	16
2.2.5 Spatial index.....	20
2.3 Related work.....	22
<b>Problem Setting</b> .....	25
3.1 SpatialHadoop vs. Hadoop .....	25
3.2 Base algorithm.....	25
3.3 Efficiency matters .....	26
<b>Algorithms</b> .....	29
4.1 First Algorithm .....	30
4.2 Second Algorithm .....	32
<b>Implementation</b> .....	35
5.1 Main method.....	36
5.2 skylineMapReduce method .....	36
5.3 CellsFilter class .....	37
5.4 Map class .....	37

5.5 Reduce class .....	39
<b>Experimental Evaluation</b> .....	42
6.1 Experiments.....	42
6.2 Algorithms.....	43
6.3 Results.....	43
6.3.1 Uniform distribution.....	44
6.3.2 Correlated distribution .....	47
6.3.3 Anti-Correlated distribution.....	50
<b>Conclusion</b> .....	56
<b>Bibliography</b> .....	58
<b>A Pseudocode</b> .....	59
A.1 Algorithm 1 .....	59
A.2 Algorithm 2.....	60
<b>B Hadoop/SpatialHadoop installation</b> .....	62

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

## List of Figures

1-1: Skyline points example .....	2
2-1: Hadoop subprojects [1].....	6
2-2: MapReduce data flow with a single reduce task [1].....	9
2-3: MapReduce data flow with multiple reduce tasks [1] .....	10
2-4: MapReduce data flow with no reduce tasks [1].....	10
2-5: R-tree index stored as a folder [6].....	20
3-1: CG_Hadoop's filters and optimization techniques .....	27
3-2: First algorithm's filters and optimization techniques.....	27
3-3: Second algorithm's filters and optimization techniques .....	28
4-1: CellsFilter example, Cell3 is dominated by Cell0 .....	30
4-2: Area dominated by map filters in algorithm 1.....	31
4-3: Area dominated by map filters in algorithm 2.....	33
5-1: High level code organization.....	35
5-2: Map flowchart of the first algorithm.....	38
5-3: Map flowchart of the second algorithm .....	39
5-4: Reduce flowchart of the first algorithm.....	40
5-5: Reduce flowchart of the second algorithm.....	41
6-1: Uniform distributed data, 1G and 10G, Non-indexed file .....	45
6-2: Uniform distributed data, 1G and 10G, Rtree-indexed file .....	47
6-3: Correlated distributed data, 1G and 10G, Non-indexed file .....	48
6-4: Correlated distributed data, 1G and 10G, Rtree-indexed file .....	50
6-5: Anti-Correlated distributed data, 1G and 10G, Non-indexed file.....	52
6-6: Anti-Correlated distributed data, 1G and 10G, Rtree-indexed file.....	53
6-7: Anti-Correlated (2) distributed data, 1G and 10G, Rtree-indexed file .....	55



## List of Tables

5-1: min_x and min_y example .....	41
6-1: Parameters .....	43
6-2: Uniform distributed data, 1G, non-indexed file .....	44
6-3: Uniform distributed data, 10G, non-indexed file .....	45
6-4: Uniform distributed data, 1G, Rtree-indexed file .....	46
6-5: Uniform distributed data, 10G, Rtree-indexed file .....	46
6-6: Correlated distributed data, 1G, Non-indexed file .....	48
6-7: Correlated distributed data, 10G, Non-indexed file .....	48
6-8: Correlated distributed data, 1G, Rtree-indexed file .....	49
6-9: Correlated distributed data, 10G, Rtree-indexed file .....	50
6-10: Anti-Correlated distributed data, 1G, Non-indexed file .....	51
6-11: Anti-Correlated distributed data, 10G, Non-indexed file .....	51
6-12: Anti-Correlated distributed data, 1G, Rtree-indexed file .....	52
6-13: Anti-Correlated distributed data, 10G, Rtree-indexed file .....	53
6-14: Anti-Correlated (2) distributed data, 1G, Rtree-indexed file .....	54
6-15: Anti-Correlated (2) distributed data, 10G, Rtree-indexed file .....	55

# Chapter 1

## Introduction

### 1.1 Overview

Over the last few years, big data has become a big deal. Recently, spatial data has gained interest. Spatial data has been used in many applications. Increasingly, the size, variety, and update rate of spatial datasets exceed the capacity of commonly used spatial computing and spatial database technologies to process the data with reasonable effort. Such data is known as Spatial Big Data. A solution to this problem is cloud computing. Cloud computing has become a viable, mainstream solution for data processing, storage and distribution. The cloud computing model is a perfect match for big data since cloud computing provides unlimited resources on demand.

The skyline of a  $d$ -dimensional dataset contains the points that are not dominated by any other point on all dimensions. The skyline operator is important for several applications involving multi-criteria decision making.

**Definition:** Given a set of points  $p_1, p_2, \dots, p_N$ , the skyline operator returns all points  $p_i$  such that  $p_i$  is not dominated by another point  $p_j$ . Skyline query is also known as Pareto optimal meaning that it returns all points  $p_i$  such that there is no other point  $p_j$  better on all dimensions. Using the common example in the literature, assume that we have a set of hotels and for each hotel we store its distance from the beach ( $x$  axis) and its price ( $y$  axis). The most interesting hotels are these for which there is no point that is better in both dimensions (An example is illustrated in figure 1-1).

When it comes to processing vast amounts of data it becomes difficult to compute the skyline points. In these situations, MapReduce is a fine solution. MapReduce is a programming model for processing large data sets with a

parallel, distributed algorithm on a cluster. A MapReduce program is composed of a Map() procedure that performs filtering and sorting and a Reduce() procedure that performs a summary. The "MapReduce System" orchestrates by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance. A popular open-source implementation is Apache Hadoop. Apache Hadoop is an open-source software framework for storage and large scale processing of data-sets on clusters of commodity hardware.

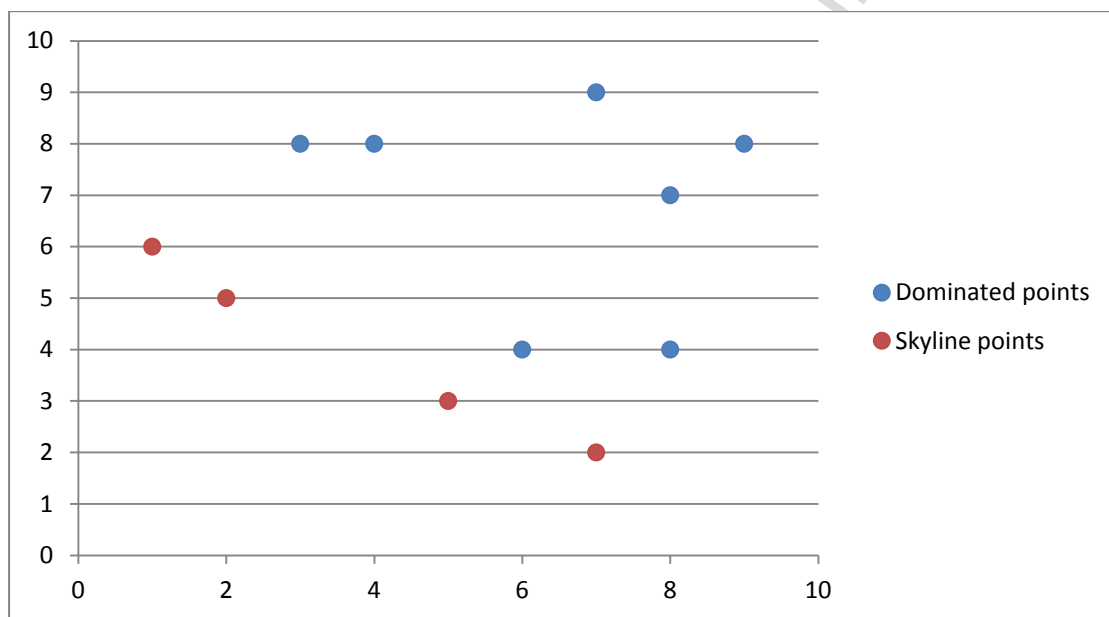


Figure 1-1: Skyline points example

Unfortunately, Hadoop is not very efficient concerning spatial data processing. A more efficient MapReduce extension to Apache Hadoop designed specially to work with spatial data is SpatialHadoop. SpatialHadoop is an open source MapReduce extension designed specifically to handle huge datasets of spatial data on Apache Hadoop. SpatialHadoop is shipped with built-in spatial high level language, spatial data types, spatial indexes and efficient spatial operations.

## 1.2 Aims

The goal of this research is to implement skyline algorithms on SpatialHadoop that will be efficient to deal with a variety of input data. The programs must take as input a file of two-dimensional points and it must output the skyline points.

The most important thing is to develop an algorithm that can compute the skyline points on different sizes of files e.g. 1G or 10G. Also, it should be able to work with various data distributions such as uniform, correlated and anti-correlated.

The challenge is that the algorithm must be efficient on all these different kind of data. It is not easy to do something like this because some algorithms perform better in some situations and less efficiently in other cases. To be more specific, we may have developed an algorithm that is very fast when it comes to uniform distributed data but not so efficient in anti-correlated distributed data or the exact opposite. So the goal of this thesis is to find out algorithms that are efficient no matter the size or distribution of a data set.

## 1.3 Thesis outline

The thesis has been divided into 7 chapters starting from this one. The remaining chapters are organized as follows:

- Chapter 2 describes how MapReduce works, explains the features of Hadoop, SpatialHadoop and presents papers on skyline query processing using MapReduce platforms.
- Chapter 3 focuses on the subject of efficient skyline query processing in SpatialHadoop.
- Chapter 4 explains the algorithms that compute the skyline points over an input file.

- Chapter 5 presents the implementation of the algorithms from chapter 4.
- Chapter 6 gives us an experimental evaluation of the implemented algorithms.
- Chapter 7 summarizes every aspect presented in this thesis.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

## Chapter 2

### Background

#### 2.1 Hadoop

Hadoop is a collection of related subprojects that fall under the umbrella of infrastructure for distributed computing. These projects are hosted by the Apache Software Foundation, which provides support for a community of open source software projects. Although Hadoop is best known for MapReduce and its distributed file system (HDFS, renamed from NDFS), the other subprojects provide complementary services, or build on the core to add higher-level abstractions. The subprojects, and where they sit in the technology stack, are shown in Figure 2-1 and described briefly here:

- Core
- Avro
- MapReduce
- HDFS
- Pig
- HBase
- ZooKeeper
- Hive
- Chukwa

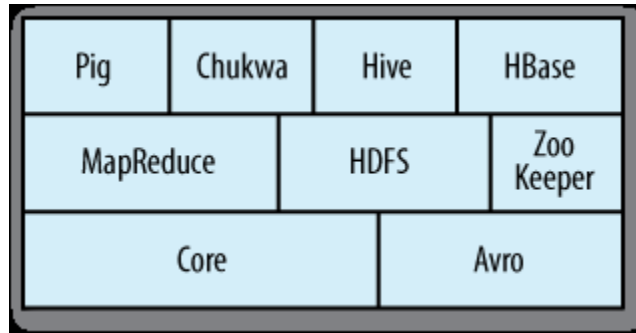


Figure 2-1: Hadoop subprojects [1]

### 2.1.1 MapReduce introduction

MapReduce is a programming model for data processing. The model is simple, yet not too simple to express useful programs in. Hadoop can run MapReduce programs written in various languages. Most importantly, MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal.

MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the map function and the reduce function.

### 2.1.2 MapReduce Data Flow

A MapReduce job is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks.

There are two types of nodes that control the job execution process: a JobTracker and a number of TaskTrackers. The JobTracker coordinates all the jobs run on the system by scheduling tasks to run on TaskTrackers. TaskTrackers run tasks and send progress reports to the JobTracker, which keeps a record of the overall progress of each job. If a task fails, the JobTracker can reschedule it on a different TaskTracker.

Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Hadoop creates one map task for each split, which runs the user defined map function for each record in the split.

Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if we are processing the splits in parallel, the processing is better load-balanced if the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained.

On the other hand, if splits are too small, then the overhead of managing the splits and of map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of a HDFS block, 64 MB by default, although this can be changed for the cluster (for all newly created files), or specified when each file is created.

Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the data locality optimization. It should now be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some of the split would have to be transferred across the network to the node running the map task, which is clearly less efficient than running the whole map task using local data.



Map tasks write their output to local disk, not to HDFS. Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete the map output can be thrown away. So storing it in HDFS, with replication, would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to recreate the map output.

Reduce tasks don't have the advantage of data locality—the input to a single reduce task is normally the output from all Mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the Reducer is normally stored in HDFS for reliability. For each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes.

The whole data flow with a single reduce task is illustrated in Figure 2-2. The dotted boxes indicate nodes, the light arrows show data transfers on a node, and the heavy arrows show data transfers between nodes.

The number of reduce tasks is not governed by the size of the input, but is specified independently.

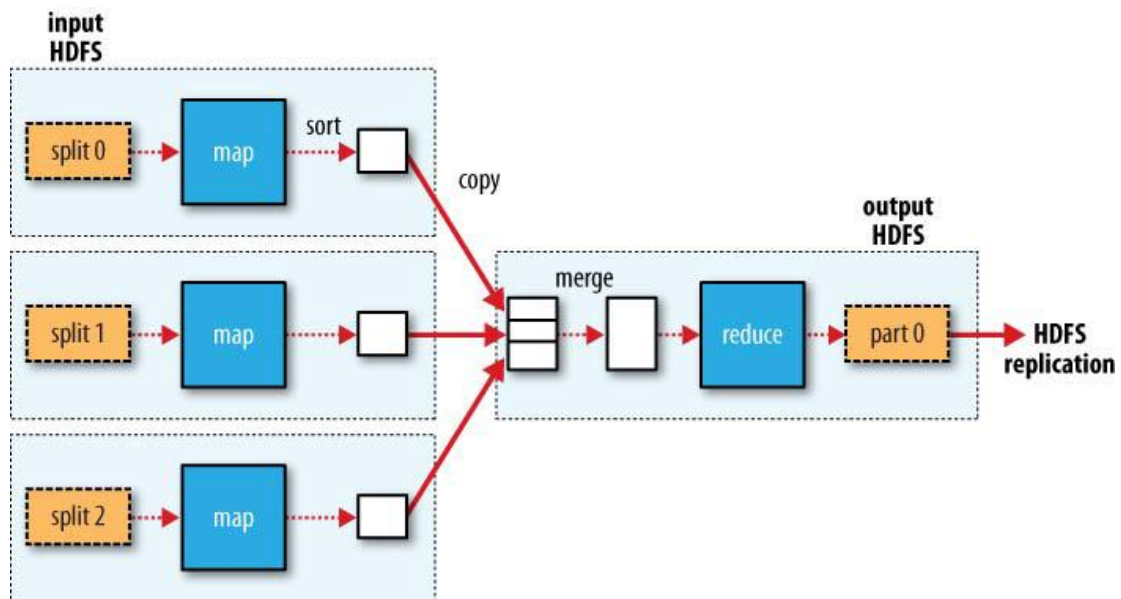


Figure 2-2: MapReduce data flow with a single reduce task [1]

When there are multiple Reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for every key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner - which buckets keys using a hash function - works very well.

The data flow for the general case of multiple reduce tasks is illustrated in Figure 2-3. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as “the shuffle,” as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time.

Finally, it’s also possible to have zero reduce tasks. This can be appropriate when you don’t need the shuffle since the processing can be carried out entirely in. In this case, the only off-node data transfer is when the map tasks write to HDFS (see Figure 2-4).

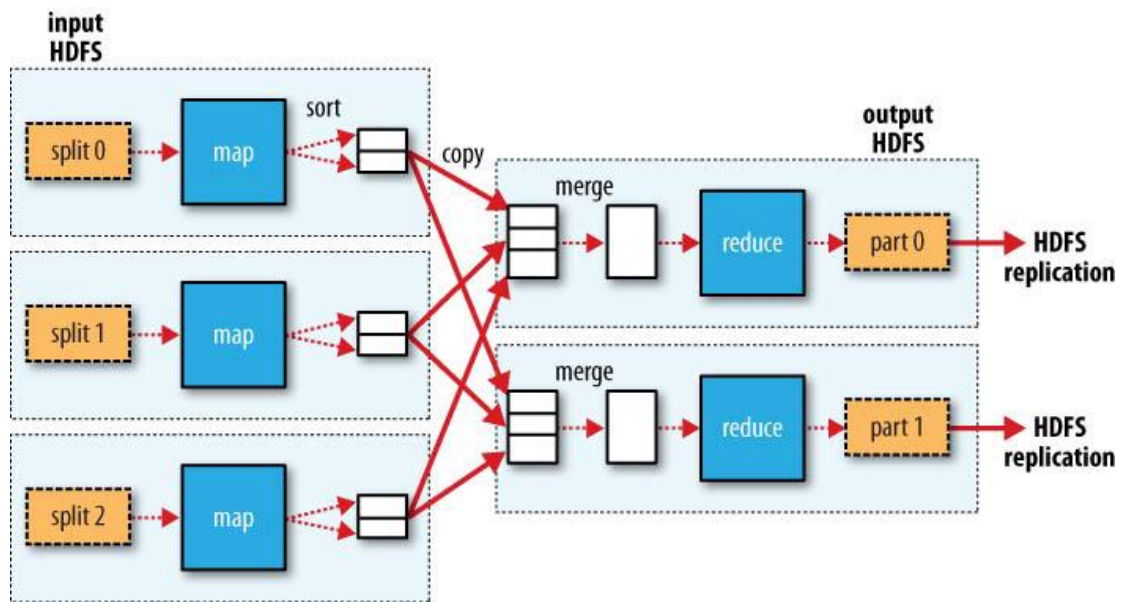


Figure 2-3: MapReduce data flow with multiple reduce tasks [1]

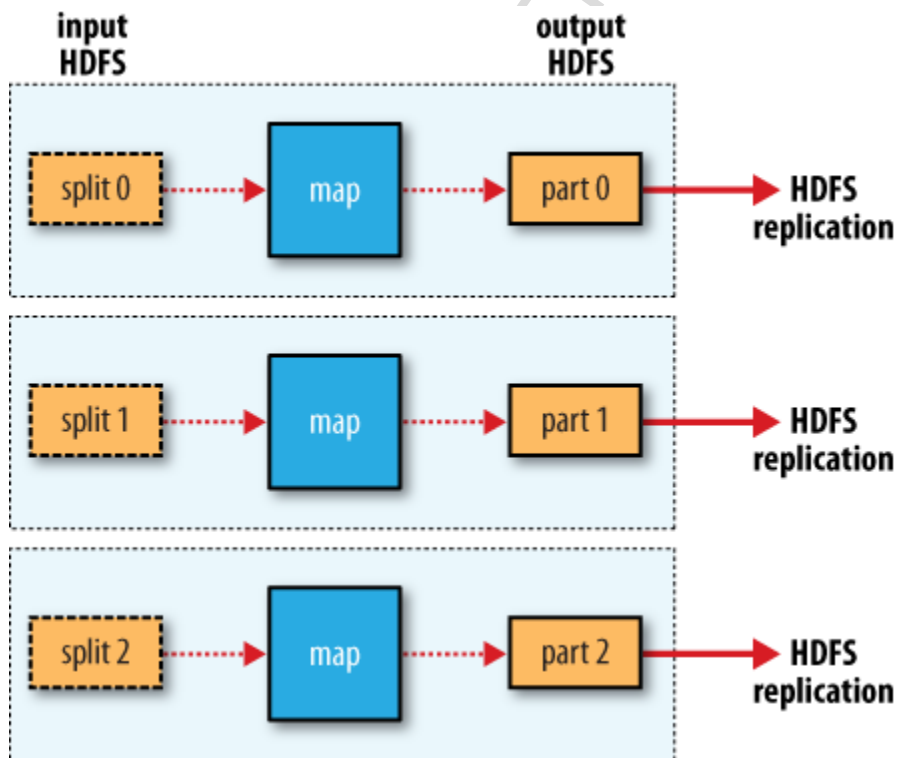


Figure 2-4: MapReduce data flow with no reduce tasks [1]

### 2.1.3 Combiner Functions

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a combiner function to be run on the map output—the combiner function’s output forms the input to the reduce function. Since the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the Reducer.

The contract for the combiner function constrains the type of function that may be used. This is best illustrated with an example. Imagine the first map produced the output:

(1950, 0)  
(1950, 20)  
(1950, 10)

And the second produced:

(1950, 25)  
(1950, 15)

The reduce function would be called with a list of all the values:

(1950, [0, 20, 10, 25, 15])

with output:

(1950, 25)

since 25 is the maximum value in the list. We could use a combiner. The reduce would then be called with:

*(1950, [20, 25])*

and the reduce would produce the same output as before. More succinctly, we may express the function calls as follows:

*max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25*

Not all functions possess this property. For example, if we were calculating mean values, then we couldn't use the mean as our combiner function, since:

*mean(0, 20, 10, 25, 15) = 14*

but:

*mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15*

The combiner function doesn't replace the reduce function. The reduce function is still needed to process records with the same key from different maps. But it can help cut down the amount of data shuffled between the maps and the reduces, and for this reason alone it is always worth considering whether you can use a combiner function in your MapReduce job.

#### **2.1.4 Hadoop Streaming**

Hadoop provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java. Hadoop Streaming uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program.

Streaming is naturally suited for text processing, and when used in text mode, it has a line-oriented view of data. Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output. A map output key-value pair is written as a single tab-delimited line. Input to the reduce function is in the same format—a tab-separated key-value pair—passed over standard input. The reduce function reads lines from standard input, which the framework guarantees are sorted by key, and writes its results to standard output.

### **2.1.5 Hadoop Pipes**

Hadoop Pipes is the name of the C++ interface to Hadoop MapReduce. Unlike Streaming, which uses standard input and output to communicate with the map and reduce code, Pipes uses sockets as the channel over which the TaskTracker communicates with the process running the C++ map or reduce function. JNI is not used.

Unlike the Java interface, keys and values in the C++ interface are byte buffers, represented as Standard Template Library (STL) strings. This makes the interface simpler, although it does put a slightly greater burden on the application developer, who has to convert to and from richer domain-level types.

## **2.2 SpatialHadoop**

SpatialHadoop is an open source MapReduce extension designed specifically to handle huge datasets of spatial data on Apache Hadoop. SpatialHadoop is shipped with built-in spatial high level language, spatial data types, spatial indexes and efficient spatial operations. You can use it to analyze your huge spatial datasets on a cluster of machines.

### 2.2.1 Extensible data types

SpatialHadoop ships with several data types including (Point, Rectangle and Polygon). There are different cases where you'll need to extend these data types or implement new spatial data types.

- Input files are not in the standard format used by SpatialHadoop
- Each record contains more information than just the shape (e.g., tags or comments)
- The application uses shapes other than the supported shapes (e.g., rounded rectangle)

### 2.2.2 Built-in data types

SpatialHadoop contains three main spatial data types, namely, Point, Rectangle and Polygon. Each data types stores just the spatial information about the shape without any extra information. All shapes are two-dimensional in the Euclidean space. A point is represented by its two dimensions (X and Y). A rectangle is represented by a corner point (X, Y) and the dimensions (Width x Height). A polygon is represented as a list of two-dimensional points.

The main storage format for spatial data types in SpatialHadoop is the text format. This makes it easier to import/export legacy formats in other applications. The standard format is a CSV format where each record is stored in one line. This format can be changed for custom data types provided that each record is stored in exactly one line. For point, a line contains two fields (X,Y) separated by a comma. For a rectangle, the tuple (X, Y, Width, Height) is stored with a comma as a separator. For polygon, each line contains number of points followed by the coordinates of each point. For example, a triangle with the corner points (0,0), (1,1) and (1,0) can be

represented as "3,0,0,1,1,1,0". All coordinates used in the standard data types are long integers (64-bit).

### 2.2.3 User-defined data types

To define your own data type, you need to define it as a new class that implements the Shape interface. For convenience, you could choose to extend one of the standard data types and built on top of it instead of building a class from scratch. For example, let's say that your files contain records represented as rectangles. Unlike the standard rectangles, each rectangle has an additional ID that precedes the coordinates of the rectangle. You can extend the rectangle class and add an additional ID field.

RectangleID.java:

```
public class RectangleID extends Rectangle {  
    private int id;
```

To define your own data type, you need to define it as a new class that implements the Shape interface. For convenience, you could choose to extend one of the standard data types and built on top of it instead of building a class from scratch. For example, let's say that your files contain records represented as rectangles. Unlike the standard rectangles, each rectangle has an additional ID that precedes the coordinates of the rectangle. You can extend the rectangle class and add an additional ID field.

The new field must be also written when an object of this class is serialized over network. This is required by SpatialHadoop (and Hadoop) when objects are transferred from mappers to Reducers. This can be done as follows:

```
public void write(DataOutput out) throws IOException {  
    out.writeInt(id);  
    super.write(out);  
}
```



```

public void readFields(DataInput in) throws IOException {
    id = in.readInt();
    super.readFields(in);
}

```

You need also to specify the format of the input file that contains objects of this type. This done by implementing two methods from `Text` and `toText`. The first method takes as input a text that represents a line read from the input file, and parses it to fill the target object. The second method does the exact opposite of this. It takes a `Text` object and serializes the information stored in this object to this text. It should not add a terminating new line as this is added by the framework itself. The implementation of these two method will look like this.

```

public void fromText(Text text) {
    id = TextSerializerHelper.consumeInt(text, ',');
    super.fromText(text);
}

```

```

public Text toText(Text text) {
    TextSerializerHelper.serializeInt(id, text, ',');
    return super.toText(text);
}

```

Once you're done with this class, you can use it with the existing operations (range query, kNN and spatial join). All you need to do is to provide its name when you call the operations using the `shape:` option. For example, you can perform a range query using the following command:

```

$ bin/hadoop hadoop-operations-*.jar input-filename rect:500,500,1000,1000
shape:RectangleID

```

## 2.2.4 Spatial Operations

Operations in `SpatialHadoop` are implemented as regular MapReduce programs. The main difference between spatial operations and regular operations is that the input file is spatially indexed. To read a spatially indexed

file, you need to provide the correct InputFormat and RecordReader. In addition, to the regular map and reduce functions, SpatialHadoop allows you to provide a filter function that performs an early pruning step that prunes away file blocks that do not contribute to answer based on their minimal bounding rectangles (MBRs). This can be useful to decrease the number of map tasks for a job. We will take you with step by step instructions to write a spatial operation.

We will use the range query as an example to describe how spatial operations are implemented. In range query, we have an input file that contains a set of shapes and a rectangular query area (A). The output is all shapes that overlap with the query area (A).

Before writing the MapReduce program for this operation, we need to think about it and decide how it should work. A naive implementation would scan over all shapes in the input file and select the shapes that overlap the query area. In SpatialHadoop, since the input file is indexed, we can utilize this index to avoid scanning the whole file. The input file is partitioned and each partition is stored in a separate block. If the boundaries of a partition is disjoint with the query area, it indicates that all shapes inside this query area are also disjoint. Hence, an initial filter step is to remove all blocks that are disjoint with the query area. This leaves only the blocks that overlap with the query area. For each overlapping block, we need to find shapes that overlap the query area. This simple algorithm is almost correct. There is one glitch that needs to be handled. As some shapes in the input file might overlap two partitions, they are replicated to each of these partitions. If the query area overlaps these two partitions and overlaps this shape, the shape will be reported twice in the answer. To avoid this situation, we implement a duplicate avoidance technique which ensures that each shape is reported once. This is done by calculating the intersection of the query area and the block boundaries (cell intersection) and the intersection of the query area and the shape (shape intersection). If the top left point of the shape intersection falls inside the cell intersection, the answer is reported, otherwise the answer is skipped.

The spatial filter function takes as input all blocks in an input file, and outputs the subset of blocks that needs to be processed. For range query, it selects the blocks that overlap the query area. The code will look like the following.

RangeFilter.java:

```
public class RangeFilter extends DefaultBlockFilter {
    public void selectBlocks(SimpleSpatialIndex gIndex,
        ResultCollector output) {
        gIndex.rangeQuery(queryRange, output);
    }
}
```

This code simply selects and returns all blocks that overlap the query range. The MBR of each block was calculated earlier when the file was indexed. Note that to access the query area in the filter function, it needs to be set in the job configuration file and read in RangeFilter#configure method.

The map function takes as input the contents of one block, and it selects and output all shapes overlapping the query area. We will show here how the map function looks like if the blocks are indexed as R-tree.

RangeQuery.java:

```
public void map(final CellInfo cellInfo, RTree shapes, final OutputCollector
output, Reporter reporter) {
    shapes.search(queryShape.getMBR(), new ResultCollector() {
        public void collect(T shape) {
            try {
                boolean report_result = false;
                if (cellInfo.cellId == -1) {
                    report_result = true;
                } else {
                    Rectangle intersection =
queryShape.getMBR().getIntersection(shape.getMBR());
                    report_result = cellInfo.contains(intersection.x, intersection.y);
                }
                if (report_result)
                    output.collect(dummy, shape);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    });
}
```

```

    });
}

```

The above code simply issues a range query against the R-tree built in this block to find all shapes overlapping the query area. For matching shapes, the duplicate avoidance test is carried out to decide whether to report this shape in answer or not. If the cell ID is -1, this indicates that there is no MBR associated with this block. This means that records in input file are not partitioned, hence, no replication and the answer should be reported. Otherwise, the test described earlier is done. Depending on the result of this test, the answer is finally reported.

Since the output of the map function is the final answer, no reduce step is needed. The reduce function is not provided for this operation.

The final step is how to configure the job. We will focus on the parts that are specific to the range query and/or SpatialHadoop.

RangeQuery.java:

```

job.setNumReduceTasks(0);
job.setClass(SpatialSite.FilterClass, RangeFilter.class, BlockFilter.class);
RangeFilter.setQueryRange(job, queryShape);
job.setMapperClass(Map.class);
job.setInputFormat(RTreeInputFormat.class);
job.set(QUERY_SHAPE_CLASS, queryShape.getClass().getName());
job.set(QUERY_SHAPE, queryShape.toText(new Text()).toString());
job.set(SpatialSite.SHAPE_CLASS, shape.getClass().getName());

```

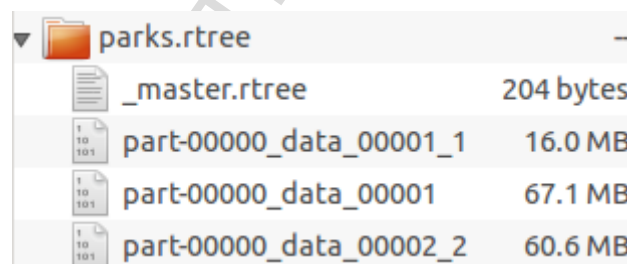
Setting number of reduce jobs to zero ensures that the output of the map function goes directly to output. The filter function is set using job.setClass method. The query range used by the filter function is set using the RangeFilter.setQueryRange method. Then, the map function is set as normal. The input format is set to RTreeInputFormat since blocks are R-tree indexed. After that, the query shape is set in job configuration to make it accessible to the map function. Finally, the SHAPE\_CLASS is set to indicate the type of shapes stored in input file.

Once the job is configured correctly, it is submitted to SpatialHadoop for processing as a normal MapReduce job. The output is stored to HDFS in the configured output file. You can check the job counters to see how many splits were created and see that only the subset of blocks overlapping the query range were processed.

## 2.2.5 Spatial index

We will show how to access the spatial indexes built in SpatialHadoop. We assume that there is an index already constructed using the index command and stored on disk. We will show how to retrieve basic information about the index and how to perform simple queries on it.

The first step to interact with the index is to understand how it is organized on disk. Let's say we issue the index command to create an R-tree index in the path 'parks.rtree'. The target will be stored as a folder that looks like the one in figure 2-5.



File Name	Size
_master.rtree	204 bytes
part-00000_data_00001_1	16.0 MB
part-00000_data_00001	67.1 MB
part-00000_data_00002_2	60.6 MB

Figure 2-5: R-tree index stored as a folder [6]

The index is stored as a set of data files where each data file contains the records in one partition. The index also contains one `_master.xxx` file which contains metadata about the global index (i.e., file partitions). Simply, it contains one line per partition which contains the boundaries of the partition and the partition file name. The extension of the *master* file indicates the type of index constructed. The supported indexes are grid, rtree and r+tree. Below is an example of a small master file with three partitions.

-179.3248215,-54.934357,6.9290401,71.2885321,part-00000\_data\_00001  
-171.7735299,-54.8114255,6.9261512,65.1485099,part-  
00000\_data\_00001\_16.9225032,-46.44586,179.3801209,78.0657531,part-  
00000\_data\_00002\_2

All other files are data files which contain data records. For a grid index, each partition file is a simple text file which contains one record per line. For R-tree and R+-tree, partition files are a little bit more complex. Records in each partition are organized in an R-tree index. Each file contains two sections. The first section stores the R-tree structure in a binary format as a list of nodes stored in level-order traversal. The second section stores data records in a text format as one record per line. The header of the R-tree contains information about the size of the tree which allows SpatialHadoop to skip over the R-tree structure and reads the records directly if the R-tree is not useful for processing.

All the information about the global index is stored in the master file. However, you don't need to parse the file yourself. You can make an API call which retrieves the global index and returns it as one object. The method `SpatialSite#getGlobalIndex(FileSystem, Path)` takes a file system and a path to a directory in that file system and returns the associated master file. If the path indicates a non-indexed data, null is returned. The returned value is of type `GlobalIndex<Partition>`. You can iterate over all partitions using the iterator method. You can also retrieve specific partitions using the `rangeQuery` and `knn` methods.

Once you retrieve a partition or a set of partitions, the next step is to read the records stored in that partition. The format of partition files is different depending on the index type. In grid index, partitions are stored as text files, while in R-tree and R+-tree, partitions organize the data in an R-tree. To simplify the parsing of the partition, the API contains an abstract class `SpatialRecordReader` which contains all the logic needed for parsing a data file. This class automatically detects the format of the partition and parses it accordingly. In addition to that abstract class, SpatialHadoop also

contains a set of concrete classes that retrieves records from the file in a specific format. All of them are instances of RecordReader which allows them to be used in MapReduce programs. To adhere with the MapReduce programming interface, each record has to be represented as a key-value pair. SpatialHadoop always uses the partition boundaries as the key represented as an object of type Rectangle. The value differs according to the specific reader instantiated/used.

### 2.3 Related work

There are several papers that propose a solution to skyline query processing in MapReduce. In this unit we present some of these and their main ideas.

*CG\_Hadoop: Computational Geometry in MapReduce* [2] by Ahmed Eldawy, Yuan Li, Mohamed F. Mokbel and Ravi Janardan. This paper proposes two algorithms, one for Hadoop and another for SpatialHadoop. The Hadoop skyline algorithm works in three steps, partitioning, local skyline, and global skyline. The partitioning step divides the input set of points into smaller chunks of 64MB each and distributes them across the machines. In the local skyline step, each machine computes the skyline of each partition assigned to it, using the traditional algorithm, and outputs only the non-dominated points. Finally, in the global skyline step, a single machine collects all points of local skylines, combines them in one set, and computes the skyline of all of them. The skyline algorithm in SpatialHadoop is very similar to the Hadoop algorithm, with two main changes. First, in the partitioning phase, it uses the SpatialHadoop partitioner when the file is loaded to the cluster. This ensures that the data is partitioned according to an R-tree instead of random partitioning, which means that local skylines from each machine are non overlapping. Second, it applies an extra filter step right before the local skyline step. The filter step, runs on a master node, takes as input the minimal bounding rectangles (MBRs) of all partitioned R-tree index cells, and prunes those cells that have no chance in contributing any point to the final skyline result.

*Parallel Computation of Skyline and Reverse Skyline Queries Using MapReduce* [3] by Yoonjae Park, Jun-Ki Min and Kyuseok Shim. This paper proposes the SKY-MR algorithm to discover the skyline  $SL(D)$  in a given data set  $D$  which consists of three phases. First is the Sky-quadtree building phase. To filter out nonskyline points effectively earlier, it proposes a new histogram, called the sky-quadtree. To speed up, it builds a sky-quadtree with a sample of  $D$  where each leaf node with non-skyline sample points only is marked as «pruned». Second is the Local skyline phase. It partitions the data  $D$  based on the regions divided by the sky-quadtree and computes the local skyline for the region of every unpruned leaf node independently using MapReduce by calling L-SKY-MR. Third is the Global skyline phase. It calculates the global skyline using MapReduce from the local skyline points in every unpruned leaf node by calling G-SKY-MR. When the number of local skyline points is small, it runs the serial algorithm G-SKY in a single machine to speed up.

*Efficient Skyline Computation for Large Volume Data in MapReduce Utilising Multiple Reducers* [4] by Kasper Møllegaard and Jens Laurits Pedersen. This paper proposes two novel algorithms, MR-GPSRS and MR-GPMRS. The main feature of the algorithms is that they allow decision making across mappers and reducers. This is accomplished by using a bitstring describing the partitions empty and non-empty state across the entire data set. In addition, the common bottleneck of having the final skyline computed at a single node is avoided in the MR-GPMRS algorithm by utilizing the bitstring to partition the final skyline computation among multiple reducers.

*Adapting Skyline Computation to the MapReduce Framework: Algorithms and Experiments* [5] by Boliang Zhang, Shuigeng Zhou, and Jihong Guan. This paper proposes MapReduce based BNL (MR-BNL), MapReduce based SFS (MR-SFS) and MapReduce based Bitmap (MR-Bitmap). The BNL algorithm (MR-BNL) is a two-stage method. The first stage is to divide the whole data into small disjoint subsets. For each subset, it runs a BNL procedure to compute the skyline. In the second stage, the local skylines are merged and filtered, thus the global skyline is obtained. In the Sort-Filter-Skyline (SFS)



algorithm the input data is sorted at first in a topological order compatible with the skyline criterion. Thus, once a record  $r$  is inserted into the window, no record following it will dominate  $r$  and  $r$  is a skyline point. Therefore, at the end of each iteration, it can output all records inside. The iteration times of SFS is optimal, i.e.,  $\lceil N/W \rceil$  where  $N$  represents the number of data records and  $W$  is the window size in number of records. In MR-Bitmap, every point is mapped into a bit vector, and the whole dataset forms the bitmap structure. The bitmap structure reflects the order of data records in all dimensions, ignoring their magnitudes. Every comparison in Bitmap is a lightweight bitwise operation. However, in order to examine a record, it needs to compare it with all the other records. Bitmap supports progressive sky line computation since a point can be returned to the user immediately once it is identified as a member of the skyline.

## Chapter 3

### Problem Setting

As mentioned, the purpose of this thesis is to implement efficient skyline query processing algorithms. The best way to accomplish this is by using the MapReduce programming model. It is ideal to process large data sets on a cluster of machines. It is possible to implement these algorithms using Apache Hadoop but SpatialHadoop is much more desirable for a lot of reasons. The “CG\_Hadoop: Computational Geometry in MapReduce” paper proposes an algorithm for this query for SpatialHadoop. This algorithm has some advantages but also many disadvantages.

#### 3.1 SpatialHadoop vs. Hadoop

The input data we want to process are 2-dimensional points. Unlike Hadoop, SpatialHadoop contains three spatial data types, namely, Point, Rectangle and Polygon as described in the second chapter. We can use the Point data type for processing our input files and to produce the output. In order to achieve a better performance, using indexed files is a good idea. Hadoop does not support indexes but SpatialHadoop does. The most important thing is that SpatialHadoop supports spatial indexes like Rtree. Rtree-indexing improves a lot the efficiency for skyline query processing. Taking into account all these this stuff, we will use SpatialHadoop to implement our algorithms.

#### 3.2 Base algorithm

First of all, we have to implement an algorithm that computes the skyline points over a file without taking into account efficiency matters. A base

algorithm that performs this task just outputs every single point from the Mapper and in the Reducer applies an algorithm to find the skyline points. This step is necessary just to make sure the algorithm computes the skyline points correctly.

### **3.3 Efficiency matters**

The “CG\_Hadoop: Computational Geometry in MapReduce” paper’s proposed algorithm applies a filter step before the MapReduce job runs. According to SpatialHadoop an Rtree-indexed file is divided into different cells. This filter step prunes those cells that have no chance in contributing any point to the final skyline result. This improves a lot the efficiency of the program. Also, it uses a combiner to reduce the total number of inputs to the Reducer. Then, the Mapper and the Reducer of the base algorithm is performed. The main problem of this algorithm is that the Mapper does not apply any filters to reduce the total number of map output records, it just outputs every point. Moreover, the Reducer does not use any optimization techniques to improve performance.

The challenge we are facing is to find additional filter steps and optimization techniques to reduce the total execution time taking into account different data distributions and volumes of data. For example, a filter step that is perfect for a uniform distributed dataset would be very bad for an anti-correlated distributed dataset with 300 thousand skyline points. The algorithms proposed in this thesis should work on both indexed and non-indexed files.

First of all, we have to apply a cells filter that prunes all the dominated cells. The second thing we have to do is to apply a filter in the Mapper to reduce the map outputs. Reducing the map outputs will improve the performance of the Reducer’s computation of the skyline points. There are two algorithms presented in this thesis and both of them apply a filter to reduce the map outputs. The filter in the first algorithm is better in cases where there are not too many skyline points and there are a lot of map input records. On the other

hand, when there many thousands of total skyline points the second's algorithm filter is better. Also, the second algorithm sorts the map output points according to their mindist that is the sum of the two dimensions. This allows us to perform some optimization techniques in the Reducer. The first algorithm also uses some optimization techniques in the Reducer which are used as filters in the Mapper. Both algorithms use a combiner. In figures 3-1, 3-2 and 3-3 are illustrated the performance steps of the CG\_hadoop and this thesis' proposed algorithms.

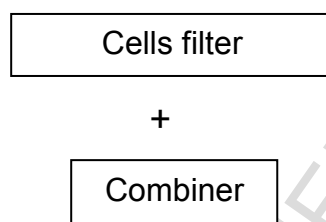


Figure 3-1: CG\_Hadoop's filters and optimization techniques

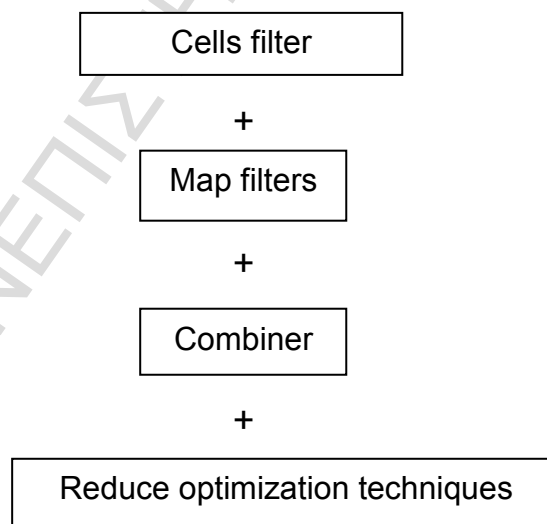


Figure 3-2: First algorithm's filters and optimization techniques

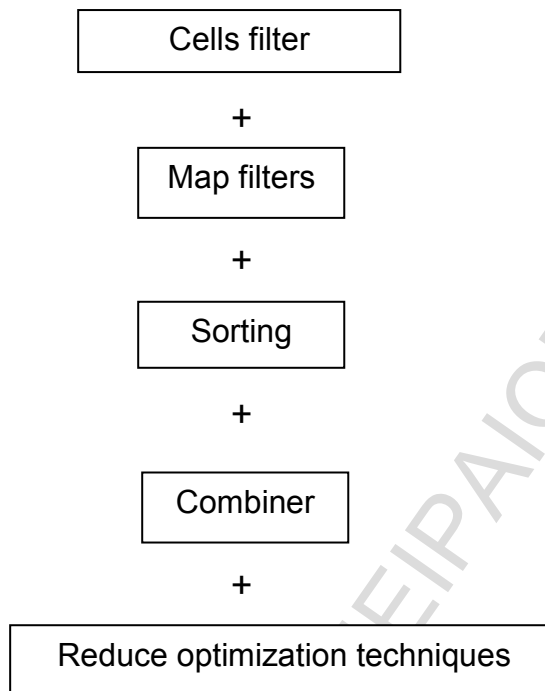


Figure 3-3: Second algorithm's filters and optimization techniques

In some cases where there are too many map input records and not too many total skyline points the sorting of the second algorithm is not preferred. Also, the filters and optimization techniques are not the same so we cannot conclude anything from these figures. As said before, everything depends on the input data we are processing. In the next chapter, we will examine more thoroughly how this thesis's algorithms work and how are they implemented.

## Chapter 4

### Algorithms

In this chapter we will discuss about the algorithms that compute the skyline points over a file. These algorithms can work with indexed and non-indexed files. We have two different algorithms with some common elements. Both algorithms use a list that keeps every single moment the current skyline points. The first algorithm uses that list in both map and reduce functions but the second one only in the reduce function. We use some filtering and optimization techniques to reduce total execution time. We will next examine thoroughly these algorithms. You can refer to Appendix A for the pseudocode of these algorithms.

First of all, we will present the common filtering techniques of these two algorithms:

- *CellsFilter*

When the input file that contains the 2 dimensional points is indexed we can use this filter. An indexed file is divided into different cells. Each cell is a rectangle e.g. starting (lower left corner) from the point (0,0) and its width is 500 and its height also 500. Many points are contained in this rectangle that their dimensions vary between 0 and 500. If another cell exists and starts at the point (1000,1000) it will be dominated by the previous one, since all its points will be dominated by that cell. In many cases, there are cells that have no chance to contain candidate skyline points so with the CellsFilter they are pruned and in this way the Mappers will have much less input records. In figure 4-1 you can see an example of a dominated cell.

- Combiner

We use as a combiner the same class we do with the Reducer. The combiner is very useful because it takes as input the output of the Mappers and computes the local skyline points. So the Reducer will be much faster because many points will have been pruned by the combiner.

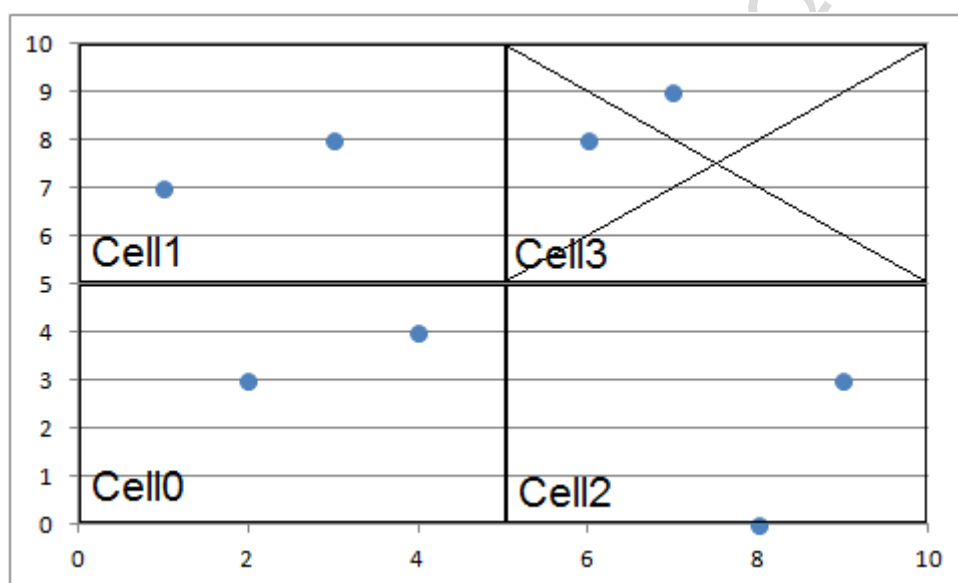


Figure 4-1: CellsFilter example, Cell3 is dominated by Cell0

#### 4.1 First Algorithm

Now, let's focus on the first algorithm. In the map function we use the list as discussed above. This list is empty before the map function begins. When the map gets first point it will output it. This point will also be inserted in the list. From now on, every point will be checked if it is dominated by the points contained in the list. If a point is dominated it will not be inserted in the list and it will not be outputted. Otherwise, we check if that point dominates some of the points contained in the list. If so, these points are removed and the new point is inserted and outputted. Our last filter in the map is to keep the point

that has the minimum mindist that is the sum of its dimensions ( $x + y$ ). Every moment we have to check if the points coming from the map are dominated from that point. This point will be also contained in the list. If a point is dominated by this one, we don't have to search the list. Otherwise, we check if the new point is the one that will be from now on the point that has the minimum mindist.

Let's see an example of how filters work in the Mappers. In this example we assume that at a certain time our list contains five points:

{20,0}, {18,1}, {15,2}, {14,4}, {13,5}

Of course, this list contains the point with the minimum mindist which is {15,2}. The point with the minimum mindist is used only to avoid searching the list. To be more specific, first we check if a point is dominated by the point with the minimum mindist. Then, we check if a point is dominated by a point contained in the list. In figure 4-2 you can see the area dominated by each filter.

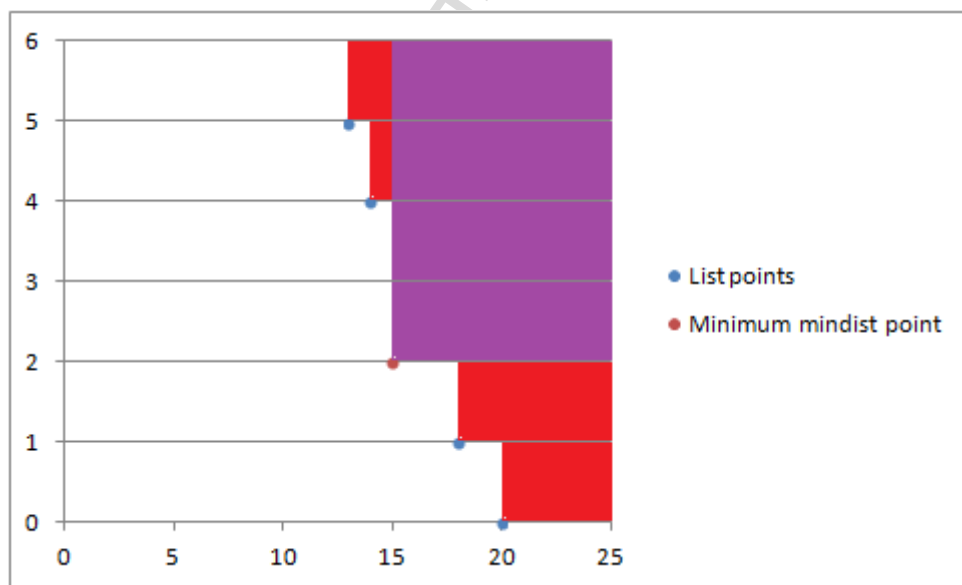


Figure 4-2: Area dominated by map filters in algorithm 1

In the Reducer we use the same filtering techniques to reduce the total execution time. What is different in the Reducer is that now every single



moment that a point is inserted in the list is not outputted at the same time. When all points outputted from the Mapper are processed in the Reducer we use an extra loop in which we output the points contained in the final version of the list. These points are the global skyline points.

## 4.2 Second Algorithm

The second algorithm works a little bit differently. The main difference is that the points are sent sorted according to their mindist in the Reducer. We achieve that by using the MapReduce sorting built-in functionality. Every time we output a point as a value we set its key as the sum of its two dimensions that is  $x$  and  $y$ . For instance, a point that its  $x$  dimension is 5 and its  $y$  is 10 is outputted with the key equal to 15. By doing this, the efficiency of the Reducer can improve because it allows us to perform some filter and optimization tricks.

The Mapper is also different than it was in the previous algorithm. We use three points to filter the output. The first one is the point with the minimum mindist(  $x + y$  ), the second is the point with the minimum  $x$  dimension and the third one has the minimum  $y$  dimension. In the beginning, these points are not being assigned to any value. When the map gets the first point its value is assigned to all three filtering points and of course it is outputted. For every new point we check if we have to update these three points. From now on, every point will be always checked by these three to decide if it will be outputted. For example, if the point with minimum mindist, the point with minimum  $x$  dimension and the point with minimum  $y$  dimension are:

{5,5}, {3,9}, {10,4}

Then the area which points will be dominated from now is shown in figure 4-3.

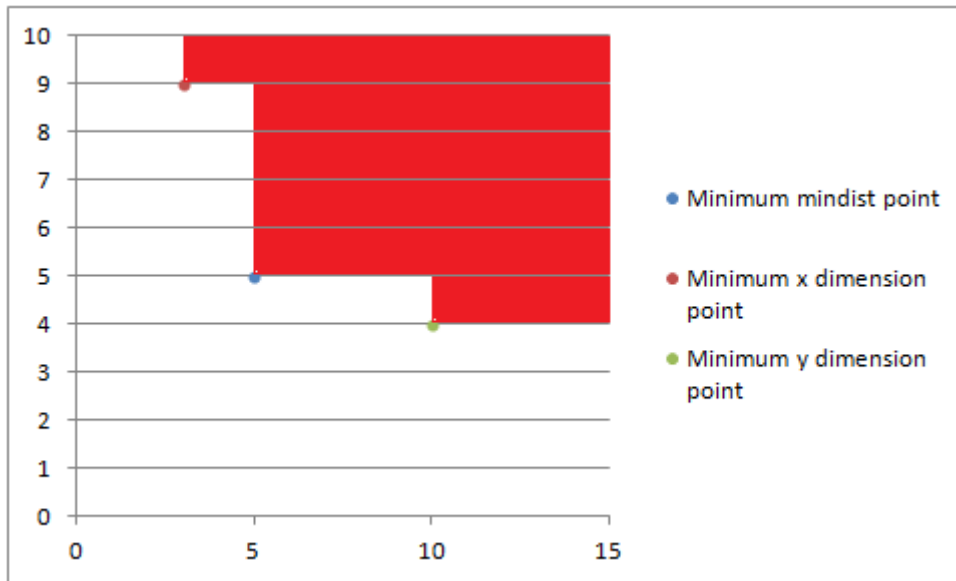


Figure 4-3: Area dominated by map filters in algorithm 2

The Reducer will take as input the points outputted from the map. All these points will be sorted in ascending according to their key which is the mindist(  $x + y$  ). This sorting allows us to implement a few techniques that were not possible in the first algorithm. The Reducer uses a list to keep the skyline points. We will only perform additions not removals in the list because the Reducer gets the points sorted so there is no chance a future point to dominate a point that has already been added in the list. Also, every point added in the list will be outputted at the same time because it is not possible to be dominated by future points so the algorithm will not spend time to output the global skyline points in an additional loop. We will also use `min_x` and `min_y` values. These values will keep the minimum x and minimum y dimensions found in the current skyline points list. The reason we want to keep these values is that we will check every new point if its x dimension is less than `min_x` or its y dimension is less than `min_y`. If so, this point is not dominated by any other points added in the list and it will be outputted and added in the list without having to search the list if it is dominated. Every moment we add a point in the list we also check if we have to update `min_x` and `min_y` values. In order to understand better how `min_x` and `min_y` works let's see an example. Assume we have found three skyline points until now:

{12,7}, {11,9}, {8,13}

In this case,  $\min_x$  is equal to 8 and  $\min_y$  is equal to 7. If a candidate point's x dimension is less than 7 e.g. 5 it will not be dominated by any of these three points and it will not dominate any of them because in this algorithm all points are sorted according to their mindist ( $x + y$ ). So if this points x dimension is 5, its y dimension will be at least 16 which is the maximum mindist found until now. This optimization technique reduces the total execution time a lot when our list contains thousands of skyline points because we don't have to search the whole list to see if a candidate point is dominated or not.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

## Chapter 5

### Implementation

In this chapter we will describe step by step how the algorithms on chapter 4 are implemented on SpatialHadoop. As explained earlier SpatialHadoop is the most suitable software to perform spatial operations in a distributed environment. We will next present the implementation of the skyline query on MapReduce adapted on both indexed and non-indexed files. In figure 5-1 is illustrated the high level code organization of both algorithms.

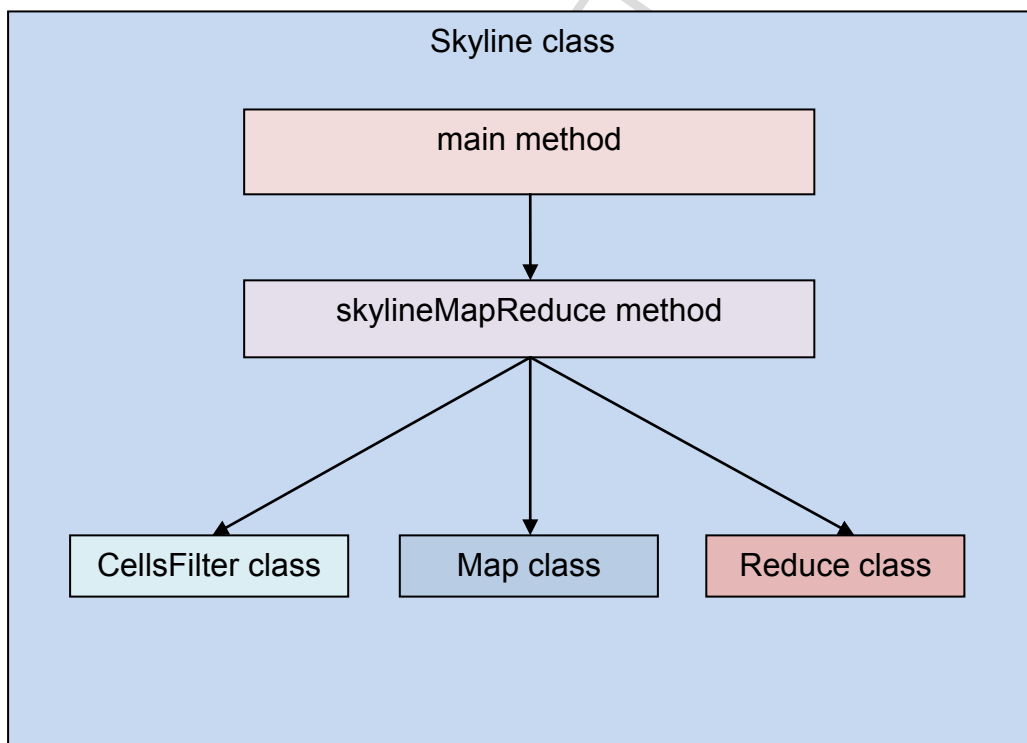


Figure 5-1: High level code organization

## 5.1 Main method

Both algorithms use a timer that calculates the total execution time. The program gets the current local machine time in milliseconds in the beginning and in the end of the main. Then, subtracts these two values and outputs it on the screen. The program takes two arguments one for the input file we want to compute the skyline points and one for the path where we want to output these points. These arguments as well as a file system object are passed into a method called `skylineMapReduce`. This method keeps the driver code and performs the cells filter step described in previous chapters.

## 5.2 `skylineMapReduce` method

The `skylineMapReduce` method contains the driver code for the job, finds the cells that are pruned and runs the job. In this unit when not mentioned, the code described will be the same for both algorithms.

In the driver code we set the number of Map and Reduce tasks, the output key and value classes. The output value class is `Point` and is the same for both algorithms because the data we are processing are 2 dimensional points. For the first algorithm, the output key class is `NullWritable` since we don't want to process the data according to their keys. On the other hand, for the second algorithm the output key class is `IntWritable` because we want to sort the data according to their keys. We also set the Mapper, Combiner and Reducer classes. By the way, the Combiner and Reducer classes are the same. Next, we set the input and output format and paths for our data. The next step is to check if the file is indexed and if so we have to find which cells are being dominated and we pass the information needed to the `CellsFilter` class. This information includes the MapReduce job object, arrays with spatial information of the cells and a Boolean array that indicates if a cell is dominated or not. All this information is passed through the `setQueryRange` method of the `CellsFilter` class. Finally, we run the job.

### 5.3 CellsFilter class

The CellsFilter class is responsible for the cells that will be processed in the map. The setQueryRange method called in the skylineMapReduce method passed the necessary information to this class. Actually, the selectBlocks method decides the cells that will be processed in our job. In this method we use the rangeQuery method and the Boolean array containing the information about the cells that have to be pruned and the other arrays with spatial information about the cells.

### 5.4 Map class

The Mapper classes are called Map in both algorithms. However, the code differs in these two algorithms. The difference is the filters used to reduce the Map outputs.

The Mapper of the first algorithm uses two different kinds of filters. The first one is a list of points called skylinePoints. This list keeps the current skyline points. The second one is the p\_minMinDist variable of Point type which is initialized to null. The p\_minMinDist keeps the point with the minimum mindist that is the sum of the x and y dimension. We also keep a Boolean variable called inserted. This variable is set to true every time we begin processing a new point in the map and is set to false if a point is dominated. In the map method we first check if the p\_minMinDist is null, if so we assign it with the first point's value. Otherwise, we check if the point is dominated by p\_minMinDist. If this condition is not true we check if the point is dominated by any point in the skylinePoints list. We also, update if needed the p\_minMinDist and the skylinePoints list. In the end, we check if the inserted variable is set to true in order to output the received point. In Figure 5-2 the map flowchart of the first algorithm is illustrated.

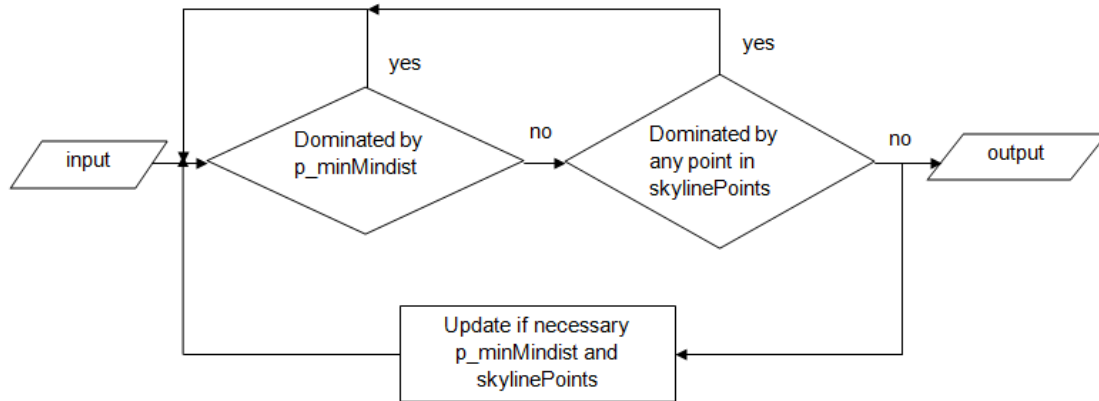


Figure 5-2: Map flowchart of the first algorithm

The only common filter in the Mapper of the second algorithm is the `p_minMinDist` point. This Mapper uses another two Point variables, the `p_minX` and `p_minY`. The `p_minX` and `p_minY` keep the points with the minimum x and y dimension respectively. In the beginning these variables are initialized to null. When the map method runs for the first time these variables are set to the first received point. Then that point is outputted. From now on, for every new received point we will check if it is dominated by any of these three points. If not, the received point is outputted. Of course, we update if needed these three variables. In Figure 5-3 the map flowchart of the second algorithm is illustrated.

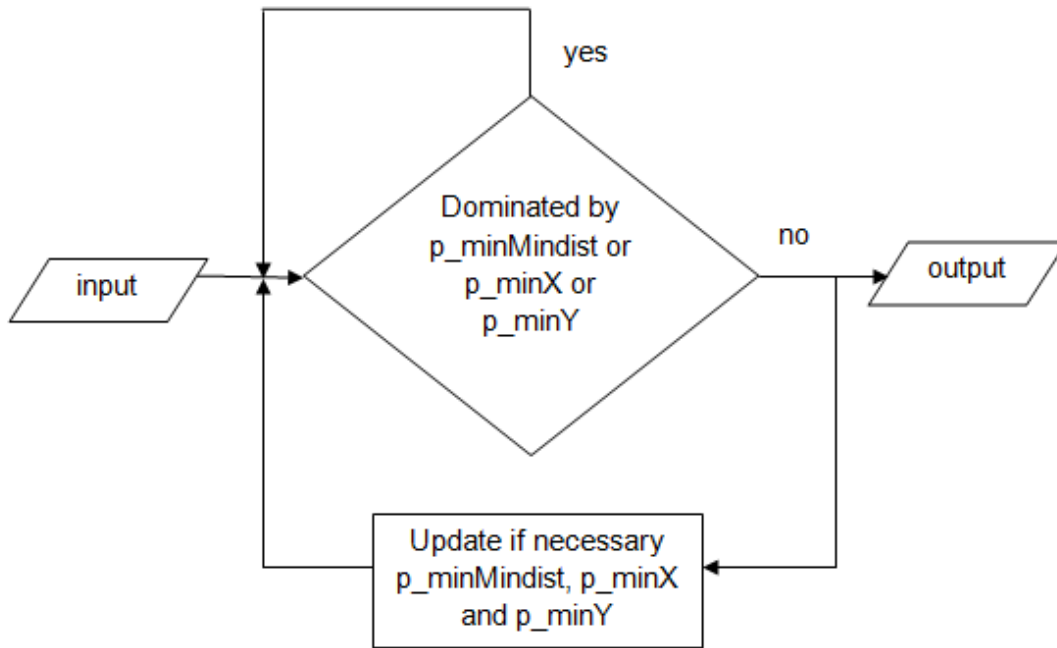


Figure 5-3: Map flowchart of the second algorithm

## 5.5 Reduce class

As said earlier, the Reducer classes are the same with the Combiner classes on both algorithms. However, the Reducer classes work in a different way in these two algorithms because the second algorithm gets the points sorted from the Mapper.

The Reducer of the first algorithm uses the variables `skylinePoints`, `p_minMinDist` and `inserted`. These variables are being updated and are used in conditions in the same way as in the Mapper. The only difference is that when a received point is added in the `skylinePoints` it is not outputted at the same time. Only if we receive all points from the Mapper in a while loop we will be sure that the `skylinePoints` list will contain the correct skyline points. When this while loop completes we output all skyline points listed in `skylinePoints` from a for loop. In Figure 5-4 the reduce flowchart of the first algorithm is illustrated.



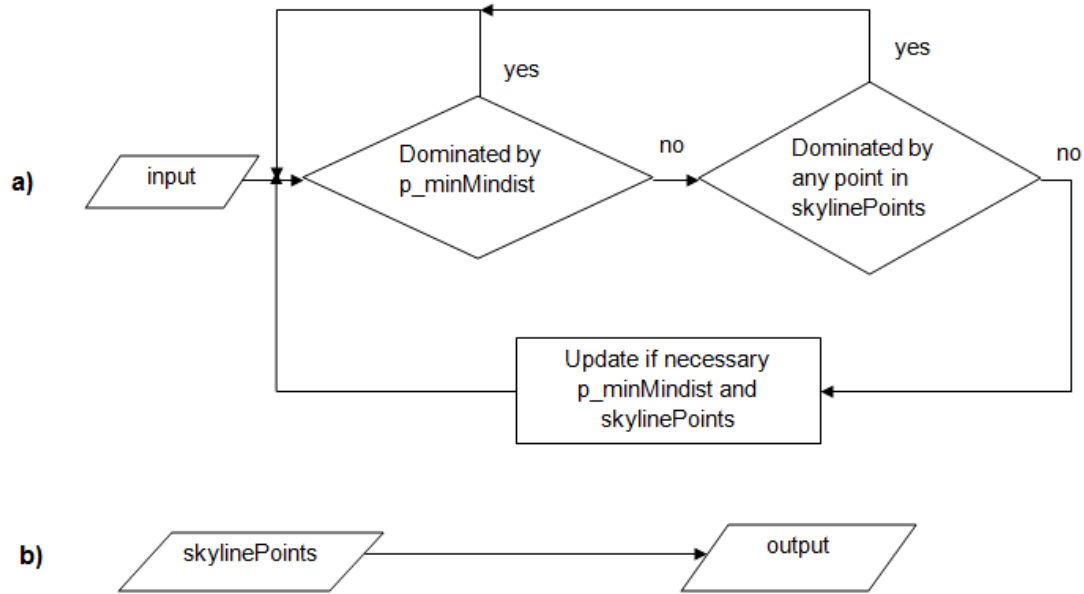


Figure 5-4: Reduce flowchart of the first algorithm

As described earlier, the second algorithm gets all the points in the Reducer sorted. This allows us to apply some important optimization techniques. The Reducer uses the skylinePoints variable like the Reducer of the first algorithm. Moreover, it uses another two variables of type long, min\_x and min\_y. These variables keep the minimum x and y found in skylinePoints list respectively and they are initialized to the maximum value that the long type can take that is 2147483647. We also use the inserted variable. The Reducer has only a while loop where it receives all points from the Mapper. In this loop, we first check if the received point's x dimension is less than the min\_x or its y dimension is less than the min\_y. If so, we know for sure that the point is not dominated by any other point. In this occasion the point is outputted. Otherwise we have to check if the received point is dominated by any of the points in the skylinePoints list. When a received point is outputted min\_x, min\_y and skylinePoints are being updated if necessary. In most cases, when points are sorted in ascending order according to their mindist, as we process points with greater mindist, one of their dimensions is less than any of the previous points. For example, if we have to process the following sorted dataset:

{10,10}, {8,14}, {16,9}, {19,7}, {5,25}, {0,32}, {35,2}, {40,0}

We can see that by using `min_x` and `min_y` variables we avoid to search the `skylinePoints` list. When we use these two variables the total number of the for loop iterations is 0. Otherwise, the number of iterations is 28. In the table 5-1 you can see this example step by step. When `skylinePoints` contains thousands of skyline points this optimization technique will improve a lot the efficiency of the computation. In Figure 5-5 the reduce flowchart of the second algorithm is illustrated.

	point		mindist	min_x	min_y	Total loop iterations	Total loop iterations without min_x and min_y
	x	y					
0	-	-	-	2147483647	2147483647	0	0
1	10	10	20	10	10	0	0
2	8	14	22	8	10	0	1
3	16	9	25	8	9	0	3
4	19	7	26	8	7	0	6
5	5	25	30	5	7	0	10
6	0	32	32	0	7	0	15
7	35	2	37	0	2	0	21
8	40	0	40	0	0	0	28

Table 5-1: `min_x` and `min_y` example

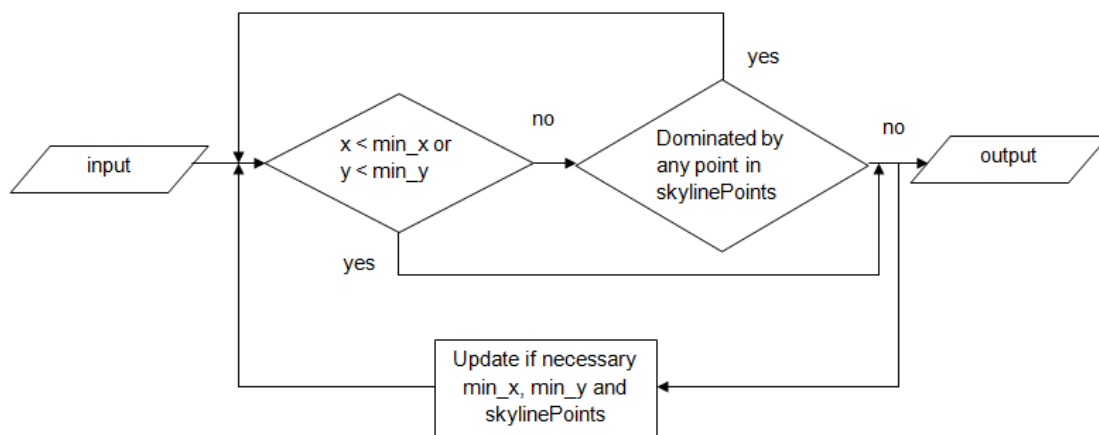


Figure 5-5: Reduce flowchart of the second algorithm

## Chapter 6

### Experimental evaluation

In this chapter we will present the experimental evaluation of the algorithms implemented by this thesis as well as the CG\_Hadoop algorithm and we will compare the algorithms presented in this thesis with the CG\_Hadoop algorithm. Also, we will present our results in tables and diagrams.

#### 6.1 Experiments

The environment we ran our experiments was a 17 nodes cluster. You can refer to Appendix B for instructions how to install and configure Hadoop/SpatialHadoop in a multi-node environment. The NameNode, SecondaryNameNode and JobTracker daemons were running on three different machines. The DataNode and TaskTracker daemons were running on all remaining machines. In order to evaluate the efficiency of the algorithms we had to run a variety of experiments with different parameters. The first parameter was the size of the input file. The two different sizes used in our experiments were 1G and 10G. The second parameter was the distribution of the data of the input file. Uniform, Correlated and two different Anti-Correlated distributions were used. The second Anti-Correlated distribution was accompanied only by indexed files because it produced many thousands of skyline points and it was too time-consuming for some algorithms. The third parameter was to use indexed and non-indexed input files. In our case, we used Rtree-indexed files. All parameters are summarized in Table 6-1. Each experiment was run 10 times and we recorded important information such as input splits, map input and output records, combine input and output records, reduce input and output records, total execution time.

Parameter	Range
Size	1G, 10G
Data distribution	Uniform, Correlated, Anti-Correlated
Index	Null, Rtree

Table 6-1: Parameters

## 6.2 Algorithms

The first algorithm as explained in the previous chapters uses a cells filter to prune the cells that are dominated. Also uses some filters that reduce the total map output points and computes the skyline points in the Reducer.

The second algorithm uses the same cells filter with the first algorithm but is different concerning the map filters. Moreover, it sorts the map output records so it computes the skyline points in the Reducer with some important optimization techniques.

The point in these experiments is to evaluate the efficiency of the algorithms described in this thesis and to compare them with the CG\_Hadoop algorithm. CG\_Hadoop algorithm also uses the cells filter step. Then, it outputs all the map output points and computes the skyline points. We have to figure out if the previous algorithms perform better than CG\_Hadoop in different circumstances.

## 6.3 Results

Firstly, we will start our experiments with the Uniform distributed data. Next, we will continue with Correlated distributed data. Lastly, we will perform experiments with two different Anti-Correlated distributions. For each,

experiment we will show a table with information. After the tables we will present diagrams comparing the efficiency of the three algorithms. Also, every diagram shows the error bars for the total execution times. With error bars we will be able to estimate if the mean represents the true total execution time.

### 6.3.1 Uniform distribution

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	16/16	16/16	16/16
<b>Map input records</b>	68000004	68000004	68000004
<b>Map output records</b>	68000004	2052	41127
<b>Combine input records</b>	68003621	2052	41127
<b>Combine output records</b>	3877	260	260
<b>Reduce input records</b>	260	260	260
<b>Reduce output records</b>	21	21	21
<b>Total execution time (10 runs)</b>	51555, 54346, 50451, 51565, 50198, 51211, 51190, 47422, 48371, 50380	41131, 43147, 43372, 42514, 35379, 35307, 41223, 43194, 41351, 41404	43150, 42153, 42170, 42153, 42352, 41306, 41379, 43183, 43126, 44329

Table 6-2: Uniform distributed data, 1G, non-indexed file

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	160/160	160/160	160/160
<b>Map input records</b>	680000020	680000020	680000020
<b>Map output records</b>	680000020	19950	417977
<b>Combine input records</b>	680035251	19950	417977
<b>Combine output records</b>	37667	2436	2436
<b>Reduce input records</b>	2436	2436	2436
<b>Reduce output records</b>	10	10	10

<b>Total execution time (10 runs)</b>	137646, 148406, 136444, 137607, 136388, 138376, 138375, 136646, 142622, 142711	93391, 94299, 91563, 94501, 90707, 97288, 90305, 92467, 90597, 96576	93603, 90275, 96332, 96667, 89499, 93335, 94320, 94252, 97340, 90527
---------------------------------------	--	--	--

Table 6-3: Uniform distributed data, 10G, non-indexed file

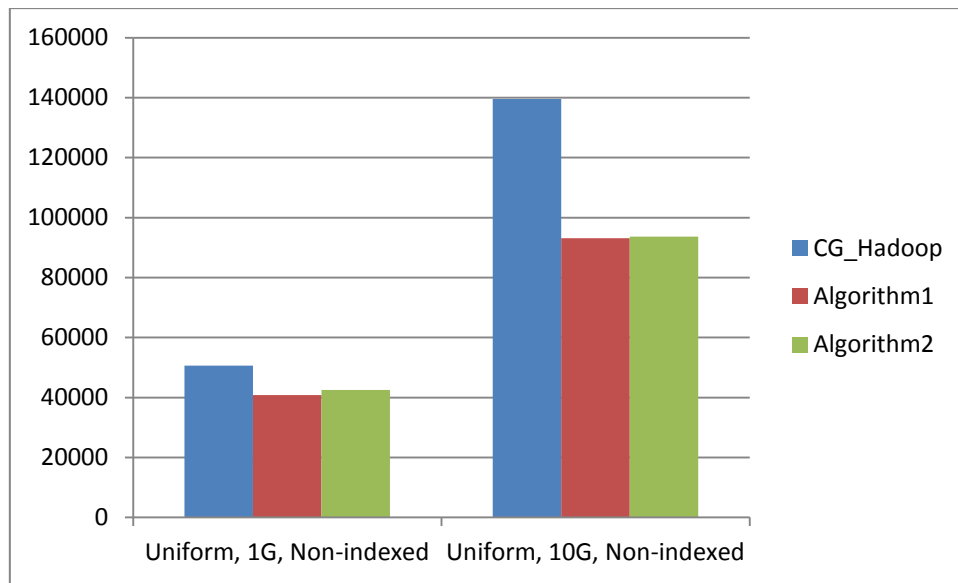


Figure 6-1: Uniform distributed data, 1G and 10G, Non-indexed file

We can see that Algorithm1 and Algorithm2 are better than CG\_Hadoop. CG\_Hadoop does not use any filters in the map so it has many points to process in the reduce. Also, in the reduce it does not use any optimization techniques. This is the reason why it performs worse. Algorithm1 is a little bit better than algorithm2 in this experiment. Algorithm2 has more map output records than algorithm1 because of the different map filters it uses. Also, it sorts the map output points and sorting is a little bit time consuming especially when we have many map output records. In this occasion, we don't have many map output records so we have only a small difference in total execution time.

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	10/20	10/20	10/20
<b>Map input records</b>	33970406	33970406	33970406
<b>Map output records</b>	33970406	1350	25840
<b>Combine input records</b>	33972175	1350	25840
<b>Combine output records</b>	1926	157	157
<b>Reduce input records</b>	157	157	157
<b>Reduce output records</b>	21	21	21
<b>Total execution time (10 runs)</b>	44408, 45167, 44163, 40116, 44400, 45156, 45536, 38095, 40124, 40415	38385, 38166, 38127, 38139, 38131, 34361, 38389, 34358, 34062, 38167	35325, 33382, 38387, 34117, 38157, 34101, 38345, 34113, 35402, 33121

Table 6-4: Uniform distributed data, 1G, Rtree-indexed file

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	28/182	28/182	28/182
<b>Map input records</b>	104528147	104528147	104528147
<b>Map output records</b>	104528147	3577	75281
<b>Combine input records</b>	104533540	3577	75281
<b>Combine output records</b>	5754	361	361
<b>Reduce input records</b>	361	361	361
<b>Reduce output records</b>	10	10	10
<b>Total execution time (10 runs)</b>	53232, 48470, 53438, 53380, 48173, 54204, 53402, 54507, 54483, 47217	41234, 42500, 41349, 41429, 41180, 41267, 41130, 47487, 41361, 44431	41450, 42186, 45387, 41416, 40476, 41155, 41157, 41197, 41358, 41349

Table 6-5: Uniform distributed data, 10G, Rtree-indexed file

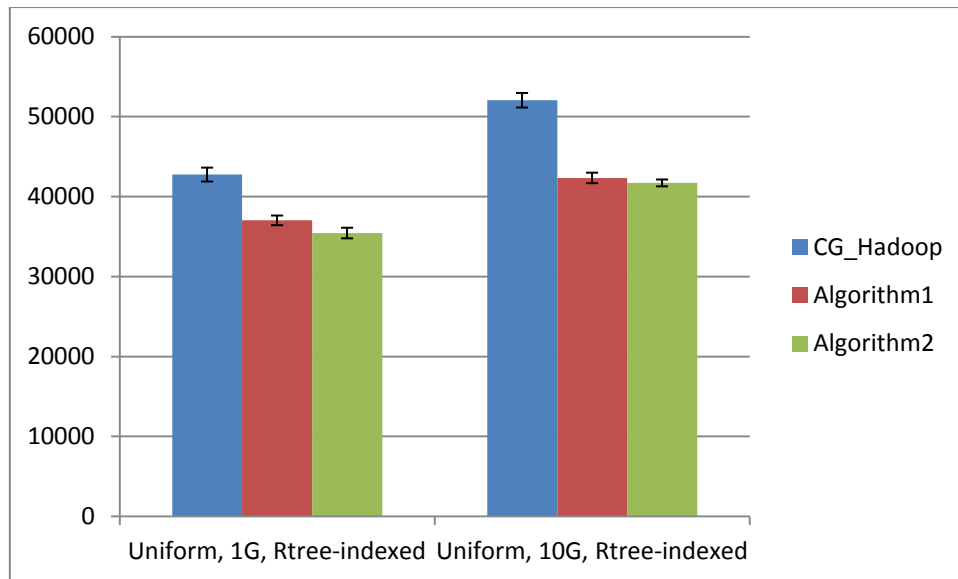


Figure 6-2: Uniform distributed data, 1G and 10G, Rtree-indexed file

Algorithm1 and Algorithm2 are better than CG\_Hadoop for the same reasons described in the previous experiment. Here, algorithm2 is a little bit better than algorithm1. Indexing helps algorithm2 to perform better with the computation of the skyline points by using some optimization techniques.

### 6.3.2 Correlated distribution

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	17/17	17/17	17/17
<b>Map input records</b>	68000002	68000002	68000002
<b>Map output records</b>	68000002	1191	4021
<b>Combine input records</b>	68002074	1191	4021
<b>Combine output records</b>	2263	191	191
<b>Reduce input records</b>	191	191	191
<b>Reduce output records</b>	9	9	9



<b>Total execution time (10 runs)</b>	49194, 49153, 51340, 49268, 52133, 49160, 51373, 49352, 49389, 51243	42152, 40119, 40274, 40338, 43341, 40123, 41478, 40352, 42423, 42344	42087, 42102, 42124, 43267, 44314, 39322, 39245, 40112, 40117, 39416
---------------------------------------	--	--	--

Table 6-6: Correlated distributed data, 1G, Non-indexed file

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	163/163	163/163	163/163
<b>Map input records</b>	680000029	680000029	680000029
<b>Map output records</b>	680000029	11062	33724
<b>Combine input records</b>	680020645	11062	33724
<b>Combine output records</b>	22313	1697	1697
<b>Reduce input records</b>	1697	1697	1697
<b>Reduce output records</b>	15	15	15
<b>Total execution time (10 runs)</b>	145765, 134410, 142410, 134611, 141576, 150592, 143449, 140583, 138403, 141899	96584, 90547, 93331, 100337, 98793, 100583, 93549, 96300, 100286, 97349	93553, 100529, 97418, 93547, 97439, 94538, 91505, 91532, 96317, 93324

Table 6-7: Correlated distributed data, 10G, Non-indexed file

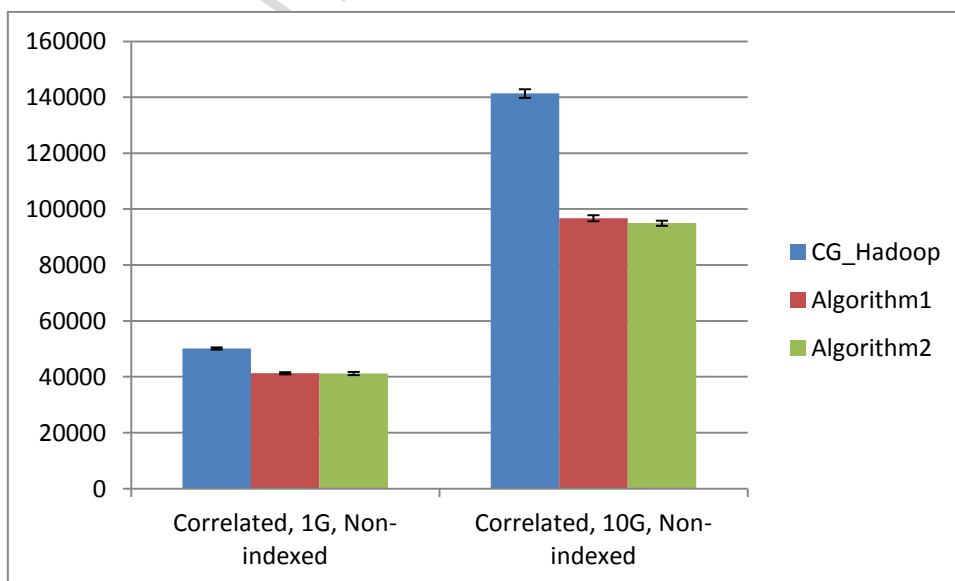


Figure 6-3: Correlated distributed data, 1G and 10G, Non-indexed file

The CG\_Hadoop is worse than the other algorithms because of the reasons described in the previous experiments. Algorithm1 and algorithm2 perform almost the same in this experiment because both of them output only a few map output records so the different processing in the map and reduce does not make a difference.

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	8/20	8/20	8/20
<b>Map input records</b>	27220734	27220734	27220734
<b>Map output records</b>	27220734	899	704094
<b>Combine input records</b>	27221989	899	704094
<b>Combine output records</b>	1356	101	114
<b>Reduce input records</b>	101	101	114
<b>Reduce output records</b>	9	9	9
<b>Total execution time (10 runs)</b>	44126, 44165, 44157, 45438, 45375, 44278, 44142, 44237, 47365, 48350	39392, 38095, 38166, 38112, 38397, 38396, 41372, 41128, 41190, 39386	35428, 35478, 38117, 38175, 35097, 38350, 38187, 38494, 38160, 38136

Table 6-8: Correlated distributed data, 1G, Rtree-indexed file

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	26/182	26/182	26/182
<b>Map input records</b>	97049893	97049893	97049893
<b>Map output records</b>	97049893	3093	4440302
<b>Combine input records</b>	97054663	3093	4440302
<b>Combine output records</b>	5080	310	504
<b>Reduce input records</b>	310	310	310
<b>Reduce output records</b>	15	15	15

<b>Total execution time (10 runs)</b>	59253, 53178, 50205, 48198, 56303, 48411, 49453, 47195, 53227, 49181	41415, 35178, 41183, 41180, 41154, 41410, 44375, 41442, 41189, 41150	51199, 47250, 47235, 47343, 47195, 47412, 47471, 50349, 47407, 41237
---------------------------------------	--	--	--

Table 6-9: Correlated distributed data, 10G, Rtree-indexed file

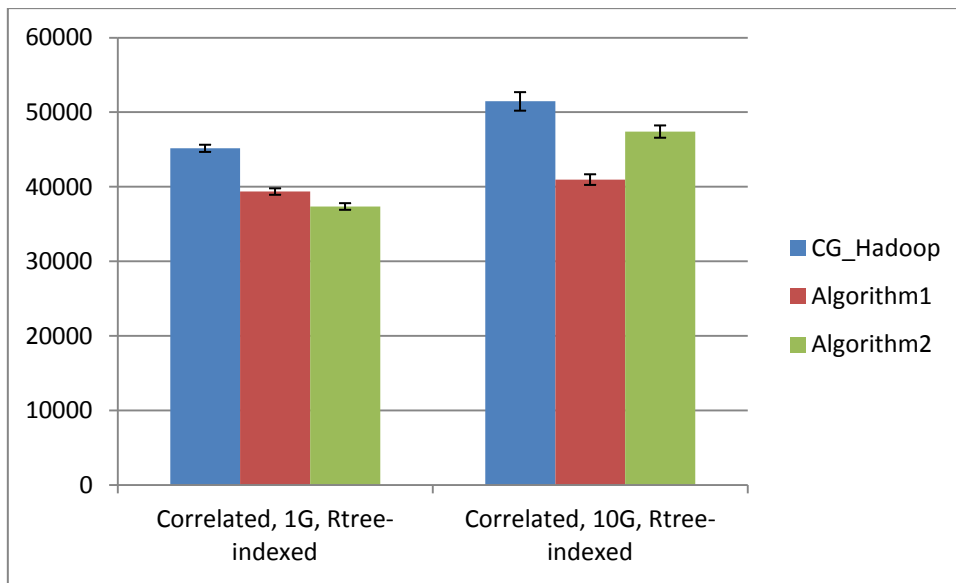


Figure 6-4: Correlated distributed data, 1G and 10G, Rtree-indexed file

Once again, algorithm1 and algorithm2 are better than CG\_Hadoop. In the 1G dataset algorithm2 is a little bit better than algorithm1 because indexing improves the performance of the reduce in a better way for this algorithm. In the 10G dataset algorithm1 is better than algorithm2. Algorithm1 map outputs are only 3093 while algorithm2 has 4440302 map outputs. Algorithm2 has much more data to process in the reduce and also it has to sort all these map output records. This is the reason why algorithm1 performs better in this situation.

### 6.3.3 Anti-Correlated distribution

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	16/16	16/16	16/16
<b>Map input</b>	68000001	68000001	68000001

<b>records</b>			
<b>Map output records</b>	68000001	313455	67585488
<b>Combine input records</b>	68749451	313455	68329019
<b>Combine output records</b>	806041	56591	800122
<b>Reduce input records</b>	56591	56591	56591
<b>Reduce output records</b>	4226	4226	4226
<b>Total execution time (10 runs)</b>	290608, 287810, 298910, 293837, 299625, 290894, 288616, 299793, 341881, 318109	435884, 446825, 451128, 397919, 388855, 410818, 401988, 467907, 428942, 427121	70382, 62202, 68399, 69434, 68433, 75451, 71422, 65222, 69150, 69244

Table 6-10: Anti-Correlated distributed data, 1G, Non-indexed file

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	160/160	160/160	160/160
<b>Map input records</b>	680000024	680000024	680000024
<b>Map output records</b>	680000024	3139739	675862632
<b>Combine input records</b>	687501173	3139739	683314695
<b>Combine output records</b>	8067244	566095	8018158
<b>Reduce input records</b>	566095	566095	566095
<b>Reduce output records</b>	4746	4746	4746
<b>Total execution time (10 runs)</b>	1349957, 1338891, 1327857, 1316040, 1280655, 1332886, 1329749, 1306915, 1346073, 1275824	1963214, 1950774, 1996999, 2001036, 1938801, 1914687, 1974196, 1962905, 1983027, 1885885	232668, 235687, 234881, 235892, 240992, 234625, 232637, 230580, 247857, 239866

Table 6-11: Anti-Correlated distributed data, 10G, Non-indexed file

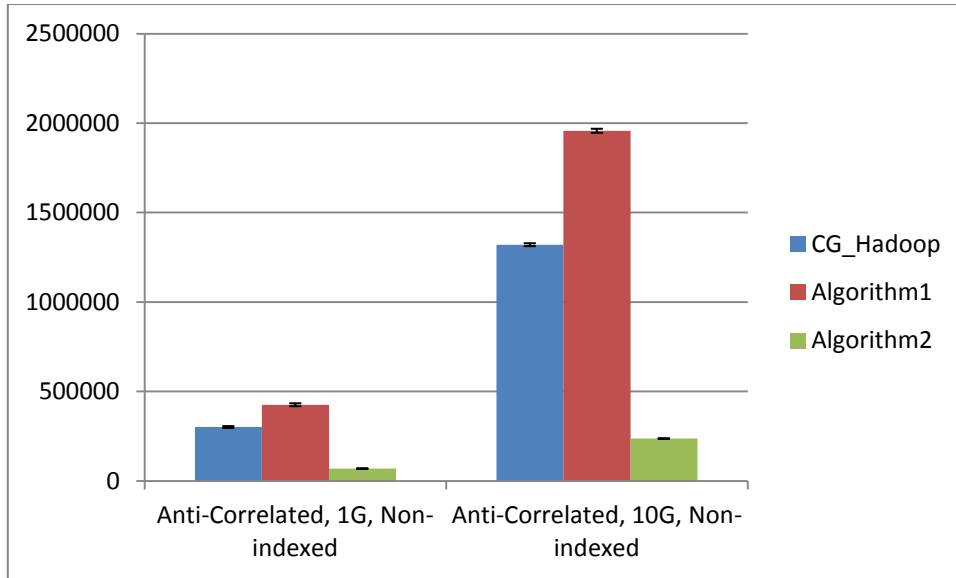


Figure 6-5: Anti-Correlated distributed data, 1G and 10G, Non-indexed file

Algorithm1 performs worse than CG\_Hadoop in this experiment because the list in the map takes a long time when there are a lot of candidate skyline points. However, algorithm2 is a lot better than CG\_Hadoop because sorting the map output records allows it to perform some important optimization techniques in the reduce.

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	20/20	20/20	20/20
<b>Map input records</b>	68000001	68000001	68000001
<b>Map output records</b>	68000001	31526	60664391
<b>Combine input records</b>	68049431	31526	60708260
<b>Combine output records</b>	53830	4400	48269
<b>Reduce input records</b>	4400	4400	4400
<b>Reduce output records</b>	4226	4226	4226
<b>Total execution time (10 runs)</b>	59256, 54502, 60357, 59409, 56171, 62125, 61182, 59392, 59331, 57400	56160, 59173, 51380, 56363, 58400, 55172, 62217, 56141, 56390, 57395	55354, 51165, 53164, 58351, 53441, 55396, 47139, 50169, 56137, 54385

Table 6-12: Anti-Correlated distributed data, 1G, Rtree-indexed file

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	160/160	182/182	182/182
<b>Map input records</b>	680000024	680000024	680000024
<b>Map output records</b>	680000024	53221	278199208
<b>Combine input records</b>	680083772	53221	278232224
<b>Combine output records</b>	89948	6200	39594
<b>Reduce input records</b>	6200	6200	6578
<b>Reduce output records</b>	4746	4746	4746
<b>Total execution time (10 runs)</b>	159538, 158788, 152727, 155908, 157522, 151624, 154797, 153853, 154729, 157502	106594, 106597, 103588, 105680, 107589, 107428, 109423, 101460, 100545, 103379	141489, 139485, 138653, 138680, 144454, 139512, 138692, 142680, 138749, 146512

Table 6-13: Anti-Correlated distributed data, 10G, Rtree-indexed file

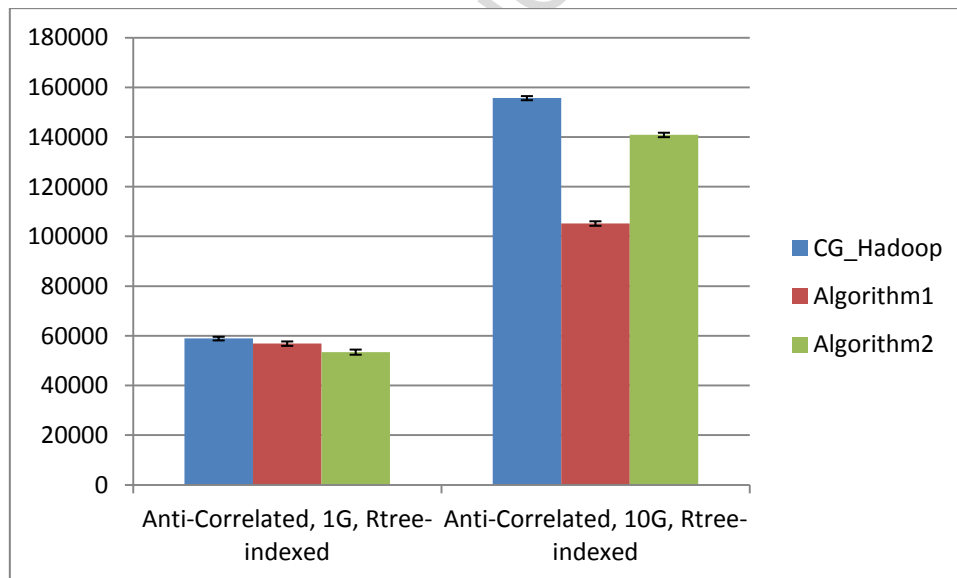


Figure 6-6: Anti-Correlated distributed data, 1G and 10G, Rtree-indexed file

In the 1G dataset algorithm2 is better than the other two algorithms for the same reasons described in the previous experiment. Indexing helps algorithm1 to produce around 90% less map output records than the previous experiment. This is the reason why algorithm1 is more efficient than CG\_Hadoop. In the 10G dataset algorithm1 is more efficient than algorithm2

because it has much less map output records than algorithm2. Algorithm2 has to sort a lot of data and has much more data to process in the reduce. These are the reasons why algorithm1 performs better than algorithm2.

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	20/20	20/20	20/20
<b>Map input records</b>	68000002	68000002	68000002
<b>Map output records</b>	68000002	1335016	67923443
<b>Combine input records</b>	70924076	1335016	70839478
<b>Combine output records</b>	3221858	297784	3213819
<b>Reduce input records</b>	297784	297784	297784
<b>Reduce output records</b>	297720	297720	297720
<b>Total execution time (10 runs)</b>	1264229, 1223253, 1237214, 1213143, 1199206, 1215345, 1167184, 1226310, 1217119, 1233394	1909542, 1997314, 1853071, 1923334, 1940543, 1930446, 1870182, 1892219, 2038671, 1876538	254559, 274801, 263537, 266580, 261823, 281795, 260781, 260488, 257518, 274512

Table 6-14: Anti-Correlated (2) distributed data, 1G, Rtree-indexed file

	<b>CG_Hadoop</b>	<b>Algorithm 1</b>	<b>Algorithm 2</b>
<b>Input splits</b>	182/182	182/182	182/182
<b>Map input records</b>	680000014	680000014	680000014
<b>Map output records</b>	680000014	2085641	672952362
<b>Combine input records</b>	684297122	2085641	677188802
<b>Combine output records</b>	4651946	354838	4591278
<b>Reduce input records</b>	354838	354838	354838
<b>Reduce output records</b>	354188	354188	354188

<b>Total execution time (10 runs)</b>	1328741, 1328777, 1335638, 1269714, 1327800, 1301545, 1290865, 1297698, 1346977, 1322697	2210032, 1806532, 1809504, 2156789, 1795359, 1794466, 2162211, 1803348, 1781537, 1805472	425269, 442986, 431952, 423284, 437244, 437146, 435275, 433221, 423098, 434211
---------------------------------------	---	---	--

Table 6-15: Anti-Correlated (2) distributed data, 10G, Rtree-indexed file

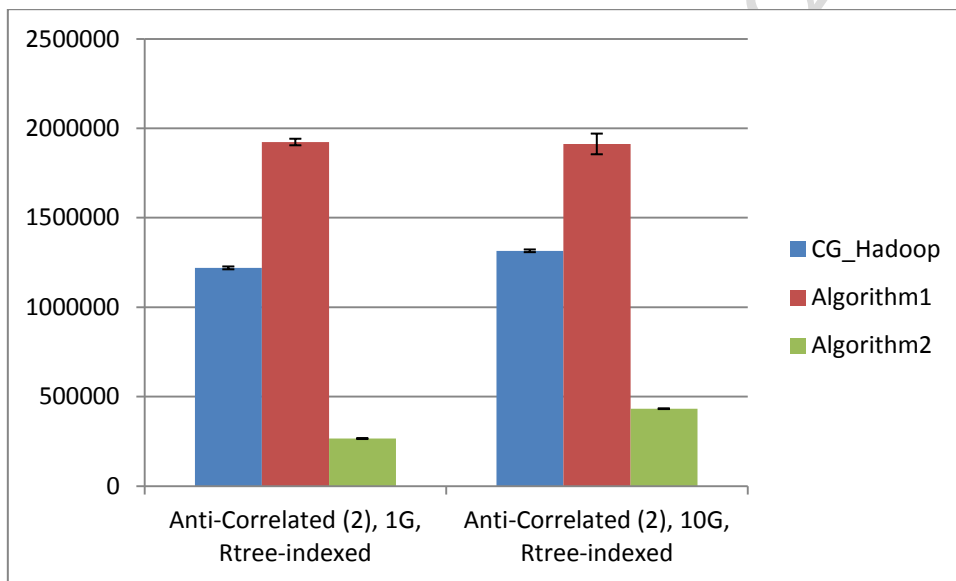


Figure 6-7: Anti-Correlated (2) distributed data, 1G and 10G, Rtree-indexed file

In these experiments we have many thousand skyline points. Algorithm2 sorts its map output records and performs some optimization techniques that improve a lot the computation that's why it performs much better than the other algorithms. Algorithm1 performs worse than CG\_Hadoop because the list it uses in the map to keep the candidate skyline points has many data to process in this situation.



## Chapter 7

### Conclusion

In this thesis, we have addressed efficient skyline query processing algorithms. These algorithms were implemented on SpatialHadoop using the MapReduce programming model. We developed two algorithms. Both algorithms have a filter step that selects only the cells that it is possible to contain candidate skyline points. The first one uses some filter techniques in the Mapper that reduces the number of total map output records to a significant degree. However, this filter can increase a lot the total execution time when it comes to anti-correlated data distributions with a huge number of total skyline points. In this occasion, the map filtering techniques of the second algorithm are preferred. The second algorithm also sorts the data. This improves the performance of the Reducer by using some optimization techniques. The idea was to combine these algorithms into a single algorithm by using a sampler to figure out which algorithm is better depending on the input data. However, the sampler was not too efficient so we ended up having two separate algorithms. We also, implemented the CG\_Hadoop algorithm in order to compare it with this thesis' algorithms. The CG\_Hadoop algorithm uses also the cells filter step described earlier. We ran a set of experiments on a cluster of 17 nodes. The most important thing was to test them in different kind of input data. Each experiment was run 10 times. From these experiments we calculated the mean, standard deviation and standard error to represent the average total execution times with error bars. In most cases, the first algorithm was better than CG\_Hadoop. The second algorithm was better than CG\_Hadoop in all cases. This thesis' algorithms perform better in different situations. To sum up, both algorithms in this thesis were efficient and it depends on what kind of data we want to process to choose which one is better. The second algorithm is much better than the first one when we

have too many total skyline points. The second algorithm performs worse than the first one only if it outputs too many map output records and for the same input data the first algorithm outputs only few map output records.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

## Bibliography

- [1] Tom White. Hadoop: The Definite Guide.
- [2] Ahmed Eldawy, Yuan Li, Mohamed F. Mokbel, Ravi Janardan. CG\_Hadoop: Computational Geometry in MapReduce.
- [3] Yoonjae Park, Jun-Ki Min, Kyuseok Shim. Parallel Computation of Skyline and Reverse Skyline Queries Using MapReduce.
- [4] Kasper Mullesgaard, Jens Laurits Pedersen. Efficient Skyline Computation for Large Volume Data in MapReduce Utilising Multiple Reducers.
- [5] Boliang Zhang, Shuigeng Zhou, and Jihong Guan. Adapting Skyline Computation to the MapReduce Framework: Algorithms and Experiments.
- [6] <http://spatialhadoop.cs.umn.edu/>
- [7] [https://hadoop.apache.org/docs/r1.2.1/mapred\\_tutorial.html](https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html)
- [8] <http://en.wikipedia.org/wiki/MapReduce>
- [9] [http://en.wikipedia.org/wiki/Apache\\_Hadoop](http://en.wikipedia.org/wiki/Apache_Hadoop)
- [10] <http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/>
- [11] <http://www.cs.umd.edu/class/spring2005/cmsc828s/slides/skyline.pdf>
- [12] <http://cloud.asperasoft.com/big-data-cloud/>
- [13] <http://www.theatlantic.com/sponsored/ibm-cloud-rescue/archive/2012/09/changing-the-world-big-data-and-the-cloud/262065/>
- [14] <http://wenku.baidu.com/view/d42d29f2524de518974b7d02.html>

## Appendix A

### Pseudocode

This appendix gives the pseudocode for the skyline algorithms of this thesis, written in a MapReduce programming paradigm.

#### A.1 Algorithm 1

```
if file is spatially indexed then
    function CellsFilter(C: Set of cells)
        for each cell c in C do
            if c is not dominated by any cell then
                Load c in Map function
            end if
        end for
    end function
end if
Initialize skylinePoints list to {}
Initialize p_minMinDist to null
function Map(p:Point)
    if p is not dominated by p_minMinDist then
        if p is not dominated by any points in skylinePoints then
            Update if necessary p_minMinDist
            Add p in skylinePoints
            Remove the points dominated by p in skylinePoints
            output(null, p)
        end if
    end if
end function
Initialize skylinePoints list to {}
Initialize p_minMinDist to null
function Combine, Reduce (null, P:Set of points)
    for each point p in P do
        if p is not dominated by p_minMinDist then
            if p is not dominated by any points in skylinePoints then
                Update if necessary p_minMinDist
```

```

        Remove the points dominated by p in
        skylinePoints
    end if
end if
end for
for each point p in skylinePoints do
    output(null, p)
end for
end function

```

## A.2 Algorithm 2

```

if file is spatially indexed then
    function CellsFilter(C: Set of cells)
        for each cell c in C do
            if c is not dominated by any cell then
                Load c in Map function
            end if
        end for
    end function
end if
Initialize p_minMinDist to null
Initialize p_minX to null
Initialize p_minY to null
function Map(p:Point)
    if p is not dominated by p_minMinDist or p_minX or p_minY then
        Update if necessary p_minMinDist
        Update if necessary p_minX
        Update if necessary p_minY
        output(x + y, p)
    end if
end function
Initialize skylinePoints list to {}
Initialize min_x and min_y to 2147483647
function Combine, Reduce (mindist, P:Set of points)
    if the x or y dimension of p is less than min_x or min_y respectively
then
        output(mindist, p)
    else
        if p is not dominated by any points in skylinePoints then
            Update if necessary min_x
            Update if necessary min_y

```

```
        Add p in skylinePoints
        output(mindist, p)
    end if
end if
end function
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

## Appendix B

### Hadoop/SpatialHadoop installation

In this appendix we will describe how to set up a linux multi-node Hadoop/SpatialHadoop cluster. To achieve this we have to follow these required steps on every single machine:

#### 1) Java 6

Java 6 is recommended for running Hadoop/SpatialHadoop. To install java 6:

```
$ sudo apt-get install sun-java6-jdk
```

#### 2) Configuring SSH

We have to generate an SSH key for our user:

```
$ ssh-keygen -t rsa -P ""
```

After this press ENTER. Then we have to enable SSH access to our local machine with this newly created key:

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

#### 3) Networking

All machines must be able to reach each other over the network. We have to update /etc/hosts on all machines. For example, if we have a cluster with six nodes. Three nodes running NameNode, SecondaryNameNode, JobTracker daemons and three slave nodes running the DataNode and TaskTracker daemons. If the IPs of these machines are 172.18.255.0, 172.18.255.1, 172.18.255.2, 172.18.255.3, 172.18.255.4, 172.18.255.5 respectively we update the hosts file with the following lines:

172.18.255.0	namenode
172.18.255.1	secondarynamenode
172.18.255.2	jobtracker
172.18.255.3	slave01
172.18.255.4	slave02
172.18.255.5	slave03

#### 4) SSH access

The user on the namenode and jobtracker must be able to connect to the users on the slave machines via a password-less SSH login. We have to upload the `id_rsa.pub` key from the namenode and jobtracker and copy them to the `authorized_keys` on every slave machine as in the step 2.

#### 5) Hadoop/SpatialHadoop upload

We must upload the Hadoop/SpatialHadoop on every machine in the cluster. This can be done with the `scp` command. For example, to upload a file called `SpatialHadoop` (located in your home folder) to the root user on namenode:

```
$ scp SpatialHadoop root@namenode:[path]
```

The path is optional. If you do not place anything after the `:` the file will be placed in the home folder.

#### 6) Hadoop/SpatialHadoop configuration

Next we have to define on which machine Hadoop/SpatialHadoop will start secondarynamenode in our multi-node cluster. To achieve this we have to update the `conf/masters` file. On the namenode we update the `masters` file with:

```
secondarynamenode
```

The `conf/slaves` file lists the hosts, one per line, where the Hadoop slave daemons (`DataNodes` and `TaskTrackers`) will be run. On both namenode and jobtracker nodes we must update the `slaves` file:



slave01  
slave02  
slave03

We must also change the configuration files `conf/core-site.xml`, `conf/mapred-site.xml` and `conf/hdfs-site.xml`, `conf/hadoop-env.sh` on all machines as follows:

#### conf/core-site.xml

First, we have to change the `fs.default.name` parameter (in `conf/core-site.xml`), which specifies the NameNode host and port.

```
<property>  
  <name>fs.default.name</name>  
  <value>hdfs://namenode:9000</value>  
</property>
```

#### conf/mapred-site.xml

Second, we have to change the `mapred.job.tracker` parameter (in `conf/mapred-site.xml`), which specifies the JobTracker host and port.

```
<property>  
  <name>mapred.job.tracker</name>  
  <value>jobtracker:9001</value>  
</property>
```

#### conf/hdfs-site.xml

Third, we change the `dfs.replication` parameter (in `conf/hdfs-site.xml`) which specifies the default block replication. It defines how many machines a single file should be replicated to before it becomes available.

```
<property>  
  <name>dfs.replication</name>  
  <value>3</value>  
</property>
```

conf/hadoop-env.sh

The only required environment variable we have to configure for Hadoop is JAVA\_HOME for example:

```
export JAVA_HOME=/usr/lib/jvm/java-6-sun
```

#### 7) Disabling IPv6

One problem with IPv6 on Ubuntu is that using 0.0.0.0 for the various networking-related Hadoop configuration options will result in Hadoop binding to the IPv6 addresses of my Ubuntu box. We can disable IPv6 for Hadoop/SpatialHadoop by adding the following line to conf/hadoop-env.sh:

```
export HADOOP_OPTS=-Djava.net.preferIPv4Stack=true
```

#### 8) Formatting the HDFS filesystem via the NameNode

To format the filesystem, run the command the following command on namenode:

```
$ bin/hadoop namenode -format
```

#### 9) Starting the multi-node cluster

Starting the cluster is performed in two steps.

- a) We begin with starting the HDFS daemons: the NameNode daemon is started on namenode, and DataNode daemons are started on all slaves:

```
$ bin/start-dfs.sh
```

- b) Then we start the MapReduce daemons: the JobTracker is started on jobtracker, and TaskTracker daemons are started on all slaves:

```
$ bin/start-mapred.sh
```

#### 10) Stopping the multi-node cluster

Like starting the cluster, stopping it is done in two steps. The workflow however is the opposite of starting.

- a) We begin with stopping the MapReduce daemons: the JobTracker is stopped on jobtracker, and TaskTracker daemons are stopped on all slaves:

```
$ bin/stop-mapred.sh
```

- b) Then we stop the HDFS daemons: the NameNode daemon is stopped on namenode, and DataNode daemons are stopped on all slaves:

```
$ bin/stop-dfs.sh
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ